



1ST EDITION

Palo Alto Networks from Policy to Code

Automate PAN-OS security policies with Python precision

NIKOLAY MATVEEV
WITH MIGARA EKANAYAKE



Palo Alto Networks from Policy to Code

Automate PAN-OS security policies with Python precision

Nikolay Matveev
with Migara Ekanayake



Palo Alto Networks from Policy to Code

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Vijin Boricha

Relationship Lead: Anindya Sil

Project Manager: Gandhali Raut

Content Engineer: Akanksha Gupta

Technical Editor: Arjun Varma

Copy Editor: Safis Editing

Indexer: Hemangini Bari

Proofreader: Akanksha Gupta

Production Designer: Aparna Bhagat

Growth Lead: Ankita Thakur

First published: August 2025

Production reference: 1270825

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-128-6

www.packtpub.com

This book is independently written and published. Palo Alto Networks, Inc. has not sponsored, endorsed, or approved any of the content within this book.

To my parents, with deep respect and enduring love. Thank you for everything.

– Nikolay Matveev

Contributors

About the authors

Nikolay Matveev holds a degree in Information Systems in Nuclear Power Engineering and has more than 25 years of IT experience, including 12 years working with Palo Alto Networks technologies. Based in London, U.K., he is a Principal Security Engineer at a major U.S. investment firm. Previously, he was a Professional Services Consultant at Palo Alto Networks and has held technical roles across financial services, consulting, and manufacturing. Nikolay is a CISSP (ISC2). He previously held certifications including PCNSC, PCNSE, MCSE, VCP-DCV, and CCNA.

Migara Ekanayake is a seasoned cybersecurity professional with over a decade of experience in cloud security, network automation, and professional services. Currently serving as a global solutions architect for cloud security automation at Palo Alto Networks, Migara has been instrumental in helping organizations worldwide secure their digital transformations. With a career spanning roles at industry leaders such as Palo Alto Networks and F5 Networks, Migara has developed a deep understanding of cloud platforms, application security, and network infrastructure. Migara's diverse background, which includes roles in software development and network security, provides him with a unique perspective on the intersections of technology, business, and security.

Acknowledgment

I am deeply grateful to the many people who, in different ways, made this book possible – some by shaping its content, others by offering support, inspiration, or encouragement when it mattered most.

In no particular order, I would like to thank:

- **Migara Ekanayake**, for writing *Chapter 6* on automation and management choices.
- **Seetal Patel** and **Ron Cowen**, for their thoughtful and constructive feedback.
- **Akanksha Gupta**, for tireless editing and steady support.
- **The wider Packt team**, for turning this manuscript into a real book — I had no idea how much work goes into publishing one!
- **Christophe Painchaud** and **Sven Waschkut**, for creating the fantastic pan-os-php toolkit.
- **James Holland**, a true Palo legend and ambassador for all things cool and inspiring.
- **Chris Motley**, the best SE ever – and an endless source of tips, advice, and sanity checks on PAN-OS when I needed them most.
- **Zia Islam**, for the sweat and blood poured into all those firewall migrations.
- **David Stern**, for encouraging creativity, offering constant inspiration, and leading by example.
- **Shehnaz Mirza**, **Colleen Valentine**, and **Anna Kandratiuk**, for making WCAR a working reality.
- **Sian Clarke**, for helping proofread some of the chapters.
- **My other half**, for putting up with the mayhem and keeping me grounded.

To all of you — thank you.

Nikolay Matveev

About the reviewers

Seetal Patel is a seasoned cybersecurity leader specializing in human and non-human identity security, with over two decades of experience in driving enterprise transformation across complex digital landscapes. Known for blending deep technical mastery with strategic foresight, Seetal has earned a reputation as a trusted advisor to global organizations navigating the convergence of IT, OT, and cloud-first architectures. Seetal has recently contributed to the NIST NCCoE Zero Trust initiative.

After a distinguished nine-year tenure at Palo Alto Networks—where he led advanced engineering, consulting, and security architecture initiatives—Seetal now serves in a senior role at Cato Networks. His focus is on empowering enterprise clients through secure platform design, strategic advisory, and operational excellence in SASE and network transformation.

When he's not deep in cybersecurity strategy, Seetal enjoys the thrill of cricket and has a lifelong fascination with military aviation.

Ron Cowen has been in the network security industry for over a decade, spanning roles at AT&T, Juniper Networks, and his current position as a senior systems engineer for Palo Alto Networks. He is based in Seattle, WA.

I want to acknowledge and thank all of those who have supported and continue to support my growth as a network security professional, as well as my wife and our two daughters.

Table of Contents

Preface	xix
Your Book Comes with Exclusive Perks - Here's How to Unlock Them	xxiv
 Part 1: Firewall Security Policy Fundamentals	 1
Chapter 1: Next-Gen Firewall Fundamentals	3
Technical requirements	4
Networking 101	4
The Open Systems Interconnection (OSI) model • 4	
<i>OSI layer 3 – Network layer • 6</i>	
<i>OSI layer 4 – Transport layer • 18</i>	
<i>OSI layers 5 to 7 – Application layer • 25</i>	
Name resolution • 28	
<i>Zones and records • 29</i>	
<i>Name resolution algorithm • 29</i>	
Load balancing • 31	
How next-generation firewalls see network traffic	36
Stateful firewalls • 37	
Example network connection • 37	
<i>Packet flow inside the firewall – App-ID • 39</i>	
<i>Packet flow inside the firewall – content identification and scanning • 53</i>	
Summary	54

Chapter 2: Navigating Real-World Firewall Management and Cyber Risks 57

Challenges of managing firewalls in real life 58

Cybersecurity risks versus you 62

Risk of unauthorized access to a restricted resource • 62

Risk of successful unauthorized external access to the network • 63

Risk of denial of service (DoS) and distributed denial of service (DDoS) attacks • 64

Risk of eavesdropping/interception of sensitive data • 64

Risk of violating applicable compliance requirements • 64

Risk of data exfiltration • 65

Conclusion • 65

Functional and non-functional requirements 66

Example company • 66

Functional requirements (FRs) • 69

Non-functional requirements (NFRs) • 70

Summary 71

Chapter 3: PAN-OS Security Policy Features: Connection Matching 73

Technical requirements 74

Security rule components 74

Connection matching 75

Rule type • 77

Source and destination zones • 77

Source and destination addresses • 78

Addresses • 78

Address groups • 80

Regions • 82

External dynamic lists • 84

Source and destination port numbers • 87

Source user • 88

Application • 88

Application categories and subcategories	90
Application characteristics and risks	91
<i>Application tags • 93</i>	
URL category • 93	
<i>Predefined URL categories • 94</i>	
<i>Custom URL categories • 95</i>	
Summary	97
 Chapter 4: PAN-OS Security Policy Features:Connection Processing	 99
Technical requirements	100
Processing the matching connections	100
Action setting • 100	
Log settings • 101	
<i>Log forwarding profile • 102</i>	
Profile setting • 106	
<i>Antivirus profile • 107</i>	
Action	107
<i>Anti-Spyware profile • 110</i>	
DNS Security algorithm • 116	
<i>Vulnerability Protection profile • 119</i>	
<i>URL Filtering profile • 120</i>	
<i>File Blocking profile • 125</i>	
<i>WildFire Analysis profile • 126</i>	
<i>Data Filtering profile • 127</i>	
Auxiliary functions • 129	
<i>Schedule • 129</i>	
<i>QoS Marking • 129</i>	
<i>Server response inspection • 130</i>	
Response pages	131
Summary	134

Chapter 5: Security Policy Design **135**

Panorama 101: device groups and templates	136
Introduction to device groups • 136	
Policy hierarchy • 139	
Introduction to templates and template stacks • 141	
Core policy principles	142
Traffic identification and classification • 142	
Policy terminology • 144	
Policy principles • 145	
Interface with end users and the IT help desk	156
Tagging and naming convention	161
Naming convention • 161	
Tagging • 163	
External dependencies: EDLs and user identity	165
User identity • 166	
External Dynamic Lists • 166	
TLS decryption	167
PKI and TLS basics • 168	
SSL Forward Proxy • 170	
SSL Inbound Inspection • 172	
SSH Proxy • 172	
Decryption policy • 173	
Legacy custom application signatures and policy rules	175
Logging	176
Summary	177

Part 2: Firewall Policy Automation **179**

Chapter 6: Firewall Automation and Management Choices **181**

Evolution of firewall management	182

CLI-based management • 182	
Journey toward full automation • 183	
APIs: The foundation of automation	184
Authentication • 184	
API scopes • 184	
PAN-OS XML API • 184	
<i>Example XML API request/response • 185</i>	
<i>Finding the XPath • 186</i>	
<i>Proxying API calls via Panorama • 186</i>	
<i>Multi-config requests • 187</i>	
PAN-OS REST API • 189	
<i>Anatomy of a REST API URL • 190</i>	
<i>Example REST API request/response • 190</i>	
XML versus REST API • 191	
Software development kits	191
Python SDK (pan-os-python) • 191	
<i>Object-oriented framework • 191</i>	
<i>Direct versus proxy firewall connection • 192</i>	
<i>VSYS awareness • 193</i>	
Go SDK (pango) • 193	
<i>Client-side filtering • 196</i>	
PHP SDK (pan-os-php) • 197	
<i>Custom scripting • 197</i>	
<i>CLI usage • 198</i>	
Infrastructure-as-code tools	198
Imperative versus declarative IaC languages • 199	
Terraform versus Ansible • 200	
Ansible for PAN-OS • 201	
PAN-OS Terraform provider • 202	
<i>Commits • 205</i>	

Choosing the right automation approach	205
Best practices and implementation guidelines	206
General automation best practices • 206	
Security considerations • 207	
Performance enhancements • 208	
Integration patterns • 208	
Summary	209
 Chapter 7: Setting Up Your Software Development Environment	 211
Configuring GitHub	212
Account creation • 212	
Copying the project source code • 212	
Setting up Python and PyCharm	214
Installation • 215	
First steps and further setup • 216	
Learning PyCharm • 218	
Project creation • 219	
Containerization with Docker	222
Docker Desktop • 222	
Docker Hub • 222	
GitHub Actions • 225	
Summary	227
 Chapter 8: Policy to Code: Foundations	 229
Technical requirements	230
First steps with the PAN-OS SDK for Python	230
Connecting to a firewall or Panorama with the PAN-OS SDK for Python • 230	
Working with applications and policies • 231	
Performance optimization • 235	
SDK methods • 243	

Separation of code and data	244
Repository structure and key definitions	245
Representation of the policy rules	254
Firewall policy creation considerations	258
Firewall policy creation algorithm	260
Summary	273
Chapter 9: Policy to Code: Advanced	275
Technical requirements	275
Dependencies and prerequisites	276
Summary of the project functionality • 276	
Zone names • 278	
Default internal addresses • 279	
Forward trust certificate • 279	
Network address translation • 286	
User-ID and Windows domain • 287	
Debugging	289
Rich traceback • 289	
IDE debugging • 291	
File logging versus console logging • 293	
Strict transactional multi-config • 294	
Logging of multi-config requests • 296	
Low-level debugging • 298	
ReplayDB • 300	
Covering SDK gaps with the XML API	303
XML API calls without XPath	310
Response pages and naming conventions	312
Response pages and Jinja2 templates • 312	
Enforcing the naming convention • 313	
Use of artificial intelligence	317
Summary	319

Part 3: Quality Assurance, Testing, and Go-Live 321

Chapter 10: Quality Assurance, Customization, and Testing 323

Technical requirements 324

Setting a user identity for testing 324

Automated tests 328

 URL filtering • 329

 DNS Security • 331

 App-ID-based rules • 334

Manual tests 343

 File blocking • 343

 Data filtering • 344

Automated incident response 345

 Geo-blocking • 351

Exceptions 352

Response pages and service desk integration 354

User acceptance testing 357

 Prerequisites for UAT • 357

 Objectives for the pilot • 358

Application coverage • 358

Experience the block pages • 358

Workflow validation • 359

 Running the test • 359

 Success criteria • 359

A note on performance 360

Summary 361

Chapter 11: Your First Cutover and Next Steps 363

Migration methodology for zero business impact 363

 Policy formation • 364

Automating preparation tasks • 367	
<i>PAN-OS-PHP utilities • 368</i>	
<i>Custom Python scripting • 373</i>	
What needs to be preserved • 376	
Deployment to a test firewall • 377	
Analyzing logs and predicting policy actions • 379	
<i>Traffic report based on App-ID • 380</i>	
<i>Report based on URL filtering • 382</i>	
The migration day • 385	
Migration of the subsequent firewalls • 387	
What you have achieved	387
Opportunities for further improvement	388
Summary	389
 Chapter 12: Unlock Your Book’s Exclusive Benefits	 391
How to unlock these benefits in three easy steps	391
Step 1 • 391	
Step 2 • 392	
Step 3 • 392	
Need help? • 393	
 Other Books You May Enjoy	 397
Index	401

Preface

Policy as Code is a way of managing firewall policies by defining them in code, much like developers manage software. This makes it possible to automate deployment and updates, reducing manual work, lowering the risk of mistakes, and improving consistency across large networks.

Palo Alto Networks' next-generation firewalls are especially prominent in this space. They extend beyond basic packet inspection, providing real-time application identification, detailed traffic analysis, threat prevention, integration with automation frameworks, and support for code-based management through APIs and SDKs.

This book introduces the fundamentals of how these firewalls operate and interpret traffic, then examines common management challenges and cyber risks. It covers policy features for connection matching and processing, strategies for effective policy design, and available options for automation.

Later chapters explain how to set up a development environment, translate policies into code at both basic and advanced levels, add quality assurance and testing, and roll out changes in a controlled manner. Examples use Python and the PAN-OS SDK for Python, with tools such as Docker and the `pan-os-php` framework included where useful.

Most examples are drawn from a typical enterprise setup where a Palo Alto Networks firewall secures internet access for users, though the concepts can be adapted to other environments.

This book does not attempt to cover every aspect of firewall configuration. Topics such as decryption, Network Address Translation (NAT), networking, device management, IoT specifics, and IT service desk workflows are only touched on or left aside. The focus is on security policy and its automation, rather than replicating the vendor's feature reference guides.

Grounded in real-world experience, the goal is to give security teams practical guidance for designing and automating firewall policies that meet business needs while upholding strong security standards.

Who this book is for

This book is intended for firewall engineers, network security engineers, technical engineering managers, technical security architects, information security consultants, and DevOps professionals aiming to automate firewall management and strengthen security practices.

Prior experience with networking, firewalls, and basic scripting is recommended to derive maximum benefit from this publication.

What this book covers

Part 1: Firewall Security Policy Fundamentals

Chapter 1, Next-Gen Firewall Fundamentals, examines the distinctions of next-generation firewalls from traditional solutions, including their methods for identifying applications and threats, alongside foundational concepts such as OSI layers and session tracking.

Chapter 2, Navigating Real-World Firewall Management and Cyber Risks, explores common obstacles in firewall administration, such as expedited implementations and ambiguous requirements, and establishes criteria for a robust policy based on a hypothetical enterprise.

Chapter 3, PAN-OS Security Policy Features: Connection Matching, details the mechanisms for traffic matching, including zones, addresses, users, applications, and URL categories, with coverage of advanced elements such as external dynamic lists.

Chapter 4, PAN-OS Security Policy Features: Connection Processing, investigates actions following a match, such as permitting or denying traffic, logging, and the application of profiles for antivirus, URL filtering, threat prevention, and other security functions.

Chapter 5, Security Policy Design, demonstrates the construction of a policy utilizing Panorama for centralized oversight, including conventions for naming, tagging, decryption, and managing legacy applications.

Part 2: Firewall Policy Automation

Chapter 6, Firewall Automation and Management Choices, reviews automation strategies, ranging from APIs and SDKs to frameworks such as Ansible and Terraform, along with their respective advantages and limitations.

Chapter 7, Setting Up Your Software Development Environment, provides guidance on configuring tools such as GitHub, PyCharm, and Docker, and acquiring the book's source code.

Chapter 8, Policy to Code: Foundations, introduces the essentials of the PAN-OS SDK for Python, project organization, data management, and a straightforward algorithm for policy creation.

Chapter 9, Policy to Code: Advanced, addresses dependencies, debugging techniques, addressing SDK limitations with the PAN-OS XML API, and customization of response pages.

Part 3: Quality Assurance, Testing, and Go-Live

Chapter 10, Quality Assurance, Customization, and Testing, focuses on testing and verifying policy features, automated incident response, accommodations for exceptions, evaluating user experience, and considerations related to performance.

Chapter 11, Your First Cutover and Next Steps, outlines a methodology for deployment with minimal disruption, requisite automation, achievements realized, and suggestions for further enhancements.

To get the most out of this book

You should have a basic understanding of networking, firewalls, and programming. Having access to a Palo Alto Networks firewall, and preferably Panorama, is essential.

Any additional setup and information required for preparation are outlined within the relevant chapters.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Palo-Alto-Networks-from-Policy-to-Code>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and X/Twitter handles. For example: “Execute the `python main.py` command.”

A block of code is set as follows:

```
from panos.firewall import Firewall  
fw = Firewall('192.168.1.1', 'admin', 'password')
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
https://{hostname}/api/?type=import&category={page_type}
```

Any command-line input or output is written as follows:

```
pip install pan-os-python
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select **System info** from the **Administration panel**.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book’s title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com/>.

Share your thoughts

Once you've read *Palo Alto Networks from Policy to Code*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Your Book Comes with Exclusive Perks - Here's How to Unlock Them

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name. Ensure it's the correct edition.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



Enhanced reading experience with our Next-gen Reader:

- ☁ **Multi-device progress sync:** Learn from any device with seamless progress sync.
- 📖 **Highlighting and notetaking:** Turn your reading into lasting knowledge.
- 🔖 **Bookmarking:** Revisit your most important learnings anytime.
- 🌙 **Dark mode:** Focus with minimal eye strain by switching to dark or sepia mode.

Learn smarter using our AI assistant (Beta):

- ⚡ **Summarize it:** Summarize key sections or an entire chapter.
- ⚡ **AI code explainers:** In the next-gen Packt Reader, click the **Explain** button above each code block for AI-powered code explanations.



Note: The AI assistant is part of next-gen Packt Reader and is still in beta.

Learn anytime, anywhere:

 Access your content offline with DRM-free PDF and ePub versions—compatible with your favorite e-readers.

Unlock Your Book's Exclusive Benefits

Your copy of this book comes with the following exclusive benefits:

-  Next-gen Packt Reader
-  AI assistant (beta)
-  DRM-free PDF/ePub downloads

Use the following guide to unlock them if you haven't already. The process takes just a few minutes and needs to be done only once.

How to unlock these benefits in three easy steps

Step 1

Keep your purchase invoice for this book ready, as you'll need it in *Step 3*. If you received a physical invoice, scan it on your phone and have it ready as either a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.



Note: Did you buy this book directly from Packt? You don't need an invoice. After completing Step 2, you can jump straight to your exclusive content.

Step 2

Scan this QR code or go to packtpub.com/unlock.



On the page that opens (which will look similar to Figure 0.1 if you're on desktop), search for this book by name. Make sure you select the correct edition.

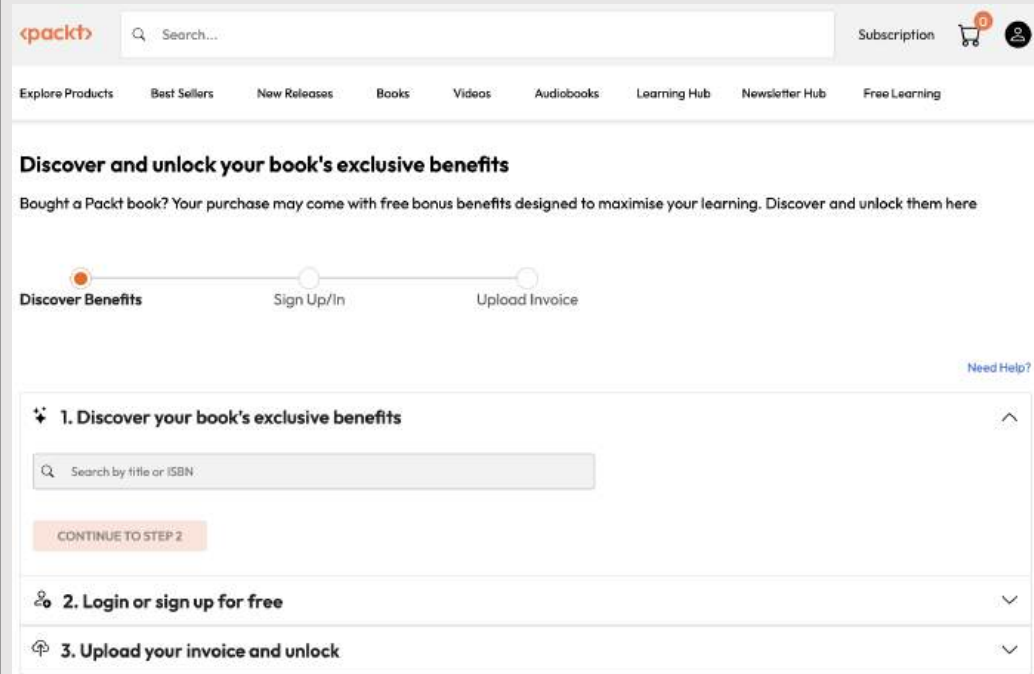


Figure 0.1: Packt unlock landing page on desktop

Step 3

Sign in to your Packt account or create a new one for free. Once you're logged in, upload your invoice. It can be in PDF, PNG, or JPG format and must be no larger than 10 MB. Follow the rest of the instructions on the screen to complete the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. The following QR code will take you to the help page directly:



Note: If you are still facing issues, reach out to customer@packt.com.

Part 1

Firewall Security Policy Fundamentals

In this first part of the book, you will build the foundation for designing and implementing firewall security policies. You will start by exploring next-generation firewall fundamentals, core networking concepts, and how PAN-OS identifies and inspects traffic. You will then examine the real-world challenges of managing firewalls and the key cyber risks they must address.

Chapters 3 and 4 provide a detailed look at PAN-OS security policy features: how the firewall matches connections and how it processes them with security profiles, logging, and auxiliary functions.

Finally, in *Chapter 5*, you will bring all this knowledge together to design enterprise-grade firewall security policies. This chapter shows how to combine traffic classification, connection processing, and user interaction into a coherent, enforceable model that can be applied consistently across your environment.

This section contains the following chapters:

- *Chapter 1, Next-Gen Firewall Fundamentals*
- *Chapter 2, Navigating Real-World Firewall Management and Cyber Risks*
- *Chapter 3, PAN-OS Security Policy Features: Connection Matching*
- *Chapter 4, PAN-OS Security Policy Features: Connection Processing*
- *Chapter 5, Security Policy Design*

1

Next-Gen Firewall Fundamentals

You may be familiar with conventional network firewalls, host-based firewalls, network access control lists, AWS security groups, and Azure network security groups. However, next-gen firewalls differ significantly from these in order to address the unique challenges organizations face in identifying and securing traffic in today's networks.

To develop a robust firewall security policy, you must understand how firewalls identify network protocols and applications, the cybersecurity risks that modern enterprise networks encounter, the core firewall security features, and how all of this correlates with business requirements.

This chapter's objective is to establish a technical foundation that will enable you to better understand what next-gen firewalls can do in the network security domain and how they accomplish this. It will also lay the essential groundwork for studying the building blocks that PAN-OS provides for creating a security policy.

We will establish requirements and create the policy in the subsequent chapters; however, now, let us focus on the following key topics:

- Networking 101 in a firewall context
- How next-gen firewalls perceive all network traffic

Technical requirements

Access to a Palo Alto Networks firewall would be beneficial, but not necessary.

Networking 101

This section provides an overview of networking to the extent required to fully understand all *network-related* building blocks of the security policy of the next-gen firewalls.



If you have a solid networking and firewall background, feel free to skip most of this section but **read the last subsection on load-balancing and name resolution**. If you have some practical knowledge but prefer a brief theory refresher or are a beginner, please read on.

The Open Systems Interconnection (OSI) model

At the high level required to understand networking in the context of a next-gen firewall security policy, the way modern networks operate is relatively simple.

First, let us start by introducing the OSI reference model, a sound conceptual framework for understanding how network communication works. The model breaks down the process of sending and receiving data into seven distinctive layers of abstraction. Each layer serves a specific purpose, relies on the layer below, and provides a foundation for the layer above. At the top of this model, we have very high-level protocols that directly interface with applications (for example, HTTP – the bread and butter of web browsers). As we go down the model, we eventually reach the Physical layer – electronic circuits, cables, and wireless media.

Here are all these layers in a table with a firewall context. Glance through the table; then, we will elaborate on the essential aspects:

Layer name	Description	Examples	Recognized by a Palo Alto Networks firewall security policy?	Recognized by a traditional firewall security policy?
Application (Layer 7)	Interacts with software applications that implement a communication component.	HTTP, SMTP, DNS, BGP	Yes	No
Presentation (Layer 6)	Responsible for data representation. Performs encoding, compression, and encryption of data.	SSL, TLS, MIME		
Session (Layer 5)	Creates, manages, and terminates sessions between two network nodes.	RPC, NetBIOS, SOCKS		
Transport (Layer 4)	Provides end-to-end communication services for higher layers. The services may include connection-oriented communication, reliability, flow control, multiplexing, and so on.	TCP, UDP, SCTP	Yes	Yes
Network (Layer 3)	Provides logical addressing and network path determination	IPv4, IPv6, ICMP, IGMP	Yes	Yes
Data Link (Layer 2)	Provides physical addressing and peer-to-peer data transfer within the same physical network segment.	Ethernet, ARP, LLDP, CDP; Wi-Fi, Zigbee	No	No

Layer name	Description	Examples	Recognized by a Palo Alto Networks firewall security policy?	Recognized by a traditional firewall security policy?
Physical (Layer 1)	Provides an electrical, mechanical, and procedural interface to the transmission medium.	Electronic circuits and chips, transceivers	No	No

Table 1.1 – OSI model in a firewall policy context

Layers 1 to 3 are known as **media layers**, and layers 4 to 7 are known as **host layers**.

As you can see from the table, the difference between next-gen firewalls and traditional firewalls is their ability to distinguish traffic attributes specific to all host layers. While traditional firewalls can see only one (Transport), the next-gen firewalls can recognize all four layers - Transport, Session, Presentation, and Application.

The OSI model is somewhat theoretical. In practice, distinguishing layers 5 to 7 from each other is often difficult, if at all possible. All programming logic of these three layers is commonly baked into a single software application – a web browser, for instance. Therefore, for the purpose of this book, we will collectively call layers 5 to 7 the Application layer – by the name of the top layer in the stack of host layers.

Now, let us elaborate on the Network, Transport, and Application layers because of their significance in the context of firewall security policy.

OSI layer 3 – Network layer

The vast majority of network communications between computer systems these days is done by means of the **Internet Protocol (IP)**. A *protocol* is the equivalent of a *language* in the human world. Subsequently, all networks where computers “speak” IP are called **IP networks**. The computer industry uses the terms *network* and *IP network* as synonyms.

All data transmitted from a hypothetical computer A to computer B is broken into small chunks called **packets**. Each packet is delivered through the network independently from other packets. Therefore, to find its way, *each* packet must have the following (at a minimum):

- The address of the *destination* (the address of computer B)
- The address of the *source* (computer A) so that a response packet can be sent if required

The IP addresses can be in two formats – IPv4 and IPv6. The former uses a 32-bit/4-byte addressing scheme (traditionally written in the so-called *dotted-decimal* form of 4 period-separated decimal numbers from 0 to 255 – for example, 198.51.100.1). When referenced, each address is often (but not always) accompanied by a **network mask** that effectively divides the address into two parts – the host’s address and the address of the network that the host belongs to. The mask, in turn, can be written either in the same dotted-decimal notation or in a so-called **Classless Inter-Domain Routing (CIDR)** notation.

For example, the address 198.51.100.1 with the mask 255.255.255.0 in the CIDR notation would be written as 198.51.100.1/24 (considerably shorter), where /24 represents the number of bits (in the binary representation of the numbers; counting from the left) that belong to the network part of the address.

The following table demonstrates how different representations of IP addresses correlate with each other.

	1st octet	2nd octet	3rd octet	4th octet
Dotted-decimal host address	198	51	100	1
Binary equivalent of the host address	11000110	00110011	01100100	00000001
Decimal mask	255	255	255	0
Binary equivalent of the decimal mask	11111111	11111111	11111111	00000000
CIDR equivalent of the mask	/24			

Table 1.2 – Representation of IPv4 address and mask

As you can see, the mask makes the most sense when it is represented in binary form. However, since decimal representation is much easier for humans to handle, it’s used most often. At the same time, the CIDR notation lets us keep a mental link to the underlying binary form (the number of significant mask bits).

In this book, we will use the shorter CIDR notation where possible.

Now, let us see how IP addresses relate to the firewall security policy.

IPv4 header in the context of firewall security policy

IP addresses and other metadata are stored at the very beginning of each packet. This section of a packet is called the **IP header**.

Here are all the fields the IPv4 header comprises and their relation to the corresponding fields in a Palo Alto Networks firewall security policy rule.

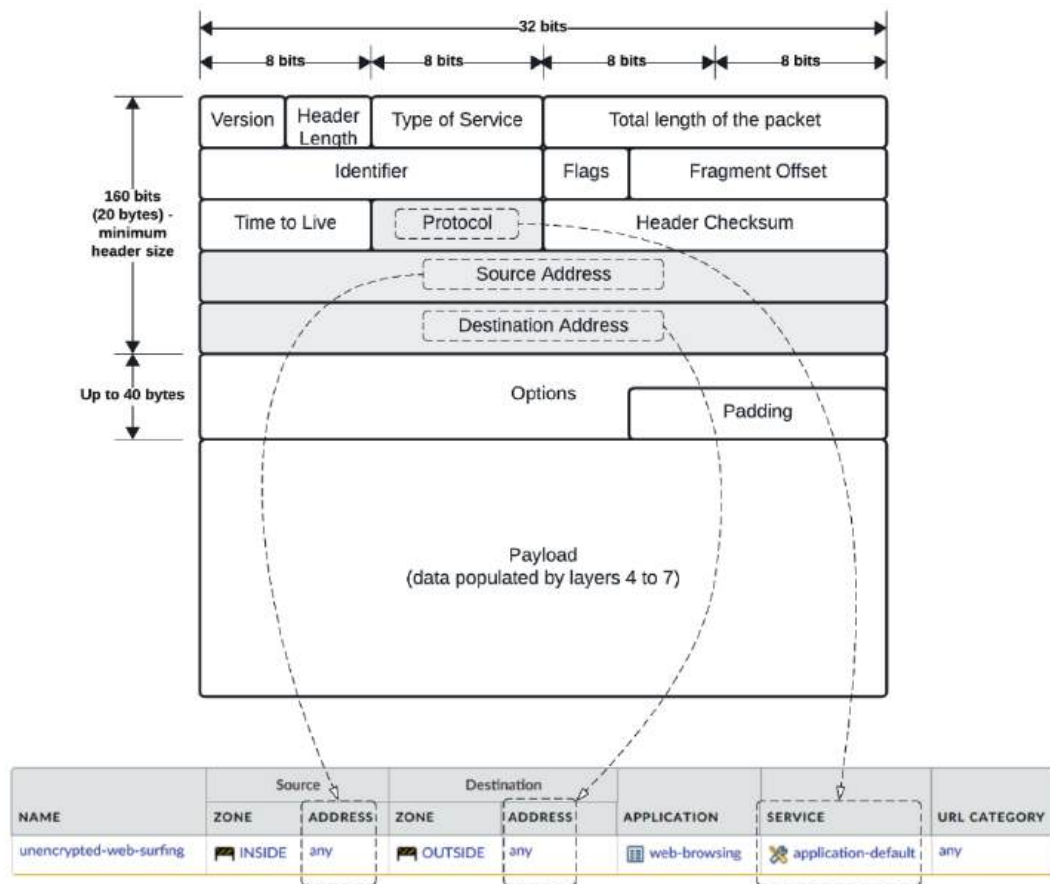


Figure 1.1 – IPv4 header and relation of its fields to the firewall security policy

As you can see, the addresses are directly mapped onto the respective fields in the policy rule. The **Protocol** specified in the header is somewhat hidden in the **SERVICE** field of the policy rule.

When the service is specified as **application-default**, the details can be found in the specified application(s) definition. A summary of the definition is available right on the security policy screen of the firewall web management console.

A little arrow will appear if you hover your mouse pointer over an application name in a firewall security policy rule. Click on that arrow for a small context menu to appear, and you will discover the application details, as shown in the next screenshot:

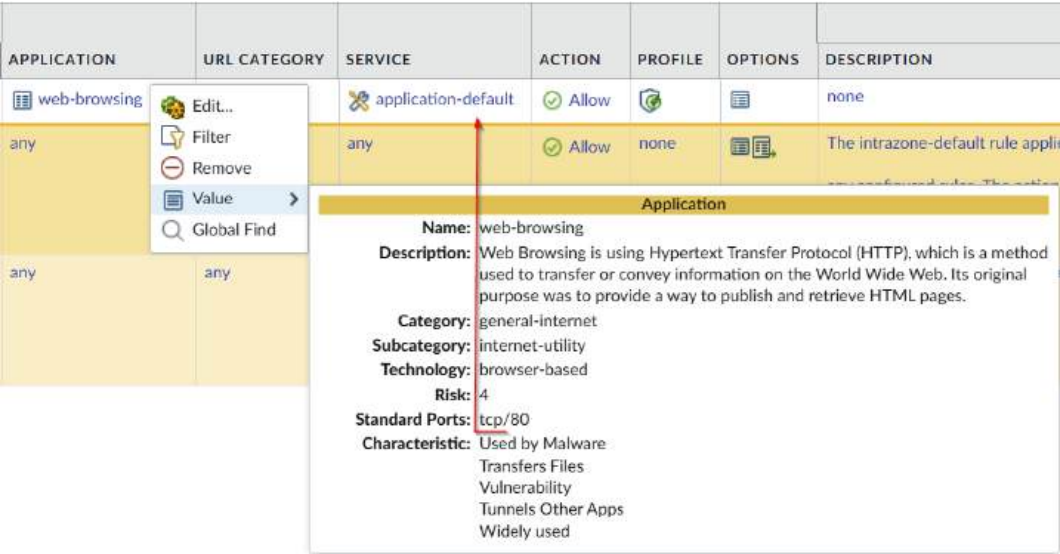


Figure 1.2 – How to find the transport protocol of an application

Among other attributes of the application, you will find the **Standard Ports** one that will tell you this application’s transport protocol (in the provided example, it is TCP). In other words, when you see the keyword **application-default**, it means that details of the IP protocol should be taken from the application definition.

Alternatively, you may have a specific service (either predefined or custom) rather than **application-default**. In such a case, you can find the same little arrow in the **SERVICE** field and expand the service definition to find its transport protocol.

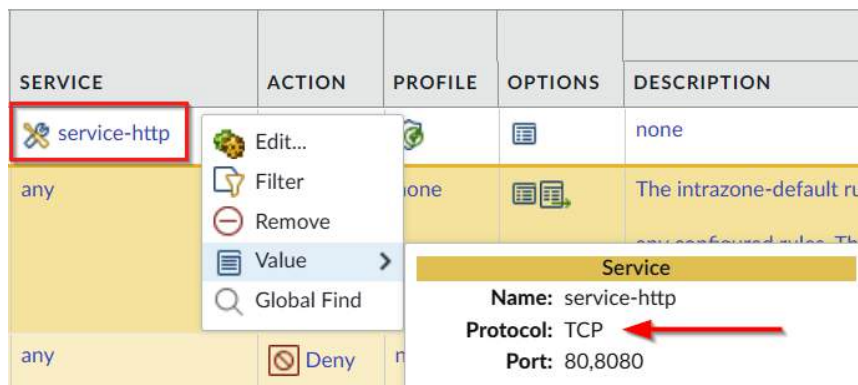


Figure 1.3 – How to find the transport protocol in a service definition

On the other hand, definitions of some applications may have *no* **Standard Ports** field.

If you look at the properties of **icmp**, for example, you will see that it has no such field in its definition. This is because there are network protocols (ICMP is one of them) that work directly over IP but do not use **Transmission Control Protocol (TCP)** or **User Datagram Protocol (UDP)** as a transport mechanism (the latter two protocols will be discussed in the next sections of this chapter). This is because such protocols do not need to carry a significant amount of data or because they have a particular use case.

They are encapsulated directly into the IP packet and therefore have their own protocol number. **Internet Control Message Protocol (ICMP)** is protocol 1. TCP and UDP are 6 and 17, respectively. IP protocol numbers are not arbitrary. They are assigned by the **Internet Assigned Numbers Authority (IANA)** and never really change.

Unfortunately, it is impossible to see the IP protocol number for an application directly from the security policy screen of **PAN-OS** (the operating system that all Palo Alto Networks firewalls run). If you try the method described previously, the web UI will display **There is no value for the selected item**. You must navigate to the application browser (**Objects | Applications | <application name>**) to see the IP protocol number corresponding to an application identified by the IP protocol number.

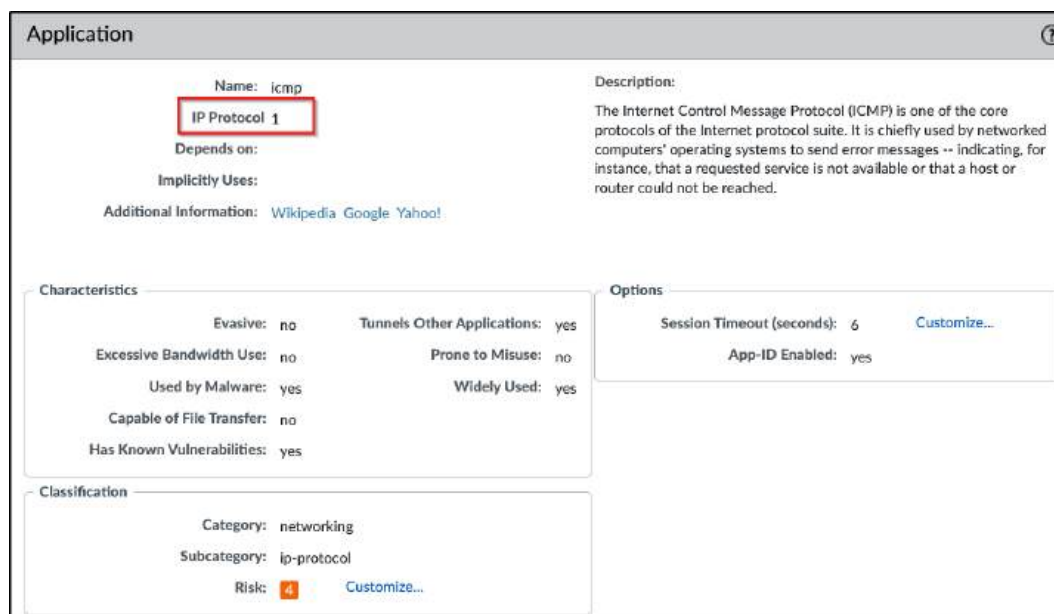


Figure 1.4 – IP protocol number of an application

There is a better way of seeing what is hidden behind the **application-default** keyword in the **SERVICE** field – we will elaborate on it later in this chapter. For now, let us finish with the structure of the IP header.

You will notice several other fields in the header:

- **Version:** Used to show the IP version; it can be only 4 or 6 at present
- **Type of Service:** Used to prioritize certain types of traffic (such as *Voice-over-IP*) when required
- **Time to Live:** Used to discard packets that crossed too many hops in the network to avoid flooding of networks by packets that cannot reach their destination for whatever reason
- **Identifier, Flags, and Fragment Offset:** Used for the *fragmentation* mechanism allowing to break a packet into fragments and reassemble them upon arrival to the destination; this can be done by intermediate routers that the packet crosses on its way to the destination
- **Header Checksum:** Used to confirm the packet's integrity; if the checksum in the packet does not match the checksum calculated by the receiving side, the packet is discarded, assuming there was a transmission error
- **Options:** Rarely used, and when used, is mostly to control the routing behavior of intermediate routers (they usually ignore this field, anyway) and for some experimental features

The firewall security policy cannot reference any of these fields.

However, the **Version** header is implicitly referenced in the system-wide firewall setting (**Device | Setup | Session | IPv6 Firewalling**) and in the IPv6 settings of individual layer 3 interfaces. Unless IPv6 is globally enabled and configured on individual layer 3 interfaces accordingly, the firewall will discard incoming IP packets with the **Version** field set to 6.

Conversely, interfaces working in the **Virtual Wire (v-wire)** mode do not explicitly specify the IP version. Thus, only the aforementioned global setting will define the firewall's ability to apply its security policy to IPv6 traffic crossing **v-wires**. We will examine interfaces and their types in a minute.

While the **Options** field of IP packets is also “invisible” for the security policy, you can still block packets with security-sensitive IP options on a per-zone basis via so-called **zone protection profiles** (found in **Network | Network Profiles | Zone Protection | <profile name> | Packet Based Attack Protection | IP Drop | IP Option Drop**), as shown in the following screenshot.

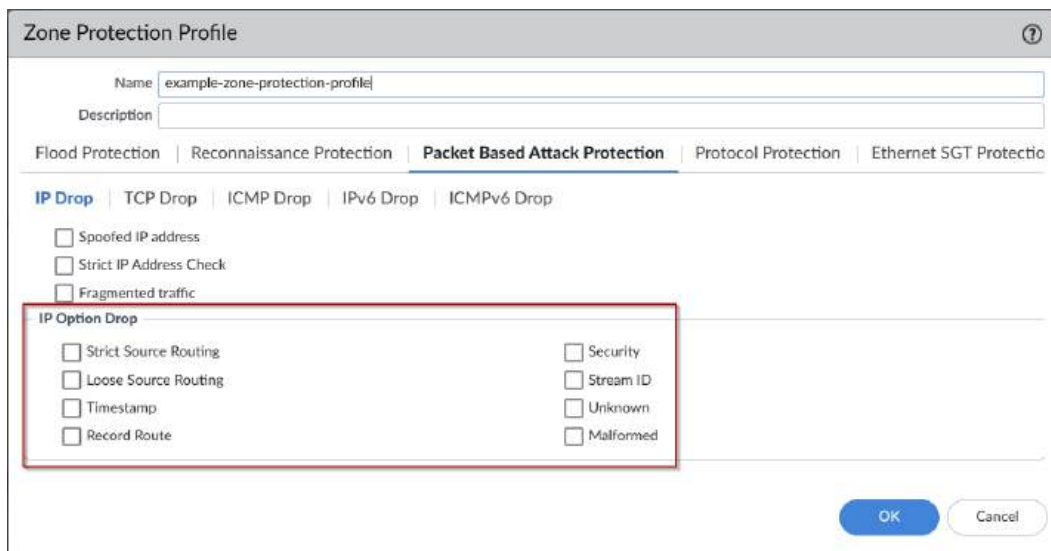


Figure 1.5 – How to discard packets with security-sensitive IP options

When you create a zone protection profile with the highlighted boxes selected and assign this profile to a *zone*, all incoming packets in that zone with such IP options set will be discarded.

Since we mentioned zones, let us briefly explain them. But before we do this, we will have to take a step back to discuss interfaces first.

Interfaces

Each device with networking capabilities connects to a network medium via a device component called a network **interface**. An interface can be either *physical* or *virtual*, and a device can have multiple network interfaces. Firewalls are prime examples of such devices.

Interfaces are rather complex constructs spanning from one to three bottom layers in the OSI model. Those that work at the Physical and Data Link layers are called Ethernet **Layer 2** interfaces.

Here is how they are represented in the PAN-OS UI:

Ethernet | VLAN | Loopback | Tunnel | SD-WAN

Layer2

INTERFACE	INTERFACE TYPE	IP ADDRESS	MAC ADDRESS	VLAN / VIRTUAL-WIRE	VIRTUAL ROUTER
 ethernet1/3	Layer2	none	dc:0e:96:fd:2f:12	LAB1	none
 ethernet1/4	Layer2	none	dc:0e:96:fd:2f:13	LAB1	none
 ethernet1/5	Layer2	none	dc:0e:96:fd:2f:14	LAB1	none
 ethernet1/6	Layer2	none	dc:0e:96:fd:2f:15	LAB1	none

Figure 1.6 – Layer 2 physical interfaces

As you can see, these interfaces do not have an IP address, but they do have a **MAC** address, which is a hardware address that each physical interface has. Unique ranges of MAC addresses are allocated to each hardware manufacturer.

Layer 2 interfaces can be grouped via a construct called **VLAN** (short for **Virtual Local Area Network**). VLAN effectively defines the boundaries of a layer 2 network – in other words, it defines how far a layer 2 broadcast or unicast packet can travel.

The following screenshot illustrates how a VLAN is defined in PAN-OS. As you can see, it groups layer 2 interfaces together:

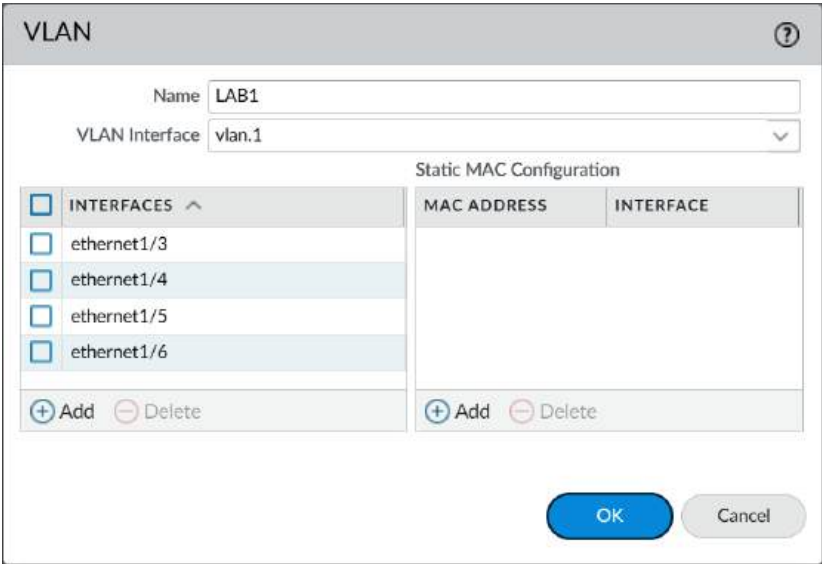


Figure 1.7 – VLAN configuration

A packet entering one of the listed interfaces can directly reach only the other interfaces within the same VLAN.

Confusingly enough, there are *virtual* interfaces that are also called VLAN – luckily, with an index at the end (vlan.1, for example) – you can also see one on the preceding configuration screen.

These interfaces have no physical components (hence the name), operate across *Data Link* and *Network layers*, and are necessary to establish communication between separate VLANs or network segments.

Here is an example of a VLAN interface:

Ethernet VLAN Loopback Tunnel SD-WAN			
Q vlan.1			
INTERFACE	IP ADDRESS	VIRTUAL ROUTER	VLAN
vlan.1	192.168.51.254/24	default	LAB1

Figure 1.8 – VLAN interface

As you can see, it has an IP address, and it is always linked with a **virtual router** and a VLAN. Effectively, this interface is a gateway between one VLAN and other VLANs and interfaces. After a packet egresses the VLAN via this gateway, it follows routes defined in the virtual router to find its destination (i.e., another VLAN or an interface).

The next, very special, interface type is a **loopback** interface. Loopback interfaces operate at layer 3 only and can be used to host various software services a firewall may provide (for example, a *global protect gateway*).

Finally, interfaces that span all three media layers—Physical, Data Link, and Network—are called Ethernet **layer 3**. Because these interfaces are simple to use (all-in-one), they are the most popular interface type in corporate firewall deployments.

INTERFACE	INTERFACE TYPE	IP ADDRESS	VIRTUAL ROUTER
 ethernet1/1	Layer3	Dynamic-DHCP Client	default
 ethernet1/7	Layer3	10.1.1.1/24	default
 ethernet1/8	Layer3	10.1.2.1/24	default

Figure 1.9 – Layer 3 physical interfaces

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Other special interface types include **tunnel**, **virtual wire**, **High Availability (HA)**, **decrypt mirror**, **aggregate Ethernet**, **SD-WAN**, and **subinterfaces**. However, they all require a more detailed description that would be outside the scope of this book focused on firewall security policy.

Zones

Interfaces cannot be included in the security policy of Palo Alto Networks firewalls because these firewalls are *zone-based* rather than *interface-based*. A **zone** is a collection of firewall interfaces of the *same* type connected to one or more network segments. It is usually considered the best practice to include network segments with endpoints of the same or similar risk/trust level and/or purpose. For example, you may want to include all your desktop computers in one zone and all your servers in another.

However, the zone-based approach is not a limitation. It does not lack granularity because you can easily create a zone with just one interface. You will better understand the role zones play as we elaborate on the packet flow inside the firewall later in this chapter.

Types of IPv4 addresses

We have now covered most of the aspects of IPv4 except the really important one – the *types* of IPv4 addresses. There are quite a few of them. The IPv4 address space of 4.3 billion possible addresses (2³²) is divided into blocks (subnets) of various sizes. Each block has its unique purpose.

Look at some of them in the following table.

Address block	Number of addresses	Description
192.168.0.0/16	65,536	Local communications within a private network
172.16.0.0/12	1,048,576	Local communications within a private network
10.0.0.0/8	16,777,216	Local communications within a private network
198.51.100.0/24, 203.0.113.0/24, 192.0.2.0/24, 233.252.0.0/24	1,024 in total	Documentation and examples
169.254.0.0/16	65,536	Link-local addresses (used for host self-configuration within a network that has neither static addresses nor DHCP service to allocate dynamic ones)
224.0.0.0/4	268,435,456	Used for multicast (one-to-many) traffic

Address block	Number of addresses	Description
240.0.0.0/4	268,435,456	Currently reserved for future use
127.0.0.0/8	16,777,216	Loopback addresses to the local host
255.255.255.255/32	1	Local segment limited broadcast address
0.0.0.0/8	16,777,216	Current network (software and special use – for example, to specify default gateway)
100.64.0.0/10	4,194,304	Shared address space for Carrier-Grade Network Address Translation (CGNAT)

Table 1.3 – Types of IPv4 addresses

There are a few other less significant small address ranges, but most of the rest of the IPv4 address space (approximately 3.7 billion addresses) comprises regular unicast **public IP addresses**.

A public IP address is an Internet Protocol address that is globally unique and routable on the public Internet, allowing devices assigned such addresses to communicate directly without translation.

Ranges of *public* IP addresses are allocated to **Internet Service Providers (ISPs)**. The ISPs, in turn, allocate subranges of the ranges they own to their customers. The size of these subranges varies from one address for small businesses and individual broadband customers to millions of addresses for big corporations. Apple, for example, owns the whole 17.0.0.0/8 subnet—a whopping 16.7 million addresses.

Network address translation

This table's first three address ranges (192.168.0.0/16, 172.16.0.0/12, and 10.0.0.0/8) are also widely known as *private* or *RFC 1918* addresses. They are *not* routable on the public internet and are used only in enterprise and home networks. To communicate with the internet, these private IP addresses must be *translated* into public addresses.

When a packet egresses from a host with a private IP address and heads toward the internet, it eventually reaches a network perimeter device that segregates the private network from the internet. In a home network, such a device is your broadband router. In a corporate network, this device is often a firewall. When it forwards the packet further to the internet, it substitutes the source address in each IP packet with the address of its own public/external interface and records this mapping in a special table in memory. When a response packet arrives, and its destination address matches the public interface address of the firewall, the firewall checks its NAT table. It then replaces the destination address with the address of the original host before forwarding the packet accordingly.

This process is called **Network Address Translation (NAT)**. While the concept is very simple at this high level, the devil is in the details. In practice, the number of internal endpoints with private addresses far exceeds the number of available public addresses your ISP allocated. This also requires translating the *source port* number to avoid clashes between hosts wanting to use the same source port number. This type of NAT is called **DIPP** (short for **Dynamic IP and Port**) NAT.

The meaning of a **port** is explained in the *OSI layer 4 – Transport layer* section further on.

IPv6

The IPv6 protocol was created to address the problem that public addresses in the IPv4 address scheme have been exhausted while the internet keeps growing as more devices get online daily.

While 3.7 billion public addresses may seem like a lot, it really is not when you consider what is currently required at the scale of the whole planet.

IPv6 uses 128-bit long addresses, thus providing for 2^{128} or some 3.4×10^{38} total addresses. This is *considerably* more than the 4.3 billion addresses that comprise the IPv4 address space. IPv6 does not differ from IPv4 in the context of a firewall security policy, but since most of you are more familiar with the latter protocol, we will use it in all examples in this book.



When you expand this 3.4×10^{38} number, it becomes 340,282,366,920,938,463,463,374,607,431,768,211,456 or approximately 340 undecillion (340 trillion trillion) addresses.

OSI layer 4 – Transport layer

Strictly speaking, hosts do not communicate with each other – processes running on these endpoints do. Many processes can run on the same endpoint simultaneously and want to communicate with remote processes on other endpoints.

On the other hand, only the endpoint as a whole (or its interfaces, to be precise) possesses IP addresses; processes do not. Thus, to address a process running on a host, we need another dimension added to the addressing scheme provided by the Network layer. This is what the protocols at the transport layer do, among other things.

Another header is added after the IP header of a packet. When the **Protocol** field in the IP header is set to **6** (for TCP) or **17** (for UDP), it means that the next header that follows is a *transport* header containing source and destination port numbers in addition to other metadata.

Effectively, a port is an address of a software process running on a host. One port can be used only by one process at a time. There can be 65,535 unique port numbers, which translates to roughly the same maximum theoretical number of different processes on a given host that can independently communicate – all at the same time. In practice, the maximum number of ports in use on a host is limited by various factors and is usually much lower than the said number. This addition of the port numbers at the Transport layer is what we referred to in the earlier OSI table as the *multiplexing* feature.

Two main protocols dominate the Transport layer – TCP and UDP. While they both support multiplexing, there is a fundamental difference between them. The former protocol is **connection-oriented**, while the latter one is **connectionless**.

Let us briefly review them both.

TCP

All application developers love TCP because it provides a range of sophisticated and helpful features that allow them to focus on application logic rather than data transfer mechanics. These features include reliable, ordered, error-free delivery, retransmission, and congestion avoidance. Unfortunately, they come at the price of the need to establish and maintain a communication channel (before any application data can be transmitted) and, thus, increased latency. The communication channel is established via a so-called **three-way handshake** – a packet exchange where peers confirm each other's readiness for the data transmission and agree on key parameters of the connection.

Then, the actual data exchange occurs, and the connection is eventually torn down. As you can see, all handshake and connection termination messages must be acknowledged, including portions of data sent in chunks called **windows** during the data exchange stage. These windows are defined by the size of the receive memory buffer that each side of the connection allocated and advertised during the handshake.

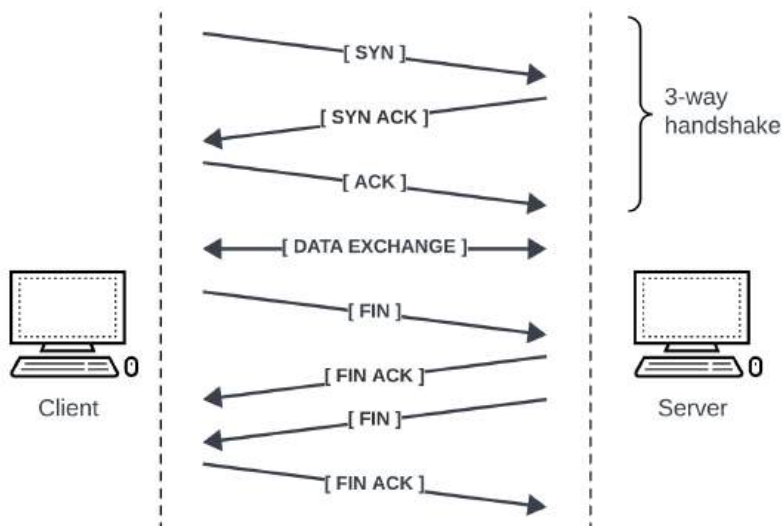


Figure 1.10 – Typical TCP connection (communication channel)

This is a *very* high-level diagram. It does not mention the names of states the connection transitions through during its lifetime. Neither does it explain important TCP mechanisms, such as acknowledgments. In fact, there is so much more that can be explained about this protocol that books have been written on this topic.

Let us just mention that in addition to the graceful termination of TCP connections shown in the preceding figure, there is a way to abruptly terminate a connection by sending a so-called RST message (TCP packet with RST flag set) to one of the peers. This is one of the methods by which firewalls can terminate a TCP connection.

Stateful firewalls track the current state of each TCP connection they pass through. This tracking ensures that all connections behave according to respective internet standards **Request for Comments (RFCs)** and do not take up too much of the firewall's resources and resources of the endpoints it secures. State tracking allows the firewall to be certain that the application data transferred across the connection will have been inspected in its entirety. The latter is the main reason why, by default, a stateful firewall would reject a TCP packet with data without seeing the respective full TCP handshake prior to it.

TCP state tracking involves strict timing of the connection states. For example, the three-way handshake phase must take no longer than 10 seconds.

There are system-wide settings defined at **Device | Setup | Session | Session Timeouts**.

Session Timeouts	
Default (sec)	30
Discard Default (sec)	60
Discard TCP (sec)	90
Discard UDP (sec)	60
ICMP (sec)	6
Scan (sec)	10
TCP (sec)	3600
TCP handshake (sec)	10
TCP init (sec)	5
TCP Half Closed (sec)	120
TCP Time Wait (sec)	15
Unverified RST (sec)	30
UDP (sec)	30
Captive Portal (sec)	30

Figure 1.11 – Global session timeouts, including TCP-related ones

Three of these timeouts can be overridden in individual applications' definitions. You can do this in **Objects | Applications | <application name> | Options | <timeout name> | Customize....**

Application

Name: web-browsing

Standard Ports: tcp/80

Secure Ports: tcp/443

Depends on:

Implicitly Uses:

Deny Action: drop-reset

Additional Information: [Wikipedia](#) [Google](#) [Yahoo!](#)

Description:

Web Browsing is using Hypertext Transfer Protocol (HTTP), which is a method used to transfer or convey information on the World Wide Web. Its original purpose was to provide a way to publish and retrieve HTML pages.

Characteristics

Evasive: no

Excessive Bandwidth Use: no

Used by Malware: yes

Capable of File Transfer: yes

Has Known Vulnerabilities: yes

Tunnels Other Applications: yes

Prone to Misuse: no

Widely Used: yes

Options

TCP Timeout (seconds): 3600 [Customize...](#)

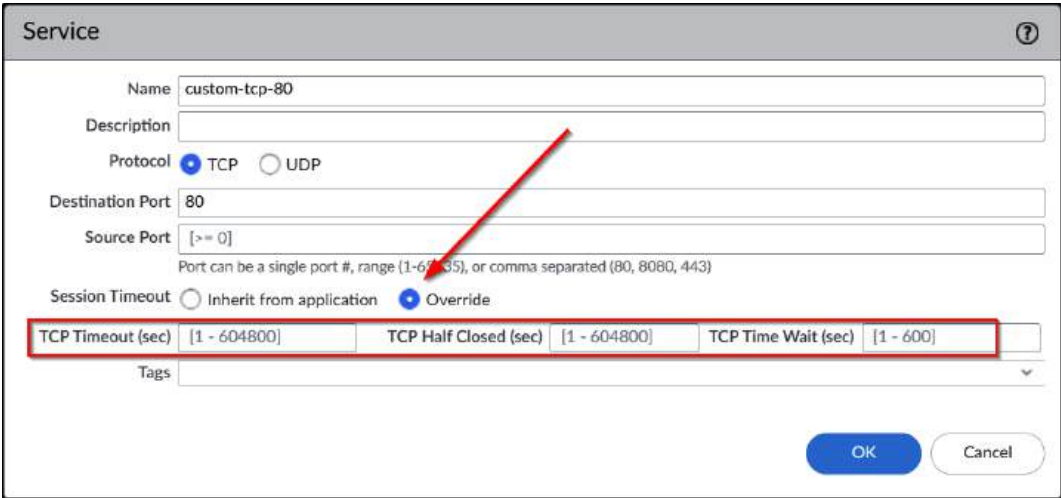
TCP Half Closed (seconds): 120 [Customize...](#)

TCP Time Wait (seconds): 15 [Customize...](#)

App-ID Enabled: yes

Figure 1.12 – How to override an application's TCP timeouts

The same can be done at the **Service** level as well:



The screenshot shows the 'Service' configuration window for a custom service named 'custom-tcp-80'. The 'Protocol' is set to TCP. The 'Destination Port' is 80, and the 'Source Port' is set to '[>= 0]'. The 'Session Timeout' is set to 'Override'. A red box highlights the 'TCP Timeout (sec)' field, which is set to '[1 - 604800]'. A red arrow points from the 'TCP Timeout (sec)' field to the 'Destination Port' field. The 'TCP Half Closed (sec)' field is set to '[1 - 604800]' and the 'TCP Time Wait (sec)' field is set to '[1 - 600]'. The 'Tags' field is empty. The 'OK' and 'Cancel' buttons are at the bottom right.

Field	Value
Name	custom-tcp-80
Description	
Protocol	TCP
Destination Port	80
Source Port	[>= 0]
Session Timeout	Override
TCP Timeout (sec)	[1 - 604800]
TCP Half Closed (sec)	[1 - 604800]
TCP Time Wait (sec)	[1 - 600]
Tags	

Figure 1.13 – How to override TCP timeouts at the service level

The timeouts defined at the application level will override the global values, and the timeouts defined at the service level will override the ones defined for the application.

Now, let us examine how the TCP header complements the IP header reviewed in the previous section of this chapter and how they both map onto the firewall security policy.

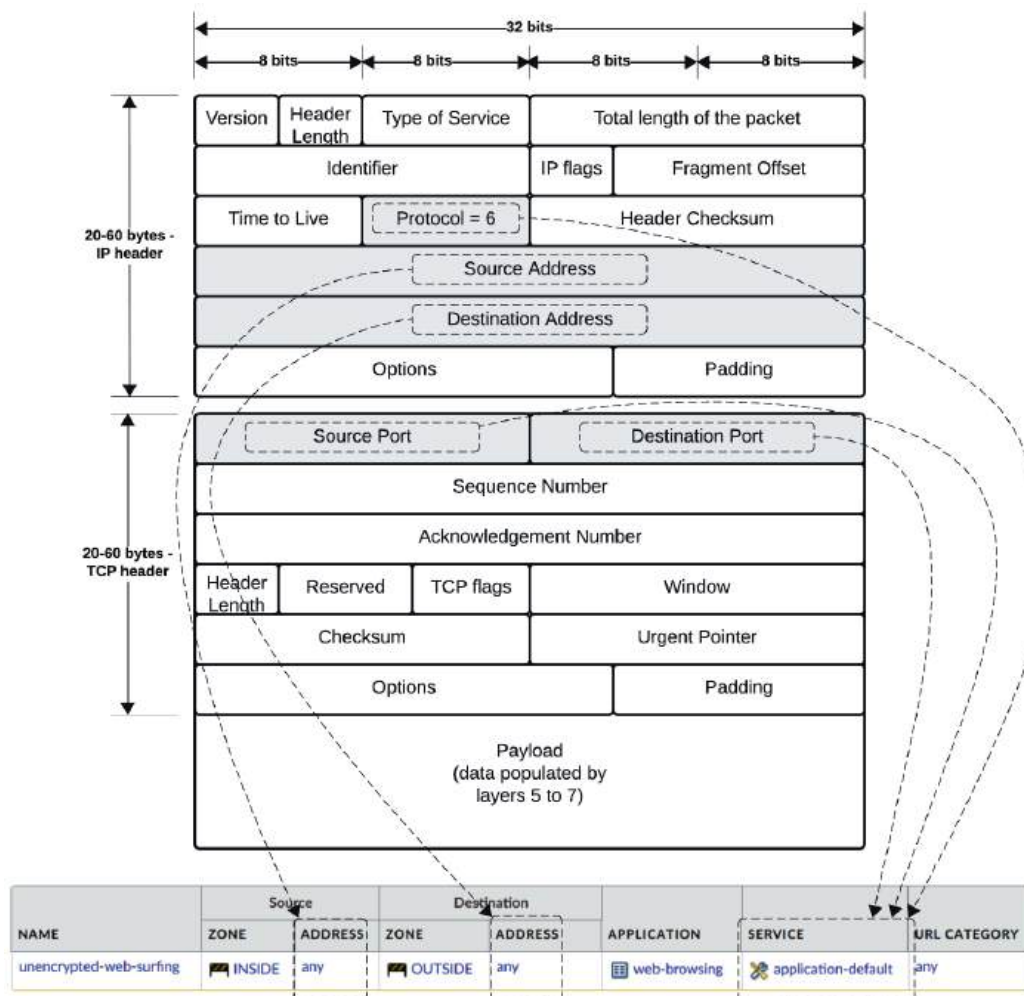


Figure 1.14 – IPv4 and TCP headers and the relation of their fields to the firewall security policy

As you can see, we have two new fields (*source port* and *destination port*) that are also hidden in the service or application definition.

UDP

UDP is a simpler transport layer protocol that does not need to set up a *communication channel* to transmit data. The first packet in a data transmission over UDP can carry actual application data. This protocol does not have any sophisticated features that its connection-oriented counterpart has. Its only transport layer features are the *multiplexing* and basic *data integrity*.

The latter functionality is simply a checksum calculated based on the packet payload. The sender calculates a value based on the payload and includes it in the packet header. The recipient performs the same operation on the received data and discards the packet if the checksum value does not match its own calculations. Unlike its connection-oriented counterpart, UDP will not care to retransmit any data damaged in transit and discarded by the receiving part – it will be the headache of the higher-level protocols and applications.

Let us study UDP packet structure to see how it differs from TCP from the firewall perspective.

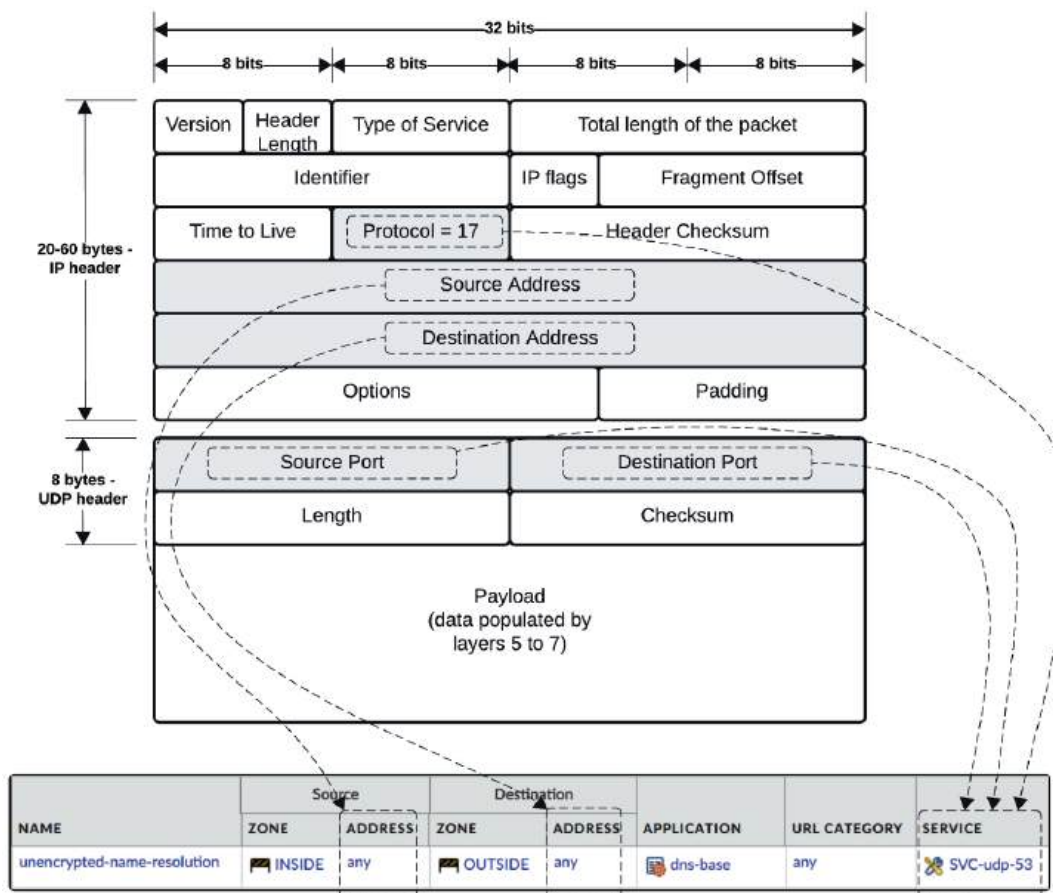


Figure 1.15 – IPv4 and UDP headers and the relation of their fields to the firewall security policy

As you can see from the header structure, the protocol is incredibly simple and straightforward, making it a good candidate for applications sensitive to network latency and those naturally resistant to packet loss. The prime examples of such applications are DNS and various audio-video applications accordingly.

Despite some challenges that handling UDP presents from the session management perspective (partially covered in the next section of this chapter), its relation to the security policy elements is identical to that of TCP.

OSI layers 5 to 7 – Application layer

Finally, we get to the most interesting part—the remaining host layers of the OSI model that handle actual application data. We are going to follow the model adopted by the **Defense Advanced Research Projects Agency (DARPA)**, the research branch of the U.S. Department of Defense, which grouped the functions pertaining to OSI layers 5 to 7 into a single Application layer.

As you can see in the previous sections, the IP header tells *exactly* what its payload is. This is done via the **Protocol** field, which contains the protocol number of the encapsulated protocol. However, the transport protocols are less friendly in this respect—they do not have an equivalent field that would tell the recipient of a packet what kind of application data the packet carries.

As we learned in the previous section, the destination port number essentially specifies an application on the receiving side for which the packet is intended. The application would know perfectly well what kind of data format it should expect in the packet payload. Therefore, it can process the data and act accordingly. A web server, for example, would expect that the payload of all packets with requests from web browsers that it receives on TCP port 80 would start with a known HTTP method name, followed by a URI, an HTTP protocol version, an **Fully Qualified Domain Name (FQDN)** of the website the request is intended for, and a number of optional protocol headers:

```
GET / HTTP/1.1
Host: example.com
```

Other applications would contain a totally different data structure. In many cases, unique patterns of bytes/symbols would clearly distinguish an application, but sometimes, there would not be anything like that—only a proprietary binary sequence that only the application itself is capable of understanding.

One may suggest that the port number is a good indicator of the type of application data encapsulated in a packet. Unfortunately, this is not the case in real life. The problem is that well-known port number allocation to applications and protocols (such as TCP 80 is HTTP, and TCP 443 is TLS) is only a guidance. Operating systems, software development kits, and programming languages virtually never dictate what port must be used for what application/protocol. Even the biggest and most respected software and hardware vendors often violate open standards because it is so tempting and convenient for them to use a destination port that is **open** (a synonym for being allowed by network security controls) in the vast majority of corporate and home networks.

Needless to say, malicious actors are even better at hiding in the woods of default well-known port numbers such as TCP/80, TCP/443, UDP/53, and many others.

The bottom line is that *absolutely nothing* in the IP or TCP/UDP header can reliably identify the type of application that communicates by means of a given packet, let alone data structures transferred by the application.

It is only by looking at the payload of the transport layer packet one *may* be able to tell what kind of application/high-level protocol is at play.

While the TCP/UDP packet payload is a meaningless chunk of data for a traditional firewall, it is an open book to a Palo Alto Networks next-gen firewall.

Now, we are armed with all the knowledge to complete the picture we started painting at the beginning of this chapter. The next figure takes HTTP as an example to demonstrate how all elements of a complete network packet that contains a web request map onto matching criteria of a security policy rule.

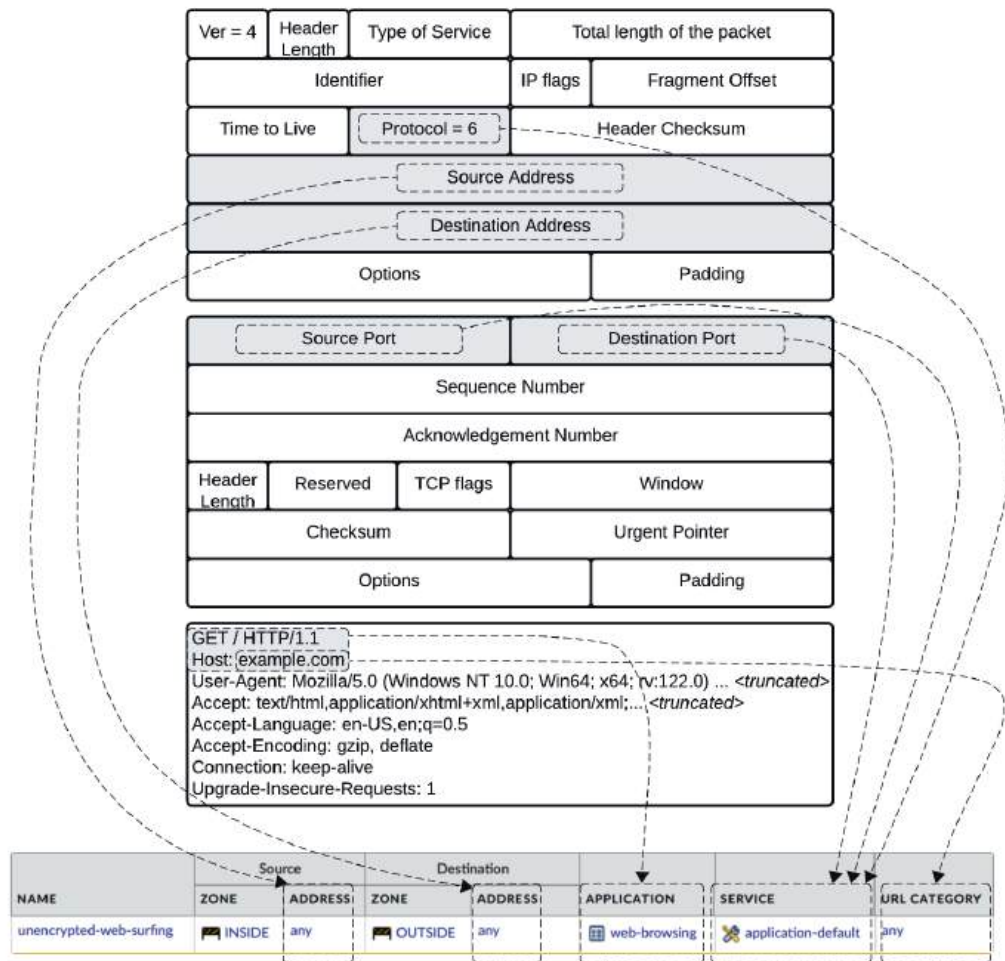


Figure 1.16 – A web request in the context of Network, Transport, and Application layers mapped onto fields of a security policy rule

The packet structure in its bottom part is simplified for readability. In fact, each symbol, including the “invisible” *Line Feed* and *Carriage Return*, would take a whole byte (thus, the word GET would occupy three consecutive bytes, and so on).

I have deliberately used HTTP because this protocol, along with its encrypted form, HTTPS, is the only one that can use the **URL Category** field as a matching criterion.

Before we proceed to study the inner mechanics of Palo Alto Networks’ proprietary App-ID technology, let us examine two other super important topics: *name resolution* and *load balancing* (in networks in general and on the internet in particular).

Name resolution

While an IP address in its numerical form is the only good format for computers and network devices, humans are not wired to easily memorize something like 93.184.216.34 (IPv4) or 2606:2800:220:1:248:1893:25c8:1946 (IPv6).

It is much easier for anyone to remember `www.example.com` than these faceless numbers.

The **Domain Name System (DNS)** is intended to solve this problem. It is a distributed hierarchical system that allows IP addresses to be represented in easy-to-remember symbolic names. Here is how it works at the internet scale.

DNS uses a hierarchical naming convention for symbolic names. Each **FQDN**, such as `www.example.com`, consists of these elements separated by dots from each other:

1. **Hostname:** `www`
2. **Subdomain(s):** None in our illustrative case
3. **Domain:** `example`
4. **Top-level domain (TLD):** `com`
5. **Invisible root domain:** Usually represented as an implicit dot

As you may guess, someone must manage this system.

Ultimately, the IANA, an **Internet Corporation for Assigned Names and Numbers (ICANN)** subsidiary, is at the top of the management pyramid. It is responsible for the root domain, which defines all TLDs and delegates their management to other organizations. There are more than 1,500 TLDs currently.

Country code TLDs (`.us`, `.fr`, `.dk`, etc.) are operated by designated organizations in the respective countries. For example, *Registry Services, LLC* manages the US TLD on behalf of the US Department of Commerce; the France TLD is managed by **AFNIC** (short for **Association Française pour le Nommage Internet en Coopération**), the Denmark TLD is managed by *Dansk Internet Forum* (also known as *Punktum dk*), and so on.

Other TLDs are operated by organizations ranging from the *US Department of Defense* (`.mil`) to big corporations such as *Microsoft* (`.microsoft`), the non-profit *Public Interest Registry* (`.org`), and the *City of Paris* (`.paris`).

In turn, these operators may delegate management of domains in their TLDs to other companies. For example, *VeriSign Global Registry Services*, which runs `.net`, delegates the management of `php.net` to the registrar called *Tucows Domains Inc.* headquartered in Toronto, Canada.

Each domain registrar, including IANA, can create name records (such as `www`) only in the domain they directly manage. For example, VeriSign itself *cannot* create a `www` record in the aforementioned `php.net` domain because it delegated the management of this domain to another company. On the other hand, VeriSign can revoke the delegation of the whole domain if the current domain owner violates VeriSign's terms and conditions (for example, if the annual registration fee is not paid on time or if the delegated domain is found to be a deliberate hosting for malware).

Zones and records

DNS is essentially a tree-like hierarchical distributed database containing records of different types.

The whole DNS namespace is broken down into so-called zones. A DNS zone is a distinct part of the namespace that has been delegated to an entity for full management. A zone can contain records for multiple domains and subdomains.

Each zone is served by a DNS server or a group of servers known as **authoritative** DNS servers. Authoritative means that they are the final authority – they do not hold cached and, thus, potentially inaccurate records – they keep the ultimate and most accurate records for a given zone.

Each record in a zone has a **Time To Live (TTL)** value assigned to it by the respective authoritative server(s), which allows resolvers and intermediate servers to cache it. This makes the system extremely scalable and efficient.

Here are some of the main types of DNS records that you must understand if you manage a fire-wall security policy:

- **A (address)**: Maps a name to an IPv4 address
- **AAAA (IPv6 address)**: Maps a name to an IPv6 address
- **CNAME (canonical name)**: Redirects one domain name to another domain name
- **NS (name server)**: Specifies authoritative DNS servers for the domain
- **PTR (pointer)**: Maps an IP address to a domain name, the reverse of an A or AAAA record

Name resolution algorithm

Each network device must be able to communicate with a DNS server to resolve names to IP addresses. Therefore, the address of a DNS server itself in the device's network settings must be specified in numeric form.

A system software component responsible for name resolution is called a **resolver**.

Here is how it typically works:

1. **Local cache check:** The resolver first checks whether the requested domain's IP address is already in the local cache. If it's there, the resolver uses this IP address, skipping further steps. You can see the content of the cache by running `ipconfig /displaydns` on Windows, and `sudo killall -INFO mDNSResponder` on macOS.
2. **System configuration files:** If the cache doesn't have the answer, the resolver checks the local system's configuration file (it is `/etc/hosts` on Unix/Linux, `/private/etc/hosts` on macOS, and `C:\Windows\System32\Drivers\etc\hosts` on Windows) for any statically defined hostname-to-IP mappings.
3. **Recursive query to DNS server:**
 - a. The resolver sends a query to the configured DNS server.

In a typical home network, this would be the address of your broadband router, which, in turn, forwards all requests to your ISP. In an enterprise environment, this would be an internal DNS server – a Windows domain controller, a Linux/Unix server running BIND, or a specialized appliance such as Infoblox. The server acts as a recursive resolver, meaning it takes responsibility for fully resolving the query.
 - b. The recursive resolver gathers the necessary IP address by consulting root name servers, TLD name servers, and, eventually, the authoritative name servers for the queried domain. Potentially, there can also be CNAME-based redirections to alternative domains, such as ones run by **Content Delivery Networks (CDNs)** – Akamai, Cloudflare, and so on.
4. **Caching the response:** Once the recursive resolver finds the IP address, it sends this information back to your local resolver, which caches the response for a certain time period defined by the TTL of the requested record. This makes future requests to the same domain much faster.
5. **Resolution complete:** The IP address is returned to the application that made the request (such as your web browser), allowing it to establish an IP-based connection to the target server.

While seemingly lengthy, this process takes milliseconds, thanks to the efficiency of caching and the distributed nature of the DNS infrastructure.

Load balancing

The performance of a single server is limited. Busy websites serve tens of thousands of complex requests per second, spiking to millions during their busiest periods. One cannot get by with a single server. There must be many, often geographically dispersed, to cope with the load and reduce latency between the website and its users.

Another reason to have multiple servers for a single application is redundancy. Servers require regular maintenance. They need to have security vulnerabilities patched, software updated, and hardware replaced. If the application's required availability is 100%, then there must be a mechanism to fulfill all these requirements.

Load balancing is a collection of technologies that make all the preceding possible. While some of these technologies are transparent from the firewall policy perspective, others are not.

When you create a security policy rule, you must choose matching criteria. Your criteria may include the destination IP address(es), either as a static address object or an FQDN object. This is where the problem is often hidden. Let me explain the issue using an example.

Look at the following figure:

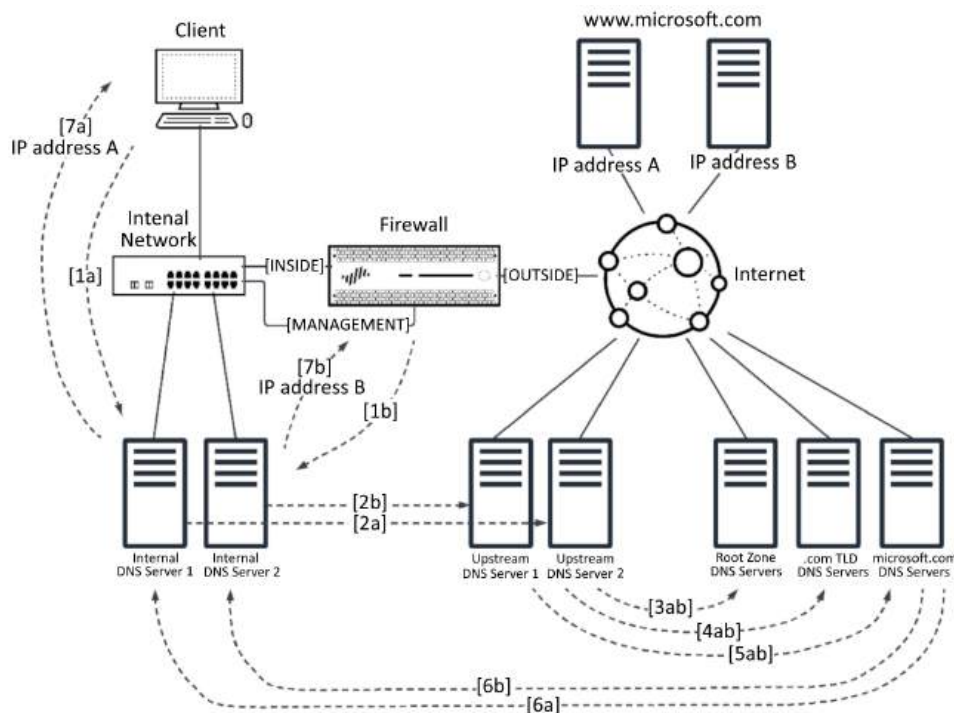


Figure 1.17 – Name resolution process

Imagine your company has two DNS servers configured to forward all client requests for public domains to two well-known public DNS servers: Google’s 8.8.8.8 and Cloudflare’s 1.1.1.1. These could have been your company’s ISP servers or other external DNS servers – the choice of these servers is not the point; the point is that you have two separate servers on the inside, which communicate to two *logical* servers on the outside. I am saying *logical* because you do not know what they are. These logical servers are often represented by hundreds (if not thousands) of individual physical or virtual servers, all hidden from you behind a couple of unicast addresses.

In the client’s network settings, you have specified the addresses of both internal servers. You have done the same in the firewall’s system settings (**Device | Setup | Services | Services | DNS Settings | Primary DNS Server & Secondary DNS Server**). This is completely in accordance with best practices to have two DNS servers configured on each client – for redundancy.

Now, imagine that you must create a firewall security policy rule allowing the client to browse `www.microsoft.com`. You go ahead and create your policy rule, as shown in the next figure.

Source		Destination		APPLICATION	URL CATEGORY	SERVICE	ACTION
ZONE	ADDRESS	ZONE	ADDRESS				
 INSIDE	any	 OUTSIDE	 FQDN-www.microsoft.com	 ssl  web-browsing	any	 application-default	 Allow

Figure 1.18 – FQDN-based matching (incorrect rule)

Is it going to work? Well, it may work, or maybe not. You cannot be sure about this rule. Let us examine why not.

Even if you are a firewall administrator and a client administrator, you cannot influence their resolvers’ DNS server selection algorithm. The client may choose to send its name resolution query (step [1a] in the diagram) to *internal DNS server 1*, while the firewall may use *internal DNS server 2* – step [1b]. These servers, in turn, may forward the requests to the same logical external server (which still could be different physical or virtual servers) or to different external servers ([2a] and [2b]). Imagine the requests go to different external servers (1.1.1.1 and 8.8.8.8 in our example). These servers would subsequently query the root DNS servers [3ab], .com TLD servers [4ab], and microsoft.com DNS servers [5ab]. The final responses, [6a] and [6b], will eventually reach the client and the firewall at steps [7a] and [7b]. These responses may contain the same or different IP addresses.

This is easy to simulate with a command-line tool such as `nslookup` or `dig`. Here is an example with the former one:

```
c:\> nslookup www.microsoft.com 1.1.1.1
```

```
Server:  one.one.one.one
```

```
Address:  1.1.1.1
```

```
Non-authoritative answer:
```

```
Name:     e13678.dscb.akamaiedge.net
```

```
Addresses: 2a02:26f0:5900:496::356e
```

```
           2a02:26f0:5900:48f::356e
```

```
           2a02:26f0:5900:490::356e
```

```
           2a02:26f0:5900:495::356e
```

```
           2a02:26f0:5900:494::356e
```

2.22.5.218

```
Aliases:  www.microsoft.com
```

```
          www.microsoft.com-c-3.edgekey.net
```

```
          www.microsoft.com-c-3.edgekey.net.globalredir.akadns.net
```

```
c:\> nslookup www.microsoft.com 8.8.8.8
```

```
Server:  dns.google
```

```
Address:  8.8.8.8
```

```
Non-authoritative answer:
```

```
Name:     e13678.dscb.akamaiedge.net
```

```
Addresses: 2a02:26f0:fd00:10bc::356e
```

```
           2a02:26f0:fd00:1091::356e
```

```
           2a02:26f0:fd00:10b9::356e
```

2.17.222.14

```
Aliases:  www.microsoft.com
```

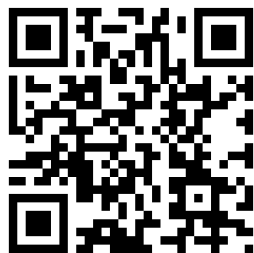
```
          www.microsoft.com-c-3.edgekey.net
```

```
          www.microsoft.com-c-3.edgekey.net.globalredir.akadns.net
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.



🔑 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



As you can see, we ran identical queries against two different public DNS servers and got completely different results – Cloudflare (1.1.1.1) told us that `www.microsoft.com` is 2.22.5.218 while Google (8.8.8.8) told us that `www.microsoft.com` is 2.17.222.14.

Thus, the client in our example may get one address, and the firewall may get a completely different address; therefore, our security policy rule will *not* match. However, in some cases, these addresses will match. Nevertheless, the result is that you have ended up with an inconsistent policy that *will* pay you back badly.

What if this were your business-critical application instead of the Microsoft home page? Someone at the first line of IT support will spend a lot of time figuring out why the app *always* works for one user but fails *every Tuesday* for her colleague (a pattern the users established and reported).

This may result in a case raised with the vendor's technical support and internal escalations; eventually, it will end up with you as a firewall engineer.

So, what should the policy look like to consistently match the desired traffic?

Since we cannot rely on the destination IP addresses because of the underlying load-balancing algorithms and complexity of the DNS infrastructure, in this case, we must rely on the available layer 7 information for HTTP(S), which is the URL.

Here is the correct rule:

Source		Destination		APPLICATION	URL CATEGORY	SERVICE	ACTION
ZONE	ADDRESS	ZONE	ADDRESS				
 INSIDE	any	 INET	any	 ssl  web-browsing	URL-LIST-microsoft.com	 application-default	 Allow

Figure 1.19 – URL-based matching (correct rule)

And here is the definition of the URL category referenced in this rule:

Custom URL Category

Name

URL-LIST-microsoft.com

Description

Type

URL List

Matches any of the following URLs, domains or host names

SITES

www.microsoft.com/

Figure 1.20 – URL-based matching (definition of the URL category)

In this rule, we allow *any* destination IP addresses because we have no reliable ways of identifying these addresses. At the same time, we are looking at the layer 7 information to ensure we allow only the URLs of interest.

Thus, we totally abstracted from all the complexities of Microsoft’s load-balancing mechanisms in this case and achieved a consistent firewall policy behavior.

In conclusion, when building a firewall policy rule, you must be mindful of potential implications caused by load-balancing algorithms an application may use. When considering a policy rule based on static IP addresses or FQDNs, make it a habit to research the application. Responsible application vendors often disclose this sort of information in one form or another, appreciating

that their customers may take network security controls seriously and pay close attention to how they provision their users' access.

Sadly, many other vendors could not care less about this subject. It is not rare to come across application network requirements suggesting something like *"We host our application on Amazon and cannot guarantee what our IP addresses might be. Just open TCP ports 443 and 1433 to all AWS public addresses, and our application will work"* or *"We cannot provide IP addresses of all our SFTP servers – just allow SFTP to *.ourdomain.com."* Seriously?

To me, this is always a good indicator of how immature the vendor is in cybersecurity in particular, and in application development in general.

How next-generation firewalls see network traffic

In the previous section, we learned networking basics in the context of firewalls. We became familiar with the Network, Transport, and Application layers of the OSI model and learned how they map onto matching criteria of a next-generation firewall security policy rule. We briefly discussed DNS and the potential implications of load balancing.

It is time to discuss how our next-generation firewalls work *under the hood* to identify applications and protocols.

How do they tell facebook-base from websocket, ldap from ftp, and web-browsing from ssl? And how does one make sense of thousands of these *apps*? How do they all *work* together in a security policy? How do we police this lot? Do we even have to?

Palo Alto Networks' proprietary technology for distinguishing applications and network protocols is called **App-ID** (short for **Application Identification**). It is so deeply integrated into PAN-OS that you cannot imagine one without the other. This section is essential for understanding how a firewall security policy should be built. I will deliberately simplify some of the steps and points where possible so that you can grasp the essence of the main ideas.



My more experienced colleagues may be tempted to skip this section altogether, but I would urge them not to. This will allow us to stay on the same page (not literally, I hope) in the chapters to follow.

Stateful firewalls

Next-generation firewall is a stateful firewall, which means that each network connection is considered by the firewall as a whole – in its entirety, so to speak. The firewall tracks the current state of a connection at each stage of its existence. Depending on the type of data the connection transmits, the firewall would engage application- and protocol-specific *parsers* or *decoders*. They may not necessarily be called such, but this is what these pieces of PAN-OS code essentially do – they parse network packets, dissecting them at different layers of the OSI model. They peel layer after layer from the onion bulb of a network packet and engage logical tracking mechanisms depending on what each next layer reveals.

For example, your regular web-browsing session to a modern website would require the firewall to track the following at a very minimum:

- The state of the NAT (as the packets of your network connection leave the private IP address space of your network and traverse the routable public IP address space of the internet)
- The state of the corresponding Transport layer connection (TCP, usually)
- The state of the corresponding Presentation layer connection (TLS encryption, for instance)
- The state of the Application layer connection (browser's HTTP requests and corresponding server responses), provided that the firewall decrypts the TLS connection

Multi-connection protocols, such as **File Transfer Protocol (FTP)**, require special parsers – so-called **Application Layer Gateways (ALGs)**. ALGs make coordinated handling of interlinked connections possible. In the case of FTP, the corresponding ALG would allow subsequent data connections based on the FTP client's commands and server responses *seen* within the FTP control connection. Such intelligence on the firewall side allows us to simplify security policy to very straightforward statements such as *Allow FTP from A to B over the default port* and not be bothered by the hidden complexity of the actual file transfer.

Example network connection

Now, let's examine what is going on under the hood. It will be easier if we consider a very simple example.

Imagine that we have a computer connected directly to a firewall, all network settings are in order, and everything works – we can browse the web. Our firewall has an *inside* interface (the one that our computer is connected to) and an *outside* interface (the one that is connected to the internet). Both interfaces are layer 3 and have valid IP addresses assigned (a private one and a public one, accordingly).

Here is a schematic representation of our example network:

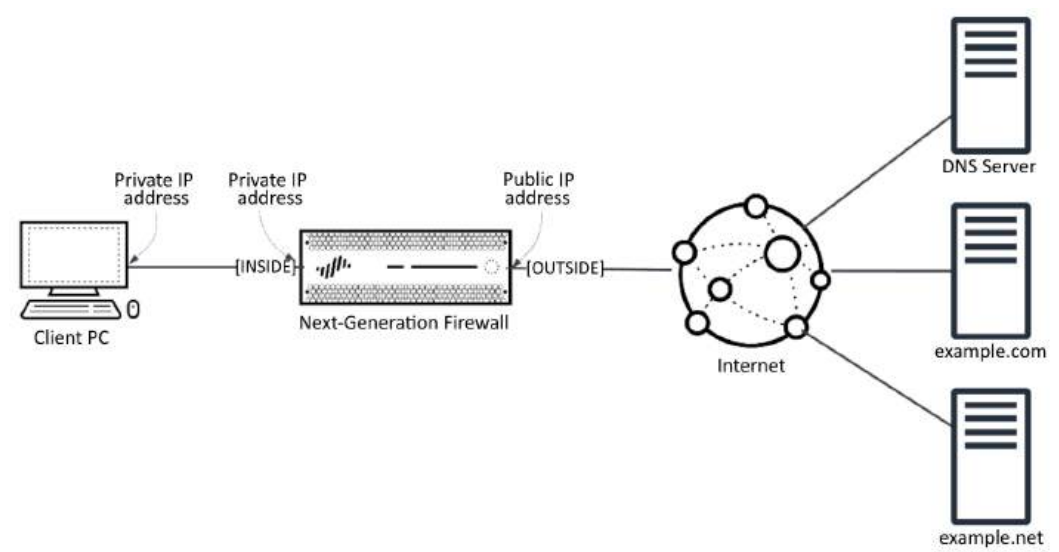


Figure 1.21 – Network topology for App-ID illustration

As you can see in the diagram, we have a firewall with two interfaces. One of them is connected to the internal network, where we also have a client PC, and the other interface is connected to the internet.

The firewall has a basic source port and address translation policy rule, as shown in the following screenshot, so traffic originating from our private network was routable on the public internet.

	NAME	Original Packet						Translated Packet	
		SOURCE ZONE	DESTINATION ZONE	DESTINATION INTERFACE	SOURCE ADDRESS	DESTINATION ADDRESS	SERVICE	SOURCE TRANSLATION	DESTINATION TRANSLATION
1	default-nat	INSIDE	OUTSIDE	any	any	any	any	dynamic-ip-and-port ethernet1/1	none

Figure 1.22 – Example NAT policy

We also have a security policy that basically says “Allow DNS protocol to all external addresses and web-browsing to example.com and example.net.” It is shown in the following screenshot:

	NAME	Source		Destination		SERVICE	APPLICATION	URL CATEGORY	ACTION
		ZONE	ADDRESS	ZONE	ADDRESS				
1	name-resolution	INSIDE	any	OUTSIDE	any	application-default	dns-base	any	Allow
2	access-to-example.com	INSIDE	any	OUTSIDE	any	application-default	ssl web-browsing	URL-LIST-example.com	Allow
3	access-to-example.net	INSIDE	any	OUTSIDE	any	application-default	ssl web-browsing	URL-LIST-example.net	Allow

Figure 1.23 – Example security policy to illustrate the App-ID algorithm

We open a browser and navigate to `http://example.net`. In fact, the browser is not the ideal client software in our use case because it won’t allow us to pause after the TCP handshake, which would be required for a deeper understanding of the policy lookup algorithm. Let us use the interactive mode of the netcat Linux utility instead of a web browser:

```
$ netcat -v example.net 80
```

The very first thing our browser will do is resolve the FQDN `example.net` to an IP address. For simplicity, let us omit the name resolution process and focus on what happens next.

Packet flow inside the firewall – App-ID

When a packet reaches the firewall ingress interface, the firewall needs to go through a few important steps first. These steps start with integrity checks and the identification of the corresponding ingress sub-interface, if there are any (a VLAN tag in the incoming Ethernet frame would be the key differentiator here). Our example is very straightforward, as we have only one internal interface.

NGFW is a zone-based firewall where each interface belongs to a zone (defined by a firewall administrator). Zones are essential elements of firewall policies. Once we know the ingress interface, we know the associated zone name.

The zone identification is followed by further checks and extraction of layer 3 and layer 4 address information (IP addresses, protocol, and port numbers). And then, a crucial thing happens – the firewall establishes the six attributes of the incoming packet, which are known as the **six-tuple**:

- Source IP address
- Source port number
- Source zone
- Destination IP address

- Destination port number
- IP protocol number

When combined, these attributes identify a unique **flow** the packet effectively represents. These attributes altogether are called a **flow key**. The firewall uses this flow key to search for a matching value in the firewall session table.

When you look at the selection of these attributes, you may guess that each session must consist of two flows to account for packets traveling in both directions of a network connection. You are right – each session combines two unidirectional flows: **client-to-server (c2s)** and **server-to-client (s2c)**. In firewall terminology, the **client** is the party initiating the connection, and the **server** is the responding party. These terms have *nothing* to do with *client* and *server* in traditional computer terminology, where they usually indicate who owns a resource (server) and who accesses that resource (client) or even the type of computer operation system (for example, Windows Server).

As you can see, the preceding attributes are pretty distinctive but not too much. Changing a single digit in a port number or address, for example, or swapping TCP for UDP will result in a completely new flow key and, thus, a different match (or no match!) in the session table. Conversely, you can have several interfaces in the same zone. The firewall will match *all* incoming packets of a network connection *regardless of the actual ingress interface* to the same flow (and, ultimately, to the same session), provided that all these interfaces belong to the same zone and that the other five attributes match. This allows for basic load balancing and thus increases performance in certain scenarios.

As you remember, we are connecting to the website `http://example.net`. The very first packet of our connection will be a TCP packet with a SYN flag set (the beginning of a TCP handshake). The firewall will establish a flow key for this packet and look it up in the session table. It will not find a suitable match because it is the first packet in our c2s flow. At this point, the firewall needs to do the following:

1. Perform a forwarding look-up (identification of the destination/egress interface – where to send the packet)
2. Create a new session (for further inspection of this packet and other packets expected to follow as part of the same network connection)

The firewall will consult a **Forwarding Information Base (FIB)** table derived from a routing table that, in its turn, is *either* a static set of network routes (configured by a firewall administrator or via DHCP) *or* a result of operation of dynamic routing protocols such as **Border Gateway Protocol (BGP)** or **Open Shortest Path First (OSPF)** *or* a combination of both approaches. The firewall

will use the FIB table to identify the egress interface to which the packet needs to be forwarded. Once the firewall identifies the egress interface, it knows what the destination zone is going to be.

With all that information, the firewall will allocate a session ID from a pool of free session numbers. It will then create a data structure linked with the session ID. The data structure will consist of two flow keys (c2s and s2c) and other metadata describing the current state of the associated network connection. All this data will essentially constitute the session and be used by protocol/application parsers and other system processes as required.

As soon as the new session ID is allocated, the firewall will perform the first security policy look-up for our network connection to determine whether it needs to be allowed or denied.

The firewall now knows the source and destination zones, source and destination IP addresses, protocol number, and destination port. It will use this to look for a matching security rule. You can forget about Application layer inspection at this stage. The firewall will start evaluating all rules in the security policy from the top to the bottom, effectively ignoring extra attributes such as URL category and application. Only OSI layer 3 (IP) and layer 4 (Transport) information is known and considered at this point. The firewall will stop parsing the policy rules when it finds the first match.

The following figure shows the policy rule fields related to layer 3:

Source		Destination	
ZONE	ADDRESS	ZONE	ADDRESS
 INSIDE	any	 OUTSIDE	any

Figure 1.24 – Layer 3 information (IP addresses and zones)

And here is the layer 4-related field:

SERVICE
 application-default

Figure 1.25 – Layer 4 information (IP protocol type and port number)

The **application-default** keyword in the **SERVICE** field of our policy rule is what you see on the surface. The firewall precisely knows the port number and IP protocol hidden behind this.

At the time of policy compilation (when you **commit** configuration changes to the firewall), the firewall would look up all metadata of the application(s) specified in each rule in the application signature database and find the IP protocol name (TCP, UDP, ICMP, etc.), port numbers and timeout value(s) specific to the application(s), and store the numbers in the equivalent rule in the resultant compiled policy on the data plane of the firewall. As a firewall administrator, you do not see that effective policy – you must learn to see port numbers and timeouts through the **Application** and **Service** fields of your policy rules.

You can find this metadata in the application browser when you navigate to **Objects | Applications** in the firewall UI.

Application

Name: web-browsing

Standard Ports: tcp/80

Secure Ports: tcp/443

Depends on:

Implicitly Uses:

Deny Action: drop-reset

Additional Information: [Wikipedia](#) [Google](#) [Yahoo!](#)

Description: Web Browsing is using Hypertext Transfer Protocol (HTTP), which is a method used to transfer or convey information on the World Wide Web. Its original purpose was to provide a way to publish and retrieve HTML pages.

Characteristics

Evasive:	no	Tunnels Other Applications:	yes
Excessive Bandwidth Use:	no	Prone to Misuse:	no
Used by Malware:	yes	Widely Used:	yes
Capable of File Transfer:	yes		
Has Known Vulnerabilities:	yes		

Options

TCP Timeout (seconds):	3600	Customize...
TCP Half Closed (seconds):	120	Customize...
TCP Time Wait (seconds):	15	Customize...
App-ID Enabled:	yes	

Classification

Category: general-internet

Subcategory: internet-utility

Risk: 4 [Customize...](#)

Tags

Web App [Edit](#)

[Close](#)

Figure 1.26 – IP protocol and port number in the application definition

As soon as the very first match is found, the firewall will apply the action defined in the matched policy rule.


```

timeout                : 5 sec
time to live           : 2 sec
total byte count(c2s)  : 74
total byte count(s2c)  : 0
layer7 packet count(c2s) : 1
layer7 packet count(s2c) : 0
vsys                   : vsys1
application             : undecided
rule                   : access-to-example.com
service timeout override(index) : False
application db          : 0
app.id                 : c2s node(0,0) s2s node(0,0)
session to be logged at end : True
session in session ager : True
session updated by HA peer : False
address/port translation : source + destination
nat-rule               : default-nat-for-lab1-1(vsys1)
software fast forwarding : False
layer7 processing       : enabled
URL filtering enabled   : True
URL category            : any
session via syn-cookies : False
session terminated on host : False
session traverses tunnel : False
session terminate tunnel : False
captive portal session  : False
ingress interface       : vlan.1
egress interface        : ethernet1/1
session QoS rule        : N/A (class 4)
end-reason              : unknown

```

We can see here that this is an active (state = ACTIVE) TCP (proto = 6) session from the address 192.168.51.10 and port 45888 to the address 192.168.50.164 and port 80. Only one packet was seen from the client to the server (layer7 packet count (c2s) = 1), and that packet was 74 bytes in size. Zero packets have been received from the destination so far. The session currently matches the access-to-example.com rule. This occurs because the firewall searches for a matching policy from top to bottom, and the first match, based on the information available at this stage, is the access-to-example.com rule.

Pay attention to the destination port in the s2c flow. It is 55957 instead of the 45888 one might have expected to see. This is because a DIPP NAT rule was applied – it translated the source port number.

The session timeout is set to 5 seconds, of which only 2 seconds are left. If you refer to *Figure 1.11*, you will see that this corresponds to the **TCP init** timeout, that is, how long the firewall waits for the SYN-ACK packet to arrive from the server in response to the client’s SYN packet.

The firewall has yet to decide what application this connection represents (application = undecided), which is logical because it has not seen any Application layer data. If there is no response received from the server, the session will be terminated by the firewalls with a traffic log entry (**Monitor | Logs | Traffic**) as follows:

IP PROTOCOL	PACKETS SENT	BYTES RECEIVED	APPLICATION	ACTION	RULE	SESSION END REASON
tcp	1	0	incomplete	allow	access-to-example.com	aged-out

Figure 1.27 – Traffic log entry for incomplete connection

The **incomplete** application means that either the TCP handshake was not completed or the handshake was completed but there was no data exchange afterward. This is typical for routing issues and issues on the server side (for example, the vendor accepts connections only from known public addresses, and yours is not on their **allow** list).

OK, let us continue our connection to `example.net`.

After the TCP handshake completes, Netcat will inform us that it is ready to transmit application data:

```
$ netcat -v example.net 80
Connection to example.net (93.184.216.34) 80 port [tcp/http] succeeded!
```

If we check the session details now, we will see that the situation has progressed:

```
timeout                : 3600 sec
time to live           : 3541 sec
total byte count(c2s)  : 140
total byte count(s2c)  : 74
layer7 packet count(c2s) : 2
layer7 packet count(s2c) : 1
vsys                   : vsys1
```

```

application          : undecided
rule                 : access-to-example.com

```

You see that the TCP handshake has successfully completed — three packets were seen (SYN, SYN-ACK, and ACK), and the firewall updated the session timeout. It is happy to wait up to 1 hour (3600 sec) for any application data to arrive across this connection (either from the client or from the server). You also can see that the firewall has not received any additional information indicating what this application might be; therefore, the application name is still undecided, and the session is associated with the same security policy rule.

However, if we do nothing on the client side for another few seconds, the server loses its patience and sends us a FIN packet, thus indicating that it does not want to wait any longer and would like to tear down the connection gracefully (to preserve its own resources). If this were to happen, the firewall would immediately update the session's timeout, as the *TCP Half Closed* state cannot last longer than 120 seconds (as per the global settings you have already seen):

```

timeout              : 120 sec
time to live         : 23 sec
total byte count(c2s) : 206
total byte count(s2c) : 140
layer7 packet count(c2s) : 3
layer7 packet count(s2c) : 2
vsys                 : vsys1
application          : undecided
rule                 : access-to-example.com

```

But we are not going to waste a good TCP connection, are we? Let us send some data. As you learned in the *OSI layers 5 to 7—Application layer* section, the minimum required data in the HTTP 1.1 protocol includes the *method*, *URI*, *protocol version*, and *host* header. This is what we are going to provide as soon as our TCP connection is established:

```

$ netcat -v example.net 80

Connection to example.net (93.184.216.34) 80 port [tcp/http] succeeded!
GET / HTTP/1.1
Host: example.net

```

If you are reproducing these commands in your environment, ensure you press *Enter* twice after the last line.

The server immediately spits out its default page as the client requested:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 291776
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Sat, 16 Mar 2024 18:37:48 GMT
Etag: "3147526947"
Expires: Sat, 23 Mar 2024 18:37:48 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (dce/268A)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1256
<!doctype html>
<html>

    <-- part of the output omitted for brevity -->
<body>
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents. You may
use this domain in literature without prior coordination or asking for
permission.</p>
<p><a href="https://www.iana.org/domains/example">More information...</
a></p>
</div>
</body>
</html>
```

So, what will the firewall do when the client starts submitting data? It will take the sequence of bytes corresponding to the string `GET / HTTP/1.1 Host: example.net` and look it up in the application signature database to see whether anything matches this pattern.

It will quickly discover that this is HTTP, which is web-browsing in App-ID terminology. The firewall will immediately engage the corresponding protocol decoder.

To see what decoder each application uses, switch to the configuration mode in the firewall CLI and run the following command:

```
show predefined application <app name> | match decode
```

Here are some examples that we wondered about at the very beginning of this section:

```
> configure
Entering configuration mode
[edit]
# show predefined application web-browsing | match decode
decode http;
# show predefined application websocket | match decode
decode generic;
applicable-decoders http;
# show predefined application ftp | match decode
decode ftp;
run-decoder yes;
applicable-decoders ftp;
# show predefined application ldap | match decode
decode ldap;
run-decoder no;
# show predefined application ssl | match decode
decode ssl;
run-decoder no;
# show predefined application facebook-base | match decode
applicable-decoders http;
```

In our example, the http parser will be used to extract the URL from the HTTP Host header and look the URL up in the URL filtering database to classify it. Our URL (example.net) will match three categories:

- computer-and-internet-info
- low-risk
- The custom one we created – URL-LIST-example.net

At this stage, the firewall becomes aware of layer 7 (Application) details, including the application name (web-browsing) and URL category (URL-LIST-example.net, computer-and-internet-info, and low-risk) in addition to previously known layer 3 (Network) and layer 4 (Transport) information.

Thus, our effective security policy visible to the policy matching algorithm becomes as follows:

Name	Zones and addresses	Port	App	URL cat	Action
name-resolution	omitted for brevity	udp/53, udp/5353, tcp/53	dns-base		Allow
access-to-example.com	omitted for brevity	tcp/443, tcp/80	ssl,web-browsing	URL-LIST-example.com	Allow
access-to-example.net	omitted for brevity	tcp/443, tcp/80	ssl,web-browsing	URL-LIST-example.net	Allow

Table 1.5 – L3/L4/L7 view of our example policy

Since the firewall now has this extra information, it attempts to find the best match for our session for the second time. Now, the rule the session matched previously (access-to-example.com) does not match anymore because that rule has a different URL category (URL-LIST-example.com). However, the next rule (access-to-example.net) does match. The firewall will now apply the action specified in this new rule to the session. In our case, the action is still **Allow**, so the firewall will keep this session in the active state.

This is how the session will look after this policy rematch:

```

timeout                : 3600 sec
time to live           : 3583 sec
total byte count(c2s)  : 504
total byte count(s2c)  : 2011
layer7 packet count(c2s) : 7
layer7 packet count(s2c) : 6
vsys                   : vsys1
application             : web-browsing
rule                    : access-to-example.net
service timeout override(index) : False
session to be logged at end : True
session in session ager : True
session updated by HA peer : False
address/port translation : source + destination

```

```
nat-rule                               : default-nat-for-lab1-1(vsys1)
software fast forwarding                : False
layer7 processing                       : enabled
ctd version                            : 10
URL filtering enabled                   : True
URL category                           : URL-LIST-example.net,
                                       computer-and-internet-info,
                                       low-risk
```

This session will remain active as long as the peers keep exchanging data and do not terminate the connection. If they stop sending data without tearing down their underlying TCP connection, the firewall will keep the session active for one hour and terminate it, recording **aged-out** as the *session end reason* in the traffic log.

The firewall keeps inspecting every single HTTP packet, looking for potential changes in the application pattern. I am sure you now understand the idea, but let us take our example one more step further to see how a session might be terminated owing to this constant reassessment of the application data traversing the TCP connection.

Let us add one more rule at the end of our security policy:

	NAME	Source		Destination		SERVICE	APPLICATION	URL CATEGORY	ACTION
		ZONE	ADDRESS	ZONE	ADDRESS				
1	name-resolution	 INSIDE	any	 OUTSIDE	any	 application-default	 dns-base	any	 Allow
2	access-to-example.com	 INSIDE	any	 OUTSIDE	any	 application-default	 ssl  web-browsing	URL-LIST-example.com	 Allow
3	access-to-example.net	 INSIDE	any	 OUTSIDE	any	 application-default	 ssl  web-browsing	URL-LIST-example.net	 Allow
4	default-block	 INSIDE	any	 OUTSIDE	any	any	any	any	 Deny

Figure 1.28 – Extra security rule to illustrate application shift

Now, we repeat all the previous steps with Netcat to get to the point where we get the home page from the web server:

```
<-- part of the output omitted for brevity -->
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents. You may
use this domain in literature without prior coordination or asking for
permission.</p>
<p><a href="https://www.iana.org/domains/example">More information...</
```

```
a></p>
</div>
</body>
</html>
```

Our connection to `example.net` remains active, and we can attempt to switch it to the `websocket` protocol, which is done by sending a specially crafted request:

```
GET / HTTP/1.1
Host: example.net
Connection: Upgrade
Upgrade: websocket

HTTP/1.1 503 Service Unavailable
Content-Type: text/html; charset=UTF-8
Content-Length: 9163

<-- HTML code of the firewall block page follows here -->
```

So, what happens here is the following:

1. The firewall's HTTP decoder keeps analyzing all subsequent client requests within the session.
2. It detects the attributes of a new protocol (connection upgrade request from the client).
3. It looks this pattern up in the application signature database and identifies this as a new application – `websocket`.
4. Since this is new Application layer information, the firewall's policy matching engine performs a new security policy look-up to find a matching rule. The **default-block** rule will now be the only matching rule in the policy because it has **SERVICE**, **APPLICATION**, and **URL CATEGORY** attributes set to **Any**.
5. The firewall sends its **Application Block** response page to the client (by substituting the server's response with the firewall's own HTML code of the response page).
6. The firewall applies the **Deny** action specified in the matched policy rule.

The **Deny** action is a generic representation of a blocking action. The exact method of blocking is defined in the application's signature of the blocked application:

```
# show predefined application websocket | match deny
deny-action drop-reset;
```


According to Palo Alto Networks documentation, if this action is taken, the session's packets will be discarded, and a TCP RST packet will be sent to let the client know that the session has been terminated. This way, the client can gracefully close the session locally.

Here is what our session will now look like:

```
total byte count(c2s)           : 1236
total byte count(s2c)           : 2126
layer7 packet count(c2s)        : 17
layer7 packet count(s2c)        : 8
vsys                             : vsys1
application                      : websocket
rule                             : default-block
service timeout override(index) : False
session to be logged at end      : True
session in session ager         : True
session updated by HA peer      : False
address/port translation         : source + destination
nat-rule                        : default-nat-for-lab1-1(vsys1)
software fast forwarding         : False
layer7 processing               : enabled
ctd version                     : 10
URL filtering enabled            : True
URL category                    : URL-LIST-example.net,
                                computer-and-internet-info,
                                low-risk

session via syn-cookies         : False
session terminated on host      : False
session traverses tunnel        : False
session terminate tunnel        : False
captive portal session          : False
ingress interface               : vlan.1
egress interface                : ethernet1/1
session QoS rule                 : N/A (class 4)
end-reason                     : policy-deny
handled by FIN proxy            : s2c, 0 packets
```

This process of changing an application during the same session is called **application shift**.

It is worth noting that when the session ends, the firewall records only the last application detected within the session in the traffic log:

SESSION ID	APPLICATION	ACTION	RULE	SESSION END REASON
31446	websocket	reset-both	default-block	policy-deny

Figure 1.29 – The traffic log only records the last application that was detected in a session

At the same time, the URL log will show each individual request made within this session along with the application name active at the time of the request:

SESSION ID	APPLICATION	URL	RULE	ACTION
31446	web-browsing	example.net/	access-to-example.net	alert
31446	web-browsing	example.net/	access-to-example.net	alert

Figure 1.30 – The URL log shows every request made within the session

As you remember, we made two requests within the session—an initial request for the website’s front page and a subsequent request to switch our connection to websocket. At the time of each request, the connection was identified as regular web browsing; therefore, this is what is reflected in these log records.

The firewall may need to allow up to 4 packets or 2,000 bytes worth of data in either direction (not including the TCP handshake) in an attempt to identify the application. If no signatures match and the traffic is based on the HTTP protocol, it is declared web-browsing. If it is based on SSL/TLS, then the application will be named as ssl. In all other cases, it will be declared unknown-tcp or unknown-udp, depending on the transport protocol. When the nature of traffic cannot be determined by signatures and protocol analysis alone, the firewall may use heuristic and behavioral analysis based on packet length, source, and session rate.

Packet flow inside the firewall – content identification and scanning

After the App-ID engine finishes identifying the application that a session’s traffic represents, the firewall engages the relevant protocol decoder. The decoder, in turn, being aware of the application’s encoding schema and types and formats of the application’s data structures, knows how to extract meaningful application data from the stream of bytes associated with the session.

Decoders differ by their capabilities. Some can identify files and some cannot. One can detect viruses and another cannot. Different factors can cause this variability. The nature of the underlying protocol plays a major role in these firewall capabilities.

For example, if you look at the capabilities of the smb decoder responsible for handling ms-ds-smb-base (Samba/Windows File Sharing), you will see that it can do pretty much everything that the Content-ID engine is capable of in principle:

```
LAB-440-1# show predefined application ms-ds-smb-base | match -ident  
  
virus-ident yes;  
file-type-ident yes;  
vulnerability-ident yes;  
data-ident yes;
```

SMB is an old, well-known protocol with clearly defined data structures suitable for streamed processing. Therefore, it was possible to create a parser that extracts files, identifies their types, and hands them over to the vulnerability and data filtering identification engines, antivirus, and WildFire analysis.

On the contrary, websocket is the exact opposite. It is essentially a proprietary stream of binary data implemented differently by different application developers. There is nothing the firewall can possibly do about this stream other than match it against a handful of vulnerability signatures:

```
LAB-440-1# show predefined application websocket | match -ident  
  
vulnerability-ident yes;
```

Thus, in many cases, the capabilities of Content-ID reflect the underlying complexity of protocols and applications it must deal with. This also proves the importance of the *Defense in Depth* principle in cybersecurity and the importance of risk-based approaches when you create a firewall security policy.

Summary

In this chapter, you explored what sets **Next-Generation Firewalls** apart from traditional network security solutions. You reviewed the OSI model through the lens of firewall policy design and observed how NGFWs, unlike their predecessors, inspect traffic all the way up to the Application layer. This deeper visibility empowers you to create security policies that consider not only IPs and ports but also application identities and behaviors, providing you with much greater control over what's allowed in your network.

You learned how NGFWs use stateful inspection, tracking each session's life cycle using a six-tuple flow key made up of source and destination IPs, ports, zones, and protocol. Through a hands-on example involving Netcat and HTTP traffic, you followed how a session is created, matched to a policy, re-evaluated when application details become available, and even denied mid-stream if the application shifts unexpectedly. These mechanics showed you how dynamic and adaptive NGFWs are compared to legacy port-based firewalls.

The chapter also helped you understand the importance of DNS resolution and the potential pitfalls of relying on static IPs or FQDNs in firewall policies. You saw why application- and URL-based rules are often more reliable, especially in the face of load balancing and CDN behavior. Finally, you explored how NGFWs use decoders to analyze traffic content, and how different protocols offer varying levels of inspectability depending on their structure and complexity.

By the end, you gained a solid grasp of how NGFWs see and evaluate traffic across the OSI stack, how applications are identified in real time, and why these capabilities matter for building secure, effective policies.

In the next chapter, you'll look at the real-world challenges of managing firewalls and responding to cybersecurity threats. You'll be introduced to a fictitious company that helps define business-aligned requirements for a firewall security policy—bringing together everything you've learned so far in a practical, risk-aware context.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



2

Navigating Real-World Firewall Management and Cyber Risks

In the previous chapter, you learned how next-generation firewalls inspect, classify, and control network traffic across multiple OSI layers using features such as App-ID and Content-ID. That knowledge gives you a solid technical foundation, but building and managing firewall policies in the real world is a different kind of challenge. It's messy, unpredictable, and influenced as much by human factors and business dynamics as by technical capabilities.

In this chapter, you'll take a critical look at the many obstacles that make firewall management difficult in practice—from rushed implementations and inconsistent migrations to vague business requirements, unreliable vendor support, and internal politics. You'll also assess what firewalls can and cannot realistically protect you from, given today's cybersecurity risks.

Finally, you'll begin to bridge the gap between theory and practice by defining both functional and non-functional requirements for a modern, risk-aware firewall security policy, using a fictitious company as a real-world scenario to anchor the discussion.

Here's what we'll cover:

- Challenges of managing firewalls in real life
- Cybersecurity risks
- Functional and non-functional requirements

Challenges of managing firewalls in real life

What makes managing firewalls difficult? There are many factors that contribute to this. I will give you a few reasons, but you can continue the list:



The challenges described here are common across many technology and cybersecurity vendors; they are not tied to any single product or brand.

- **Technical complexity:** Modern firewalls are anything but simple. Unfortunately, they cannot be simple. Their complexity reflects the complexity of the enterprise environments they operate in and the evolving cybersecurity landscape.

Next-generation firewalls have hundreds of features and configuration options. Learning how these features *can* be configured takes time, and it takes even longer to learn how these features *should* be configured. Engineers and their managers often do not fully comprehend and appreciate the complexity of the devices they have acquired.

This complexity requires careful consideration throughout the device life cycle, including initial design, deployment, migration, integrations, feature implementations, upgrades, day-to-day administration, and decommission.

- **Lack of experience and quality training:** Speaking of technical complexity, how often are firewall engineers sent to vendor training courses? This depends on the department's budget, team size, and management's appreciation of the value such training provides.

A required training course may not be immediately available and, therefore, may not fit well into a firewall project timeline. Even if the engineer makes it to the training course, they may discover that the course curriculum is not meeting their expectations. The curriculum may prove to be insufficiently deep, misaligned with the engineer's prior knowledge and experience, or imbalanced - lacking necessary hands-on lab exercises or essential theoretical background. Finally, the trainer may appear to be a pure academic with a lack of practical experience with the product.

The bottom line here is that the personal hands-on experience of the engineer and their ability to ingest a variety of reference documentation, coupled with a good background knowledge of networking and application ecosystem, play a vital role.

- **Sketchy and inconsistent initial implementations:** Firewalls are often implemented hastily with a limited set of criteria for measuring success. The cables are connected, the admin management interface is set up, the routing is configured, and a NAT rule is in place. To finish the task, a security policy with a handful of *any-any*-style rules and a default set of security profiles is created. Once this is done, the firewall is marked as complete on the project plan, and the team moves on.

The situation worsens when a company operates multiple locations, each implementing its firewall differently, without a uniform standard. At the same time, even a low-quality one would have made subsequent improvements much easier.

- **Incomplete migrations:** Similar stories happen with firewall migrations. Many IT teams eventually realize that the firewall their company purchased a decade ago is no longer fit for purpose. They begin researching the market, comparing products, and eventually decide to buy a next-generation firewall.

However, replacing an old firewall with a new one without disrupting the production environment is not a simple task. Success in such a change is usually measured by how non-disruptive it is. Even if the vendor's professional services consultant is engaged, their work mostly focuses on seamlessly provisioning the new firewall. The consultant may help the customer with some security policy rules and offer guidance on further steps, but their engagement is limited. Unfortunately, a handful of good security policy rules can hardly be called an enterprise security policy. Further work is always required, but it is often postponed (if not completely forgotten) for months and years.

- **Compounding inadequate application connectivity requirements:** Firewall security policies are rarely created as a single configuration element from the onset of the firewall deployment. Instead, they are often developed *on the fly* in response to ad-hoc business requirements. For instance, access to specific applications may need to be granted or blocked, or particular individuals or teams may need to be prohibited from using certain websites.

These requirements are added one by one over time. Unfortunately, not all of them are of good quality, and they *never* take into account previous changes and the existing firewall policy as a whole. Influenced by the nature of applications and the maturity of respective application vendors in the context of networking and cybersecurity, application requirements can be vague and very generic.

Often complemented by a lack of experience and knowledge of the implementing firewall administrators, this results in a poorly connected and inconsistent set of policy rules that comprise the entire firewall security policy. Because of the complexity of the application-oriented policy matching algorithm (think of the *application shift* processing that we covered in the previous chapter), such policies start behaving inconsistently, and firewall administrators find themselves in a neverending loop of fixing access issues for business users. This neither helps their popularity among their immediate colleagues and a wider business nor improves network security.

- **Product development challenges:** Cybersecurity ventures often start with a spark of innovation from engineers, capturing the market's attention and securing early success. As these companies evolve, the drive to stay ahead in a very competitive market leads to the development of new features aimed at attracting further investment and expanding market share.

This growth brings challenges. The effort to market these innovations escalates, with marketing teams expanding and employing the latest industry buzzwords to showcase their products. Meanwhile, pre-sales engineers who once had a deep understanding of the core product now struggle to navigate an ever-expanding feature set, often relying on these buzzwords to communicate value to customers.

The impact on customers can be significant. The promise of new and essential features often comes with a hidden cost: the burden of navigating through a myriad of issues and obvious software bugs, transforming what should be a seamless experience into an extension of the vendor's quality assurance process. The quality and innovation that once defined these products can seem overshadowed by the sheer volume of features, affecting their usability and efficiency in customers' production environments.

Despite these issues, the financial success of these vendors often masks the underlying issues from stakeholders, with impressive sales figures and market share growth presenting a picture of unmitigated success.

- **Suboptimal product support:** Ideally, initiating a support case would be automated and streamlined. Key details would be pre-filled directly from the product interface, sparing customers from the cumbersome process of manually answering numerous, often non-relevant questions.

However, reality often involves complex support systems coupled with extensive technical support teams, inefficient escalation procedures, and multiple tiers of service agreements. While this structure aims to provide comprehensive assistance, it may actually reflect areas where further investment in product development quality could reduce the need for such extensive support mechanisms.

- **Incomplete product documentation:** Navigating product documentation for new features can sometimes lead to more questions than answers. Engineers might find themselves consulting knowledge base articles or the built-in documentation of a firewall, only to encounter gaps or inconsistencies.

Reaching out to the assigned system engineer for clarity might not always yield immediate solutions, despite their expertise in showcasing the feature's benefits.

- **Lack of sense of ownership and sloppy administration:** Although it is easy to blame the vendor for all the sins of the world, sometimes the saying *problem in chair, not in computer* is very true. Humans are prone to making mistakes, and firewall administrators, like anyone else, can unintentionally make errors for a variety of reasons.

These errors range from innocent mistakes, such as omitting a standard suffix in the name of an address object, to the attitude of *Why would I care if nobody else does?*. If the latter is the case, then the department has a bigger problem, of course.

Over time, these errors start taking a compound effect, resulting in issues ranging from duplicate address objects to overly permissive and generic rules at the beginning of the policy shadowing and invalidating everything below these rules.

- **False positives:** When a firewall security feature mistakes a benign network connection for a malicious one, therefore blocking it, and this connection turns out to be part of a key application, the escalation within the business does not take long.

When a security feature, initially implemented with good intentions, consistently generates false positives, it is ultimately disabled permanently, regardless of the vendor's best practice suggestions. This inevitably results in the security risk that the feature was intended to mitigate being left unaddressed.

- **Undue business influence and underappreciation:** Security decisions, including changes to firewall security policies, sometimes succumb to business pressures, risking compromise on many fronts—from breaking the integrity of the security policies to loss of standardization, massively increased administrative overhead, and dangerous exposure of the business to cyber threats.

Advocating for security as a core business function can help align priorities and ensure adequate resources, but not every company is fortunate enough to have a CISO or information security manager capable of achieving this.

Additional hurdles are lack of standardization, inadequate business requirements (or their absence), configuration drift, low staff retention, lack of executive support, and budget constraints; these issues will resonate with many readers.

Cybersecurity risks versus you

With all the complexities of managing firewalls in real life that we briefly covered in the previous section, we must not forget why we use firewalls in the first place. Let us discuss what next-generation firewalls can and cannot do and what risks firewalls can mitigate or even remediate.

Then, we will see how we can translate the risks and challenges into a set of requirements that will allow us to proceed with building the actual policy.

I will list some of the apparent risks (in no particular order) and briefly elaborate on what we can and cannot expect from a firewall.

Risk of unauthorized access to a restricted resource

A resource can be either a sensitive application (internal or external) or a category of applications or URLs.

Firewalls *can* identify users and the groups they belong to; devices; applications, their capabilities, characteristics, and risk levels; URLs, their categories, and risk levels; and leverage custom application signatures. They can also identify virtual machines in public and private clouds, dynamically identify and tag IP addresses based on various connection attributes, and, subsequently, use this tagging in policies.

Firewalls *cannot* (or have a limited ability to) authenticate individual network connections. Strictly speaking, firewalls only establish *IP-to-user* (or *IP-and-port-range-to-user*) and/or *IP-to-device* mappings using various methods. These methods may involve initial authentication done by the firewall itself (for example, using a GlobalProtect internal gateway), but subsequent connections from an application or a different user from the same IP may not be identified correctly. For instance, an application launched via *Run As* in a user's Windows session or a background service running under a domain service account will not change the firewall's perception of the identity of all connections originating from the computer after GlobalProtect initially authenticates the session user.

Risk of successful unauthorized external access to the network

Such access can generally happen via the following attack vectors:

- A compromise of an application (a website, for example) exposed to the internet or a third party through the firewall
- A compromise of a firewall feature exposed to the internet (such as GlobalProtect – a feature providing remote access VPN functionality for company employees)
- A phishing/spear-phishing attack against a user leading to the compromise of the user's PC
- Malware planted on endpoints out of band (i.e., in a way that bypassed firewall inspection – a USB memory stick, compromise of a laptop while off the corporate network, etc.)
- Supply-chain attacks
- Compromised third-party source code and software packages used in the company's in-house applications

Firewalls *can* decrypt encrypted connections, identify protocols and applications, block known malicious payloads (for example, popular exploit kits used by hackers), block known application layer attacks (for example, SQL injection), block known malicious sources and destinations of traffic, block risky file types, block risky geolocations, use big data and machine learning techniques to block associated DNS traffic, use sandboxing to assess unknown files, and detect and block submission of user credentials.

Firewalls *cannot* (or have a limited ability to) do the following:

- Analyze proprietary or custom protocols and data structures (websocket, for example)
- Decrypt connections when software actively counteracts this by using certificate pinning or mutual certificate-based authentication
- Use custom OS images for sandboxing (which reduces the value of sandboxing)
- Always effectively resist firewall evasion techniques (for example, HTTP partial response, necessary for the usability of many web applications, directly impacts a firewall's ability to detect and scan data structures such as files within HTTP transactions)
- Patch/remediate vulnerabilities of their own features
- Identify the fact of malware planted on endpoints out of the firewall's line of sight
- Identify malware in the source code

- Identify malware in software packages signed by trusted certificate authorities
- Distinguish between trusted and untrusted software repositories

Risk of denial of service (DoS) and distributed denial of service (DDoS) attacks

Such attacks are aimed at disrupting the availability of a resource and would usually target an application (a website, for example) exposed to the internet through the firewall.

Firewalls *can* decrypt encrypted connections, identify protocols and applications, block known application layer DoS attacks (based on vulnerability signatures), protect application resources by limiting the number of concurrent sessions (total and/or per client), and block TCP SYN, UDP, ICMP, and other IP floods.

Firewalls *cannot* (or have a limited ability to) block volumetric DoS attacks and effectively distinguish and allow legitimate connections within a high-volume flood of incoming malicious requests.

Risk of eavesdropping/interception of sensitive data

Sensitive data may be intercepted and misused by threat actors while it is in transit between a user and a remote application.

Firewalls *can* identify and block encrypted sessions with expired certificates and untrusted issuers (therefore, removing the ability of end users to bypass browser security warnings), and they can identify and block connections transmitting sensitive data to unauthorized destinations—this can be done based on Microsoft’s **Azure Information Protection (AIP)** labels, for example.

Firewalls *cannot* identify the sensitivity of files within encrypted archives and enforce encryption policy based on application.

Risk of violating applicable compliance requirements

A company may be obliged to follow a certain compliance regime that requires it to enforce record-keeping of all communication of a regulated activity. For example, stock traders may be required to use only a particular corporate instant messaging application but not a consumer version of the same application. Another example is the need to ensure that credit card data never leaves a PCI DSS-certified network segment.

Firewalls *can* identify applications, classify and block them accordingly based on the user’s identity, detect and block data patterns (such as credit card numbers) in transmitted files, and classify URLs, thus allowing the company to manage user access to restricted website categories.

Firewalls *may* have limited ability to distinguish tenants/instances within certain multi-tenant SaaS applications that require granular control according to corporate policies.

Risk of data exfiltration

Data can be exfiltrated either by a hacker or an insider. They can use different methods to achieve their goals. A disgruntled employee may simply use a web browser to upload sensitive information to their personal Google Drive, while an advanced threat actor can use highly sophisticated techniques that leverage loopholes in popular network protocols such as DNS to exfiltrate the same data.

Firewalls *can* decrypt connections, identify and classify applications based on their characteristics (such as the ability to transfer files), detect files, their types, and data patterns within, use cloud-based machine learning and techniques based on big data analysis to detect DNS-based data exfiltration, and correlate some events.

Firewalls *cannot* or have limited ability to analyze human behavior, distinguish insiders, and identify novel/zero-day exfiltration techniques that do not yet have vulnerability/spyware signatures.

Conclusion

This list of risks is incomplete, of course. However, the point I am trying to make is obvious—a firewall is not a silver bullet for network security. We must contemplate a firewall, no matter how sophisticated, only as one of the elements (albeit a very important one) of the enterprise security program.

When we consider a use case where a firewall is used as a company internet gateway (the scenario covered by this book), we must focus on building a solid baseline policy that does the following:

- Integrates well with other security controls, wider I.T. infrastructure, and processes
- Allows us to be risk-centric and efficient
- Considers all the challenges of managing firewalls in real life, which we discussed in the previous section of this chapter

All these bullet points, and especially the latter one, encapsulate a lot of considerations.

In the following section, we will wrap up this chapter by developing a formal set of requirements that our security policy must fulfill to effectively address all risks and challenges.

Functional and non-functional requirements

Requirements in IT architecture are usually divided into two distinctive groups with somewhat confusing names: **functional** and **non-functional** requirements. The former category is focused on what the designed subject must achieve (*behavior*), and the latter concentrates on *how* the functional requirement must be implemented from a technological perspective.

If you have ever tried to ask a business stakeholder about the requirements for a network firewall policy and that stakeholder gave you more than “Ehh... *be secure?*”, then you are a lucky person. The information security department is usually just a service function whose unobtrusiveness and ability to stay on top of everything are the main traits. Nobody wants their business processes to be interrupted or delayed. At the same time, everyone expects their business process (no matter how badly designed) to be absolutely safe from a cybersecurity perspective.

Therefore, when it comes to getting functional requirements signed off, the business usually follows what its trusted subject matter expert suggests. And since you bought this book, this person is YOU—congratulations!

Let us reflect on the cybersecurity risks we face in the modern world and the challenges we have with managing firewalls in real life, and put together a list of sensible requirements defining how our security policy should *behave*.

We will need some specifics when setting requirements. Therefore, we will use a fictional company with distinctive characteristics and requirements resembling those of a typical medium-sized global business.

Example company

Here is a high-level diagram representing the global footprint of our company:

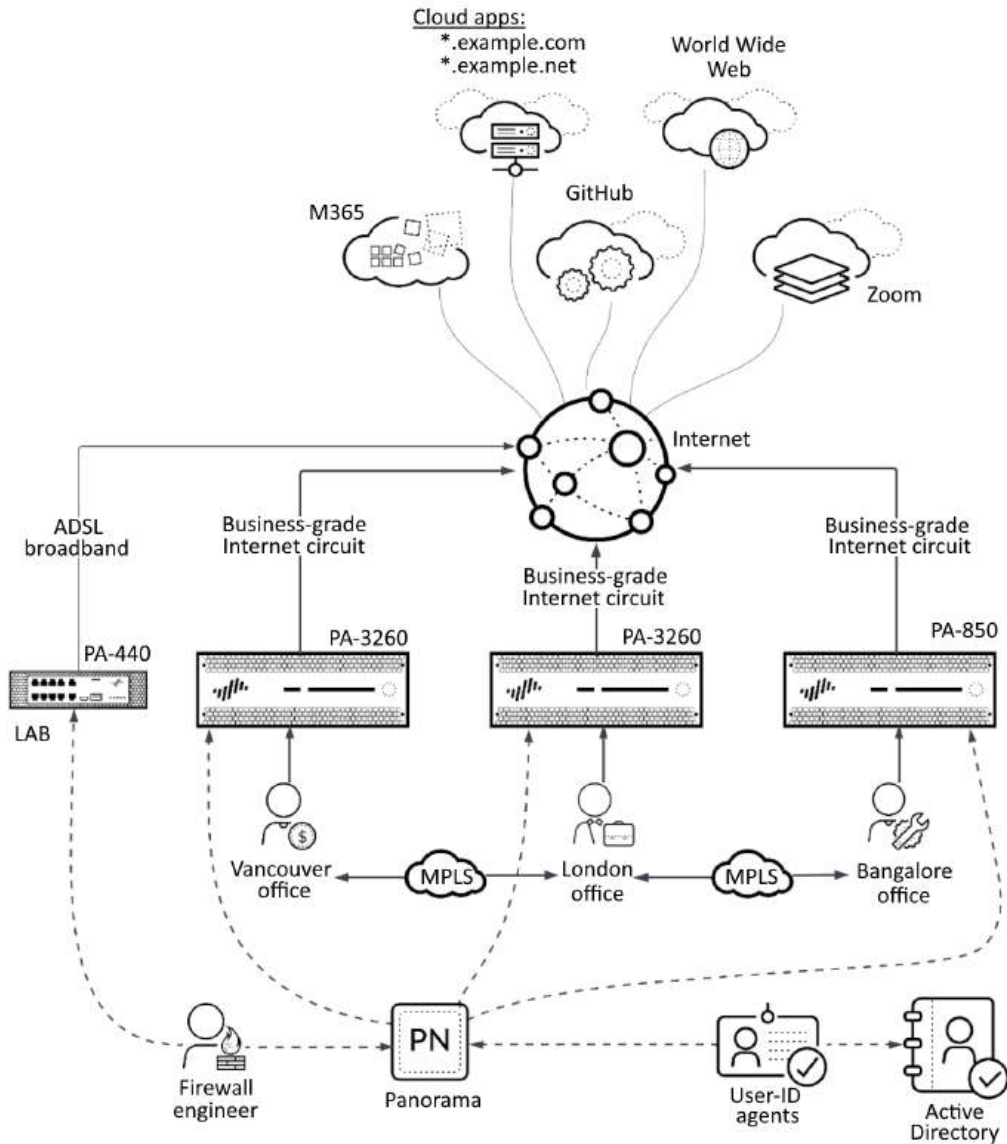


Figure 2.1 – Example company

In the preceding figure, consider the following:

- **Name of the company:** ACME.
- **Number of employees:** ~1,000; some of the employees regularly travel to different offices.

- **Type of business:** A regulated international business that operates in three different countries and offers services to the public and other companies.
- **Specifics of the regulations:**
 - Among other requirements, the industry regulator dictates that the business must strictly control access to email and instant messaging applications so that all electronic communications of employees with customers and business partners are conducted via enterprise email only and, therefore, can be recorded.
 - The company is also required to have no business whatsoever with countries and territories currently under government sanctions. The list currently consists of one territory: *Afghanistan*.
- **Number of physical sites:** Three sites located in Vancouver (Canada), London (UK), and Bangalore (India).
- **Network architecture:** All offices are connected to each other over an MPLS network. Each office has its own direct internet access provided by a local ISP. Inter-site traffic does not traverse the firewalls at present, but this is planned to be implemented in the future.
- **Firewall infrastructure:** Each office is connected to the internet via a locally deployed Palo Alto Networks next-generation firewall. The Vancouver and London offices run on PA-3260 hardware appliances, while Bangalore runs on the older PA-850 model. There is also a lab facility with a PA-440 firewall and a dedicated broadband connection. Several Windows-based User-ID agents have been deployed across the firm, and firewalls are able to identify users.
- **Firewall management:** All production firewalls are linked to an on-premises Panorama management appliance. Meanwhile, the Lab firewall is a standalone device.
- **Application infrastructure:** The company's infrastructure is hybrid and mainly based on Microsoft products. The firm uses Active Directory with a single domain. Microsoft 365 and OneDrive are the primary office productivity tools. Zoom is used for video conferencing, while the in-house software developers use GitHub extensively. The company has several of its own applications hosted in the cloud in the `example.net` and `example.com` domains.
- **Current firewall security policy:** All three production firewalls run different security policies deployed by different employees and consultants over several years, all of whom have now left the firm. Security policies comprise hundreds of rules, often overlapping and shadowing each other. Nobody in the information security department can tell how *exactly* the policies currently operate. The lack of structure in the policies resulted in many lengthy troubleshooting sessions which always ended with yet another *allow* or *deny* rule added on top of the existing policy rules.

Functional requirements (FRs)

The ACME new CISO stated that the next-generation firewall security policy must meet the criteria as follows:

1. The policy should be unified across all current and future company offices but allow minor site-specific deviations. Employees should have consistent experience accessing applications and the internet across all offices.
2. The policy should prevent unauthorized access to restricted resources. Resources may include in-house and third-party online applications, websites, and categories thereof.
3. The policy must be able to control standard and custom categories of applications and websites.
4. The policy must protect internal endpoints from exploitation of their vulnerabilities and from other associated threats.
5. The policy should prevent users from downloading arbitrary software from the internet.
6. The policy should be able to identify sensitive company data (files tagged with Microsoft's AIP sensitivity labels) and prevent potential exfiltration of this data via unauthorized file-sharing applications and webmail.
7. The policy must block known bad sources, destinations, and malware.
8. The policy must block connections to and from geographical territories and countries in accordance with current regulations.
9. The policy should be risk-centric in policy decisions where possible – apply stricter controls and produce more detailed logs for riskier connections.
10. The policy should provide a flexible mechanism for declaring an application as either sanctioned or unsanctioned, allowing the CISO, CIO, and compliance department to gauge and control application usage and associated risks.
11. The policy should automatically identify and isolate compromised network hosts and automatically raise a security incident when this happens.
12. The policy must be user-friendly – make it easy for users to understand why their access was blocked and make it easy to raise these issues with the IT service desk.
13. The policy should be non-disruptive and reliable. Key business applications must always be accessible. There should be the ability to fully test new revisions of the policy in the lab environment.

14. The policy must be admin-friendly and require low maintenance.
15. The policy must have a mechanism to make temporary exceptions from the policy controls.
16. All policy changes must be version-controlled and easily traceable.

Non-functional requirements (NFRs)

Looking ahead a little and assuming we know PAN-OS's core capabilities, let's consider *how* we can implement these functional requirements.

You may not be familiar with some of the technological features mentioned here, but rest assured we will cover them in detail in the next two chapters:

1. The policy must leverage an automation framework to satisfy FRs 1, 13, and 16 (refer to the numbered functional requirements in the previous section).
2. User-ID must be used to control access to restricted resources to satisfy FR 2. Access control must be implemented using user groups rather than individual user accounts in accordance with industry best practices. In the context of authentication, all resources should be divided into three types: those available to groups of users, those available to authenticated users regardless of their group membership, and those requiring no authentication.
3. Generic web applications and sites should be classified based on their distinctive characteristics and risk levels. Therefore, they should be described using application filters and controlled at this level of granularity rather than individually. The classification mechanism should resort to URL filtering when a web application does not have an applicable App-ID signature. Custom application signatures and URL categories should be used when standard vendor categories do not provide the required scope of access control. This satisfies FRs 2 and 3.
4. TLS decryption must be a prerequisite to satisfy FRs 4, 5, 6, and 7.
5. Vulnerability and Anti-Spyware profiles must be used to satisfy FR 4.
6. File Blocking profiles must be used to satisfy FR 5.
7. Data Filtering profiles and custom data patterns differentiated at the application filter/group level should be leveraged to fulfill FR 6.
8. Built-in external dynamic lists with known malicious addresses along with Antivirus profiles and DNS Security should be used to satisfy FR 7.
9. Policy rules with geo-blocking based on built-in geolocation database must be used to satisfy FR 8.

10. Policy rules must determine the strictness of the applied security profiles and Log Forwarding profiles based on the risk level of the destination URL (for web-based applications) – this will satisfy FR 9.
11. The policy should leverage the application signature tagging mechanism to provide custom selection criteria in application filters and visibility via built-in PDF reports. This will address FR 10.
12. The policy must use the **Built-in Actions** mechanism in Log Forwarding profiles to isolate hosts exhibiting strong indicators of compromise, such as matching threat signatures for *command-and-control* traffic. HTTP profiles must be used to raise security incidents. This will cover FR 11.
13. When the policy blocks a user's connection and displays a firewall response page, it must generate a contextualized message and a hyperlink providing necessary metadata about the issue to the IT service desk system. This NFR covers FR 12.
14. To satisfy FR 13, the policy must explicitly provision access to the key business applications.
15. The policy should use a strict naming convention and group tagging of policy rules to visually distinguish policy sections. Chosen selection criteria in the application filters should automatically account for new application signatures released by the vendor. This addresses FR 14.

The policy must use external dynamic lists, auto-tagging, and dynamic address groups to satisfy FR 15. Now, we have defined the *what* (FRs) and *how* (NFRs) of our policy.

Summary

In this chapter, you explored the complex realities of managing firewalls in real-world enterprise environments. You examined a wide range of challenges—from technical complexity and lack of training to inconsistent deployments, rushed migrations, and administrative oversights. These issues are often compounded by weak documentation, subpar vendor support, business pressure, and human error. Together, they create an environment where even the most capable firewall can become challenging to manage effectively without a clear structure, ownership, and planning.

You also assessed the limitations and capabilities of modern firewalls in the context of today's cybersecurity threats. While firewalls can block known threats, decrypt traffic, enforce compliance, and mitigate specific attack vectors, they cannot do everything — especially when it comes to advanced or insider threats, encrypted payloads, or zero-day techniques.

With this understanding, a comprehensive set of requirements for a new firewall policy was defined, using a fictional global company as the reference point. These requirements now form the baseline for a policy that is not only technically sound but also aligned with business needs and risk priorities.

In the next chapter, you'll dive into the fine details of security policy elements and the PAN-OS features behind them, giving you the insight needed to make the right design decisions as we move forward.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



3

PAN-OS Security Policy Features: Connection Matching

Building on the challenges and requirements discussed in the previous chapter, this one focuses on the mechanics that drive security enforcement in PAN-OS. Now that you have a clear understanding of the practical limitations of firewalls and the strategic needs of a modern enterprise, it's time to examine how traffic is matched to policy rules—a critical step in designing adequate firewall controls.

This chapter introduces the complete set of connection-matching features available in PAN-OS. You'll learn how the firewall uses various attributes—such as zones, addresses, users, applications, and URL categories—to determine whether a network connection matches a security policy rule. In addition to conventional criteria, we'll also explore advanced options such as dynamic objects, external lists, and geolocation.

By the end, you'll have a solid grasp of all the matching options you can use when constructing security rules. This knowledge will serve as the technical groundwork for the next chapter, where we'll turn to how PAN-OS processes matched connections and applies security controls.



As you read this chapter, you may want to find a more detailed description of one of the mentioned firewall features. I highly recommend you read *Mastering Palo Alto Networks* by Tom Piens from *Packt Publishing* as a comprehensive yet succinct guide on configuring PAN-OS. Meanwhile, this chapter and the following one aim to provide the depth of detail minimally necessary to comprehend material in the subsequent chapters, all focused on building a firewall security policy.

This chapter will categorize all security rule elements into three distinct groups and discuss the one responsible for connection matching, which includes the following elements:

- Rule types
- Source and destination zones
- Source and destination addresses
- Source and destination port numbers
- Source user
- Application
- URL category



Be aware that this chapter contains several hyperlinks, as copyrighted materials cannot be reproduced here.

Technical requirements

If you have access to a Palo Alto Networks firewall while reading this chapter, it can really help you visualize, understand, and remember all the features we're discussing.

Security rule components

Speaking of the security policy, let's begin by referring to the following screenshot to visualize the location of the three functional groups of features of a security policy rule.

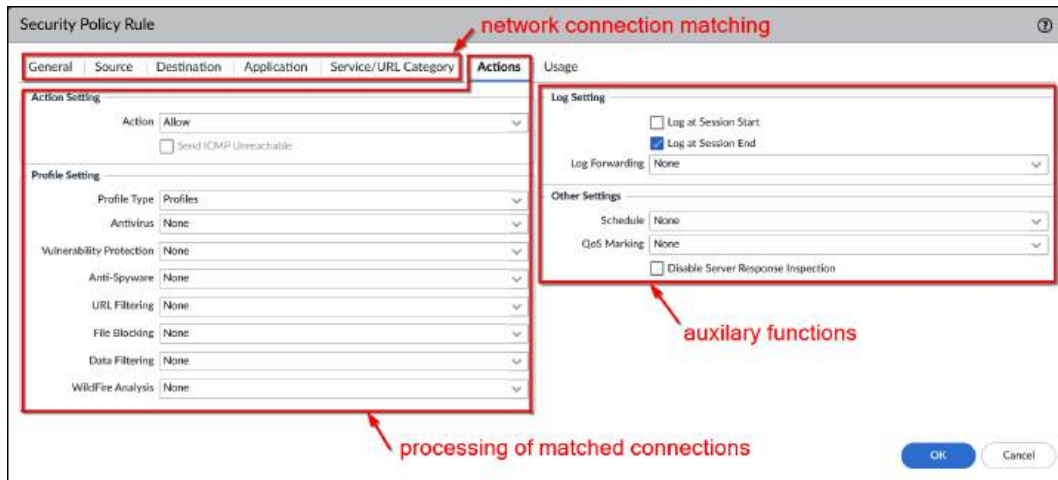


Figure 3.1 – Security policy feature groups

As you can see here, the security rule configuration window has seven tabs.

The first five tabs (**General**, **Source**, **Destination**, **Application**, and **Service/URL Category**) have configurable attributes related to *connection matching*. The **Actions** tab contains configuration items related to the *processing of the matched connections* and some *auxiliary* functions.

On the contrary, the **Usage** tab does not have any configurable features. It shows when the rule was created and edited, when it was hit for the first and last time, the applications that match this rule so far, and the total traffic handled by the rule over the past 30 days. It also helps you compare the applications specified in the rule with those actually detected.

Let's thoroughly examine each of the configurable elements.

Connection matching

As you learned in the *Networking 101* section of *Chapter 1*, a network connection can be matched to a policy rule based on any combination of the connection attributes, as follows:

- Source and destination zone(s)
- Source and destination address(es)
- Source and destination port number(s)
- Application(s)
- URL categories (for HTTP- and HTTPS-based applications only)

However, there are additional matching criteria. They are not derived from the incoming packets but are a result of the operation of other firewall features—User-ID, GlobalProtect, IoT Security, and 5G Security:

- **User-ID** provides source users or groups
- **GlobalProtect** and **Prisma Access** provide source **Host Information Profiles (HIPs)**
- **IoT Security** provides device identity information, that is, Device-ID
- **5G Security** provides source subscribers, source equipment, and network slices

Out of these four elements, we will cover only the Source User provided by the User-ID subsystem of PAN-OS. IoT Security, 5G Security, Global Protect, and Prisma Access are beyond the core set of firewall capabilities and products described in this book.

All connection-matching attributes of a **Security Policy Rule** are grouped under the tabs shown in the following screenshot:

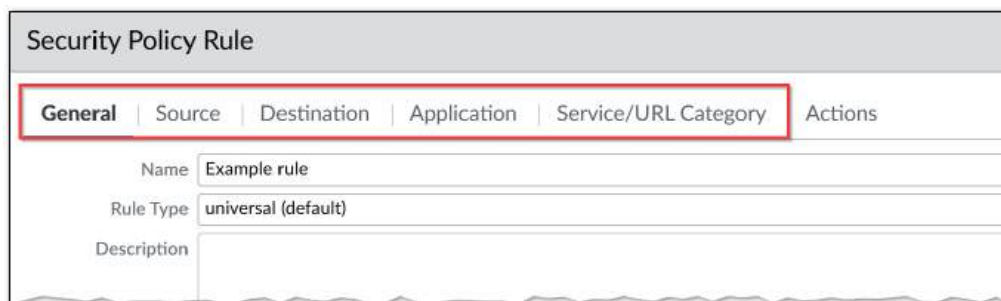


Figure 3.2 – Security Policy Rule tabs related to connection matching

Note that the **Usage** tab is missing when you create a new rule. However, the tab will appear if you re-open the rule you have just created.

The default value for all connection-matching attributes except **Service** is **Any**. The default value for **Service** is **application-default**, which means that port number(s) and applicable timeout values must be derived from the application definition. Therefore, each application matching this rule will be allowed only over the specified port numbers and with timeouts specified in the application's definition.

Let us take a closer look at each of these attributes.

Rule type

Rule types differ by the way they treat specified zones:

- The **universal (default)** type will match traffic between any zone specified as the source and any zone specified as the destination (including **intrazone** traffic when the same zone is the source and the destination simultaneously). This is the default rule type.
- The **intrazone** type will match only traffic within the same zone. If you specify more than one zone, the rule will match only traffic within each of these zones but not between them.
- The **interzone** type will match traffic from any zone specified as the source to any zone specified as the destination, provided it is not the same zone.

All these security rule types are illustrated in the following screenshot:

	NAME	TYPE	Source	Destination
			ZONE	ZONE
1	universal-rule	universal		
2	intrazone-rule	intrazone		(intrazone)
3	interzone-rule	interzone		

Figure 3.3 – Security policy rule types

Source and destination zones

After defining the rule type (or accepting the **universal (default)** type), you must define source and destination zones. The latter is not needed in the intrazone rules.

The source zone essentially defines a set of interfaces where we expect to receive the incoming packets of the connections of interest. You can specify as many zones as you want. If an incoming packet hits any interface of any of the specified zones, the source zone matching criterion will be satisfied.

To meet the destination zone matching criterion, packets must exit the firewall via an interface belonging to one of the zones specified in the **Destination Zone** field. The zone-matching algorithm will depend on the rule type, as shown in the preceding screenshot. For intrazone rules, you can specify only the source zone because, as the name implies, the destination zone must match the source zone.



If you remember from the *Packet flow inside the firewall – application identification* section in *Chapter 1*, the firewall will identify the actual egress interface (and, therefore, the destination zone) based on the **Forwarding Information Base (FIB)** table. You also need to bear in mind that you may have a **Network Address Translation (NAT)** policy in place that, in turn, is always evaluated *before* the security policy. The NAT policy may change the egress zone of the packet. Therefore, as you create security policy rules, follow the empirical rule “pre-NAT addresses, post-NAT zones.” In other words, this means that you need to disregard the NAT policy when you specify source and destination addresses in your security rules but take it into account when you specify source and destination zones.

Rules matching multicast traffic must use a special destination zone—**multicast**. This book does not cover this type of traffic and zone; however, if you wish to learn more, you can visit <https://docs.paloaltonetworks.com/pan-os/11-1/pan-os-networking-admin/ip-multicast>.

Finally, you can specify the **Any** keyword, which is self-explanatory because it means any zone defined on the firewall.

Source and destination addresses

When you specify the source and destination addresses, you can use *any combination* of the following object types.

Addresses

This object type is created via **OBJECTS | Addresses**. Objects in this category can be defined as **IP Netmask**, **IP Range**, **IP Wildcard Mask**, or **FQDN**.

IP Netmask is defined as an IPv4 or IPv6 address in CIDR notation. If you omit the mask, it is assumed to be /32. It is considered best practice to always specify the mask.

IP Range is defined as a continuous range of IPv4 or IPv6 addresses. For example, 93.184.216.34–93.184.216.36 is a valid range of three addresses.

IP Wildcard Mask is defined as an address and a wildcard mask. To determine whether an address matches the object of this type, you need to convert the address to its binary representation and reset to 0 all its bits that correspond to bits set to 1 in the specified mask. After you have reset (that is, masked) all these bits, compare the result to the address part in the wildcard object. Provided they are identical, you have a match.

Creating these objects with masks aligned at the octet (byte) boundary is easier. For example, the wildcard object defined as `10.0.1.0/0.255.0.255` would effectively represent addresses `10.<0...255>.1.<0...255>`. The prime use case for such objects is a multisite deployment where the IP address schema has a site and a VLAN/subnet type encoded into the address. Address objects of this type are very powerful, as the following example demonstrates.

Let's assume that your company has 256 or fewer offices, each with a number of /24 standard VLANs. All site networks match the standard `10.<site code>.<vlan number>.0/24` pattern. With this address schema, you need to create *only one* wildcard mask object per VLAN (for instance, `10.0.21.0/0.255.0.255` for VLAN 21 at each site) that would cover all addressed from VLANs with that number at every single office. If you were to address the same subnets with regular **IP Netmask** objects, you would need to create as many objects as you have sites.

The only unfortunate limitation of this address object type is that it can be used *only* in security policies. It cannot be used in NAT, QoS, decryption, or other policy types.

Fully Qualified Domain Name (FQDN) objects represent another important type of address object that we'll explore next.

As we learned in the *Networking 101* section of *Chapter 1*, this object type is tricky. DNS-based load-balancing of an application may cause the FQDN to resolve to different addresses on the firewall and on the client, thereby not achieving the desired result of a policy rule where such an object is referenced from.

The firewall will initially resolve an FQDN to an IP address at the time of configuration commit using the **DNS Servers** or **DNS Proxy Object** setting specified in **DEVICE | Setup | Services | DNS Settings**. The firewall will respect the TTL value of the DNS server response (and refresh the object accordingly) provided that this value is greater than the value specified in **DEVICE | Setup | Services | DNS Settings | Minimum FQDN Refresh Time (sec)** (30 by default), which makes FQDN-based objects very unreliable in security policy context for destinations with very low TTLs. You can set the minimum refresh time to 0 to ensure the firewall always respects the TTL value provided by the DNS server, no matter how low that TTL value is. If you do this, beware that having many low-TTL FQDNs in the policy can strain firewall resources.

FQDN objects that cannot be resolved because DNS servers are unreachable are declared *stale*. The last known IP address(es) of a stale FQDN objects will be kept in the firewall cache for the length of time specified in **DEVICE | Setup | Services | DNS Settings | FQDN Stale Entry Timeout (min)** (1440 by default).

Here are some examples of valid address objects:

NAME	TYPE	ADDRESS
FQDN-example.com	FQDN	www.example.com
IP-NETMASK-example.com_93.184.216.34_32	IP Netmask	93.184.216.34/32
IP-RANGE-example-93.184.216.34_93.184.216.35	IP Range	93.184.216.34-93.184.216.35
IP-WILDCARD-example	IP Wildcard	10.0.1.0/0.255.0.255

Figure 3.4 – Example address objects

Address groups

This type is created via **OBJECTS | Address Groups**. Groups are subdivided into the following:

- Static
- Dynamic

The former group type can contain individual address objects and other address groups (both static and dynamic). As the name suggests, the content of a static group cannot change at runtime—whatever you put in the group stays there. The dynamic objects included in a static address group (FQDN objects and dynamic address groups) will be updated at runtime accordingly. The following screenshot demonstrates an example of a valid group that contains objects of different types:

Address Group

Name

STATIC-GROUP-example

Description

Type

Static

Addresses

☐ ADDRESS ^

☐ DYNAMIC-GROUP-example

☐ FQDN-example.com

☐ IP-NETMASK-example.com_93.184.216.34_32

☐ IP-RANGE-example-93.184.216.34_93.184.216.35

☐ SUB-GROUP-example

Browse

Add

Delete

Tags

OK

Cancel

Figure 3.5 – Example static address group

On the other hand, dynamic address groups do not have a preconfigured list of members. Instead, they define only a selection criterion—a logical expression comprised of tags and the AND and OR logical operands.

For example, imagine you have tagged address objects and a dynamic address group based on these tags, as shown in the following screenshot:

	NAME	TAGS	ADDRESS
☐	FQDN-host-a	Linux	host-a.example.com
		Server	
☐	FQDN-host-c	Windows	ho
		Server	
☐	IP-host-b	Linux	19
		Workstation	
☐	IP-host-d	Windows	19
		Workstation	

Address Group

Name

DYNAMIC-GROUP-all-linux-servers

Description

Type

Dynamic

Match

'Server' and 'Linux'

Figure 3.6 – Example address objects with tags and a dynamic group referencing these tags

Here, we have four address objects, each with two assigned tags. The address group also references two tags.

Only one of these objects—**FQDN-host-a**—will match the specified criterion because it is the only address object with the **Linux** and **Server** tags. The following screenshot demonstrates this.

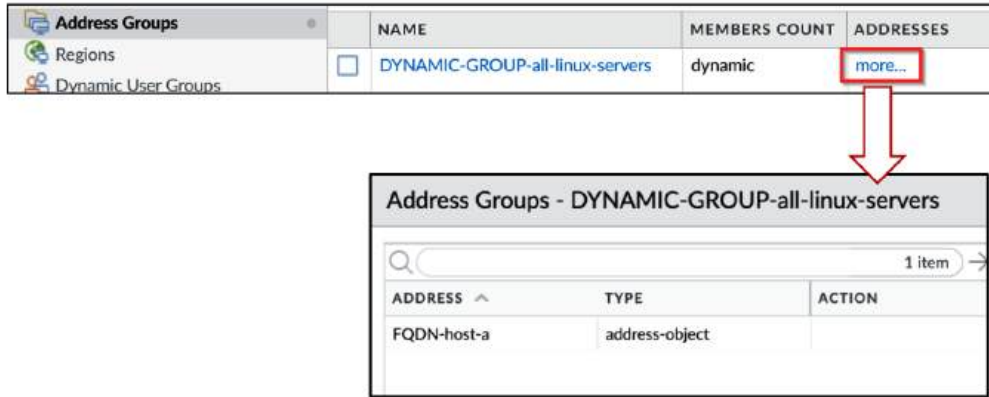


Figure 3.7 – How to view dynamic address group members

The screenshot shows how to view the effective members of a dynamic address group.

The preceding example demonstrates how dynamic groups work with statically tagged objects. Note that you can tag not only individual address objects but also static and dynamic address groups. If a group had the same **Linux** and **Server** tags used in the preceding example, it would also have been included in the example dynamic group.

Another, more powerful, use case for dynamic groups is their ability to include IP addresses tagged by the firewall at runtime. This is achieved via the feature called **built-in actions** in the log forwarding profiles. We will discuss it in *Chapter 4*.

Regions

There is a special type of address object called a **region**.

Next-generation firewalls contain an IP geolocation database regularly updated as part of **Applications and Threats** dynamic updates. This database is built and based on information retrieved by Palo Alto Networks directly from the respective regional internet registries such as the **American Registry for Internet Numbers (ARIN)** and **Réseaux IP Européens Network Coordination Centre (RIPE NCC)**.

All public IP addresses in this database have a geolocation attribute that matches one of approximately 300 possible “regions.” A region is not necessarily a country in its own right—it can be a disputable territory, an internationally governed one (Antarctica, for example), or a special category such as **Anonymous Proxy** or **Satellite Provider**. There is also a set of regions defined as IP ranges. Some examples are the following:

- 10.0.0.0-10.255.255.255 (private IP range)
- 172.16.0.0-172.31.255.255 (private IP range)
- 192.168.0.0-192.168.255.255 (private IP range)
- 100.64.0.0-100.127.255.255 (carrier-grade NAT)
- 224.0.0.0-224.255.255.255 (multicast)

You will recognize these IP ranges from the *Networking 101* section of *Chapter 1*.

You can create custom regions via **OBJECTS | Regions**. A custom region includes a name and a list of IP subnets and/or IP ranges. You can also specify the region’s latitude and longitude.

Sometimes, you may observe the incorrect geolocation of an IP address. You be certain that an address belongs to your business partner’s website hosted in Virginia, US, but the firewall would claim that the address is somewhere in Spain. To overcome this issue, you need to override the vendor’s value with your own one, as shown in the following example:

Region ?

Region

☒ Shared

☐ **Geo Location**

Latitude

Longitude

☐ **ADDRESS**

☐	203.0.113.1/24

Figure 3.8 – How to override the region for an IP address

Geolocation information may be useful from an observability perspective. The world maps displayed in **Monitor | App Scope | Traffic Map** and **Threat Map** use this database.

Therefore, you can create custom regions for your data centers and/or offices, assign a latitude and longitude to them, and enjoy a world view of your enterprise traffic and threats. Firewalls, however, won't be placed on the map purely based on the regions their interfaces belong to—you will have to define each firewall's location explicitly. This can be done in **Device | Setup | Management | General Settings | Latitude and Longitude**.

Note that custom regions override the built-in ones and do not work like address objects.

For instance, imagine you have a rule in your security policy that is based on the destination address specified as the standard region 10.0.0.0-10.255.255.255. Then, you create an identical rule below that rule but based on a custom region named 10.1.1.0-10.1.1.255 and define it as 10.1.1.0/24:

	NAME	Source	Destination	APPLICATI...	SERVICE	ACTION
		ADDRESS	ADDRESS			
1	rule-based-on-a-standard-region	any	 10.0.0.0-10.255.255.255	any	any	 Allow
2	rule-based-on-a-custom-region	any	 10.1.1.0-10.1.1.255	any	any	 Deny

Figure 3.9 – Illustration of custom region use

Now, if you have a suitable network connection to 10.1.1.1, logically, you would expect it to match rule 1 because the address 10.1.1.1 is within the range 10.0.0.0...10.255.255.255, matching the standard region in the first rule.

However, it would only work like this if you had regular address objects, not regions. In fact, the connection will match the second rule—the one with your custom region, despite the fact that it is placed below a more generic rule, also seemingly matching the destination.

This may seem counterintuitive, but it is not. When you consider region-based rule matching, you need to do the following:

- Look at IP-based region names as strings rather than the implied meaning of these strings
- Bear in mind that when IP addresses of a custom region overlap with IP addresses of a standard region, the former takes precedence

External dynamic lists

External Dynamic Lists (EDLs) are created via **OBJECTS | External Dynamic Lists**.

EDLs are regular text files hosted on a web server that the firewall can access. The firewall retrieves these files at regular set intervals and immediately applies their content to the rules where the

EDLs are referenced—without requiring a configuration commit. Based on their content, EDLs are distinguished into three types: **IP**, **URL**, and **Domain**.

IP-based EDLs can be used in the source and destination address fields of the security policy rules.

These EDLs can include individual IP addresses, subnet addresses (address/mask), or ranges of IP addresses. The list also can include comments and special characters such as *, :, ;, #, or /.

The syntax for each line in the IP-based EDL is as follows:

```
<address> OR <address/mask> OR <start_address-end_address> <comment>
```

Enter every IP address, range, or subnet on a new line. A subnet or an IP address range, such as 198.51.100.0/24 or 203.0.113.10-203.0.113.20, counts as one IP address entry and not multiple IP addresses (in the context of capacity planning). You should add comments where possible. Each comment must be on the same line with the IP address, range, or subnet it relates to.

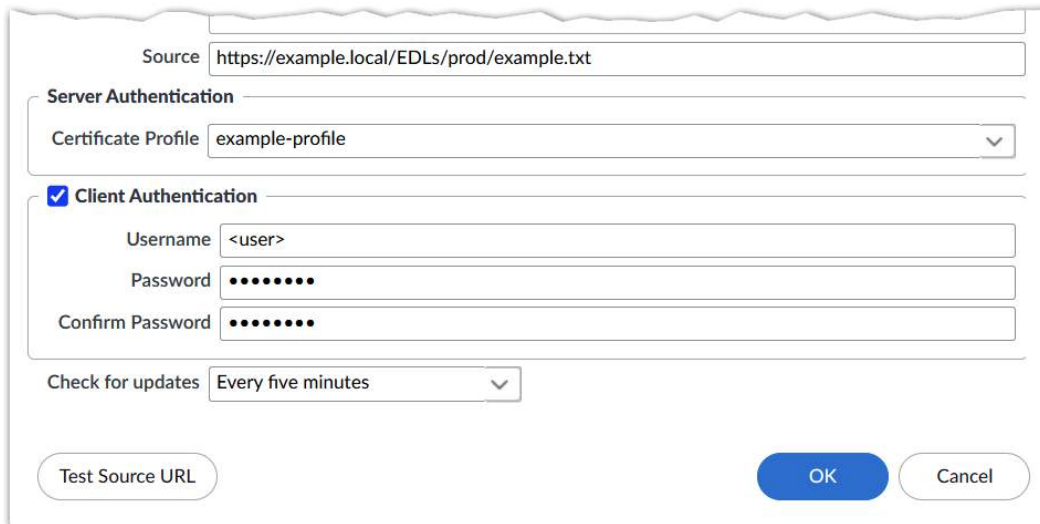
EDLs are defined in **OBJECTS | External Domain Lists**, as shown in the following screenshot:

The screenshot shows the 'External Dynamic Lists' configuration window. The 'Name' field is 'EXAMPLE-IP-EDL'. The 'Type' is 'IP List'. The 'Description' is 'Example description'. The 'Source' is 'http://'. The 'Certificate Profile' is 'None'. The 'Check for updates' is set to 'Every five minutes'. The 'Test Source URL' button is present. The 'Check for updates' dropdown is open, showing options: 'Every five minutes', 'Hourly', 'Daily', 'Weekly', and 'Monthly'. The 'OK' and 'Cancel' buttons are at the bottom right.

Figure 3.10 – Definition of an example EDL

It is best practice to access EDLs over the HTTPS protocol and authenticate the server using a certificate profile. This will allow you to ensure that the matching criteria you use in your security policy rules (IP addresses contained in an EDL) come from a verified secure source and that a malicious **man in the middle** is not potentially serving the firewall with its own content.

Once you have specified a certificate profile to authenticate the server, you can also specify basic client authentication based on the username and password (for example, if you do not want to expose the content of your EDL to anyone but the firewall on your internal network):



Source:

Server Authentication

Certificate Profile:

☒ **Client Authentication**

Username:

Password:

Confirm Password:

Check for updates:

Figure 3.11 – Client authentication option appearing after a certificate profile has been selected

There are four predefined IP EDLs and one URL EDL that you can use in your firewall policy:

- **Tor exit IP addresses**
- **Bulletproof IP addresses**
- **High risk IP addresses**
- **Known malicious IP addresses**
- **Authentication Portal Exclude List**

Each built-in EDL object has a detailed description.

It is best practice to block all connections to and from the addresses in these lists. Predefined EDLs are updated as part of the dynamic updates for antivirus (configured in **Device | Dynamic Updates | Antivirus**).

Palo Alto Networks also provides a free EDL hosting service that supplies IP lists for Microsoft 365, various Azure, AWS, and GCP services, Salesforce, Zoom, GitHub, Okta, Akamai, and so on.

EDL mechanism is very reliable. Once successfully retrieved, EDLs are cached until the next successful connection to the hosting web server. “Successful” means the web server responds with an HTTP 2xx code. Cached content of an EDL will survive the reboot of the whole firewall (or its management plane and data plane), upgrade/downgrade of PAN-OS, and unreachability of the hosting web server (as well as any errors it may generate because of a fault—such as HTTP 4xx or 5xx).

Unfortunately, capacity limitations may prevent you from using EDLs for all possible use cases in a particular network. Regardless of the hardware or virtual firewall model, you can have only 30 custom EDLs with unique sources. The total combined number of IP entries in all configured EDLs is 150,000 in PA-3200, 5200, 5400, and 7000 series, but this number drops to 50,000 for all other platforms.

You can check a firewall’s current capacity by navigating to **OBJECTS | External Dynamic Lists** and clicking on **List Capacities** at the bottom of the screen.



By default, the firewall retrieves the EDLs using its management interface. You can change this behavior in **DEVICE | Setup | Services | Services Features | Service Route Configuration**.

Source and destination port numbers

As discussed in the *Networking 101* and *Packet flow inside the firewall – application identification* sections of *Chapter 1*, source and destination UDP/TCP port numbers are encoded in the **Service** and **Application** fields of a security policy rule.

The **Service** field can contain a particular predefined or custom service, the **Any** keyword, or the **application-default** keyword. The latter means that the port number(s) should be retrieved from the application definition, the **Any** word means that the port number is anything from 0 to 65535, and a predefined/custom service would explicitly list port numbers in its definition.

You can also use custom services to override session timeouts set for applications.



You can specify IP addresses, subnets, and ranges (but not IP wildcard masks or FQDNs) directly in the policy rules without creating the respective objects. This is considered bad practice because addresses referenced in this way are not manageable at scale. They cannot be grouped, reused, and found in the **OBJECTS** menu. However, this technique is helpful for quick ad hoc rule creation when you're in a rush or running some testing in a lab environment.

Source user

You can specify individual users or user groups **source user**. Before specifying user groups, you must configure LDAP integration in **DEVICE | User Identification | Group Mapping Settings**.

When creating a group mapping entry, it's best practice to explicitly specify groups used in your policy on the **Group Include List** tab. This will ensure you do not oversubscribe the firewall and the LDAP server.

You can also utilize user accounts and groups created locally on the firewall in **DEVICE | Local User Database** or **Dynamic User Groups**. The local user database coupled with a captive portal configured in **DEVICE | User Identification | Authentication Portal Settings** and an authentication policy configured in **POLICIES | Authentication** can be a good option for testing your policy in a lab environment that lacks integration with the production enterprise user directory.

Application

All applications the firewall can distinguish are presented in **OBJECTS | Applications**. This screen allows you to find applications by name or attribute(s).

Applications are grouped into categories and subcategories. A subcategory can belong to more than one category. For example, if you click on the **auth-service** subcategory, you will see that applications within it are categorized as either **business-systems** or **saas**:

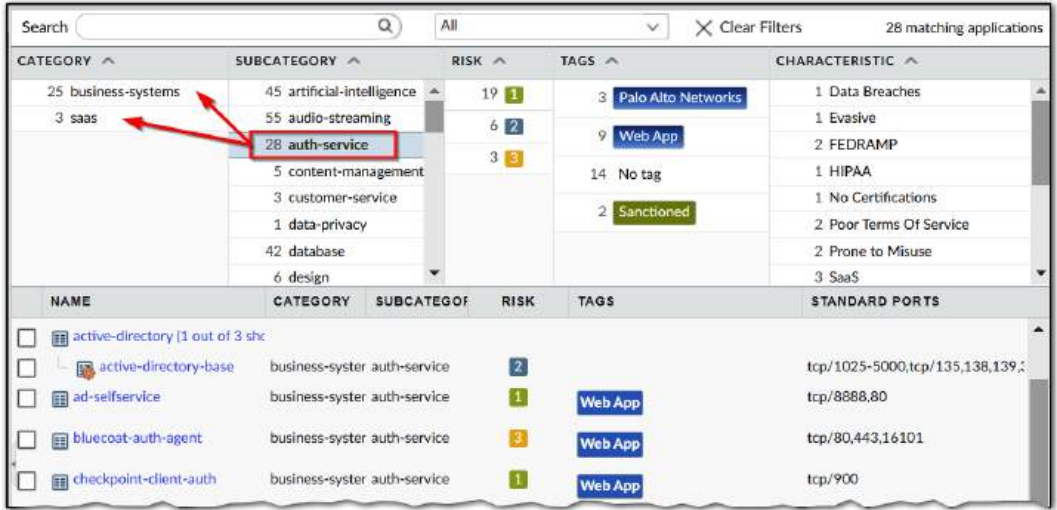


Figure 3.12 – Application browser

As you can see in the screenshot, each application also has **RISK**, **TAGS**, and **CHARACTERISTIC** attributes. If you click on different values in the respective columns, you will see how this further narrows down the selection of applications matching the chosen criteria.

Not all columns have the same logic when you include their values in your selection criteria. In our example, if you click on 2 and 3 in the **RISK** column, this will narrow the selection down to nine applications because the firewall will include all applications that have risk level 2 OR risk level 3:

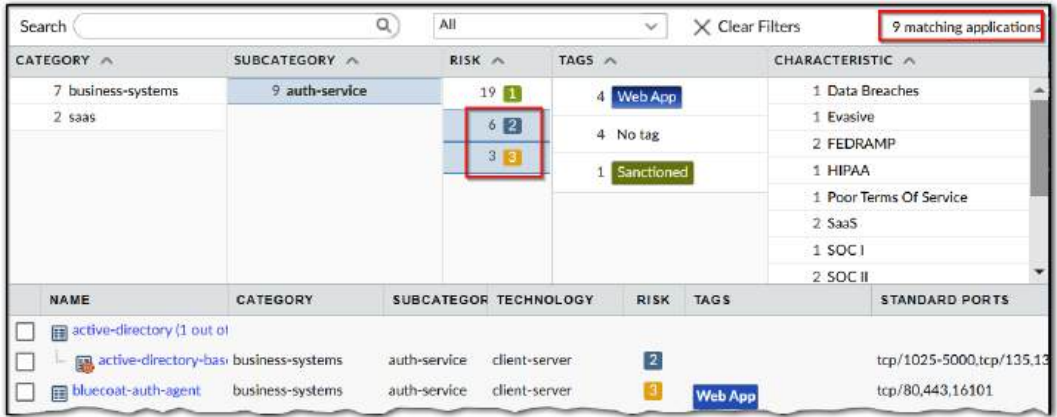


Figure 3.13 – The OR logic applies in the RISK column

In contrast, the **TAGS** column uses AND logic when you select multiple values. For example, if you click on the **Web App** tag, four applications are shown. If you then also select **Sanctioned**, the number of matches drops to just one instead of increasing to five. This happens because applications can have more than one tag, and here the firewall shows only those that include all of the selected tags.

The same AND logic applies to the **CHARACTERISTIC** column. If you select more than one characteristic, only applications that include all of them will appear in the results.

What you do on this screen can also be saved as a custom application filter and consequently used in a security policy rule. Navigate to **OBJECTS | Application Filters**, click **Add**, specify **Name**, and select matching criteria for your filter. When creating an application filter, you can exclude individual signatures by clicking on the **[X]** icon in the **EXCLUDE** column.

Let us review all possible categories, subcategories, characteristics, tags, and risks.

Application categories and subcategories

There are seven different categories available:

- **business-systems**
- **collaboration**
- **media**
- **networking**
- **saas**
- **general-internet**
- **unknown**

The categories are very broad and generic, so their practical use is limited. However, they may be useful as additional selection criteria when used in conjunction with the much more granular classification that subcategories provide.

At the time of writing this book, 44 subcategories are available. The list gradually expands as new use cases emerge. For example, in 2023, the world witnessed an explosive growth of generative AI. Palo Alto Networks introduced a new subcategory, **artificial-intelligence**, to help customers control this controversial class of applications.

A license for the **App-ID Cloud Engine (ACE)** feature enables additional subcategories; however, the feature is not discussed in this book. If you would like to learn more about this

feature, please refer to the description in the PAN-OS Administrator Guide here: <https://docs.paloaltonetworks.com/pan-os/11-1/pan-os-admin/app-id/cloud-based-app-id-service>.

Unfortunately, the firewall UI, PAN-OS Administrator Guide, and built-in help pages do not contain a definition of the App-ID categories, subcategories, and predefined tags. However, you can find their descriptions and examples in a LIVEcommunity article here: <https://docs.paloaltonetworks.com/pan-os/11-1/pan-os-admin/app-id/cloud-based-app-id-service>.

Application characteristics and risks

Each application has a set of characteristics that contribute to its overall risk level. Every standard characteristic is assigned a weight, and the weights are summed. The total is then mapped into risk bands ranging from 1 (lowest risk) to 5 (highest risk).

The following screenshot illustrates the risk calculation:

Application

Name: apple-maps
 Standard Ports: tcp/80,443
 Depends on: web-browsing
 Implicitly Uses: ssl
 Deny Action: drop-reset
 Additional Information: Apple Maps Wikipedia Google Yahoo!

Description: Maps is a mapping service application developed by Apple Inc. for its iOS mobile operating system.

Characteristics

- 3 Evasive: no
- 2 Tunnels Other Applications: no
- 1 Excessive Bandwidth Use: **yes**
- 2 Prone to Misuse: no
- Used by Malware: no
- 1 Widely Used: **yes**
- 4 Capable of File Transfer: no
- 3 Has Known Vulnerabilities: no

Options

- TCP Timeout (seconds): 3600 [Customize...](#)
- TCP Half Closed (seconds): 120 [Customize...](#)
- TCP Time Wait (seconds): 15 [Customize...](#)
- App-ID Enabled: yes

Classification

- Category: general-internet
- Subcategory: internet-utility
- Risk: **1** [Customize...](#)

Tags

- Web App

Risk bands

- 0...3 => 1
- 4...6 => 2
- 7...9 => 3
- 10...13 => 4
- 14+ => 5

Calculation: 1 + 1 = 2 → 0...3 => 1

Figure 3.14 – Application risk calculation

If you disagree with a calculated risk score, you can override it per application. The preceding screenshot shows how the risk value was derived. To override it, click **Customize...** next to the risk value.

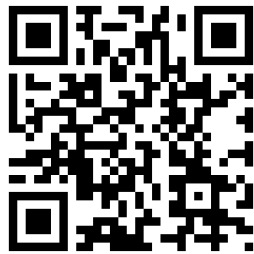
SaaS applications extend this model with additional attributes, as shown below:

SaaS Characteristics	
Certifications:	FEDRAMP, HIPAA, PCI, SOC I, SOC II
Data Breaches:	no
IP Based Restrictions:	yes
Poor Financial Viability:	no
Poor Terms Of Service:	no

Figure 3.15 – SaaS characteristics of a SaaS application (application shown is amazon-polly)

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



These attributes, together with compliance, identity, and security factors maintained by Palo Alto Networks, contribute to the SaaS application's final risk score.

When an application is SaaS and/or belongs to the **artificial-intelligence** subcategory, the risk calculation follows more nuanced algorithms than the simple weighted-band method described earlier.



For a complete description of how Palo Alto Networks calculates SaaS and AI application risks, see the Knowledge Base article:

<https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000C1mQCAS>

Application tags

You may notice that subcategories alone, even when combined with their risk factors and characteristics, are not very flexible in differentiating applications. You may want to add another dimension based on an alternative set of criteria. PAN-OS gives you a mechanism to achieve this via tagging. A tag is a label that you can assign to an application and then use as one of the application selection criteria in application filters.

You can use several predefined application tags. The PAN-OS administrative interface displays all these predefined tags in *blue*.

You *cannot* assign any of these tags to an application as you wish. If you discover a discrepancy or an incorrectly tagged application, you will have to raise this issue with Palo Alto Networks TAC. However, you *can* use any of these blue tags as matching criteria for an application filter.

There is also a special predefined tag named **Sanctioned**. It is *green*; therefore, it *cannot* be used as a matching criterion in an application filter, but you *can* assign it to any application.

This tag implements a very good concept promoted by Palo Alto Networks. The concept suggests breaking down all applications a company uses into three big groups: *sanctioned*, *tolerated*, and *blocked*. Applications that you label with this **Sanctioned** tag are then marked as *sanctioned* in a very informative predefined PDF report that covers SaaS application usage at the network level (**Monitor | PDF Reports | SaaS Application Usage**).

We will discuss the practical use of this tag and other application attributes in the next chapter.

URL category

URL category is the last possible network connection attribute we can use as a matching criterion in a security policy rule. When we specify a URL category in a policy rule, this immediately implies that all applications referenced in the rule must be based either on HTTP or SSL/TLS protocols. The firewall's administrative UI will not tell you that, but the moment you specify a URL category in a rule and commit the change, subsequent policy lookups will ignore all applications in the rule that are not based on HTTP or SSL decoders.

There are predefined URL categories and custom ones. Custom ones can be defined either in **OBJECTS | Custom Objects | URL Category** or be sourced externally in the form of a URL-based EDL (defined in **Objects | External Dynamic Lists**).

Predefined URL categories

Each URL in the Palo Alto Networks URL database can have up to four categories assigned to it:

- Content-based
- Function/behavior-based
- Risk
- Other security-related category

For example, a personal blog about AI can be classified as “personal sites and blogs” and “artificial intelligence” simultaneously. The website will also have a risk level based on its recent activity.

You can find the full list of all standard URL categories with their descriptions at <https://docs.paloaltonetworks.com/advanced-url-filtering/administration/url-filtering-basics/url-categories>.

It is recommended to block the following categories:

- **command-and-control**
- **compromised website**
- **copyright-infringement**
- **dynamic-dns**
- **extremism**
- **grayware**
- **malware**
- **parked**
- **phishing**
- **proxy-avoidance-and-anonymizers**
- **questionable**
- **ransomware**
- **scanning-activity**
- **newly-registered-domains**
- **unknown**



You need to be careful when blocking URLs classified as **unknown**. You must be careful that none of the URLs of your business-critical web applications are classified as such. This is very difficult to ensure, especially for URLs that are behind an “authentication wall” and, therefore, cannot be classified by PAN-DB in advance.

Depending on the nature of your business, the **newly-registered-domains** category may contain many legitimate websites essential for your company. Nevertheless, you must exercise caution when you provision this access—consider using stricter content inspection for this category.

Custom URL categories

There are two types of custom URL categories: **Category Match** and **URL List**.

The **Category Match** type is defined as a combination of two, three, or four predefined categories. A URL must be classified into all specified categories simultaneously to match a custom category of this type. A useful application of this type of custom category is a risk-based treatment of the content/function/behavior-based categories. For example, you can create a custom category defined as a match of **business-and-economy** and **high-risk** to apply a stricter set of security profiles for content inspection and a more detailed logging profile. This will allow you to reduce the number of false positives while remaining risk-centric in your security controls.

The **URL List** type is defined as a list of exact URLs or URL patterns.

You can add entries individually or import them from a text file. Each entry in the file must start on the new line. The basic principles are as follows:

- Do not include the protocol (HTTP/HTTPS).
- Each entry can be up to 255 characters long.
- Entries are *not* case-sensitive.
- Each entry comprises **tokens**—alphanumeric strings—separated by the following symbols: . ; / ? & = +

These symbols are called **token separators**.

- A token can be replaced with a wildcard symbol—either * or ^:
 - ^ will match only a *single* token
 - * will match *one or more* tokens
- If you use a wildcard symbol, there must be no other symbols next to it other than the token separators.

If an entry does not end in an asterisk (*) or a forward slash symbol (/), the firewall will automatically add a trailing forward slash. This is a default firewall behavior starting from PAN-OS 10.2. This can be modified in **DEVICE | Setup | Content-ID | URL Filtering | Append Ending Token**. This addition is invisible when added by the firewall, so it is recommended that you always add the ending forward slash manually.

If this feature is disabled (or you are running an earlier version of PAN-OS) and the trailing slash is not present, the firewall will add an implicit trailing token separator and any number of tokens afterward (or no tokens at all) up to the maximum allowed length of 255 symbols.

URL list entry	Append Ending Token feature enabled	Effective patterns the URL list entry translates to	Examples of matching URLs	Examples of not-matching URLs
example.com	no	example.com example.com.* example.com?* example.com=* example.com+* example.com/*	example.com example.com/ example.com/sub example.com.uk example.com.uk/sub example.com?q=123	www.example.com example.company
example.com	yes	example.com/	example.com/ example.com/sub1	example.com www.example.com example.com.uk example.company

Table 3.1 – How the ending token affects URL matching

You may find it confusing that the table claims that the *example.com* URL does not match a URL category with the *example.com* entry while the **Append Ending Token** feature is enabled. This result was verified with the test `custom-url` command and is correct. Here is the explanation.

If you remember from *Chapter 1*, a regular HTTP 1.1 request looks as follows:

```
GET / HTTP/1.1
```

```
Host: example.com
```



Therefore, the effective full URL for matching purposes will be constructed as the value of the host header concatenated with the path, as follows: `example.com + /` = `example.com/`.

This means that in real life, a valid HTTP request will always result in a URL with a trailing forward slash *as a minimum* because the path after the GET method cannot be empty—it is either the forward slash or the forward slash followed by other symbols. For this reason, it is not really a problem. If you enter *example.com* in a web browser and press *Enter*, the effective URL will also contain the ending forward slash and match the URL category with the *example.com* entry and the appended forward slash.

This concludes the list of connection attributes we can use to match network connections to a security policy rule.

Summary

In this chapter, you explored the full range of connection-matching capabilities in PAN-OS security policy rules. You learned how the firewall uses various attributes—such as zones, addresses, users, applications, and URL categories—to determine whether a session matches a rule. We also covered advanced constructs, such as dynamic address groups, EDLs, and custom regions, highlighting their flexibility and potential pitfalls. Together, these features form the logic that defines which traffic is subject to further inspection or control by the matching policy rule.

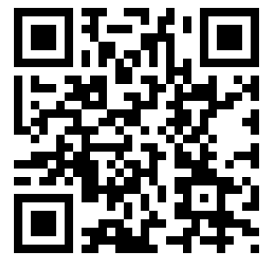
Understanding how connections are matched is essential before you can define how matched traffic should be handled. In the next chapter, we'll shift focus from matching to processing. You'll examine the **Actions** tab of the security rule and the security profiles that apply content inspection and threat prevention. The chapter will also cover critical functions such as logging, alerting, blocking threats, and custom response pages.

With this, you'll be ready to design policy rules that not only identify relevant traffic but also secure it effectively.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



4

PAN-OS Security Policy Features: Connection Processing

In the previous chapter, you examined how PAN-OS determines whether a session matches a security policy rule by evaluating attributes such as IP addresses, zones, users, applications, and URL categories. That matching logic forms the basis for any security decision the firewall makes. Now that you understand how connections are identified, it's time to explore what happens next.

This chapter focuses on connection processing—specifically, the configuration options available on the **Actions** tab of a security rule. Here, you'll define what the firewall should do with matched connections: allow or block them, inspect them for threats, log them, or apply additional controls. You'll also learn how to configure and apply security profiles such as antivirus, anti-spyware, URL filtering, and more.

By the end of this chapter, you'll be able to apply layered security controls to any session matched by a rule. You'll understand how each profile contributes to threat prevention and how these settings support the design of a resilient, risk-aware security policy.

Here is the list of features covered in this chapter:

- Action setting
- Log settings
- Profile setting
- Auxiliary functions
- Response pages

Technical requirements

If you have access to a Palo Alto Networks firewall while reading this chapter, it can really help you visualize, understand, and remember all the features we're discussing.

Processing the matching connections

Once a session matches a rule based on a combination of the matching criteria described in the previous section, other firewall features come into play. They collectively hide under the name of **Content-ID** and are intended to process matched connections to identify and mitigate security threats.

Each security rule has an **Actions** tab that allows you to do the following:

- Specify what to do with the matched connection in principle. A connection can be allowed or denied, and the latter can be done in a few different ways.
- Configure logging options—whether to log a connection after it ends, immediately after it starts, or both. Log records can also trigger tagging actions and be forwarded elsewhere for further processing.
- Define connection security processing by *Antivirus*, *Anti-Spyware*, *URL filtering*, *File Blocking*, *Data Filtering*, *WildFire*, and *Vulnerability Protection* subsystems.
- Set a schedule for when the rule should be active, enable quality of service marking, and disable server response inspection.

Let's take a closer look at each of these configuration elements.

Action setting

The drop-down **Action** list presents a selection of actions that can be applied to matched connections. You can either *allow* or *block* a connection. While the **Allow** action is straightforward, the latter can be executed in a few different ways:

- **Deny** means that the actual method of blocking the connection will be retrieved either from the application definition *or* from the security profile associated with the decoder that processes the identified application *or* is hardcoded in the decoder. For example, if you run the following command, you will see that, say, for web-browsing, the deny action is drop-reset, but for dns-base, there is no deny action defined in the application definition:

```
show predefined application <application name> | match deny-action
```

The action for this application can differ depending on the categories of identified threats. This is configured in the **DNS Security** settings of the anti-spyware profile assigned to the security rule.

drop-reset is equivalent to the **Reset client** action described later in this list.

- **Drop** means that the firewall will silently drop connection packets. However, if you opt to send an **ICMP Unreachable** message, a TCP reset will also be sent to the client.
- **Reset client** means that the firewall will send a TCP reset message to the client.
- **Reset server** means that the firewall will send a TCP reset message to the server.
- **Reset both client and server** will reset both sides of the connection.



For every blocking action, you can optionally choose to send an **ICMP Unreachable** message to the client to allow it to terminate the connection on the application side quickly and gracefully. This ICMP option is potentially very taxing on firewall resources, so it is rate-limited. You can see the default value and change it in **Device | Setup | Session | Session Settings | ICMP Unreachable Packet Rate (per sec)**.

Log settings

There are three options for logging matched connections:

- **Log at Session Start**
- **Log at Session End**
- **Do not log**

The golden rule of information security is to log everything. Palo Alto Networks' best practice suggests logging at the session end. Logging at the session start is advisable only for long-lived connections. For example, you may want to see BGP and MS SQL connections appear in the logs straight away—not days or months later after they naturally finish. Bear in mind the *application shift* concept we discussed in *Chapter 1*, if you choose to log a session at its inception, the firewall will likely not know the application name yet; therefore, you'll likely see **incomplete** as the logged application name at this stage.

Log forwarding profile

Besides logging the session locally, the firewall can forward log records to external consumers such as Syslog, HTTP, SNMP, email servers, and Cortex Data Lake. This is configured in **Log Forwarding Profiles (DEVICE | Log Forwarding)**. Once you have created a profile in this section, it becomes available in the **Log Forwarding** drop-down list in the **Actions** tab of security policy rules.

Each log-forwarding profile can contain up to 64 match lists. These lists define what needs to be forwarded and where. Let's explore them.

Log forwarding profile match list

The best thing about log forwarding is that you can forward log records selectively – only those that match the selection criteria set in **Log Forwarding Profile Match List**. For example, here is how you would forward all traffic log entries for blocked new App-ID signatures to a Syslog server:

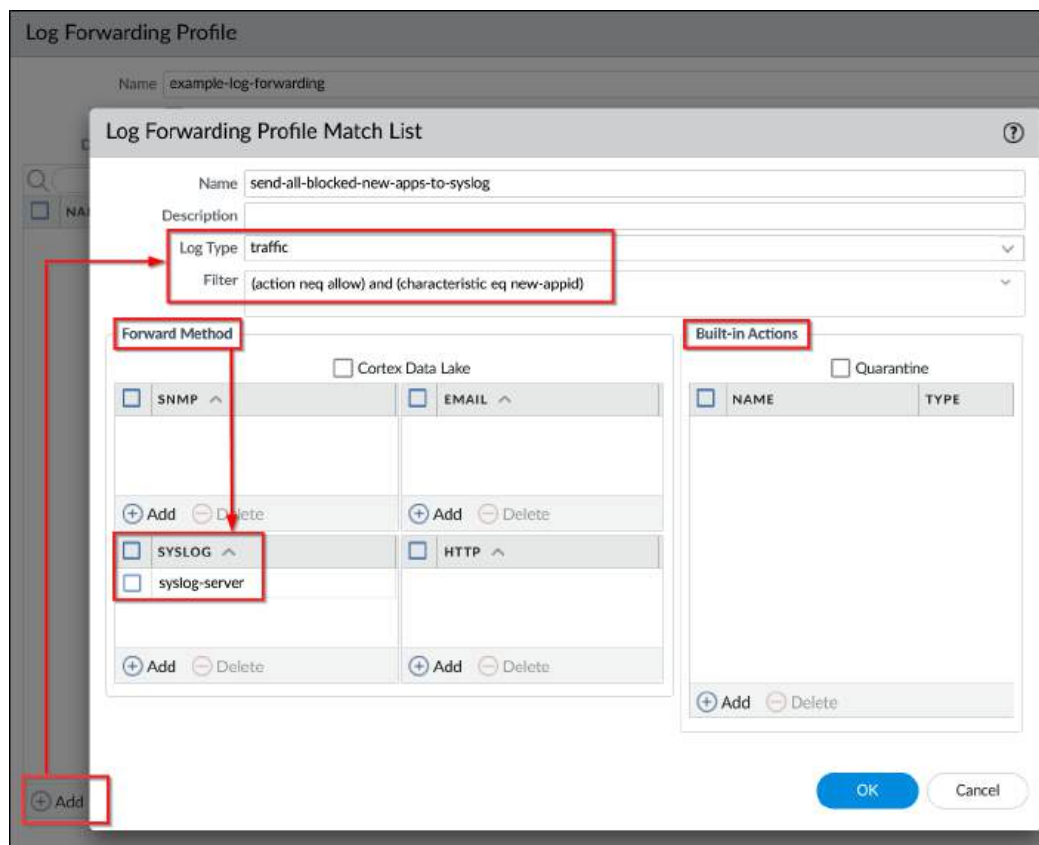


Figure 4.1 – Selective log forwarding

As the screenshot shows, you can specify **Log Type**, **Filter**, **Forward Method**, and **Built-in Actions** for each match list you create. The **Built-in Actions** field is empty in this example. We'll discuss the possibilities of this option in a minute.

Not only can you forward selectively but the format in which log records are forwarded is highly customizable. You can choose to forward either a limited selection of available log fields or all of them. Moreover, you can forward not the actual log record but a custom hardcoded message or an SNMP trap. This is invaluable for automation purposes.

For example, as shown in the following screenshot, you can create an HTTP server profile (**DEVICE | Server Profiles | HTTP**) that sends a REST API call with custom HTTP headers and a payload to turn a smart light on. Then, you can reference this HTTP profile in a log forwarding profile match list that looks for a specific security event.

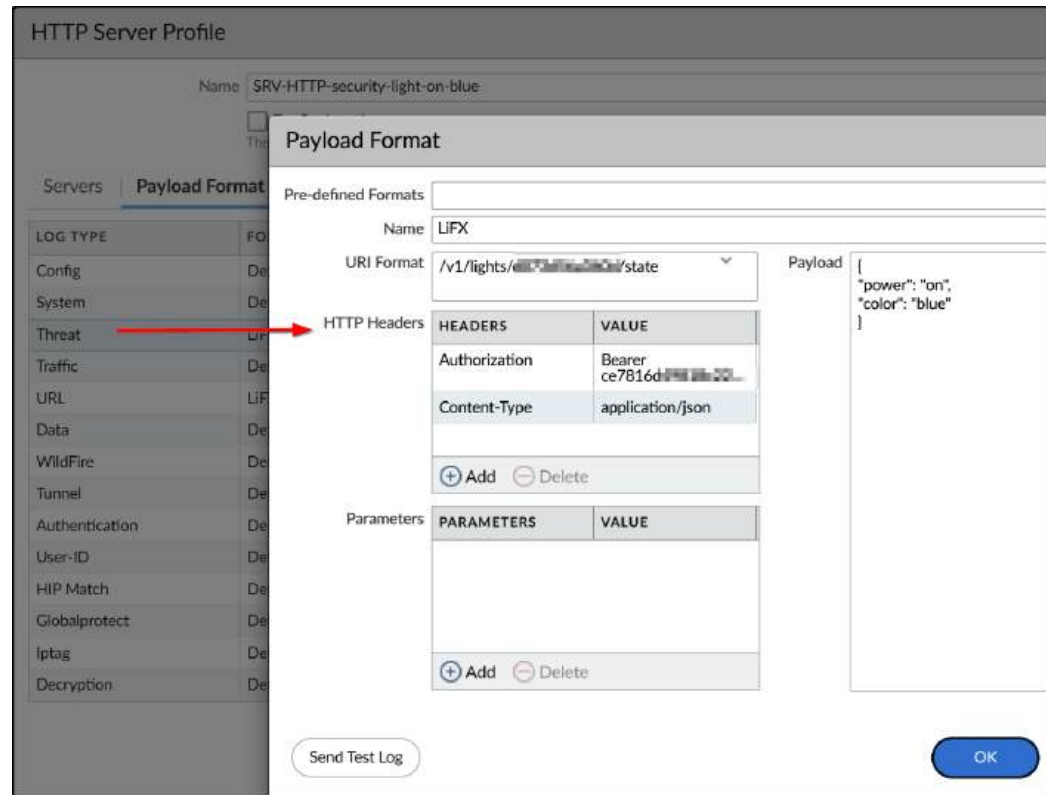


Figure 4.2 – Customization of the payload format in the HTTP server profile

Thus, you could easily add color to your security operations center, turning lights on and off based on security events and associated firewall actions.

If you don't feel creative enough, you can simply choose one of the predefined payload formats, such as a ServiceNow incident or VMware NSX tagging.

Built-in actions

Another powerful feature of log forwarding profiles is **Built-in Actions**, which are configured individually for each log forwarding profile match list. There are three actions available:

- Quarantine a device (works only for devices identified by Global Protect or IoT features and is therefore not covered in this book)
- Add a tag
- Remove a tag

The latter two functions can assign custom tags to one of the following attributes of the event being logged:

- Destination address
- Source address
- User
- **X-Forwarded-For (XFF)** address

This tagging functionality is designed to be used in conjunction with *dynamic address groups* and *dynamic user groups*, which we covered earlier.

For example, you can tag users whose network connections meet certain criteria and subsequently invoke a multi-factor authentication policy for them (by referencing the dynamic user group based on the chosen tag in the authentication policy).

Another useful scenario is to tag the IP addresses of websites that the firewall couldn't decrypt for some reason (you can create log forwarding profile match lists based on session end reasons). Then, you can exclude a dynamic address group with these IP addresses from the decryption policy, allowing your policy to become self-healing and error-resistant.

Tagging is a function of the *User-ID* subsystem. When you specify this built-in action type, you must specify the target (refer to the following screenshot) – an instance of the *User-ID* service running either locally on the firewall itself, on Panorama, which manages the firewall, or on a Windows host running User-ID agent software.

The screenshot shows a configuration window titled "Action". It contains several fields and a "Tagging" section. The "Name" field is set to "tag-addr-failed-to-be-decryptd". The "Tagging" section includes a "Target" dropdown set to "Destination Address", an "Action" section with "Add Tag" selected, a "Registration" dropdown set to "Local User-ID", a "Timeout (min)" field set to "10080", and a "Tags" dropdown showing "TLS decryption exception" with a red error bar. "OK" and "Cancel" buttons are at the bottom right.

Figure 4.3 – Tagging action configuration

As you can see in the preceding screenshot, you can also set a timeout for the tags you assign. The value can range from 0 (never expires) to 43200 (30 days).

Tagged addresses can be redistributed between firewalls the same way the User-ID information is redistributed. Depending on the use case, traffic pattern, and volume, the timeout value can negatively impact your security policy.

For example, you may have a central office equipped with a mid-range PA-3260 and a small regional office running an entry-level PA-820. The former has a capacity of 200,000 (two hundred thousand) tagged addresses while the latter has a capacity of only 1,000 (you have read it correctly—one thousand) addresses. This is a striking 200-times difference!

This means that if you start tagging, say, destination IP addresses that failed to be decrypted and configure tag redistribution between the offices, you may very quickly max out the capacity of the regional firewall, and it will stop all tagging completely (while the central office will keep tagging just fine).

In such a use case, you must experimentally work out a timeout value low enough to keep the total number of tagged addresses across both offices always below 1,000. Alternatively, you may choose not to redistribute tags. Unfortunately, there is no way to reserve capacity for specific tags—they all share the same cap.

Depending on the use case, the **DEVICE | Data Redistribution | Include/Exclude Networks** feature might also help control the redistribution of the mappings.

You can find the capacity of your firewalls by running the following command:

```
> show system state | match cfg.general.max-registered-ip-address  
cfg.general.max-registered-ip-address: 0x30d40
```

The hexadecimal value of 0x30d40 in this example is equal to decimal 200,000.

Now, let us review the most important section on the **Actions** tab – **Profile Setting**.

Profile setting

This section of the policy rule allows us to specify security profiles. Each security profile essentially defines the configuration of the relevant subsystem of the firewall in the context of connections matching the rule the profile is specified in. You can consider the profiles as mini sub-policies invoked from the security policy rule for each network connection matching the rule.

You can assign up to seven profiles:

- **Antivirus**
- **Vulnerability Protection**
- **Anti-Spyware**
- **URL Filtering**
- **File Blocking**
- **Data Filtering**
- **WildFire Analysis**

Profiles can be specified individually or as a group. The benefit of using a profile group is the ability to simultaneously update profile settings for all security rules the group is assigned to – you can replace a profile in the profile group, which will effectively apply to all rules the profile group is already assigned to. However, this benefit becomes less obvious if you use automation to create your policy.

Despite their obvious convenience, profile groups make it somewhat difficult to understand what profiles are assigned to rules as you glance through a long policy. If you compare **rule1** and **rule2** in the following screenshot, you will see that with the latter rule, you can simply hover your mouse over a particular icon representing a profile, and a small hint will appear saying the profile type and name. You do not even need this hovering to understand the types of profiles assigned to the rule – you can tell this by just looking at the profile icons:

NAME	PROFILE	ACTION
rule1		 Allow
rule2	  File Blocking Profile: basic file blocking	 Allow

Figure 4.4 – Representation of profiles and profile groups in the security policy

rule1, on the contrary, just gives you one icon, and you cannot tell what is hidden behind it without clicking and opening another window.

You can manage profiles and profile groups in **OBJECTS | Security Profiles**, and **OBJECTS | Security Profile Groups**, respectively.

Antivirus profile

The **Antivirus** profile is responsible for configuring the firewall's actions when it encounters a payload matching an antivirus signature, a WildFire signature, or when a WildFire inline machine learning model triggers.

There are three tabs in the profile. Let's review each of them.

Action

On the **Action** tab, you define signature actions for each of the protocols (decoders that are integrated with the antivirus engine) and each of the detection engines. You can also make firewall capture packets that contain matched signatures.

This tab also allows you to create exceptions at the application signature level:

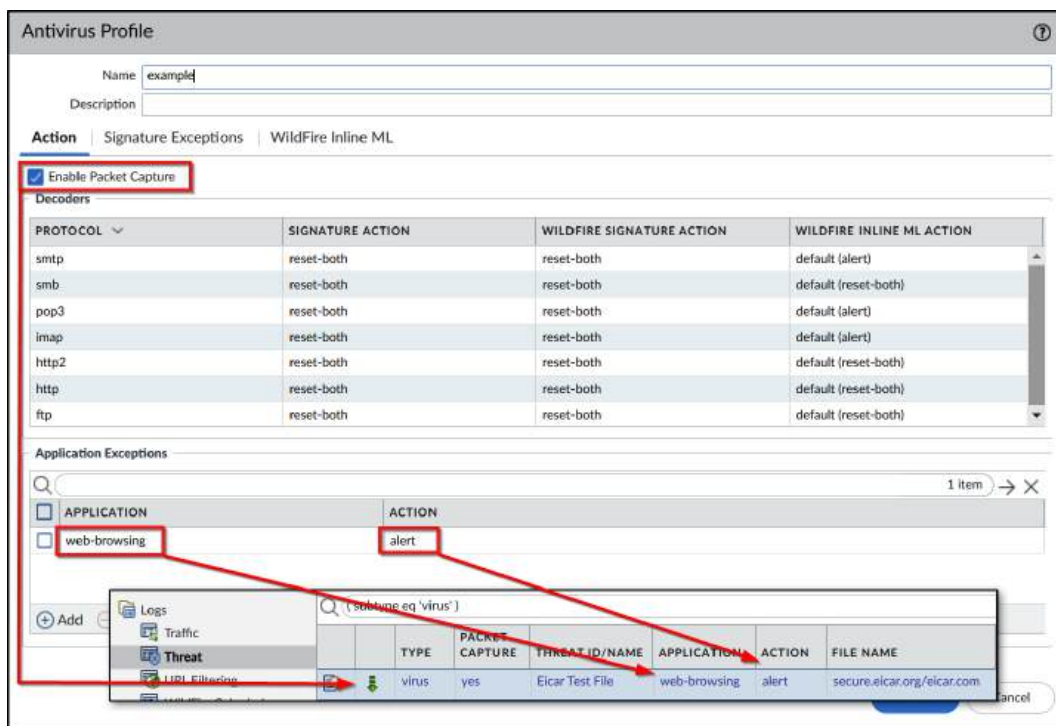


Figure 4.5 – Antivirus profile – application exception

This screenshot illustrates how an exception defined for an application is reflected in the **Threat** log. For example, if you add **web-browsing** with the action set to **alert** in the **Application Exceptions** section, the firewall won't block any viruses in generic web traffic hitting a rule with the given **Antivirus** profile. The little green arrow in the **Threat** log allows you to download the packet capture the signature triggered on.

Signature Exceptions

This tab allows you to create exceptions for individual signatures. If you discover a false positive, you can grab the signature ID in the **Threat** log and specify it on this tab of the corresponding **Antivirus** profile:

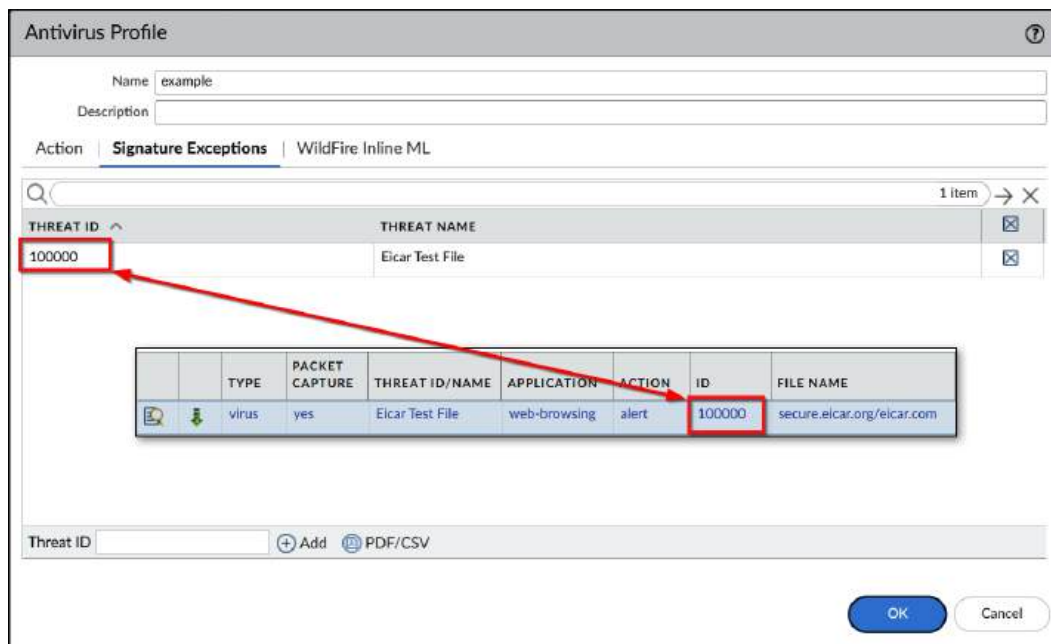


Figure 4.6 – Antivirus profile – signature exception

This screenshot illustrates the connection between an entry in the Threat log and the list of signature exceptions in the Antivirus profile.

WildFire Inline ML

This tab allows you to specify actions for inline **machine learning (ML)** models capable of detecting certain threats in Windows and Linux executables, PowerShell scripts, MS Office, and Linux Shell scripts. If required, you can make exceptions for known benign files. This is done based on **partial hash** and filename. The former can be found in the Threat log when the file is detected by the firewall.

It is worth mentioning that the reason why protocol actions are differentiated between antivirus and WildFire signatures is the potential difference in fidelity of these signatures. All WildFire signatures are autogenerated within a few minutes after a new malware sample is detected in the WildFire sandboxing environment. After a signature has been released and therefore used by all WildFire customers for approximately 24 hours, it becomes an antivirus signature. The antivirus signature database also contains human-made signatures created by Palo Alto Networks specialists. Thus, in environments with a low tolerance for false positives, you may opt for a more conservative approach to WildFire signatures.



The default action for the SMTP decoder is cautiously set to **alert**. However, the firewall is very smart when it comes to interrupting an SMTP connection. It would inject a valid SMTP response to the sending **Mail Transfer Agent (MTA)** before resetting the connection.

It is advisable to thoroughly test this feature with your particular email gateways before production use.

Anti-Spyware profile

The **Anti-Spyware** profile allows you to configure protection against spyware, adware, browser hijacking, data theft, keyloggers, and backdoors. It can also detect and block botnets, net worms, command-and-control communications, crypto miners, downloaders, phishing kits, hacking tools, web shells, and post-exploitation activities. This profile also manages the **DNS Security** policy (when licensed).

There are five tabs in the profile. Let's explore each of them.

Signature Policies

This tab contains a set of ordered rules that define actions applied when a spyware signature of a certain type matches the connection's payload. Rules are evaluated from the top to the bottom, and policy processing stops after the first match—similar to how the main security policy is evaluated. Therefore, more specific rules should be placed before the more generic rules.

Each rule includes the rule name, threat name, threat category, threat severity, required action, and packet capture.

The threat name must either be an exact match to a threat signature name or a substring of one. Threat names are case-insensitive. The reserved keyword **any** would match all known signatures.

Once you have created your policy, navigate to the **Signature Exceptions** tab, click **Show all signatures**, **PDF/CSV**, and **Export**. The exported spreadsheet will contain all signatures and, among others, the **POLICY** column. This column will indicate what policy rule within the profile each signature matches. Disregard **ACTION** field in the spreadsheet – it shows only the default value in the signature definition. The real action will be the one defined for the profile policy rule.

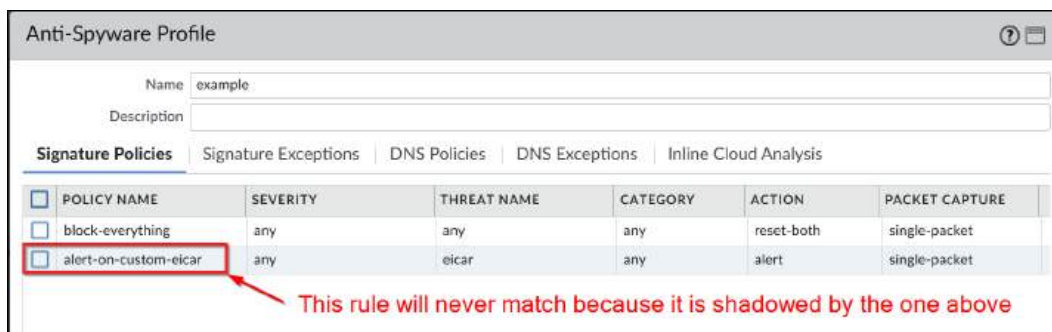


Figure 4.7– Anti-Spyware profile rule shadowing example

This screenshot shows that the bottom rule will never be reached because it is “shadowed” by a more generic rule above.



When you create your signature policy, you may be tempted to rely solely on **CATEGORY** and **SEVERITY** attributes. Do not do this without testing – after creating a policy, always export all signatures and glance through the matching rule names to see whether the result meets your expectations.

For example, you may assume that all command-and-control traffic is covered by the **command-and-control category**. However, if you set the category to any and enter command and control (with spaces instead of dashes) in the **THREAT NAME** field, you will discover that four times more signatures now match. This is because many threats of this nature belong to other categories such as spyware, botnet, backdoor, hacktool, and so on, and the category of a threat is defined by its primary function (while it may have many).

If you choose to take a packet capture in a policy rule, bear in mind that the **extended-capture** option will capture only five packets by default. However, you can increase this number to 50 in **DEVICE | Setup | Content-ID | Content-ID Settings | Extended Packet Capture Length (packets)**.

Signature Exceptions

This tab allows you to create exceptions from the policy rules created on the **Signature Policies** tab.

To create an exception, click **Show all signatures**, enter a string to search for, and press Enter. The search will be performed in all available fields; thus, you can enter threat ID, name, category, severity, or action. You can narrow the search scope by hovering your mouse over a value in the **THREAT NAME**, **IP ADDRESS EXEMPTIONS**, or **ACTION** field, and clicking on the small arrow and then on **Filter**.

Anti-Spyware Profile

Nameexample

Description

Signature Policies

Signature Exceptions

DNS Policies

DNS Exceptions

Inline Cloud Analysis

Q ('ip address exemptions' contains 'x.x.x.x')

ENABLE	ID	THREAT NAME	IP ADDRESS EXEMPTIONS	POLICY	CATEGORY
☒	15001	custom-eicar-spyware	1	everything	
☐	10001	Autorun User-Agent Traffic		block-everything	spyware
☐	10002	Suspicious User-Agent Traffic		block-everything	spyware

Figure 4.8 – Anti-Spyware profile – signature exceptions

Once you have identified the required signature, tick the **ENABLE** box against it. Then, click on the value in the **ACTION** column and modify it as required – this will override the action set in the matching policy rule (shown in the **POLICY** column).

You can narrow down the scope of the exception by clicking on the **IP ADDRESS EXEMPTIONS** field and entering either the source or destination IP address(es) that the exception must apply to. You can add up to 100 addresses. Subnets or address ranges are not supported here, unfortunately.

DNS Policies

This tab defines the configuration of the DNS-related security features.

The **DNS Policies** section on this tab contains up to three entries:

- **External Dynamic Lists:** This is optional; it appears only if you create a custom domain External Dynamic List (EDL) in **OBJECTS | External Dynamic Lists**.

- **Palo Alto Networks Content:** This contains a list of approximately ~1,000 known malicious FQDNs populated via **Dynamic Content Updates** as part of the **Advanced Threat Prevention** subscription.
- **DNS Security:** This subsection contains a list of categories and respective actions that the DNS Security cloud would use to classify FQDNs submitted by the firewall in real time. Possible actions are default (defined by Palo Alto Networks), **allow**, **block**, and **sinkhole**.
- The latter's idea is to account for use cases when there is no firewall between a client and its DNS server, but the firewall is deployed between the DNS server and the internet. From a firewall perspective, all DNS queries will be egressing from the same IP address – the address of the internal DNS server. Therefore, to identify the client that attempts to resolve a malicious FQDN, we must direct it to a preconfigured known address and intercept this connection on the firewall. This is exactly what the **sinkhole** action does:
 1. A client attempts to resolve a malicious FQDN via an internal DNS server.
 2. The firewall detects this query when it comes from the server and injects a pre-configured address into the response from an upstream DNS server to the internal DNS server.
 3. The internal DNS server forwards this response to the client.
 4. The client attempts to connect to the address from the response (instead of the real IP corresponding to the malicious FQDN).
 5. The firewall intercepts this connection and thus can identify the address of the client.

The following screenshot highlights two settings on the **DNS Policies** tab: a custom EDL and the sinkhole address.

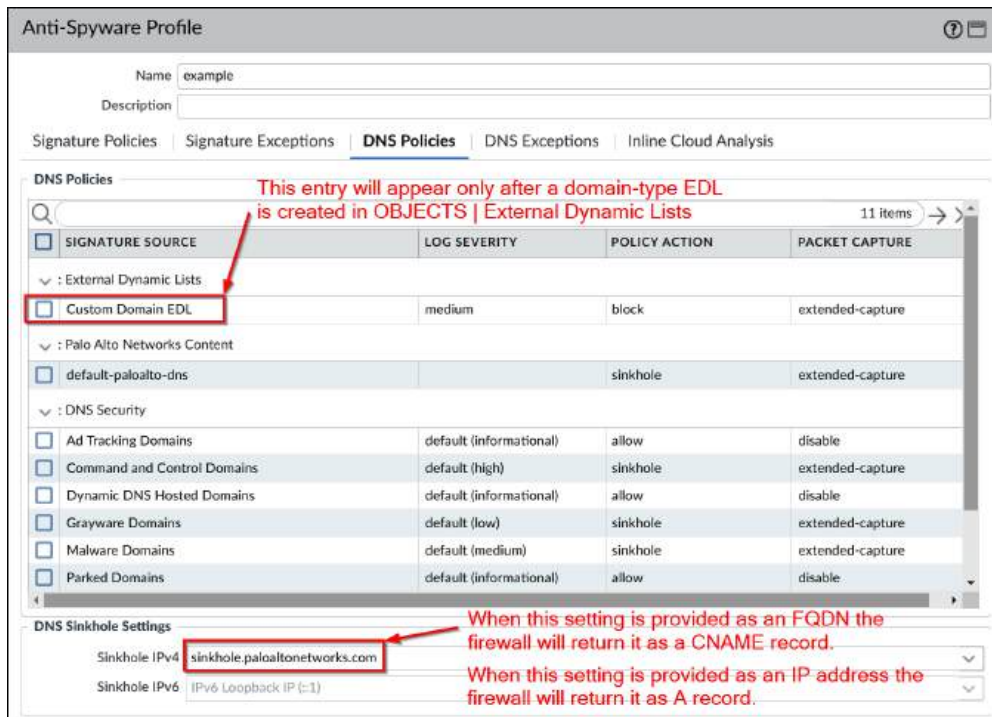


Figure 4.9 – Anti-Spyware profile – DNS Policies

The **DNS Sinkhole Settings** section defines IPv4 and IPv6 addresses that the firewall injects into the upstream DNS server response as part of the sinkhole action.

As the preceding screenshot explains, there is a subtle difference in firewall response between the IP address and FQDN used as the sinkhole target. When you specify the sinkhole as FQDN, the firewall will respond as follows:

```
$ nslookup test-dga.testpanw.com 208.67.222.222
Server:      208.67.222.222
Address:     208.67.222.222#53
Non-authoritative answer:
test-dga.testpanw.com canonical name = sinkhole.paloaltonetworks.com.
```

However, when you specify a raw IP address, the response will be as follows:

```
$ nslookup test-dga.testpanw.com 208.67.222.222
Server:      208.67.222.222
Address:     208.67.222.222#53
Non-authoritative answer:
Name:        test-dga.testpanw.com
Address: 72.5.65.111
Name:        test-dga.testpanw.com
Address: ::1
```

 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.



 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



This difference may be important for some clients, as they may expect to receive an A record only and may not be willing to resolve the provided CNAME to A record and then to IP address.

DNS Exceptions

This tab allows you to provide exceptions for domains, FQDNs, and individual DNS Security signatures. Unfortunately, you cannot browse DNS signatures like you can regular spyware signatures—you have to observe them in the **Threat** log first.

Inline Cloud Analysis

This tab allows you to configure a cloud service capable of real-time detection and blocking advanced **Command and Control (C2)** threats. By default, all detection engines are configured with the alert action. You can specify custom URL categories, URL EDLs, IP EDLs, individual IP addresses, subnets, and ranges of addresses to be exempt from the analysis.

This concludes the overview of all tabs of the **Anti-Spyware** security profile. However, before we explore the next profile type, let's review one important aspect of firewall operation.

DNS Security algorithm

As you can see, the Anti-Spyware profile has an overwhelming number of configuration options, especially those related to DNS Security. You may feel lost trying to understand the sequence of actions the firewall performs when it processes DNS queries. Let me spell out the algorithm here so that you can understand how configuration elements interrelate with each other:

1. A DNS query traverses the firewall, and it makes a copy of the query for analysis.
2. The query is looked up in the custom domain EDL (**DNS Policies | External Dynamic Lists**). If a match is found, the defined action is immediately taken unless this action is set to **allow**. The latter won't take priority over the cloud verdict – processing continues to the next step if the EDL action is set to **allow**.

Thus, to use the EDL method for exceptions, you must set the action to **alert**. In this case, the processing will stop, and the query will be considered benign.

3. The query is looked up in the list of DNS exceptions. If a match is found, no further checks are performed – the query is considered benign.
4. The query is looked up in the malicious FQDN list referenced in **DNS Policies | Palo Alto Networks Content | default-paloalto-dns**. If a match is found, the specified action is taken. The query is considered malicious.
5. The query is looked up in the DNS verdict cache kept locally on the firewall. If a match is found, it is cross-referenced with the categories listed in the **DNS Policies | DNS Security section**, and the specified action is taken.

6. If the query is missing from the cache, the firewall starts a 100 ms timer (this value can be changed in **DEVICE | Setup | Content-ID | Realtime Signature Lookup | DNS Signature Lookup Timeout (ms)**) and sends the query to the DNS Security cloud (bear in mind that, in the meantime, the original DNS query has already been sent to the upstream DNS server by this point).
7. The firewall waits for a response from the DNS Security cloud until the timer expires. If a response from the upstream DNS server arrives earlier than the response from the DNS Security cloud and before the timer expiration, then this response is discarded, and the query (of which the firewall has a copy, as you remember) is sent to the upstream DNS server again. This is done to preserve firewall resources.
8. If the timer expires but the firewall still has not received a response from the cloud, the next response of the upstream DNS server is relayed to the client.
9. If the firewall receives a verdict from the DNS Security cloud before the timer expires (*which happens most of the time*), the verdict is cross-referenced with the categories listed in the **DNS Policies | DNS Security** section, and the specified action is taken.

You may also wonder how common DNS threats and techniques map onto the categories listed in the **DNS Security** section and how you could test your policy. Here is a brief breakdown of spyware profile DNS Security policy rules, relevant DNS threats and techniques, and FQDNs you can use to test your policy:

DNS security policy	Technique	Test FQDNs
Ad tracking domains	Ad tracking	test-cname-cloaking.testpanw.com
	CNAME cloaking	test-adtracking.testpanw.com
C2 domains	C2	test-c2.testpanw.com
	DNS tunneling	test-dnstun.testpanw.com
	Domain generation algorithms (DGAs)	test-dns-infiltration.testpanw.com
	NoneXistent Name Server (NSNS) attacks	test-dga.testpanw.com
	DNS rebinding	test-nxns.testpanw.com
		test-dns-rebinding.testpanw.com

DNS security policy	Technique	Test FQDNs
Grayware Domains	Grayware	test-grayware.testpanw.com
	Fast flux	test-fastflux.testpanw.com
	Malicious NRDs	test-malicious-nrd.testpanw.com
	Dangling	test-dangling-domain.testpanw.com
	Wildcard abuse	test-wildcard-abuse.testpanw.com
	Strategical aging	test-strategically-aged.testpanw.com
	Stockpile	test-stockpile-domain.testpanw.com
	Cybersquatting	test-squatting.testpanw.com
	Subdomain reputation	test-subdomain-reputation.testpanw.com
Malware Domains	Malware	test-malware.testpanw.com
	Ransomware	test-ransomware.testpanw.com
	Compromised DNS	test-compromised-dns.testpanw.com
Newly Registered Domains (NRDs)	NRDs	test-nrd.testpanw.com
	Dynamic DNS	test-ddns.testpanw.com
Parked Domains	Parked domains	test-parked.testpanw.com
Phishing Domains	Phishing domains	test-phishing.testpanw.com
Proxy Avoidance and Anonymizers	Proxy and anonymization	test-proxy.testpanw.com

Table 4.1 – Mapping between DNS Security policy rules, DNS attack techniques, and corresponding test FQDNs



Before implementing firewall DNS Security controls, make sure that your DNS server uses source port randomization – each request to the upstream DNS server *must* originate from a new source port number, thus forming a new session. When the firewall applies the **block** or **sinkhole** action to a malicious DNS query, it transitions the corresponding DNS session into the **discard** state for 60 seconds. Thus, all subsequent packets (other, potentially benign, DNS queries) from the same port number will match the session in the discard state and will be silently dropped by the firewall.

Now, let's move on to the next profile type: Vulnerability Protection. It is less complicated than Anti-Spyware but also extremely important for network security.

Vulnerability Protection profile

As its name suggests, the **Vulnerability Protection** profile is intended to detect and stop malicious attempts to exploit system flaws and vulnerabilities or gain unauthorized access to systems.

There are two tabs in the profile. Let's explore each of them.

Rules

This tab is very similar to the **Signature Policies** tab in the **Anti-Spyware** profile. It consists of ordered rules. Each rule consists of a rule name and matching criteria as follows:

- **THREAT NAME:** Similar to anti-spyware signature policies, you can specify either the keyword **any** or a substring to search for in the vulnerability signature names.
- **CATEGORY:** Here, you specify the keyword **any** or one of the available categories: **app-id-change**, **brute-force**, **code-execution**, **code-obfuscation**, **command-execution**, **dos**, **exploit-kit**, **info-leak**, **insecure-credentials**, **overflow**, **phishing**, **protocol-anomaly**, **scan**, or **sql-injection**.
- **VENDOR ID:** Here, you can specify the vendor IDs for the vulnerabilities of interest. Not all vendors release public IDs for their vulnerabilities, and, unfortunately, this field completely lost its appeal in the context of Microsoft. The Redmond vendor stopped publishing security advisories with distinguishing names such as **ms13-009** that Palo Alto Networks used in the **VENDOR ID** field of the corresponding vulnerability signatures.
- **CVE:** CVE stands for **Common Vulnerabilities and Exposures**, a standard method for referencing publicly known vulnerabilities and exposures. Most vendors use it. Values you enter in this field are considered substrings that the firewall will try to find in the CVE of all signatures it has. Therefore, you can set this field to **CVE-2024-**, for example, to find all vulnerabilities with CVE identifiers assigned in 2024.
- **HOST TYPE:** This attribute defines the type of host to which the vulnerability signatures relate. It can be set to **client**, **server**, or **any**.
- **SEVERITY:** This attribute defines the severity of vulnerability signatures that you want the rule to match. It can range from **informational** to **critical** or be set to **any**.

The rule definition must also specify the required action and may optionally enable packet capture.

Exceptions

This tab allows you to create exceptions for individual signatures. It works very similar to the exceptions in the **Anti-Spyware** profile – you can override the action and the packet capture attribute of any signature matching a rule in your policy. The only difference is that some of the vulnerability protection signatures, especially those in the **brute-force** category, provide you with the ability to specify thresholds, as shown in the following screenshot:

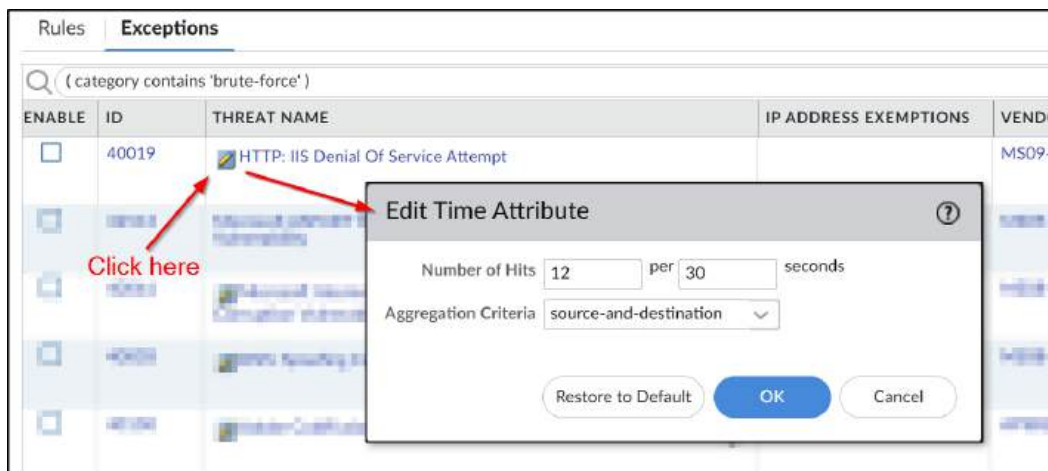


Figure 4.10 – How to change thresholds for a vulnerability signature

There is not much else to say about the **Vulnerability Protection** profile. Let's move on to the **URL Filtering** profile.

URL Filtering profile

This profile applies to all HTTP(S)-based applications matching the rule to which the profile is assigned. Compatible decoders extract the URL from the HTTP Host header, HTTP URI, the TLS SNI (*Server Name Indication*) field in the TLS handshake, or the CN (*Common Name*) field of the website certificate. Once the full URL is established, it can be processed by the Advanced URL Filtering engine.

Earlier in this chapter, we learned how Palo Alto firewalls classify URLs and how custom categories can be created. Now, it is time to learn what a **URL Filtering** profile allows us to do once we have matched an HTTP(S) connection by a security policy rule (which can be done using a URL category as one of the matching criteria!).

There are five tabs in this profile. Let's explore each of them.

Categories

This tab contains all available URL categories arranged in three distinctive groups:

- Custom URL categories
- External dynamic lists
- Predefined URL categories

The number of custom categories can be quite significant, so to make navigation easier, you can collapse and expand each of these groups as needed.

You can set *actions* for **SITE ACCESS** and **USER CREDENTIAL SUBMISSION** for each category on this tab. As you know, each URL can have up to four predefined categories assigned to it. It can also match more than one custom category. You can also apply desired actions to all or selected categories by accessing a context menu hidden behind a small arrow appearing when you hover your mouse over the column names.

Now, imagine you have a situation when the matching criteria of a security policy rule do not include a URL category, and you have a URL to process by a **URL Filtering** profile assigned to the rule. The URL matches several different predefined categories and several custom categories. Some of these categories have identical actions, some different. What is the firewall going to do?

The firewall will always identify the actionable category as follows. In order of precedence, it will try to find a match for the URL in the following:

1. Custom URL categories
2. External dynamic lists
3. Predefined URL categories

If more than one match is found at the same level (for example, more than one custom category matched), the firewall will choose the match with the strictest action. The order of precedence for actions is as follows:

1. **Block:** The URL will be blocked, and the **URL Filtering and Category Match Block** page will be generated and returned to the user instead of the original web response. The firewall will log the matched category and the action.
2. **Override:** The firewall will generate the **URL Filtering Continue and Override** page, prompting the user to enter a password to proceed to the page. There is only one system-wide password configured in **DEVICE | Setup | Content-ID | URL Admin Override**. The firewall will log the matched category and the action.

3. **Continue:** The firewall will generate the **URL Filtering Continue and Override** page, prompting the user to click a **Continue** button to proceed to the page. The firewall will log the matched category and the action.
4. **Alert:** The firewall will allow the URL and log the matched category.
5. **Allow:** The firewall will allow the URL but will not log it.

Custom URL categories also have the **None** action, which means that the category will be ignored in the context of action determination and logging in the **CATEGORY** field of the **URL Filtering** log. However, the category will still be logged as the first in the **URL CATEGORY LIST** field of the **URL Filtering** log.

If the URL is found in multiple categories of the same precedence level and all matched categories have the same action set, the alphabetical order becomes the tiebreaker.

You may have seen the effect of this final tiebreaker in the logs when URL Filtering profiles have the alert action set both for all benign content/behavior-based categories *and* for the risk-based categories. If this is the case, then a URL classified as *legal and low-risk*, for example, will be logged as *legal*; however, a URL classified as *military and low-risk* will be logged as *low-risk*. This effect completely ruins all built-in reporting and logging alike, as it's very difficult to compare "apples" to "oranges" when you attempt to analyze web traffic. There is a solution to this problem, which we will discuss when we get to building our policy in *Chapter 5*.

The **USER CREDENTIAL SUBMISSION** column defines what the firewall should do when it detects a submission of a user's corporate credentials via a web form (we will cover this feature in a minute). This column has a choice of four actions, as follows:

- **Alert:** Submission will be allowed and logged.
- **Allow:** Submission will be allowed and not logged.
- **Block:** Submission will be blocked. The **Anti Phishing Block** page will be displayed.
- **Continue:** Submission will be paused. The **Anti Phishing Continue** page will be displayed, providing an option to continue the submission.

URL Filtering Settings

This tab defines the level of logging (the **Log container** page only, **User-Agent**, **Referrer**, **X-Forwarded-For** options) and **Safe Search Enforcement**.

Depending on your logging requirements, you may want to log all information available or be selective, to save log storage and reduce the load on the firewall. The best practice is to log everything, at least for the URLs with medium and high-risk levels.

When enabled, the **safe search** option blocks search results from supported search providers (Bing, Google, Yahoo, Yandex, and YouTube) unless the user's browser is configured to perform safe searches only. The **URL Filtering Safe Search Block Page** template will be used for the response page.

User Credential Detection

The firewall feature corresponding to the settings on this tab allows to minimize risks of phishing attacks. When a user attempts to submit their credentials to a website, the firewall will apply the configured detection method and the action specified for the site's category.

Needless to say, decryption must be configured for this feature to identify submitted credentials in the HTTPS traffic.

There are three detection methods available:

- **IP User:** This is the most basic method for checking whether the user's name (from the IP-to-address mapping provided by the User-ID subsystem) is present in the web form submission request. The **password** field is completely ignored in this method.
- **Group Mapping:** The firewall checks whether the submitted username is a member of any of the groups referenced in **DEVICE | User Identification | Group Mapping Settings**. Therefore, you can block the submission of login names not only of the submitting user but any login name known to the firewall by means of the specified **Group Mapping Settings**. The **password** field is completely ignored in this method.
- **Domain Credential:** This is the most comprehensive method that allows the firewall to check both the username and the password. It requires a Windows User-ID agent with a User-ID credential service add-on installed on a **Read-Only Domain Controller (RODC)**. **The User-ID agent is referenced in DEVICE | Data Redistribution | Agents** on the firewall.

The credential service scans the Active Directory for usernames and password hashes and deconstructs them into a bit mask called a **bloom filter**. For lack of a better term, you can consider it a very smart hash.

The firewall retrieves the bloom filter from the agent at regular intervals and then checks whether the hashes of each submitted username and password are *probably* present in the original dataset used to generate the bloom filter. This mathematical method is very precise, albeit not absolute.

This technology ensures that the actual passwords or their hash values are never stored on the firewall and never transmitted across the network (from the User-ID agent to the firewall).

In addition to the detection method, you must select the severity of events that will be generated in the **URL Filtering** log when credential submission is detected.

- **HTTP Header Insertion:**

The **header insertion** feature allows you to enforce the use of specific tenants (controlled by your organization) in the supported SaaS applications. The firewall comes with templates for Dropbox, Google Apps, and Office 365. If that's not enough, you can define up to five custom headers.

The feature can also be used to enforce a restricted YouTube mode or dynamically populate the **X-Authenticated-User** header with the username and domain values associated with the source IP of the session to pass them onto an upstream security appliance. The username and domain values are available for custom headers as well. You can reference them as (\$domain) and (\$user).

Unfortunately, the **header insertion** feature works only with the HTTP/1.x protocol. HTTP/2 is not supported. The supported HTTP methods are GET, POST, PUT, and HEAD.

One of the domains specified in the **DOMAINS** field must be present in the HTTP Host header of the original request for the insertion to take place. You can use an asterisk (*) as a wildcard character to match subdomains.

- **Inline Categorization:**

The last tab in the profile allows us to enable real-time URL analysis using ML algorithms:

- **Enable local inline categorization:** This option enables firewall-based models to detect phishing and JavaScript exploits
- **Enable cloud inline categorization:** This option enables the forwarding of suspicious web page contents to the cloud, which complements the analysis models used by local inline ML

You can also create exceptions through custom URL categories or external dynamic URL lists.

File Blocking profile

This profile is responsible for controlling file uploading and downloading. You can control this based on the application, the direction of file transfer, and the file type. **File Blocking** profiles support more than 200 different file types.

Some of these file types work only in one direction – upload or download. Unfortunately, this is not always reflected in the file type name. For example, the possible-activex-cab file type is supported for downloading only, which is not reflected in the name, but png-upload, along with a few others, does have the indication of the supported direction right in its name.

It is also impossible to see the description of each file type in the firewall UI or in the built-in help pages.



You can search for the Palo Alto Knowledge Base article named *FileType list with the Threat-ID number* to see a full list of all supported file types, their corresponding threat ID, and a short description.

There is a special file type, `Multi-Level-Encoding`, that represents files with the number of layers of encoding in the file exceeding four – the maximum that the firewall supports. For example, an Excel document, compressed with ZIP and attached to an email message, will produce three levels of encoding. Therefore, by blocking this particular file type, you are making sure that the firewall policy related to file filtering is not evaded.

Do not confuse encoding with encryption in this context. While PAN-OS can identify many encrypted file types (for example, password-protected and encrypted ZIP or PDF files), it cannot scan them to detect malware or data patterns.

Rules are applied in order of precedence defined by the rule actions:

1. Block
2. Continue
3. Alert

These actions work similarly to the actions in the profiles we already discussed.

The **Continue** action has a caveat, though. It prompts the user only when used with the **web-browsing** application; with all other applications, it is identical to the **Block** action.

The best practices recommend using predefined file filtering profiles (named **basic file blocking** and **strict file blocking**) as a good starting point for your own profiles. You can clone them and make changes accordingly.

It is worth noting that the UI interface makes it very difficult to build complex **File Blocking** profiles because of the sheer number of available file types, the lack of their description, and non-resizable windows. Automation, which we will cover in later chapters, will allow us to mitigate these unfortunate limitations.

WildFire Analysis profile

This profile defines what files get uploaded to the WildFire cloud (or on-premises WildFire appliances) for sandbox analysis. WildFire sandboxing is a technology that **detonates** (executes or opens with a corresponding application) potentially dangerous file types in specially built replicas of common operating systems and software—Windows, Linux, macOS, and Android. The system and network activity associated with the detonated file is carefully analyzed from a security standpoint, and a verdict is distributed to all firewalls with active WildFire subscriptions.

I strongly advise you to read the *Advanced WildFire Administration* guide by Palo Alto Networks to learn more about this technology, as it has many caveats and nuances.

The profile itself is very simple. It consists of unordered rules that specify the following attributes:

- **APPLICATIONS**
- **FILE TYPES**
- **DIRECTION**
- **ANALYSIS**

The former three attributes define the matching criteria for the files that must be submitted for analysis, and the latter attribute specifies the target environment – either the **public-cloud** service or **private-cloud** (on-premises hardware appliances).

The profile works in conjunction with the system-wide settings in **DEVICE | Setup | WildFire**. This screen defines, among other configuration parameters, the maximum file sizes of the uploaded files, logging options for benign and grayware files, and specific details of the session associated with the file transmission.

Data Filtering profile

This final profile configures the engine that looks for data patterns in the transmitted files.

Before you can create a **Data Filtering** profile, you must create data patterns that the profile will be configured to look for. You can define these patterns in **OBJECTS | Custom Objects | Data Patterns**.

Each data pattern consists of a name, description, pattern type, and then the actual pattern. Definitions of the patterns vary depending on the selected pattern type. Here is the list of pattern parameters grouped by pattern type:

- **Predefined Pattern:**
 - **NAME:** A drop-down list of predefined names such as **Credit Card Numbers**, **Social Security Numbers**, and so on.
 - **DESCRIPTION:** This field is auto-populated when you select one of the predefined names.
 - **FILE TYPE:** This field specifies the types of files where the selected predefined pattern will be searched for. You can specify the keyword **any** or select one of the supported file types, which currently include **HTML** and **Microsoft Office Documents**.
- **Regular Expression:**
 - **NAME:** A custom name for the expression.
 - **FILE TYPE:** This field specifies the types of files where the regular expression will be searched for. You can specify the keyword **any** or select one of the supported file types, which currently include **HTML** and **Microsoft Office Documents**.
 - **DATA PATTERN:** This is the actual regular expression to look for. We will not be covering the syntax of regular expressions here because it is very well described in the online documentation.
- **File Properties:**
 - **NAME:** A custom name for the combination of file property and value.
 - **FILE TYPE:** This field specifies the types of files where the file property will be looked for. You can specify one of the following types: **Adobe PDF**, **AIP Protected Microsoft Excel**, **AIP Protected Microsoft PPTX**, **AIP Protected Microsoft Word**, **Microsoft Excel**, **Microsoft PowerPoint**, **Microsoft Word**, and **Rich Text Format**.

- **FILE PROPERTY:** This is the file property to look for. The values in this drop-down list change depending on the selected **FILE TYPE** option. For example, for the **Azure Information Protection (AIP)** Office files, the only available file property is **Microsoft MIP Label**.
- **PROPERTY VALUE:** This is the value of the selected file property.

Once you have created the data patterns of interest, you can proceed with the creation of a **Data Filtering** profile.

The profile definition comprises the following:

- **Name.**
- **Description.**
- Option to capture actual file data that matched the specified pattern(s).
- A list of rules for data patterns that you created in the previous step. Each rule has parameters as follows:
 - **DATA PATTERN:** This drop-down list contains all patterns you defined before
 - **APPLICATIONS:** Here, you select the keyword **any** or specific applications to be subjected to data filtering
 - **FILE TYPE:** Select the keyword **Any** or select specific file types
 - **DIRECTION:** Select **upload**, **download**, or **both**
 - **ALERT THRESHOLD:** Define the minimum number of times the data pattern must be detected in a file to trigger an alert
 - **BLOCK THRESHOLD:** Define the minimum number of times the data pattern must be detected in a file for the file transmission to be blocked
 - **LOG SEVERITY:** Define the log severity of **Data Filtering** log records that will be created when the patterns are detected and actioned accordingly

This concludes the list of security profiles. Let's move on to the remaining features directly related to the security policy.

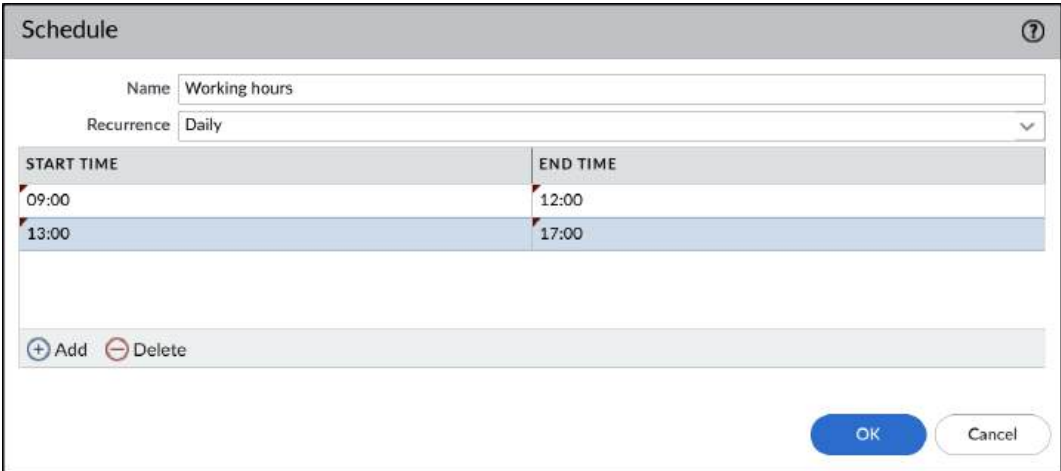
Auxiliary functions

The remaining elements of the security policy rule allow us to specify when the rule must be activated, the required quality of service processing, and turn off server response inspection if required. Let's explore each of these options.

Schedule

Each rule can reference a schedule that defines when the rule is active. If you do not specify a schedule, the rule will always be active.

The schedule recurrence can be **Daily**, **Weekly**, or **Non-recurring**. The following screenshot demonstrates an example of a daily schedule.



The screenshot shows a 'Schedule' dialog box with a title bar containing a question mark icon. Inside, there is a 'Name' field with the text 'Working hours' and a 'Recurrence' dropdown menu set to 'Daily'. Below these is a table with two columns: 'START TIME' and 'END TIME'. The table contains two rows of time slots: 09:00 to 12:00 and 13:00 to 17:00. At the bottom of the table are '+ Add' and '- Delete' buttons. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

START TIME	END TIME
09:00	12:00
13:00	17:00

Figure 4.11 – Example schedule

As you can see, it is possible to define several time slots.

You can create schedules in **OBJECTS | Schedules** and then reference them on the **Actions** tab of security policy rules.

QoS Marking

Each security policy rule can optionally perform **quality of service (QoS)** marking of the IP packets as they egress the firewall.

You can use either **IP DSCP** or **IP Precedence** for **QoS Marking**.

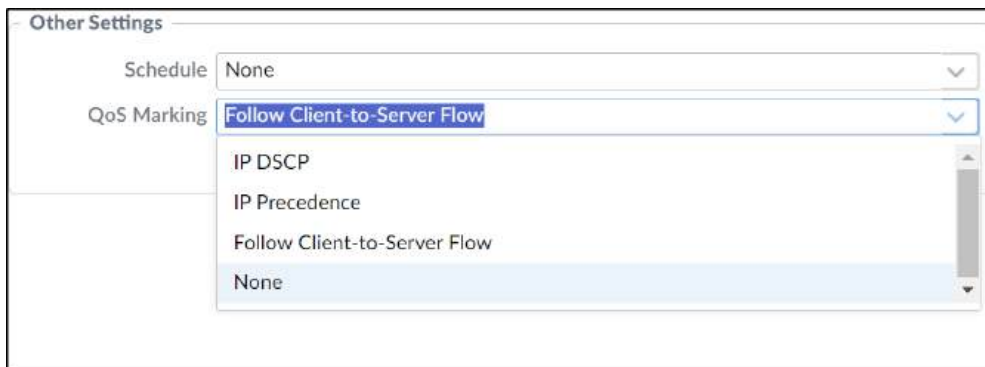


Figure 4.12 – Options available for QoS Marking

When you choose one of these options, all packets matching the session in either direction will have the specified bits (code points) set in the IPv4 ToS byte.

Another option is to **Follow Client-to-Server Flow**, which means that the firewall will mark packets in the s2c flow of the session with the same **Type of Service (ToS)** value that was found in the c2s flow of the session.

QoS Marking settings in the security policy rules are just one of many QoS features supported by PAN-OS. However, this technology is outside the scope of this book, which is focused on security policies.

Server response inspection

Disable Server Response Inspection is the last configurable setting on the **Actions** tab of security policy rules. When enabled, it makes the firewall stop any security inspection of the server-to-client flow of the sessions matching the rule.

This option concludes our review of the security policy rule elements.

Let's finish this chapter by exploring the firewall response pages. This feature directly relates to the firewall's actions resulting from security policy evaluation.

Response pages

When the firewall performs an action that interrupts the normal flow of a network connection (for example, a URL needs to be blocked), it can inject its own response instead of the original server response. The client software usually (but not always) cannot tell the server response from the firewall response – it simply displays whatever is received in response to its request. Thus, the user gets to see firewall block pages.

These firewall web pages are called **response pages**. There are 18 different types of response pages. We'll cover a selection of these pages related to firewall security and decryption policies:

- **Anti Phishing Block Page:** This page is displayed when the user login credentials submission is blocked
- **Anti Phishing Continue Page:** This page is displayed when user login credentials submission is paused
- **Antivirus/Anti-spyware Block Page:** This page is displayed when the firewall detects a virus or spyware
- **Application Block Page** (disabled by default): This page is displayed when an application is blocked
- **Data Filtering Block Page:** This page is displayed when the data filtering profile blocking action triggers (the blocking threshold for a data pattern is exceeded)
- **File Blocking Block Page:** This page is displayed when a file transmission is blocked
- **File Blocking Continue Page:** This page is displayed when a file transmission is paused
- **SSL Certificate Errors Notify Page:** This page is displayed when a TLS certificate error is encountered (for example, the certificate is revoked or expired)
- **SSL Decryption Opt-out Page** (disabled by default): This page warns users that their traffic will be decrypted and allows them to opt out
- **URL Filtering Continue and Override Page:** This page is displayed when a user accesses a URL that matches a category with the *override* or *continue* actions set
- **URL Filtering and Category Match Block Page:** This page is displayed when a user accesses a URL that matches a category with the *block* action set

- **URL Filtering Safe Search Block Page:** This page is displayed when a user attempts to use Bing, Google, Yahoo, Yandex, or YouTube while the **Safe Search** option is not enabled in their browser and the matching **URL Filtering** profile in the policy is configured to enforce the **Safe Search** feature

Each page has a corresponding template (located in **DEVICE | Response Pages**) that can be customized to suit your requirements. This allows you to use your own branding and verbiage.

A template is an HTML file that must not exceed 17,999 bytes. There is no limitation on the actual content of the HTML code; therefore, it can contain references to external resources (for example, external JavaScript files, pictures, and CSS files) and embedded JavaScript code. Unfortunately, there is no way to store the external resources (for example, a custom logo) on the firewall itself. You can work around this limitation by using Base64 encoding of images and thus embed them directly into the HTML code. Alternatively, you can use an external web server.

Templates can reference special PAN-OS variables. Before sending a response page to the user, the firewall will replace the variable reference with the actual runtime value. When the user's browser receives the page, it renders it and executes the embedded JavaScript code, which can leverage the runtime values of the variables.

The availability of the variables depends on the type of response page. Refer to the following table for the full list of variables:

Variable	Availability	Description
<threatname/>	Antivirus/Anti-spyware Block Page	This variable contains the ID and the name of the blocked threat.
<rulename/>	Application Block Page	This variable contains the security policy rule name that triggered the block action.
<user/>	All pages	This variable contains the username mapped to the source IP of the blocked session. If there is no mapping, it will contain the source IP address.
<url/>	URL Filtering Continue and Override Page URL Filtering and Category Match Block Page	This variable contains the requested URL (can be the destination IP address for TLS decryption).

Variable	Availability	Description
<category/>	URL Filtering Continue and Override Page URL Filtering and Category Match Block Page SSL Certificate Errors Notify Page	This variable contains the category of the requested URL.
<appname/>	Application Block Page	This variable contains the name of the blocked application.
<pan_form/>	URL Filtering Continue and Override Page Anti Phishing Continue Page	Depending on the context, this variable is substituted by HTML code corresponding to the Continue button and password field (for the Override action).
<fname/>	Antivirus/Anti-spyware Block Page File Blocking Block Page	This variable contains the name of the blocked file.
<certname/>	SSL Certificate Errors Notify Page	This variable contains the certificate name.
<issuer/>	SSL Certificate Errors Notify Page	This variable contains the CN of the certificate issuer.
<status/>	SSL Certificate Errors Notify Page	This variable contains the certificate status.
<reason/>	SSL Certificate Errors Notify Page	This variable specifies the cause of the certificate error.

Table 4.2 – Response page variables and their availability in response pages

As you can see, there are some unfortunate limitations. For example, you can display the category of a blocked URL, but you cannot display the category or subcategory of a blocked application. There is a workaround for this issue, though. We will cover it in the next chapter.



When customizing response pages, it's important to be very cautious when referencing external resources. These resources should be highly trusted and under your full control to avoid serving users with malicious content in case the resource is compromised.

Summary

In this chapter, you explored how PAN-OS handles traffic once it has been matched to a security policy rule. You learned how to configure the action taken on a session—whether to allow or block it—as well as how to apply logging, tagging, and external log forwarding. We also examined the suite of security profiles available in PAN-OS, including **Antivirus**, **Anti-Spyware**, **URL Filtering**, and others, each designed to enforce specific layers of protection and inspection.

Together, these features define how matched traffic is processed, monitored, and controlled. You now have the tools to apply granular inspection to sessions and shape their handling according to business and security requirements. In the next chapter, you'll put all of this into practice by designing a complete security policy that reflects the functional needs, risks, and technical constraints discussed earlier in the book.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



5

Security Policy Design

In *Chapter 3* and *Chapter 4*, we thoroughly examined all components of PAN-OS security policy rules. You can now go ahead and experiment with various types of objects and policy rules in a lab environment.

One of the things we learned about firewall security policy is that it is a flat structure—an ordered set of rules where the first match wins, and processing stops thereafter. **Application Shift**, covered in *Chapter 1*, certainly adds a layer of complexity; however, the policy remains the same flat structure, which seems challenging to adapt to an often somewhat structured organizational landscape.

The Panorama management appliance is designed to fill this gap, among many other valuable features. It enables centralized administration of all firewalls in a company, regardless of the size of the infrastructure. You will benefit from it even if you only have two to three firewalls, but its true value becomes evident in large networks with tens, hundreds, or even thousands of firewalls. In this chapter, we will cover the basics of Panorama in the context of creating a firewall policy, including **device groups**, **templates**, and **template stacks**.

Then, we will proceed with the policy design to meet the requirements outlined at the end of *Chapter 1*. We will discuss policy principles, its distinctive sections, naming conventions, response pages, integration with a ticketing system, external dependencies, TLS decryption, application signatures, and logging.

In this chapter, you will cover the following topics:

- Panorama 101: device groups and templates
- Core policy principles
- Interface with end users and the IT help desk
- Tagging and naming convention
- External dependencies
- TLS decryption
- Legacy custom applications and policy rules
- Logging

By the end of this chapter, our policy will begin to take shape, and we will be ready to automate its creation in the subsequent chapters.

Panorama 101: device groups and templates

To better understand our policy design options, we need to learn core Panorama capabilities related to policy structuring. We will cover device groups, templates, and template stacks.

If you do not have Panorama in your environment, you can skip this section. The policy and subsequent automation we will build will work equally well on standalone firewalls and Panorama-managed ones.



In case you would like to learn how to set up Panorama from scratch, I recommend *Panorama Administrator's Guide*, Palo Alto Networks (<https://docs.paloaltonetworks.com/panorama>), and the *Managing Firewalls Through Panorama* chapter of *Mastering Palo Alto Networks*, Tom Piens, Packt Publishing.

Introduction to device groups

Device groups are logical containers that allow you to form a tree-like hierarchy. Each container contains policies and objects applicable to other containers and/or firewalls placed within it. The structure of the device groups defines the resultant effective policies and objects applied to firewalls.

The hierarchy of custom device groups can be up to four levels deep, as shown in the following figure:

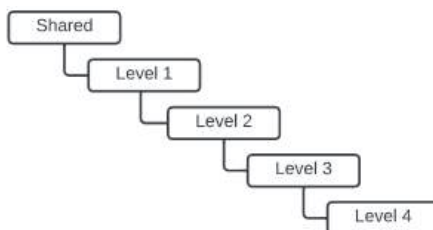


Figure 5.1 - Device group hierarchy levels

Each device group can have an alphanumeric name, except for the root (topmost) level, which has a predefined name, **Shared** that cannot be changed. In total, you can have up to 1,024 device groups.

After Panorama is installed, you need to create at least one custom device group. It will be placed at level 1, as a direct descendant of the **Shared** group. Other device groups—**Level 2**, **Level 3**, and **Level 4**, in the example shown—are optional.

Device groups are created in **Panorama | Device Groups**. The following figure shows the creation of a new device group:

The screenshot shows the 'Device Group' configuration window. The 'Name' field is 'Level 1 device group example'. The 'Description' field is empty. The 'Parent Device Group' is 'Shared'. The 'Devices' section has a 'FILTERS' sidebar with 'Device State' (Connected (2)), 'Platforms' (PA-850 (2)), and 'Templates'. The main list shows two devices: PA850-1 and PA850-2, both selected. Below the list are checkboxes for 'Select All', 'Deselect All', 'Group HA Peers', and 'Filter Selected (2)'. The 'User ID Master Device' section has a dropdown set to 'PA850-1' and a checkbox for 'Store users and groups from Master Device if Reporting and Filtering on Groups is enabled in Panorama Settings'. The 'REFERENCE TEMPLATES' section has a checkbox for 'test template'. At the bottom are 'Add' and 'Delete' buttons, and 'OK' and 'Cancel' buttons.

Figure 5.2 - Creation of a new device group

As the screenshot shows, when creating a new group, you need to fill in the **Name** and **Parent Device Group** fields. You can also optionally provide a description, reference templates (or template stacks), a reference source of user ID information, and, most importantly, devices (firewalls) that will belong to this device group. We will discuss templates in a minute.

It's okay to have an empty group without any firewalls or nested device groups. You can keep the group as a placeholder and reserve it for future use in your hierarchy.

You must create at least one custom device group on a newly deployed Panorama instance. Two new tabs—**POLICIES** and **OBJECTS**—will appear in the administrative interface as soon as you do. These tab names imply what is kept inside. Their meaning and purpose are identical to that of the same tabs of a standalone firewall. A **Device Group** drop-down list with all defined device groups will also appear under the tabs.

The selection you make in this field changes the current focus of the **POLICIES** and **OBJECTS** tabs to the content of the selected device group. This is shown in the following screenshot:

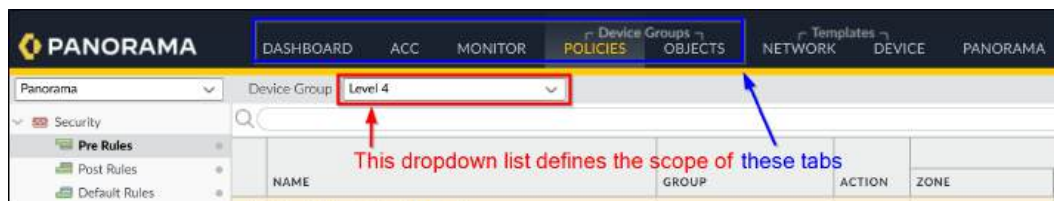


Figure 5.3 - Scope selection in Panorama administrative interface

You will be able to *view and modify* all configuration artifacts defined in the selected device group (**Level 4** in the shown example), and *view but NOT modify* all configuration artifacts in its parent device groups (**Level 3**, **Level 2**, **Level 1**, and **Shared**). The **DASHBOARD**, **ACC**, and **MONITOR** tabs will display data received from the firewalls located in the selected device group and in all device groups lower in the hierarchy.

A device group placed at a lower level in the hierarchy inherits all objects from its parent groups—all the way up to and including the **Shared** level. You may choose not to share unused address objects and service objects with devices (other object types are always shared). This system behavior is defined in **PANORAMA | Setup | Management | Panorama Settings | Share Unused Address and Service Objects with Devices**.

By default, objects defined at the lower levels of the device group hierarchy take precedence over objects with the same name defined at higher levels. This system behavior can be changed in **PANORAMA | Setup | Management | Panorama Settings | Ancestor Objects Take Precedence**.

Now, you will explore how policy rules work with Panorama.

Policy hierarchy

As you will remember, a firewall policy is a *flat*, ordered list of rules. The policy visible on a firewall fully managed by Panorama is determined by the firewall's position in the Panorama device group hierarchy, as well as the policy rules established within each device group from the level of the device group that the firewall belongs to and all its parent groups.

Therefore, it is crucial to understand how this resultant set of ordered rules is constructed out of policy rules defined at different levels of the device group hierarchy.

Each device group has two mandatory policy sections for each type of policy: **Pre Rules** and **Post Rules**. The security policy (but not other policy types) also has a special section called **Default Rules**. This is demonstrated in the following figure:

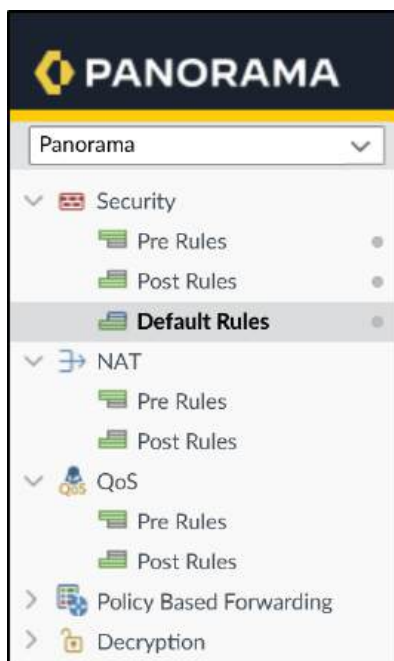


Figure 5.4 - Pre Rules, Post Rules, and Default Rules policy sections

If we place a firewall in the **Level 4** device group from our example and create security rules in each group's **Pre Rules** and **Post Rules** sections, the resulting effective security policy pushed to the firewall will look as shown in *Figure 5.5*:

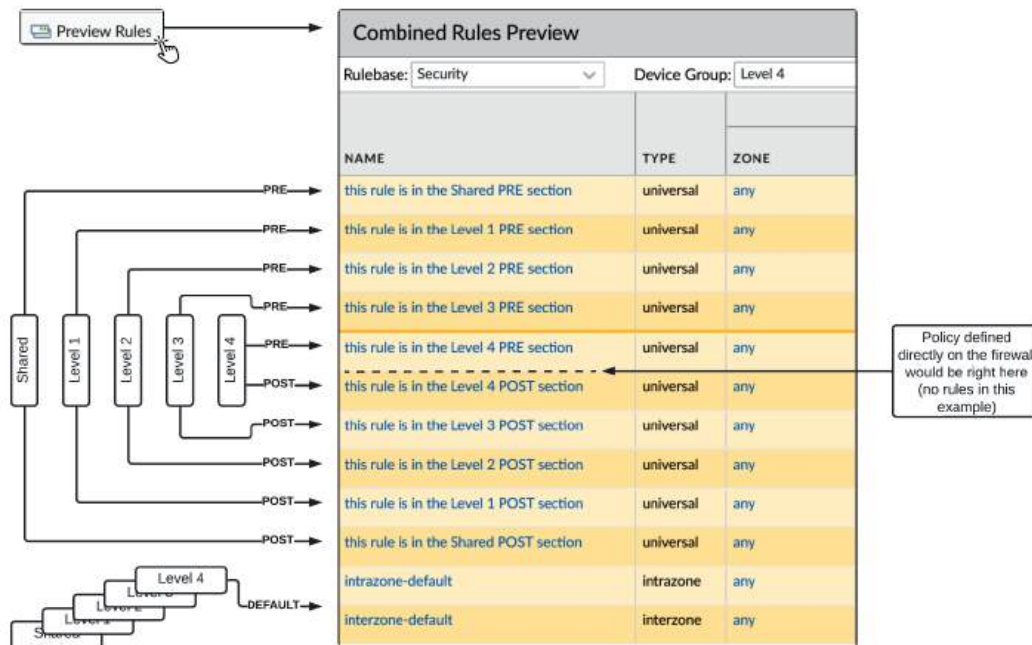


Figure 5.5 - Effective firewall policy formation

You can preview an effective firewall policy like this by clicking on **Preview Rules** at the bottom of the Panorama **POLICIES** screen. As you can see, the rules from the **Pre Rules** and **Post Rules** sections of higher-level device groups embrace the rules defined at lower levels of the hierarchy.

Now, take a moment to reflect on the principles outlined above. Consider your organization's structure and how you could leverage a multi-layered hierarchical approach to building firewall policies. For example, if you run your business in different geographical locations with rather specific requirements, you could create some global allow/block rules at level 1, then create regional groups with deviations from the global policy at level 2, and individual site groups at level 3.

A device group hierarchy can also serve as a basis for administrative **Role-Based Access Control (RBAC)**. For example, enterprise admins can have full access to **Shared** and **Level 1** device groups, while regional and site admins can work only at **Level 2** and **Level 3**, respectively.

The **Default Rules** section in each device group always contains only two rules—replicas of the **Intrazone-Default** and **Interzone-Default** rules on a firewall. You can override either of these rules in any device group by selecting a rule and clicking on **Override** at the bottom of the screen.

This override action allows you to change the rules' **Tags**, **Group Tag**, **Audit Comment**, **Action**, **Profile Setting**, and **Log Setting**. The changes you make to these rules in a lower-level device group will take precedence over those at a higher level.

It is considered best practice to always enable logging in the **Interzone-Default** rule (it is disabled by default).

Introduction to templates and template stacks

Templates (**PANORAMA | Templates**), coupled with the device groups, are essential to achieve a consistent firewall configuration in a multi-site enterprise environment.

Unlike device groups, templates are not directly related to policies. However, they are logical containers for many policy dependencies. **Templates** define everything found in the **NETWORK** and **DEVICE** tabs of the firewall administrative UI. Having these settings defined in the form of a template allows you to set global, regional, or site-specific standard settings once and then assign the template with these settings to as many firewalls as needed.

Many template settings can be defined as variables. The number and type of settings that can be defined as variables are growing as new PAN-OS versions are released. This feature allows you to configure various settings without knowing the actual runtime values, defining them only when you assign firewalls to the template. For example, you can specify `$ldap` instead of the actual server address in an LDAP server profile, assign the template to multiple firewalls, and then set the actual variable values for each firewall—this is done via a straightforward CSV file.

Templates can be organized into template stacks, which are then assigned to firewalls instead of individual templates. In recent PAN-OS versions, this is now the only option—you cannot assign a single template to a firewall anymore; it must be a stack. The order of templates within the stack is crucial, as the final settings applied to the firewalls are determined by merging the templates from bottom to top. Single-value settings from the templates positioned higher in the stack override those defined in the templates below. Multiple-value settings with different names will all be included in the merged result. For example, if two templates in a stack each define an LDAP server profile with a unique name, the firewall assigned to the stack will end up with both server profiles in its running configuration. A template does not have to define all network and device settings of a firewall. It may well define just a single parameter, thus relying on other templates or locally

defined firewall configuration for other essential settings. For example, you can create a template that contains all the settings for a new project, like the security policy we're building in this book. Then, as you onboard new sites, you can simply assign this functional template to the template stacks of the sites being migrated and thus ensure that all your policy dependencies are in place.

The template settings minimally required for our policy to operate are as follows:

- Custom response pages (**DEVICE | Response Settings**)
- Group mapping settings (**DEVICE | User Identification | Group Mapping Settings**)
- LDAP profile used in the group mapping settings (**DEVICE | Server Profiles | LDAP**)
- HTTP profile used in the log forwarding profiles (**DEVICE | Server Profiles | HTTP**)

Finally, after all the theory and relevant reference information covered in the previous chapters, it is time to start building your security policy.

Core policy principles

We have learned how next-generation firewalls process network connections and identify network protocols and applications. We reviewed the core components of firewall security policies and Panorama features related to policy structuring. We also presented a business case and compiled a set of technical requirements.

Let's now examine how to leverage this knowledge to design a robust enterprise-grade firewall security policy. This will be our primary focus for the remainder of this chapter. After finalizing the design, we will proceed to automate the project in the subsequent chapters.

To establish policy principles, it is important to review how next-generation firewalls recognize traffic and to introduce new terminology.

Traffic identification and classification

As you may remember, the firewall uses two primary methods of traffic identification and classification:

- App ID application signatures that are broken into categories and assigned distinctive attributes and characteristics
- URL categorization of *web* traffic

The former mainly depends on human research, offering higher accuracy. The latter relies predominantly on machine classification, making it more susceptible to errors.

At a very high level, the firewall will follow this algorithm (refer to *Figure 5.6* as you go through this list):

1. It will always attempt to find a matching App ID signature first. The two most prominent groups of application signatures are those based on HTTP and HTTPS (depicted as **HTTP(S) app signatures** on the pie chart) and those working directly over TCP, UDP, or IP (depicted as **Non-HTTP(S) app signatures**).
2. If the firewall fails to find a matching signature but has enough information to conclude that the underlying protocol is HTTP or HTTPS, it will resort to two particular generic signatures: **web browsing** and **ssl**, respectively. They are shown on the pie chart as **Generic HTTP (web-browsing)** and **Generic TLS (ssl)**.
3. Depending on the transport protocol used, if the traffic is neither HTTP nor HTTPS, the firewall will resort to two other generic signatures—**unknown-udp** and **unknown-tcp**. These are shown on the chart as **Generic UDP (unknown-udp)** and **Generic TCP (unknown-tcp)**.
4. For HTTP(S) traffic (both generic and signature-based), the firewall will always perform the categorization of detected URLs.

Thus, the firewall's view of all possible categories of network traffic it processes, as described in the preceding steps, can be illustrated in the pie chart shown in the following figure:

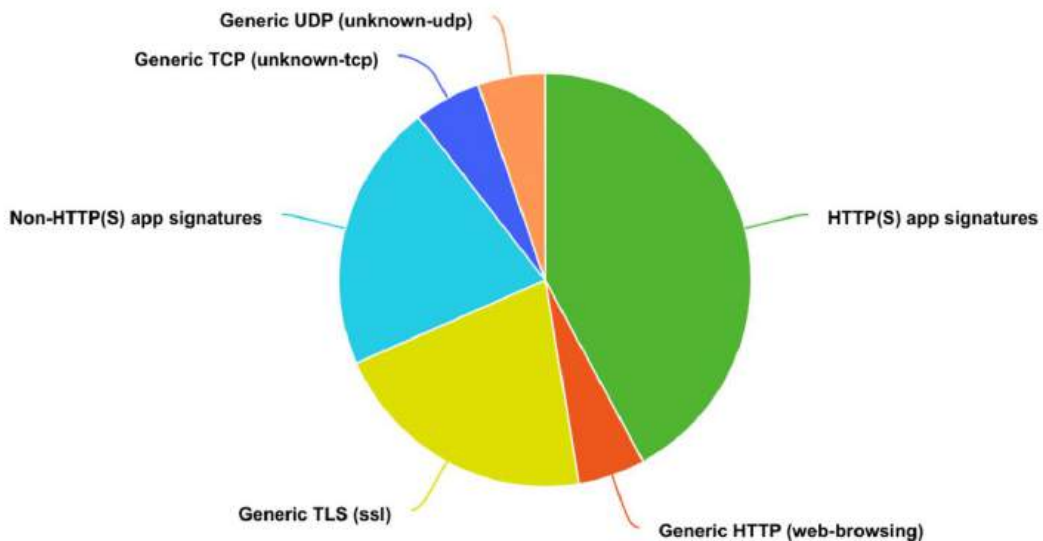


Figure 5.6 - Representation of all traffic types our security policy must handle

The sizes of the slices in this pie chart are arbitrary because these proportions will vary from one network to another, depending on the specific set of applications and websites used. This is what our firewall security policy must work with. The policy must instruct the firewall on how to handle each slice of the traffic pie carefully, in a security-conscious way, and in accordance with business needs.

The following section introduces new terminology that will help us formulate policy principles.

Policy terminology

As you may recall from earlier chapters, application signatures in PAN-OS are organized into categories, which can be viewed in the firewall's administrative interface under **OBJECTS | Applications**.

Some of these categories—such as file-sharing or email—carry clear business, cybersecurity, or regulatory implications. For these, we must be selective about **which applications are permitted and who is allowed to use them**, with user access controlled through Active Directory groups. Let's refer to these as **managed categories**.

Other categories—such as marketing or photo-video—have little direct impact on business operations. In these cases, we are still selective about **which applications are permitted**, but we make no distinction about **who can use them**: any authenticated user may do so. We will call these **non-managed categories**, reflecting the fact that access is not tied to user identity.

Finally, there are categories that should be blocked outright. Examples include encrypted tunnels and databases. We will call these **non-sanctioned categories**.

In short, both **managed** and **non-managed** categories are sanctioned for use, but with different levels of oversight, while **non-sanctioned** categories are explicitly denied.



Our final solution will be fully customizable, allowing you to decide which controls apply to specific categories in your particular environment. Here, we use straightforward examples that can be leveraged further in the chapter to explain policy principles.

Now, spend a few minutes browsing the available applications under **OBJECTS | Applications**. Click on items in the **SUBCATEGORY** field to see which applications belong to each subcategory. You will quickly notice that some subcategories are overly broad. In most business environments, you'll want more granular controls. For background on how to achieve this, refer back to the Applications section in *Chapter 3*.

In short, the **TAGS**, **RISK**, and **CATEGORY** attributes typically provide the level of granularity you need. If the built-in tags are not sufficient, you can create custom tags under **OBJECTS | Tags** and use them as additional selection criteria.

It follows that **managed** and **non-managed** applications that do not meet your selection criteria (e.g., tag, risk, etc.) will also fall into the **non-sanctioned** category. Put differently, all of the above can be summarized in the mathematical formula shown in the following figure:

$$\text{Non-generic apps} = (\text{sanctioned} \cap \text{managed}) \sqcup (\text{sanctioned} \cap \text{unmanaged}) \sqcup (\text{unsanctioned})$$

Figure 5.7 - Mathematical representation of policy terms and their interrelationships

This formula can be restated as follows.



Here, non-generic means specific App-ID signatures; the generic fallbacks used when App-ID reverts to URL filtering—e.g., web-browsing, ssl, soap, websocket—are outside this classification and are governed by URL filtering policy

Each non-generic app signature falls into exactly *one* of these groups:

- **Sanctioned and managed** – applications we oversee and restrict by user identity
- **Sanctioned and non-managed** – applications pre-approved for all authenticated users
- **Non-sanctioned** – applications that are off-limits to everyone and everything

The following section explains how this approach can be expressed in the form of an actual firewall security policy.

Policy principles

Now that you understand what the policy must cover in principle and have settled on the terminology, move on to the implementation. Take an iterative approach: build the policy step by step, adding complexity and functionality with each pass.

Begin with a straightforward first iteration. Convert the basic formula developed at the end of the previous section into a security policy (see the following figure):

	NAME	ACTION	Source	APPLICATION	URL CATEGORY	PROFILE
			USER			
1	managed-apps-file-sharing	 Allow	 U-file-sharing	 AF-file-sharing	any	
2	managed-apps-email	 Allow	 U-email	 AF-email	any	
3	non-managed-app-categories	 Allow	 known-user	 AF-marketing  AF-photo-video	any	
4	non-sanctioned-apps	 Deny	any	any	any	none

Figure 5.8 - Core policy principles – first iteration

Addresses and zones are omitted in this screenshot for brevity. This simple security policy uses the example categories mentioned in the Policy terminology section. Here is what each of the policy rules does:

- *Rule 1* allows users from the **U-file-sharing** group to access all applications defined by the **AF-file-sharing** application filter. This rule controls the first managed category of applications: file sharing.
- *Rule 2* allows users from the **U-email** group to access all applications defined by the **AF-email** application filter. This rule controls the second managed category of applications: e-mail.
- *Rule 3* allows all authenticated users to access all applications defined by the **AF-marketing** and **AF-photo-video** application filters. This rule controls the non-managed categories: marketing and photo/video.
- *Rule 4* blocks all other applications (non-sanctioned ones, in policy terms).

This policy effectively manages only two slices of the traffic pie chart shown in *Figure 5.6*: **HTTP(S) app signatures** and **Non-HTTP(S) app signatures**. All remaining slices will be simply denied by *Rule 4*.

Now, you need to account for all generic HTTP(S) traffic. In Palo Alto Networks firewalls, this is represented by two special signatures: **web-browsing** and **ssl**. When these two applications are detected, URL categorization must be applied. All URL categories should be divided into the same three groups as with non-generic apps: *managed*, *non-managed*, and *non-sanctioned*.

Using the closest URL-based equivalents to the example application categories introduced above, add controls for this generic web traffic to the policy. The policy will expand as follows:

	NAME	ACTION	Source		APPLICATION	URL CATEGORY	PROFILE
			USER				
1	managed-apps-file-sharing	Allow	U-file-sharing	AF-file-sharing	any		
2	managed-apps-email	Allow	U-email	AF-email	any		
3	managed-url-file-sharing	Allow	U-file-sharing	ssl web-browsing	online-storage-and-backup		
4	managed-url-email	Allow	U-email	ssl web-browsing	web-based-email		
5	non-managed-app-categories	Allow	known-user	AF-marketing AF-photo-video	any		
6	non-managed-url-categories	Allow	known-user	ssl web-browsing	any		
7	non-sanctioned-apps	Deny	any	any	any	none	

Figure 5.9 - Core policy principles – second iteration

As you can see, rules 3, 4, and 6 have been added. Here is what they do:

- **Rule 3** allows users in the **U-file-sharing** group to access all websites in the **online-storage-and-backup** URL category, as long as these sites do not have a special app ID signature and thus match the generic **web-browsing** or **ssl** application signatures.
- **Rule 4** allows users in the **U-email** group to access all websites in the **web-based-email** URL category, as long as these sites do not have a special app ID signature and thus match the generic **web-browsing** or **ssl** application signatures.
- **Rule 6** allows all authenticated users to access all websites except the ones in the non-sanctioned URL categories, provided that these sites do not have a special app ID signature and thus match the generic **web-browsing** or **ssl** application signatures.

Policy actions for non-managed and non-sanctioned URL categories are hidden in a URL filtering profile assigned to **Rule 6**.

Important



If you followed the same approach you used with other applications and leveraged the **URL CATEGORY** matching criterion and the **ACTION** parameter of the rule itself to block *non-sanctioned* URL categories, you would have ruined the associated firewall response page because the page would have shown only the blocked application name (i.e., **ssl** or **web-browsing**) but not the name of the blocked URL category.

The approach to blocking unwanted URLs via URL filtering profiles rather than rule action allows you to access a response page variable containing the name of the blocked URL category. You can leverage this variable to provide users with a contextualized text message and integrate the incident with a helpdesk system. This aspect of the policy will be covered in the *Interface with end users and the IT help desk* section.

This is good progress, but what about the risk-based approach that the policy requirements demanded?

First, you can incorporate the risk levels of applications into the selection criteria of the respective application filters. However, this method is relatively static. It only keeps you updated with newly released application signatures. What if you encounter a higher-risk URL embedded in an application's domain?

You can address this by adding an extra policy rule above each rule for managed and non-managed application categories. This additional rule will have an additional matching criterion: **URL CATEGORY**. Two standard URL categories should be specified in that policy field: **high-risk** and **medium-risk**.

Here is an example:

	NAME	ACTION	Source		APPLICATION	URL CATEGORY	PROFILE
			USER				
1	managed-apps-file-sharing-risky	✔ Allow	👤 U-file-sharing	📁 AF-file-sharing	high-risk medium-risk	🛡️	
2	managed-apps-file-sharing-regular	✔ Allow	👤 U-file-sharing	📁 AF-file-sharing	any	🛡️	
3	managed-apps-email-risky	✔ Allow	👤 U-email	📧 AF-email	high-risk medium-risk	🛡️	
4	managed-apps-email-regular	✔ Allow	👤 U-email	📧 AF-email	any	🛡️	

Figure 5.10 - Distinguishing higher-risk URLs embedded in managed applications

As you can see, instead of one rule per application category, you now have two rules. The additional rule can enforce stricter security profiles and leverage the risk category of the URLs that comprise the web applications that match the application filter in the rule. This risk-specific approach will only be available for HTTP(S)-based applications, which is acceptable. Non-HTTP(S) applications will always match the regular rule for the application category.

Policy rules based on URL categories need a different approach to become risk-aware. Let's explore how you can do this.

As you remember from *Chapter 3*, each URL can belong to up to four *standard* categories, one of which indicates the URL's risk level. Also, there is a special *custom* category type called **Category Match**. A URL will match such a category if it is simultaneously classified as each of the specified standard categories. Essentially, this feature allows you to have hardcoded combinations of standard categories. You can leverage this feature to create risk-aligned versions of managed URL categories. You can create them as shown in the following figure:

NAME	TYPE	MATCH
UM-online-storage-high-risk	Category Match	online-storage-and-backup high-risk
UM-online-storage-medium-risk	Category Match	online-storage-and-backup medium-risk
UM-webmail-high-risk	Category Match	web-based-email high-risk
UM-webmail-medium-risk	Category Match	web-based-email medium-risk

Figure 5.11 - Risk-aligned managed URL categories

For instance, a URL will match the **UM-webmail-high-risk** category shown in the screenshot *only* if it is classified as **web-based-email** *and* **high-risk** simultaneously.

Armed with these new categories, you can further expand the policy by creating two rules for each managed URL category: one for high- and medium-risk traffic, and another for all other traffic (the latter effectively includes low-risk URLs and URLs without the risk attribute). This separation by risk level enables you to use different security profiles and apply stricter controls to riskier traffic.

The next logical iteration of the policy will look as follows:

	NAME	ACTION	Source		APPLICATION	URL CATEGORY	PROFILE
			USER				
1	managed-apps-file-sharing-risky	✓ Allow	U-file-sharing	AF-file-sharing	high-risk medium-risk		
2	managed-apps-file-sharing-regular	✓ Allow	U-file-sharing	AF-file-sharing	any		
3	managed-apps-email-risky	✓ Allow	U-email	AF-email	high-risk medium-risk		
4	managed-apps-email-regular	✓ Allow	U-email	AF-email	any		
5	managed-url-file-sharing-risky	✓ Allow	U-file-sharing	ssl web-browsing	UM-online-storage-high-risk UM-online-storage-medium-risk		
6	managed-url-file-sharing-regular	✓ Allow	U-file-sharing	ssl web-browsing	online-storage-and-backup		
7	managed-url-email-risky	✓ Allow	U-email	ssl web-browsing	UM-webmail-high-risk UM-webmail-medium-risk		
8	managed-url-email-regular	✓ Allow	U-email	ssl web-browsing	web-based-email		
9	non-managed-app-categories-risky	✓ Allow	known-user	AF-marketing AF-photo-video	high-risk medium-risk		
10	non-managed-app-categories-regular	✓ Allow	known-user	AF-marketing AF-photo-video	any		
11	non-managed-url-categories-risky	✓ Allow	known-user	ssl web-browsing	high-risk medium-risk		
12	non-managed-url-categories-regular	✓ Allow	known-user	ssl web-browsing	any		
13	non-sanctioned-apps	✗ Deny	any	any	any	none	

Figure 5.12 - Core policy principles – third iteration

As you can see, six new rules have been added. They have risk-aligned versions of the URL categories and stricter security profiles. They are also placed above their low-risk versions. This is important. All rule names also have corresponding suffixes to clearly distinguish them in policy and log records.

The policy has begun to take shape. You have effectively covered all sections of the traffic pie chart, except for the **Generic UDP** and **Generic TCP** slices. Put them aside for now—if you do nothing, they will naturally match the rule for non-sanctioned applications, which is precisely what you would typically want in a controlled environment like the one you strive to build here.

What else is missing?

Well, if you have managed web gateways and proxy servers long enough, you will have noticed that HTTP (**web-browsing**) and HTTPS (**ssl**) do not always appear alone on modern websites. **WebSocket** and **Simple Object Access Protocol (SOAP)** protocols often accompany them. Consequently, you need to add corresponding applications, **websocket**, and **soap**, to all our rules that manage access based on URL categories.

Another aspect to consider is that some generic websites are hosted on **Platform-as-a-Service (PaaS)** solutions, such as Google App Engine, AWS Beanstalk, and Azure App Service. The latter two do not have standard application signatures (they are identified by firewalls as regular **ssl** and **web-browsing**), but the former does—**google-app-engine**.

You should manage all PaaS-hosted websites in the same manner as regular websites. Therefore, you must add these applications to all policy rules that manage access based on URL categories.

The **google-app-engine** application has a dependency—**google-base** that, in turn, is highly prolific and also a dependency for many of Google’s own applications falling under different, potentially managed application categories. Let’s add it as well.

While this sounds logical at first glance, you may notice that it somewhat contradicts the main idea depicted in our traffic pie chart because none of these new signatures are truly generic. Even the generic ones—**ssl** and **web-browsing**—belong to an app ID category. If we are not careful enough, they all *may* match an application filter and hit a rule created for managed and non-managed categories above the URL-filtering rules. This would ruin the policy logic. Therefore, this observation must be addressed to retain policy consistency.

Take a closer look at the key attributes of all application signatures that you intend to use for URL-filtering rules.

Application	(Sub)category	Risk	Threats can be detected
web-browsing	internet-utility	4	Yes
ssl	encrypted-tunnel	4	Yes, provided the traffic is decrypted
websocket	internet-utility	3	Limited

Application	(Sub)category	Risk	Threats can be detected
soap	general-business	2	Yes
google-app-engine	internet-utility	3	Yes, provided the traffic is decrypted
google-base	internet-utility	4	Yes, provided the traffic is decrypted

Table 5.1 - Applications for URL-filtering rules

As shown in the table, four applications from your list intended for URL-filtering policy rules are classified as **internet-utility** and have a risk level above 2. When creating an application filter for this category—assuming you treat **internet-utility** as a *non-managed* category—you need to restrict the risk level to 2 or employ one of the following methods to prevent these applications from being included in the **internet-utility** application filter:

- Use a custom tag as one of the filter selection criteria, and make sure this tag is not assigned to **web-browsing**, **websocket**, **google-app-engine**, or **google-base**. The downside of this method is that newly released application signatures won't automatically match the filter and thus will be treated by the policy as non-sanctioned. You will have to manually review applications when they get released by Palo Alto Networks and tag them accordingly.
- Exclude these individual applications from the filter by clicking the **X** icon next to the application name in the application filter definition. This method also has a downside: viewing the excluded applications in the administrative UI is problematic, and not all automation frameworks support this functionality.
- The **ssl** protocol belongs to the **encrypted-tunnel** category, which all best practices advise blocking by default. Following this recommendation, you do not have to do anything else.

Finally, you must address the fact that the **websocket** protocol is essentially a proprietary, often binary, full-duplex data flow. While it is beneficial for the performance and functionality of modern web applications, it is detrimental to security, as the firewall cannot identify threats within this protocol to the same extent that it can in regular HTTP. Thus, you must not allow it on riskier websites.

All of the preceding considerations lead you to create two distinctive application groups: one for regular web browsing and one for riskier websites.

These groups will look as shown in the following screenshot:

NAME ^	APPLICATIONS
APG-web-browsing-regular	web-browsing
	ssl
	google-base
	google-app-engine
	soap
	websocket
APG-web-browsing-risky	web-browsing
	ssl
	google-base
	google-app-engine
	soap

Figure 5.13 - Application groups for regular and higher-risk web browsing

Now, all you need to do is replace **ssl** and **web-browsing** with these new application groups. An example is shown in the following screenshot.

NAME	ACTION	Source		APPLICATION	URL CATEGORY	PROFILE
		USER				
non-managed-url-categories risky	Allow	known-user		AG-web-browsing risky	high-risk medium-risk	
non-managed-url-categories regular	Allow	known-user		AG-web-browsing regular	any	

Figure 5.14 - Risk-aligned application groups in URL-filtering policy rules

Now you understand the following main ideas:

- The policy controls all traffic that the firewall can identify
- It breaks all applications into categories: sanctioned (divided into **managed** and **non-managed**) and **non-sanctioned**.
- Definitions of application filters describing these categories can be as granular as needed by leveraging various application attributes, including **Risk** and **Tags**.

- When the policy cannot identify a specific application, it resorts to generic URL filtering, which uses the exact category-based risk-aligned control mechanism as the application-based part of the policy.
- The policy applies stricter security profiles to riskier URLs.
- The policy automatically allocates newly released applications to respective rules via the application filter mechanism.

Let's call the set of rules we have just built a **dynamic** one. It must be such to reflect the complexities of the internet and to cope with the constantly evolving threat landscape.

This sounds very appealing, but somewhat idealistic and incomplete. In real life, we must undertake a few more steps to make our policy suitable for production use.

In short, we must ensure that core infrastructure, network, and business services and applications dependent on firewall security policy decisions are rock-solid. We cannot afford a critical application or a URL to be recategorized as part of a regular firewall content update, resulting in users losing access to a vital resource.

Crucial policy functionality generally includes, but is not limited to, policy rules that do the following:

- Always allow name resolution for authorized internal hosts (for example, Active Directory domain controllers) while enforcing DNS security
- Always allow authorized network routing protocols for selected endpoints (routers, switches, firewalls, etc.)
- Block known malicious sources and destinations
- Handle security incident response (by isolating a compromised host, for instance)
- Block traffic to/from countries and territories under government sanctions
- Force Chromium-based browsers to use regular TLS protocol instead of QUIC by blocking the latter
- Allow software updates, including firewalls' dynamic updates
- Always allow **Operational Technology (OT)** traffic, such as building controls or CCTV
- Allow core-business applications for all authenticated users (for example, M365 or Google Docs)

The latter category can easily grow to tens or even hundreds of rules, depending on the number of applications used. Each enterprise network will have its unique requirements. The good news is that the vast majority of these applications would usually work just fine through the rules in the dynamic section of our policy, provided you're not too restrictive in the filters' selection criteria.



I suggest you put the book aside and spend 30-60 minutes drafting your unique list of business-critical applications that require dedicated policy rules. Your enterprise network may have specific requirements, but if not, you can use the preceding bullet points as guidelines.

Let's collectively refer to this set of rules, specific to your organization, as **static**.

Since the firewall evaluates policies from top to bottom, more specific rules must always be placed above less specific ones to prevent them from being overshadowed. Therefore, the firewall policy must be organized in the following order (the sequence is crucial):

1. Static rules
2. Dynamic rules

Also, remember that no static rules you create can fully shadow a dynamic one.

For example, suppose you **manage** access to the **cryptocurrency** URL category in your dynamic policy section. In that case, you cannot create a rule in the static section that allows all users access to the same category. However, it's perfectly fine to create a custom URL category with a selection of cryptocurrency websites and then have a rule based on this custom category in the static policy section. The same principle applies to application-based rules.

In the context of the Panorama device group structure, you can place all **static** rules into the **Pre Rules** section of a device group and place **dynamic** rules in the **Post Rules** section of the same group:

1. Device group pre: static rules
2. Device group post: dynamic rules

This approach allows you to introduce region-specific or site-specific rules at lower-level device groups. For example, your enterprise policy may look as follows:

- Level 1 device group pre: global static rules
 - Level 2 device group 1 pre: site-specific static rules for site 1
 - Level 2 device group 2 pre: site-specific static rules for site 2
- Level 1 device group post: global dynamic rules

Now that you understand your policy's structure and core principles, there are a few other key aspects we still need to discuss.

Interface with end users and the IT help desk

Response pages play a crucial role in how your security policy interacts with end users. If a user is blocked for any reason, we want to explain why this happened and make it easy for them to raise the issue with the organization's IT service desk.

Firewall response pages allow us to achieve both of these goals. Let's leverage the built-in variables to deliver contextualized messages that explain why access has been blocked. The response pages your policy generates will address scenarios such as:

- The user attempted to access a managed *App-ID* category, but was not a member of the AD group associated with the category
- The user attempted to access a managed *URL* category, but they were not a member of the AD group associated with the category
- The user attempted to access a non-sanctioned application
- The user was not authenticated
- A file download or upload was blocked
- Malware was detected
- Sensitive data was detected in a file
- A TLS certificate error occurred

Refer to *Table 4.2* in the *Response pages* section of *Chapter 4, PAN-OS Security Policy Features: Connection Processing* to see what variables are available in each response page type. As you can see, you have many variables at your disposal. However, the exact set of available variables differs from page to page.

While it is easy to display a message saying, **You are not authorized to access cryptocurrency websites** because you can use the `<category/>` variable in **URL Filtering and Category Match Block Page**, the same approach will not work for application categories because there is no equivalent variable available for use in **Application Block Page**.

However, you can work around this issue by leveraging another variable: `<rulename/>`.

Divide your generic blocking rule for all non-sanctioned applications into specific rules for each application category. Each rule will only block applications that match the filter for that category. You can incorporate the application category name directly into the rule name and use JavaScript embedded in the response page to extract this name (as a substring) from the runtime value of the variable.

The blocking rules at the bottom of the policy will now look as follows:

	NAME	ACTION	Source		SERVICE
			USER	APPLICATION	
1	managed-apps-file-sharing-risky	Allow	U-file-sharing	AF-file-sharing	application-default
2	managed-apps-file-sharing-regular	Allow	U-file-sharing	AF-file-sharing	application-default
12	non-managed-url-categories-regular	Allow	known-user	AG-web-browsing-regular	application-default
13	not-authorized-for-file-sharing	Deny	known-user	AF-file-sharing	application-default
14	not-authorized-for-email	Deny	known-user	AF-email	application-default
15	non-sanctioned-apps	Deny	any	any	any

Figure 5.15 - Category-specific blocking rules for processing by response page JavaScript code

As the screenshot demonstrates, the policy should have two new blocking rules—one for each managed category. Other than the **ACTION**, **NAME**, and **Source USER**, these rules are exact replicas of the **Allow** rules for the same application categories.

The JavaScript code for the application response page to handle these new rules will be as follows:

```
<html>
...
<body><h1 id="headingText">Application blocked</h1>
<p><b>User:</b><user/><br><b>Application:</b><appname/></p>
<p id="messageText">This is a generic firewall response page - enable
JavaScript in your browser to see a detailed error message</p>
<script>
  var servicedesk = 'https://servicedesk.example.com/sp?';
  var rule      = '<rulename/>';
  var category  = rule.replace('not-authorized-for-', '');
  category      = category.replace('-', ' ');

  if (rule.includes('not-authorized-for-')) {
    document.getElementById("headingText").innerHTML = "Not authorized";
    document.getElementById("messageText").innerHTML = "You are not
authorized to access '" + category + "' applications. Click <a href='" +
servicedesk + "user=<user/>&reason=not-authorized&category=" + category +
"&details=<appname/>'>here</a> to request approval.";
  }
  ...
</script></body></html>
```

Let's discuss this in detail.

Imagine a user named **Joe** who is a member of the **U-file-sharing** group. Suppose he accesses a file-sharing application, **boxnet-base**, that matches the **AF-file-sharing** application filter. In that case, his connection will naturally hit the **managed-apps-file-sharing-risky** rule or **managed-apps-file-sharing-regular**.

What if we remove the user from the group and ask him to access the same application again?

The firewall will identify the user and the application. However, instead of matching one of the first two rules bound to an Active Directory group, the connection will now match the **not-authorized-for-file-sharing** deny rule. This rule's matching criteria differ from the other two rules in the **Source USER** attribute.

The firewall will block the connection and return its response page to the user's browser instead of the original web server response. When generating the response page, the firewall will substitute variables specified in the response page template with their runtime values, such as actual username (**Joe** in place of `<user/>`), application name (**boxnet-base** in place of `<appname/>`), and the rule name above (**not-authorized-for-file-sharing** in place of `<rulename/>`). The browser will receive and render the HTML code of the response page and execute the embedded JavaScript code along the way.

The JavaScript code will take the **not-authorized-for-file-sharing** string and convert it to **file sharing** by stripping the suffix and replacing the dash with a space. Then, it will replace the generic default header text with the **Not authorized** string and the default message paragraph with the You are not authorized to access '**file sharing**' applications. Click `<a href='https://servicedesk.example.com/sp?user=Joe&reason=not-authorized&category=filessharing&details=boxnet-base'>here</a>` to request authorization string.

The browser will render the resultant HTML code as follows:

```
Not authorized
User: Joe
Application: boxnet-base
You are not authorized to access 'file sharing' applications. Click here
to request authorization.
```

Then, the user may click the generated link, initiating an HTTP GET request with embedded parameters: **username, application name, application category name, and reason for blocking**. The request will be sent to the company's IT service desk portal, which will process it and present the user with a pre-populated form based on the metadata submitted in the HTTP request. Then, the user will submit the form to initiate a request for approval workflow. As a result of the workflow, the service desk system will include (subject to approvals) the user in the AD group associated with the managed category and notify the user.

This is how we link our policy actions with the enterprise IT service desk system.

What if the user tries to access a non-sanctioned application, or the firewall does not have an IP-to-user mapping? Let's handle these use cases by adding a few more policy rules and JavaScript code.

	NAME	ACTION	Source		APPLICATION	SERVICE
			USER			
1	managed-apps-file-sharing-risky	Allow	U-file-sharing		AF-file-sharing	application-default
2	managed-apps-file-sharing-regular	Allow	U-file-sharing		AF-file-sharing	application-default
12	non-managed-url-categories-regular	Allow	known-user		AG-web-browsing-regular	application-default
13	not-authorized-for-file-sharing	Deny	known-user		AF-file-sharing	application-default
14	not-authorized-for-email	Deny	known-user		AF-email	application-default
15	non-sanctioned-file-sharing	Deny	U-file-sharing		AF-file-sharing-all	any
16	non-sanctioned-email	Deny	U-file-sharing		AF-email-all	any
17	non-sanctioned-app-category	Deny	known-user		any	any
18	non-authenticated-connection	Deny	unknown		any	any

Figure 5.16 - Additional blocking rules to handle non-sanctioned applications and anonymous connections

The screenshot shows four additional rules: **non-sanctioned-file-sharing**, **non-sanctioned-email**, **non-sanctioned-app-category**, and **non-authenticated-connection**.

The former two rules leverage new application filters, which, as their names suggest, include *all* applications from the respective managed categories. Therefore, they will only trigger for applications that do not match the application filters we used in the respective **Allow** rules. In other words, they will match only **non-sanctioned** applications. We've also changed the **SERVICE** value from **application-default** to **any**. Thus, the same two rules will match applications that use non-standard port numbers.

The **non-sanctioned-app-category** rule will only be reached for applications from non-sanctioned categories, regardless of the port numbers they use. Finally, the **non-authenticated-connection** rule will trigger on any connection, provided the firewall does not have an IP-to-user mapping for the connection source because the **Source USER** attribute is set to **unknown**.

We can handle these new rules by expanding the `if` statement in the response page JavaScript code. We will look for two additional substrings—**non-sanctioned** and **non-authenticated**—in the `<rulename>` variable and change the error message header and body accordingly.

Similarly, we can handle other errors, such as blocked files or malware, blocked URL categories, and TLS-related issues. Each of these use cases will be handled by its own response page type. The verbiage will be contextualized, and the **IT Service Desk system** hyperlink will be populated with relevant metadata.

For complete response page templates, refer to the book's GitHub repository.

Now that we have determined the principles that our policy must follow, it's time to turn our attention to some of its less obvious yet important aspects.

Tagging and naming convention

Two essential ingredients are often missing from the firewall configuration kitchen. They may seem optional at first glance; however, the more your environment grows, the more you benefit from two simple traits of your firewall policy:

- Uniform naming of all policy objects
- Tagging of policy elements for their logic grouping and visual aesthetics

This section will introduce a naming convention and tagging to further improve your policy.

Naming convention

While naming conventions may seem like a nice-to-have but not absolutely necessary, in large corporate environments with multiple administrators in charge of making firewall changes, a good naming convention is a *must-have*.

In addition to the apparent benefit of a clean and appealing look in the administrative UI, a good naming convention will help you avoid duplicate objects, easily locate policy objects, simplify automation, enhance policy readability, and expedite troubleshooting.

Each firewall administrator has their preferences, and there is no universal convention that will make everyone happy. For this book, let's follow the convention outlined here:

- Each policy object, except the policy rules, will have a capitalized prefix acronym indicating the type and meaning of the object:
 - **H-** (Host), **N-** (Network), and **R-** (Range) for the **IP Netmask** type of address objects
 - **W-** (Wildcard) for the **Wildcard** type of address objects
 - **FQDN-** for the **FQDN** type of address objects
 - **AG-** for static Address Groups
 - **UG-** for static User Groups
 - **DAG-** for Dynamic Address Group
 - **APF-** for Application Filters
 - **APG-** for Application Groups
 - **UL-** (URL List) for custom URL categories of the **URL List** type
 - **CM-** (Category Match) for custom URL categories of the **Category Match** type
 - **AVP-** for Antivirus Security Profile
 - **ASP-** for Anti-Spyware Security Profile
 - **VPP-** for Vulnerability Protection Security Profile
 - **UFP-** for URL Filtering Security Profile
 - **FBP-** for File Blocking Security Profile
 - **WFP-** for WildFire Analysis Security Profile
 - **DFP-** for Data Filtering Security Profile



You may notice some inconsistencies in the prefixes I use for certain object types across the book, both in text and in screenshots. For instance, Application Filters may appear with the prefix **AF-** in some places and **APF-** in others. The intended meaning should always be clear from the context, but I want to acknowledge the discrepancy.

- A short object description will follow the prefix. It will be in a lowercase and abbreviated if necessary. Dashes and underscores will separate words to avoid spaces.

We may need to abbreviate and shorten words because of PAN-OS limitations on the maximum length of some object names. For example, when you get to automating the creation of risk-aligned versions of your managed URL categories, you won't be able to simply run in a loop through all categories, concatenating a prefix with the category names, because you will hit the maximum length of 31 characters very quickly:

The screenshot shows the 'Custom URL Category' configuration interface. The 'Name' field is filled with 'CM-online-storage-and-backup_high-risk'. A red error message box is overlaid on the 'Description' field, stating 'The maximum length for this field is 31'. Below the 'Type' field (set to 'Category Match'), there is a section titled 'Matches all of the following categories'. This section includes a search bar, a '2 items' indicator, and a list of categories: 'CATEGORIES', 'online-storage-and-backup', and 'high-risk', each with an unchecked checkbox.

Figure 5.17 - Object name length limitation example

Another disadvantage of having long custom object names (such as custom URL categories, EDLs, Panorama device groups, etc.) is that they may be truncated in logs or partially hidden in non-resizable drop-down lists in the firewall or Panorama administrative UI, making reporting and troubleshooting more difficult.

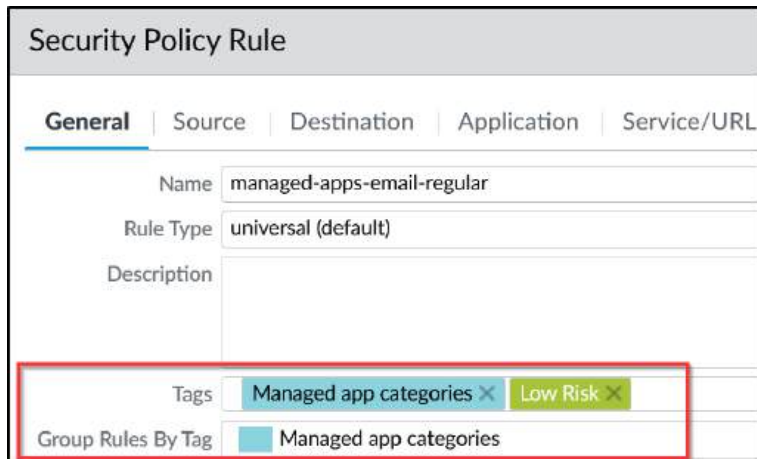
- Policy rules will have no prefixes, will always be in lowercase, and will use dashes and underscores instead of spaces.

Tagging

As you learned in *Chapter 3*, many objects can have custom tags assigned to them. This feature is invaluable. It allows you to visually label and logically distinguish policy elements such as address objects, address groups, services, service groups, and, most importantly, policy rules.

Although it might appear insignificant in small setups, a policy expanded to hundreds or thousands of rules becomes almost unreadable without tags.

As the following screenshot shows, every policy rule can have several custom tags assigned to it, including one **group** tag.



Security Policy Rule

General | Source | Destination | Application | Service/URL

Name: managed-apps-email-regular

Rule Type: universal (default)

Description:

Tags: Managed app categories × Low Risk ×

Group Rules By Tag: Managed app categories

Figure 5.18 - Policy rule tagging

Once you have assigned the desired tags, you can switch the default policy view to a tag-based or group tag-based view. This can be done on the **POLICIES** screen, as shown in the following screenshot:

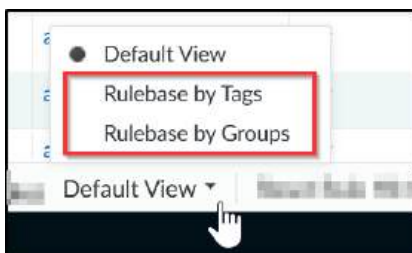
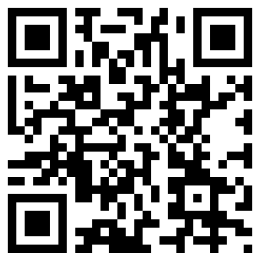


Figure 5.19 - Tag-based policy view

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



The **Rulebase by Tags** option activates a so-called **Tag Browser** that you can use to dynamically change the policy view to include only the rules with tags you select. The **Rulebase by Groups** option creates a folder-like representation of your policy sections, indicating a number of rules in each *folder*. You can click on any of these *folders* and see only the rules from the respective group.

Rest assured that this feature changes only the policy view for the administrator but not the actual order of policy rules deployed to the respective firewalls. You can always flip the switch back to the default view.

Now that we've covered these essential aspects, let's set aside our firewalls for a moment and briefly discuss external dependencies.

External dependencies: EDLs and user identity

Any enterprise firewall policy will inevitably depend on external systems. Ours is not an exception—at the very least, it needs a reliable source of user identity and entitlements, as well as hosting for **External Dynamic Lists (EDLs)**—the invaluable feature for ad hoc exceptions and incident response that we covered in *Chapter 3*.

This section will cover the sources of user identity and EDL hosting.

User identity

User authentication and identification are complex topics, and we cannot cover them here in any more detail than they are already covered in the original *PAN-OS Administrator's Guide*, Palo Alto Networks, and in the *Identifying Users and Controlling Access* chapter of *Mastering Palo Alto Networks*, Tom Piens, Packt Publishing.

Let's just remember that there are several sources and methods of user identification available to us: Captive Portal, GlobalProtect, server log monitoring, port mapping on terminal servers, syslog listening, Cloud Identity Engine, XML API, XFF header parsing, and endpoint probing. All of them can be configured or referenced in the **DEVICE | User Identification** section of the firewall local configuration or a Panorama template.

Once the firewall becomes aware of a user's identity, it can enumerate all the groups to which the user belongs. Once we know all the groups the user is a member of, we can match the user's connections to the correct rules in our security policy. Group enumeration is configured in **DEVICE | User Identification | Group Mapping Settings**, which, in turn, references LDAP integration defined in **DEVICE | Server Profiles | LDAP**. As mentioned in the *Introduction to templates and template stacks* section earlier in this chapter, all these settings can be stored in a single functional template that you can subsequently reference in template stacks of sites you cut over to the new policy.

Establishing users' group memberships is entirely independent of the user identification process. Therefore, when you change a user's group membership, you do not need to ask them to log off and log back in—the firewall will pick up this updated membership information at the next scheduled LDAP refresh.

External Dynamic Lists

The EDL, is a handy and seemingly simple feature that, unfortunately, proves difficult to scale in an enterprise environment. First, you have the regrettable limitation of a maximum of 30 EDLs you can reference in a policy.

Next, only IP-based EDLs can have inline comments. The format of the URL and domain-based EDLs does not allow us to embed any comments about the entries added to the lists over time. This presents a significant challenge for large corporate environments with multiple engineers and analysts modifying the EDLs. If you rely on these text files alone, you can never tell who made a change, when, and why.

One possible solution to this problem is to host EDLs in a **Version Control System (VCS)** where permissions and versions can be strictly controlled, and all changes can be recorded, commented on, and traced back to the individuals who made them. However, it may not be the best idea to subsequently make firewalls pull the EDLs directly from the repository (and not all VCSs would allow you to do so anyway) because they are not designed to operate as web servers.

Thus, you may end up building a custom solution in which, for example, each change committed to the master branch of your EDL repository initiates a CI/CD pipeline that pulls all files to subsequently host them on a web server running in a Docker container.

Regardless of how long we try to avoid the next topic, it will inevitably come up on our security agenda. So we'd better start sooner rather than later.

TLS decryption

According to Google's Transparency Report, the percentage of encrypted pages fetched by Chrome browsers worldwide currently ranges from 81 to 97, depending on the operating system. If you do not inspect encrypted web traffic on your firewalls, you're pretty much blind to all modern threats. Additionally, firewalls cannot identify applications and application functions determined by metadata and attributes hidden in the encrypted payload of network packets.

This topic is vast and complex, and it deserves a book of its own. However, whether you perform decryption in your network or not, the security policy principles we introduced earlier in this chapter remain the same. Let us cover the essentials of decryption relevant to web gateways, such as the one we build in this book.

Next-generation firewalls support three types of traffic decryption:

- SSL Forward Proxy
- SSL Inbound Inspection
- SSH Proxy

Fundamentally, they all do the same thing—when an encrypted connection is detected, the firewall attempts to decrypt it, inspect the decrypted payload, re-encrypt it after inspection, and forward the encrypted packets to the destination. However, each of these decryption methods covers a unique use case. Before we delve deeper into this topic, let's explore how **Public Key Infrastructure (PKI)** and the **Transport Layer Security (TLS)** protocol—the web encryption ecosystem—work.

PKI and TLS basics

PKI relies on **asymmetric encryption** technology. This technology uses an encryption algorithm with two interlinked keys—public and private. The public key encrypts data, and the associated private key decrypts it. As its name suggests, the public key is not a secret—every web server exposes it to the world as part of a **certificate**. On the contrary, the private key is known only to the server. Anyone who wishes to send an encrypted message to a server must retrieve the server's public key first. This retrieval takes place during the TLS protocol handshake phase.

A client establishes a TCP connection to a server and sends a TLS Hello message with an embedded server name and a list of its capabilities, such as the protocol version used, supported ciphers and signature algorithms, compression methods, and so on. The server responds with its choice of the client's capabilities and, most importantly, its certificate, also known as a **leaf certificate**.

The certificate is issued by a **Certificate Authority (CA)** organization beforehand in response to a **Certificate Signing Request (CSR)** generated on the web server. Several well-known public CAs exist, such as VeriSign, DigiCert, IdenTrust, and Let's Encrypt. The CSR contains the public key, subject name, and other attributes for which the server's administrator wants the CA to issue a certificate.

The certificate is a data structure that contains the server's public key, the name of the certificate issuer, the certificate subject and purpose, and other metadata. The type of certificate used in the TLS protocol is known as the **x509** certificate. The certificate is digitally signed with the issuer's private key.

When the server responds to the client's TLS Hello message, it also sends the issuer's certificate, known as the **intermediate certificate**, so that the client can verify the signature of the leaf certificate. Each intermediate certificate, in turn, is issued by a higher-level intermediate CA or a **root CA**. The latter always uses self-signed certificates. These self-signed certificates are at the end of the chain of certificates, where each certificate proves the authenticity of the certificate before it.

Since the root certificates are always self-signed, the *only* way for the client to trust them is to acquire them beforehand. Software vendors have special operating procedures that allow them to retrieve root certificates from trusted, well-known public CAs and securely embed them into OSes, browsers, and other software.

When a client connects to a web server, and the latter presents a certificate chain to prove its own authenticity, the client always ignores the root certificate if one is present in the chain. The client verifies the trustworthiness of the whole chain, invariably looking for the certificate of the respective root CA *in its own Root Certificate Store*.

Here is what the root certificate store looks like in Windows:

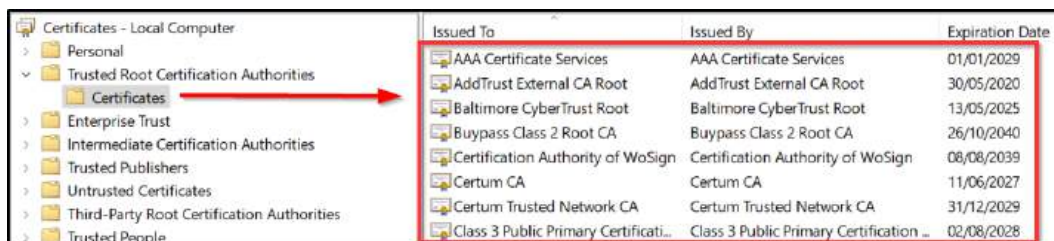


Figure 5.20 - The root certificate store in Windows OS

After the certificate chain is validated as **Trusted**, the client also checks whether any certificates in the chain have been revoked by their respective issuer. The two most common methods to do so are the **Online Certificate Status Protocol (OCSP)** and the **Certificate Revocation Lists (CRLs)**. Browser vendors, such as Google, may implement proprietary certificate revocation check mechanisms that are not reliant on the aforementioned standard protocols. This is done mainly for performance reasons.

When your browser tells you that a website is **secure**, this means the following:

- The browser entrusted the OS vendor to retrieve and maintain a list of root certificates of public CAs
- The OS vendor trusted the security controls and procedures of the public CAs enough to consider their root certificates non-compromised and authentic; then, it embedded these certificates into the OS
- The web server administrator generated a private-public encryption key pair and had a public CA to execute the public key as a certificate
- When the CA issued the certificate, it used a standard operating procedure to validate the ownership of the domain the certificate was requested for
- The web server was configured to present this certificate along with certificates of the issuing CAs to any client wanting to connect to the server over the TLS protocol

- Upon connection to the website, the browser successfully validated the presented certificate chain to one of the root CA certificates in its local certificate store
- The browser also ensured that the certificate has not expired and has not been revoked by the issuer



Sometimes, software application vendors rely on their own root certificate stores instead of the OS's built-in one. For example, Java runtime and Firefox have their own root certificate stores.

The outcome of a successful TLS handshake is the generation of the so-called **session key**. This key is used for subsequent **symmetric** encryption/decryption of all data exchanged between the client and the server.

Thus, relatively slow asymmetric PKI-based cryptography is used to securely generate and exchange a key for very fast and efficient symmetric cryptography, where a single key is used to encrypt and decrypt all data.

Let's discuss how our firewall can penetrate this ecosystem, decrypting and re-encrypting TLS connections while convincing clients that their connections are still secure.

SSL Forward Proxy

This decryption method is the most popular and relevant for web gateways. The mention of the SSL protocol in this method's name is misleading because this protocol is obsolete and insecure, and thus barely used anywhere nowadays. In fact, this method decrypts both the TLS and the legacy SSL protocols.

When a client attempts to connect to a website over TLS, the firewall, instead of passing the traffic through unchanged, effectively establishes two TLS connections over a single TCP connection: one connection between the firewall and the server and another one between the firewall and the client.

The firewall effectively becomes a transparent proxy between the client and the server. When the TLS handshake takes place between the firewall and the server, it's a regular handshake, identical to what we studied in the previous section. The most interesting part of this proxy session is the handshake between the client and the firewall.

When the client starts the handshake, it is completely unaware of the firewall's presence. It expects to receive a valid certificate from the server and consequently expects that the server (the firewall, in reality) will be able to decrypt with its private key all TLS handshake metadata that the client has encrypted using the public key from the server's certificate. The problem is that if the firewall were to simply present the original server certificate to the client, it would not be able to decrypt any of the client's handshake messages because it does not have the original private key (that never leaves the server).

The solution to this dilemma is for the firewall itself to become a CA trusted by the client and dynamically generate certificates for every website that needs to be decrypted.

None of the well-known public CAs will ever issue you a certificate that effectively allows you to become its sub-CA and consequently issue certificates for any domain in the world. Therefore, the only two solutions at your disposal are the following:

- Provision your own enterprise PKI and issue a sub-CA certificate for the firewalls to perform decryption
- Make the firewall a CA in its own right and make clients trust it

The former is often already the case, especially in Windows-based corporate environments. Therefore, you won't need to do anything to make domain-joined computers trust the firewall's issued certificates because they already trust the enterprise root CA.

The latter will require additional steps to distribute the firewall's self-signed root CA certificate to all clients whose traffic will be decrypted.

Regardless of your chosen method, once the firewall has a CA certificate trusted by the client, it can successfully issue certificates for the client-to-firewall leg of every decrypted TLS connection. The clients, in turn, won't see a difference between the original server certificate and the server certificate issued by the firewall because they trust both.

There are scenarios when this decryption method won't work. These include cases when an application uses certificate pinning, certificate-based client authentication, or encryption ciphers not supported by the firewall. For that reason, the firewall decryption policy must have mechanisms to make exceptions.

SSL Inbound Inspection

The **SSL Inbound Inspection** method is needed when we must decrypt TLS traffic from the Internet to a web server hosted behind our firewall. In this use case, we have no control over the clients as they do not belong to our organization. This means we cannot make these clients trust the CA certificate that our firewall uses to re-encrypt traffic after decrypting it for inspection. Thus, the SSL Forward Proxy method won't work in this scenario.

The only solution when we have no control over the clients' trust is to allow the firewall to eavesdrop on the client-server TLS handshake and establish the symmetric key the same way the server does. In order to do so, the firewall *must* have the original private key.

We won't go into further details of this method in this book. However, if you'd like to learn more about it, you can go to <https://docs.paloaltonetworks.com/pan-os/10-2/pan-os-admin/decryption/configure-ssl-inbound-inspection> to further explore its configuration and caveats.

SSH Proxy

Secure Shell (SSH) is a complex protocol used for many purposes: secure interactive shell for remote system administration, file transfer, and tunneling of other protocols. Similar to TLS, it can use x509 certificates, but more often, it uses SSH certificates or SSH key pairs.

SSH certificates are purpose-designed for the SSH protocol and are typically used in environments where the organization controls the CA. The trust model is more straightforward, without a need for a hierarchical chain of trust typical for x509 certificates used in TLS. Trust is established by directly configuring the server with the CA's public key.

The more popular (albeit poorly scalable) authentication is key-based. The server provides its host key to the client, who verifies the server's host key against its list of known hosts. If the key doesn't match or is not recognized, the client may prompt the user to take action.

Unfortunately, when configuring our firewall to decrypt SSH traffic, we cannot link it to the SSH CA. The firewall automatically generates a key that it uses to encrypt the client-firewall leg of the proxied SSH connection.

We won't discuss this method in this book. However, if you'd like to learn more, you can go to <https://docs.paloaltonetworks.com/pan-os/10-2/pan-os-admin/decryption/configure-ssh-proxy>.

Now, as we briefly discussed the decryption concepts, let's run through the main elements of the decryption policy that will support the core security policy designed earlier in this chapter.

Decryption policy

Similar to the security policy rule, each decryption policy rule consists of connection attributes that all must match, the type of decryption, and the desired action (*decrypt* or *not decrypt*). The policy is evaluated from top to bottom, and the first match wins—the policy processing for a connection stops after the first match is found.

Unlike the security policy, the matching criteria available in the decryption policy lack the **Application** attribute because decryption is engaged at the very beginning of a connection—after the TCP session is established, but before the firewall has a chance to properly identify the application. All you can specify is the port number (**Service**) and **URL Category** (since it can be identified from the URL present in the SNI field of the client's TLS Hello message).

When we design a decryption policy, it does not have to be dynamic in the same sense that the security policy has to be. Yet, we'd like it to be as flexible as possible. It must allow us to make exceptions on all possible matching conditions:

- Source IP addresses
- Destination IP addresses
- URLs

We can implement these exceptions using **External Dynamic Lists** (refer to *Chapter 2* for more details about this feature).

The decryption itself should be applied to users based on AD group membership to allow for a staged rollout. Let's also focus on decryption on TCP port 443.

What would also be extremely helpful is a mechanism that automatically exempts destinations from further decryption if we attempted to decrypt them but failed to do so because of unsupported parameters or other errors. We can achieve this by leveraging the **Built-in Actions** feature of **Log Forwarding Profiles** and **Dynamic Address Groups**.

The action we will create will look for entries in the traffic log with the session end reason set to **decrypt-error** or **decrypt-unsupported-param** and tag these addresses with a special custom tag. Then, we'll create a **Dynamic Address Group** matching all addresses with this tag and use this group as a destination in a decryption policy rule with the **no-decrypt** action.

The following screenshot demonstrates how this can be configured in a log forwarding profile.

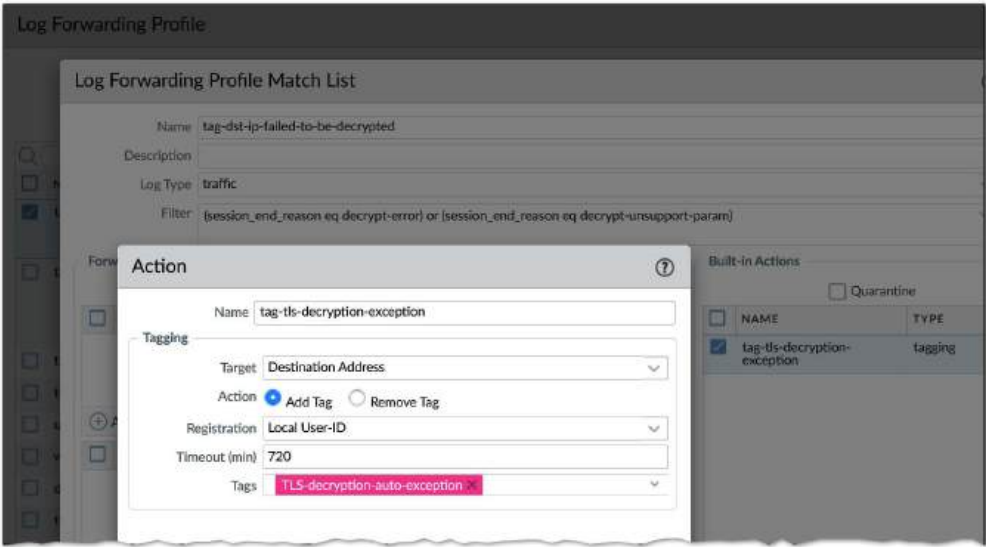


Figure 5.21 - Automatic tagging of IP addresses that failed to be decrypted

As you can see, we need to specify the following:

- Tag name (you can assign more than one tag)
- How long the tag is assigned for
- What address we tag (source or destination)
- Where we perform the registration (on the firewall itself, on Panorama, or another user ID agent)

With all the preceding considerations in mind, our minimally required decryption policy becomes as shown in the following screenshot.

NAME	Source		Destination		URL CATEGORY	ACTION
	ADDRESS	USER	ADDRESS			
1 de-not-decrypt-src-ip-manual-exceptions	EDL-tls-decryption-exceptions-src-ip	any	any		any	no-decrypt
2 de-not-decrypt-dst-ip-manual-exceptions	any	any	EDL-tls-decryption-exceptions-dst-ip		any	no-decrypt
3 do-not-decrypt-dst-ip-auto-exceptions	any	any	DG-tls-decryption-auto-exceptions		any	no-decrypt
4 do-not-decrypt-url-manual-exceptions	any	any	any		EDL-tls-decryption-exceptions-url	no-decrypt
5 default decryption	any	U-tls-decryption	any		any	decrypt

Figure 5.22 - Minimally required TLS decryption policy

Unfortunately, the book's format does not allow us to depict all policy rule fields that must be shown while keeping the text in the screenshot readable. Just bear in mind that each of the shown policy rules also has relevant source and destination zones, decryption type set to **ssl-forward-proxy**, decryption and log-forwarding profiles assigned, and it is also configured to log unsuccessful TLS handshakes.

Depending on the type of applications you have in the static section of your security policy, you may need to include a few more exceptions. For example, certain SaaS applications, such as Microsoft's M365 suite of office applications, require some of their URLs to be excluded from TLS inspection for performance reasons.

We've almost finished discussing the key aspects of our policy. Only three important topics remain uncovered: legacy application signatures, legacy policy rules, and logging.

Legacy custom application signatures and policy rules

As we build our new policy, we must not forget that it is going to replace a legacy policy that may have many rules for business-critical applications, potentially complemented by custom application signatures.

Custom application signatures can easily be included in our new enterprise policy. We will create a new custom tag specifically for all these legacy application signatures and tag them all. Then, we will create an application filter with this tag as the only selection criterion. This application filter, in turn, will be used in a single **Allow** rule in the static section of our security policy.

One benefit of this approach is that we can apply identical policies to all web gateways in our organization, regardless of the site-specific custom application signatures created in the legacy policies on these firewalls. For example, the fictitious firm we introduced in *Chapter 1, Next-Gen Firewall Fundamentals* can have a custom application signature for a local accounting app in India, a signature for a franking machine in London, and five extra signatures created by interns for the Canadian office. All these apps can be easily tagged ahead of the cutover to the new policy, while the policy itself will remain identical.

The unique policy rules may be more challenging to address. In the book's final chapters, we will discuss one possible methodology. In short, we will need to disable all generic (aka *any-any*) legacy rules to ensure that none of them fully shadow rules in the dynamic section of our new policy. Meanwhile, the non-generic legacy rules (those that have specific destination addresses and/or custom URL categories as matching criteria or non-generic app ID signatures) can be placed

in a site-specific device group so that they appear right in the middle of the resultant firewall policy—between its global static and dynamic sections.

To understand how this will pan out, revisit the *Panorama 101: device groups and templates* section earlier in this chapter, where we discussed the hierarchy of device groups.

You’ve almost reached the end of the chapter. How about we finish it by discussing traffic logging and finally start converting our ideas to code in the following chapters?

Logging

Logging is one of the cornerstones of information security. Consistent log records go a long way in making any network troubleshooting or security incident investigation easier.

While it is important to log as much information as possible, there can also be performance and log retention concerns. The more we log, the shorter the time span of logged events, considering that the firewall and Panorama disk space are limited.

Since our policy is risk-centric, we can leverage this in the URL-filtering profiles by ensuring that for low-risk URLs, we use a profile that logs only the container page and the **User-Agent** HTTP header, while for riskier websites, we use a profile that logs the full URL and **User-Agent**, **Referer**, and **X-Forwarded-For** headers.

Similarly, when we detect the submission of user credentials, we can use different severity levels of log records depending on the risk level of the URL where the credentials are being submitted.

Another important aspect of logging is the URL category that the firewall logs in the **CATEGORY** field of the URL filtering logs. As you remember from *Chapter 2*, each URL can belong to up to four different categories. *All* of them will always be logged in the **URL CATEGORY LIST** field, and *one of them* that the policy matched will be logged in the **CATEGORY** field.

However, we do not want the latter field to be populated with a mixture of risk, custom, and content/behavior categories. Unlike the **URL CATEGORY LIST** field, the **CATEGORY** field can be used in custom reporting (**MONITOR | Manage Custom Reports**). If you attempt to group report entries by category, you’ll quickly discover that **URL CATEGORY LIST** is entirely unsuitable because the built-in reporting engine cannot group using the individual categories within this complex multi-value field.

Realistically, you can leverage only the **CATEGORY** field, and the values must be consistently of the same type (i.e., either only content/behavior, risk only, or custom only, but not a random mixture of all these types). The most logical and helpful choice here is to log only the URL's content/behavior category while ignoring the risk and custom categories.

RULE	URL	CATEGORY	URL CATEGORY LIST
non-managed-uri-categories-regular	example.com/	computer-and-internet-info	URL-LIST-example.com,computer-and-internet-info,low-risk

Figure 5.23 - Logging of URL categories

We can achieve this by setting the **None** action against the custom categories and the **Allow** action against the risk categories in our policy's URL-filtering profiles.

We have already discussed another important logging feature: the **Built-In Actions** feature of **Log Forwarding Profiles**. Let me just add here that we can use it not only to tag IP addresses that failed decryption but also to tag addresses that exhibited dangerous behavior (i.e., a virus was detected, or a suspected Command and Control communication was observed).

In addition to the tagging of potentially compromised endpoints, we can reference **HTTP profiles** in **Log Forwarding Profile Match Lists** to raise security incidents in our enterprise ticketing system. You can refer to *Chapter 2* for more details about this feature.

Summary

In this chapter, we explored how Panorama device groups can be used to organize firewall policies in a hierarchical manner and the role of templates and template stacks.

Then, we introduced the core principles of our security policy and gradually expanded on them using PAN-OS features discussed in *Chapter 2, Navigating Real-World Firewall Management and Cyber Risks* to meet the requirements outlined in *Chapter 1, Next-Gen Firewall Fundamentals*. We discussed the importance of naming conventions and tagging, external dependencies, the interface with our end users, and integration with a service desk system.

We covered TLS decryption, focusing on the SSL Forward Proxy method as a significant contributor to the security performance of our policy. Finally, we introduced a method for dealing with legacy custom application signatures and policy rules and reiterated essential aspects of PAN-OS features related to logging.

The upcoming chapter will delve into firewall automation and management. We will compare the web UI and CLI, talk about CLI automation, discuss available APIs and SDKs, and briefly review our options with Ansible and Terraform.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

***Note:** Keep your purchase invoice ready before you start.*



Part 2

Firewall Policy Automation

This part begins with an overview of the automation landscape for Palo Alto Networks firewalls. In *Chapter 6*, you will explore all the available management options—from traditional CLI scripting to XML and REST APIs, SDKs, and infrastructure-as-code tools such as Ansible and Terraform. This chapter provides the context you need to choose the right automation approach for your environment.

The rest of this part is entirely practical. In *Chapter 7*, you will set up a complete development environment with Python, GitHub, PyCharm, Docker, and GitHub Actions, preparing the foundation to work with the code used throughout the book. *Chapter 8* introduces the fundamentals of translating firewall policies into code, including repository structure, SDK usage, and performance optimization. Finally, *Chapter 9* builds on this foundation with advanced techniques, covering dependencies, debugging, strict multi-config transactions, and extending SDKs with XML API calls.

This section contains the following chapters:

- *Chapter 6, Firewall Automation and Management Choices*
- *Chapter 7, Setting Up Your Software Development Environment*
- *Chapter 8, Policy to Code: Foundations*
- *Chapter 9, Policy to Code: Advanced*

6

Firewall Automation and Management Choices

By Migara Ekanayake

Imagine being responsible for the management of a fleet of firewalls spanning across physical data centers and multiple cloud environments with thousands of rules. How do you ensure consistent policy management across your environments while adhering to security best practices? How do you keep up with the increasing number of firewall changes with a limited number of resources? This segues our conversation into the topic of automation and how automation has become a necessity rather than a convenience.

Organizations are moving fast, and manual firewall configuration and management are no longer sustainable. Organizations need robust, scalable, and efficient ways to automate their security infrastructure.

This chapter will walk you through how the automation landscape has evolved over the years, map that onto Palo Alto Networks terminology and tools, and present you with the pros and cons of each approach. By the end of this chapter, you will have an understanding of the methods and tools available to you and be able to choose the right tool to solve your use cases.

In this chapter, we will cover the following topics:

- Evolution of firewall management
- APIs: The foundation of automation
- Software development kits
- Infrastructure-as-code tools

- Choosing the right automation approach
- Best practices and implementation guidelines

Evolution of firewall management

The management of firewalls has evolved significantly over time due to the increasing complexity of network environments and the demand for greater agility and efficiency. This section outlines this evolution by highlighting key shifts in management paradigms. You will examine a range of management options, from traditional, manual methods to modern, automation-focused approaches, to provide a foundation for understanding the tools and techniques available for Palo Alto Networks devices. The following list presents a set of management options that can be incorporated into your automation workflow:

- **Command-Line Interface (CLI)-based management:** Using text-based commands to configure and monitor devices
- **Web interface (GUI) management:** Interacting with the firewall through a graphical, browser-based portal
- **API-driven automation:** Using code to programmatically control and manage the firewall, enabling integration with other tools
- **Configuration as code:** Managing firewall settings through code and **version control systems (VCSs)**, treating configuration like software
- **Intent-based networking:** Defining the desired outcome (the “intent”), and letting the network automatically translate that into the necessary configuration and policy enforcement

These management choices are easier to start with, without additional tools or libraries, and might be the best approach for small-scale deployments, increasing efficiency and reducing the time spent on repetitive tasks. However, these approaches require a human operator to orchestrate and require domain-specific knowledge.

CLI-based management

All the Palo Alto PAN-OS devices contain a CLI that you can access via SSH or the serial console, which allows you to configure and monitor your devices. There are two CLI modes – operational and configuration. Operational mode enables you to view the device status, monitor traffic traversing through a firewall, and manage software processes. Configuration mode provides you with the ability to view and modify the configuration. The commands are organized into a hierarchical order for easier navigation and can be paired up with the `find` command keyword

to search specific commands. The available commands may vary based on the PAN-OS version; the complete available list for a particular version can be found by running the `find` command.

Typically, network administrators apply the crawl, walk, run approach when using the CLI to automate tasks. During the crawl phase, you will craft the CLI commands to perform specific tasks and then combine a set of commands to achieve the desired outcome, whether providing security to an application workload or setting up a new firewall with a base configuration. An example set command to add the `Example-Group` security profile group to the `example-rule-1` security rule in the `example-device-group` device group is as follows:

```
set device-group example-device-group pre-rulebase security rules example-  
rule-1 profile-setting group Example-Group
```

One key benefit of using CLI commands is idempotency, which allows you to craft a set of commands that you can use multiple times that yield the same results as performing them once.

Journey toward full automation

As you build complex infrastructure and are required to solve specific use cases that cannot be orchestrated manually, you will have to start looking into the tools available to achieve these tasks. These tools allow you to abstract away the complex domain-specific knowledge required to automate specific tasks. Some of the deciding factors to move toward full automation are the following:

- Automated validation of a configuration to reduce human error
- Dynamic requirements gathering from other systems (e.g., **IT Service Management (ITSM)** or **IP address management (IPAM)**)
- Integration requirements with other automated pipelines or workflows within your environment
- Simplify the tech stack used to build automation

We will categorize the tools and libraries available for automating PAN-OS devices into three categories for simplicity:

- First-class **Application Programming Interfaces (APIs)**
- **Software Development Kits (SDKs)**
- **Infrastructure as Code (IaC)** tools

Each category serves different use cases and comes with its own set of advantages and considerations. Let's explore each in detail.

APIs: The foundation of automation

APIs are essential for modern automation, providing a programmatic way to control and monitor network devices. For Palo Alto Networks firewalls and Panorama, the PAN-OS APIs play a crucial role by enabling interaction with third-party tools, custom scripts, and orchestration platforms. These APIs support a wide range of tasks, from routine monitoring and reporting to complex configuration changes and policy enforcement. PAN-OS offers two main types of APIs to address different needs and integration scenarios: the comprehensive XML API and the modern REST API. Understanding these APIs is key to unlocking the full potential of firewall automation. In the following section, we will delve into how authentication is managed and how access can be scoped.

Authentication

Every API request must be authenticated, and you can specify authentication using one of the following three methods:

- Pass the API key as part of the X-PAN-KEY HTTP header
- Pass the API key as the query parameter key
- Basic authorization with a username and password

As a security best practice, using an API key with a short lifetime to authenticate and avoid accidental exposure of account usernames and passwords is recommended.

API scopes

It is recommended that an admin role be created with the required API scope and associated with the admin account used for API access. For the XML API, you can only enable/disable permissions based on the action type, such as report, log, configuration, operational requests, commit, user ID agent, export, or import, and that will apply to all the XML nodes that support that action. Meanwhile, with the REST API, you can apply more granular access controls (enable, read-only, or disable) per PAN-OS resource type, such as address objects or network zones.

PAN-OS XML API

PAN-OS configuration is organized into a hierarchical XML nodes tree structure, and the XML API provides the functionality to query and make changes to these nodes by specifying corresponding XPath. XPath is an expression language that provides the capability to query or modify the information stored in XML nodes, in our case, PAN-OS configuration.

An XML API request is comprised of four key parameters:

- **type:** The type of action you would like to perform against the device, varying from making configuration changes to running an operation command to importing/exporting a file from the system
- **action:** The action to perform when type is config or op (e.g., set, edit, delete, show, or get)
- **xpath/cmd:** Specify the XPath of the configuration node (when type is config) and the operational command to run (when type is op)
- **key:** The API key to authenticate the request against the device

Both the GET and POST HTTP methods can be used with the XML API. The decision between the two depends on the query size and/or when passing non-ASCII characters as part of the query.

When using the XML API, you will have to parse the status and error fields in the HTTP response to determine the status of the operation rather than rely on HTTP response codes. For example, an HTTP 200 status code does not guarantee that the operation has been successful on the device.

Example XML API request/response

To illustrate a basic API call, the following example uses curl to send an operational command that retrieves system information from a firewall. The request specifies type as op (operational) and cmd as <show><system><info></info></system></show>:

```
curl -X POST 'https://<firewall|panorama>/  
api?type=op&cmd=<show><system><info></info></system></show>'  
<response status="success">  
  <result>  
    <system>  
      <hostname>example-firewall-hostname</hostname>  
    </system>  
  </result>  
</response>
```

While this example shows a straightforward API call, constructing the correct query, especially the XPath, can sometimes be intricate. To assist with this, PAN-OS provides several methods for identifying the precise XPath needed for your API requests.

Finding the XPath

While the XML API is extremely powerful, sometimes it can be challenging to determine the correct XPath to use with your API requests. You can use one of the following methods to identify the correct XPath:

- **API browser:** You can access the API browser by simply appending `/api` to the hostname of your firewall or Panorama device when accessing it via a web browser. The API browser allows you to browse the XML node hierarchy and test requests against a selected node.
- **CLI debug:** This method is quite helpful if you know the CLI command to perform an operation and want to find the corresponding XPath. Once you are logged in to the CLI of a firewall or Panorama, you must first enable debug mode by running the `debug cli` command. The next time you run a CLI command, the response will also contain the XPath.
- **Web UI debug:** This method allows you to locate a corresponding XPath while navigating through a specific area on the PAN-OS web UI. First, authenticate against the firewall or Panorama web UI. Next, open another web browser tab and navigate to `https://<firewall>/debug` (this is not a public link; please replace the placeholder with your own firewall management IP address, which should be a private IP) to launch the debug console and ensure the **Debug** and **Minimize JavaScript** checkboxes are enabled. Now, go back to the web UI, perform the required action, and click the **Refresh** button on the debug console to view the resulting XML syntax.

Once you are comfortable with finding XPaths, you might look for ways to streamline interactions, especially when managing multiple firewalls through Panorama.

Proxying API calls via Panorama

The `target` parameter in Panorama enables you to redirect queries to a managed firewall by specifying the serial number of the target firewall. This redirection helps reduce the time and number of steps required to issue repetitive commands. You can also make several API calls in parallel to the target firewalls, such as performing read operations, which can enhance the performance of your scripts.

For example, if you want to run the `show system info` command on a firewall directly, you can proxy that request through your Panorama using the following command:

```
$ curl -X POST 'https://<panorama>/api?type=op&cmd=<show><system><info></info></system></show>&target=<device-serial-number>'
```

Proxying calls is efficient for individual commands, but for more complex configurations involving multiple steps, PAN-OS offers a way to bundle these into a single transaction.

Multi-config requests

The PAN-OS XML API allows multiple configuration API requests in a single transaction, simplifying complex multi-step API configurations. This feature was introduced in PAN-OS 10.0. `multi-config` is enabled by default starting in PAN-OS 10.1; however, in PAN-OS 10.0, `multi-config` is disabled by default. You can enable it by running the following command:

```
request multi-config enabled
```

To leverage this powerful feature, it's important to understand the specific XML structure required for multi-config requests.

Multi-config XML document format

To utilize the `multi-config` feature effectively, API requests must adhere to a specific XML structure. This structure allows multiple configuration actions to be defined within a single XML document. The following code outlines the general format and key elements involved in constructing such a request:

```
<multi-config>
  <action id="action-id" xpath="xpath">
    <element-xml>
    </element-xml>
  </action>
  <action2 id="action2-id" xpath="xpath">
    <element-xml>
    </element-xml>
  </action2>
  <actionN id="actionN-id" xpath="xpath">
    <element-xml>
    </element-xml>
  </actionN>
</multi-config>
```

The root `<multi-config>` element can have any name, and the `id` attribute is optional, used to identify the corresponding action's response.

Beyond simply grouping commands, multi-config operations also offer robust transactional capabilities to ensure configuration integrity.

Transactional support

Transactional support guarantees that none of the actions within the multi-config operation will be performed if one of the actions fails.

The following two rules are enforced when the optional `strict-transactional` parameter is set to `yes`. By default, this is set to `no`:

1. When a commit operation is active or pending, the operation will fail:

```
<response status="error" code="15"><msg><line>Commit in Progress or
Pending</line></msg></response>
```

2. At the point of execution of a multi-config operation, if there are uncommitted changes (in the candidate configuration) made by the same user who is performing the multi-config operation, the device will first roll back those uncommitted changes before executing the multi-config actions.

Example request/response

The following example demonstrates a multi-config request that modifies three address objects within a single transaction. The request is sent to the API endpoint using `curl`, with the XML payload read from a local file (`multi-config.xml`) and the `strict-transactional` parameter set to `yes` to ensure atomicity:

The following code outlines the payload of a multi-config request that modifies three address objects within a single transaction:

```
$ cat multi-config.xml
<multi-configure-request><edit xpath="/config/devices/entry[@
name='localhost.localdomain']/device-group/entry[@name='multi-config']/
address/entry[@name='foo']"><entry
name="foo"><description>foo example</description><ip-netmask>1.1.1.1</ip-
netmask></entry></edit><edit
xpath="/config/devices/entry[@name='localhost.localdomain']/device-
group/entry[@name='multi-config']/address/entry[@name='hello']"><entry
name="hello"><description>bar
example</description><ip-netmask>3.3.3.3</ip-netmask></entry></edit><edit
xpath="/config/devices/entry[@name='localhost.localdomain']/device-
group/entry[@name='multi-config']/address/entry[@name='bar']"><entry
name="bar"><description>bar
```

Then the request is sent to the API endpoint using curl, with the XML payload and the strict-transactional parameter set to yes to ensure atomicity:

```
example</description><ip-netmask>2.2.2.2</ip-netmask></entry></edit></multi-configure-request>
$ curl -k https://<firewall|panorama>/api -d type=config -d action=multi-config --data-urlencode element@multi-config.xml -d strict-transactional=yes

<response status="success" code="20">
<response status="success" code="20">
<msg>command succeeded</msg>
</response>
<response status="success" code="20">
<msg>command succeeded</msg>
</response>
<response status="success" code="20">
<msg>command succeeded</msg>
</response>
</response>
```

While the XML API offers comprehensive control, PAN-OS also provides a more modern alternative: the REST API.

PAN-OS REST API

The PAN-OS REST API is a RESTful implementation that transfers a representation of the state of the PAN-OS resources and supports both JSON and XML for request and response formats; it defaults to JSON.

The API supports GET, POST, PUT, and DELETE methods for the CRUD operations of PAN-OS configuration. In addition to this, you can rename resources and move a policy rule using POST. Only a subset of resources is supported in the REST API compared to the XML API. The documentation can be browsed by appending `/restapi-doc` to the hostname of your firewall or Panorama device. All API requests require query parameters, and the required query parameters depend on the resource method.

To effectively use these methods, you'll need to construct the API request URL correctly, which has a specific structure.

Anatomy of a REST API URL

The REST API URL contains the base path followed by a PAN-OS resource URI:

- **Base path:** This is a static path for the firewall or Panorama you're pointing to (for example, `https://<firewall-host-name>/restapi/<PAN-OS-Version>`). **Note:** This is not a public link. Please replace the placeholder with your own firewall management IP address, which should be a private IP.
- **Resource URL:** Corresponds to the PAN-OS resource path you will be working with (for example, `Objects/Addresses` if you want to work with address objects).

Example REST API request/response

The following request retrieves all address objects configured in the exampleDG device group:

```
$ curl -X GET \
  'https://<panorama-hostname>/restapi/v11.2/Objects/
Addresses?location=device-group&device-group=exampleDG&output-format=json'
{
  "@status": "success",
  "@code": "19",
  "result": {
    "@total-count": "3",
    "@count": "3",
    "entry": [
      {
        "@name": "foo",
        "@location": "device-group",
        "@device-group": "exampleDG",
        "@loc": "example device group",
        "description": "foo bar example",
        "disable-override": "no",
        "ip-netmask": "2.2.2.2"
      },
      .....
    ]
  }
}
```

The HTTP response format differs slightly between a successful and an error response, which can be distinguished based on the `@code` attribute in the JSON or XML payload.

XML versus REST API

When choosing between XML and REST APIs, the following is a list of criteria you may want to consider:

- **Feature coverage:** The REST API only covers a subset of the supported features compared to the XML API, and this will be the key deciding factor between the two flavors to fit your use case.
- **API access control:** The REST API allows you to control access per PAN-OS resource type. In contrast, the XML API is limited to specific actions that apply to all the resources.
- **Data format:** The REST API supports JSON and XML data formats, whereas the XML API is strictly limited to XML. You might find that when integrating with other tools, it's easier to parse JSON data than XML.

While direct API interaction offers maximum flexibility, SDKs can significantly simplify the development process by providing higher-level abstractions.

Software development kits

In the context of this book, we refer to SDKs as a set of tools/libraries built using specific programming languages such as Python, Go, or PHP that can be used to extend a workflow or build an application to achieve a goal or a task against a PAN-OS-based firewall or a Panorama device. SDKs provide useful abstractions on top of the first-class APIs and seamlessly handle aspects such as authentication.

Python SDK (`pan-os-python`)

Python has become the de facto language for network automation, and the `pan-os-python` SDK provides a powerful interface for programmatic firewall management. The SDK provides an object-oriented framework and follows a similar structure to how configuration is structured in the **Graphical User Interface** (GUI) and CLI.

Object-oriented framework

Every configuration item, whether a firewall, panorama, device group, address object, or rule, is a `PanObject`, allowing you to easily construct a configuration tree before sending it to the live device. These objects also inherit common configuration methods such as `add()`, `remove()`, `find()`,

and `findall()` for building the configuration tree; `apply()`, `create()`, and `delete()` for pushing the changes to the device; and `refresh()` and `refreshall()` for reading the configuration from the device:

```
# Instantiate the Firewall Object
fw = Firewall("10.0.0.1", "admin", "password")
# Instantiate the AddressObject
webserver = AddressObject("Apache-webserver", "5.5.5.5",
description="Company web server")
# Build the configuration tree by adding the AddressObject to the Firewall
fw.add(webserver)
# Send the configuration to the firewall
webserver.create()
```

This object-oriented approach allows for flexible connection methods, whether you're targeting a firewall directly or managing it through Panorama.

Direct versus proxy firewall connection

When you want to make changes directly on the firewall and your firewall management IP is reachable from the host where your script is running, you can instantiate a firewall object with its hostname and credentials:

```
# Instantiate a Firewall with hostname and credentials
fw = firewall.Firewall("10.0.0.1", "admin", "mypassword")
```

However, if you manage several firewalls via Panorama, proxying the connection through Panorama becomes handy from a convenience and security best practice point of view. You can use the Panorama management IP to connect via the script without worrying about the connectivity between your script and each firewall. You can also avoid secret sprawl by not having to maintain credentials for all your firewalls; instead, you can use a single set of credentials for Panorama:

```
# Instantiate a Firewall with serial
fw = Firewall(serial="000123456789")
# Instantiate a Panorama with hostname and credentials
pano = Panorama("10.0.0.5", "admin", "mypassword")
# Add the Firewall as a child of Panorama
pano.add(fw)
# Create the AddressObject on the Firewall via Panorama
fw.add(AddressObject("Server", "2.2.2.2")).create()
```

In addition to direct or proxied connections, the SDK also provides mechanisms to effectively manage firewalls configured with multiple **Virtual Systems (VSYS)**.

VSYS awareness

A single physical firewall can be partitioned into multiple logical instances, called virtual systems, or VSYS. Each VSYS operates as an independent, standalone firewall with its own security policies, network interfaces, and administrative management. This feature is particularly useful in multi-tenant environments or for segmenting networks with distinct security requirements within a single organization. The pan-os-python SDK provides two flexible methods for managing configurations on firewalls with multiple VSYS:

1. Instantiate a firewall instance for each VSYS:

```
vsys2 = firewall.Firewall("10.0.0.1", "admin", "mypassword",
vsys="vsys2")
vsys3 = firewall.Firewall("10.0.0.1", "admin", "mypassword",
vsys="vsys3")
```

2. Use a single firewall instance with multiple VSYS instances:

```
fw = Firewall("10.0.0.1", "admin", "mypassword", vsys="shared")
vsys2 = device.Vsys("vsys2")
vsys3 = device.Vsys("vsys3")
fw.add(vsys2)
fw.add(vsys3)
address = vsys2.add(AddressObject("MyIP", "2.2.2.2"))
address.create()
```

The complete source code and additional resources for the SDK are available on GitHub:

<https://github.com/PaloAltoNetworks/pan-os-python>

For development teams that prefer Go, a similar level of abstraction and control is available through the pango SDK.

Go SDK (pango)

For organizations using Go for their development needs, the pango module offers a way to interact with PAN-OS-based firewalls and Panorama. pango is also used as the underlying SDK for the panos Terraform provider.

PAN-OS resources are organized into packages that reflect the configuration structure in the GUI and CLI. Each package contains the following:

- **Service (service.go):** Every resource type implements create, read, update, delete, and list methods.
- **Entry/config (entry.go/config.go):** The entry and config structs represent all the attributes that can be configured for a particular PAN-OS resource.
- **Location (location.go):** A specific PAN-OS resource can exist in multiple locations; for example, an address object can be configured in a shared location, a VSYS, or a device group. All create, read, update, delete, and list methods require a location parameter to be passed in, allowing you to specify explicitly where the resource should be created.

To install the latest version of the pango SDK, run the following `go get` command:

```
go get github.com/PaloAltoNetworks/pango@main
```

To get started, create a client connection and pass it along with the `Entry` and `Location` parameters to the service you want to use.

The following code demonstrates how to create an address and address group and read an existing address from the device:

```
package main
import (
    "context"
    "log"
    "github.com/PaloAltoNetworks/pango"
    "github.com/PaloAltoNetworks/pango/objects/address"
    address_group "github.com/PaloAltoNetworks/pango/objects/address/group"
    "github.com/PaloAltoNetworks/pango/util"
)
func main() {
    ctx := context.Background()
    con := &pango.Client{
        Hostname: "10.0.0.1",
        Username: "admin",
        Password: "mypassword",
        SkipVerifyCertificate: true,
    }
    if err := con.Setup(); err != nil {
```

```
log.Printf("Failed to setup client: %s", err)
return
}
if err := con.Initialize(ctx); err != nil {
log.Printf("Failed to initialize client: %s", err)
return
}
log.Printf("Initialize ok")
// Create an Address Object
addressObject := &address.Entry{
Name: "example-address",
IpNetmask: util.String("1.1.1.1"),
}
var addressLocation *address.Location
if ok, _ := con.IsPanorama(); ok {
addressLocation = address.NewDeviceGroupLocation()
addressLocation.DeviceGroup.DeviceGroup = "go_device_group"
} else {
addressLocation = address.NewSharedLocation()
}
// Create a new address service and pass the client connection to it
addressSvc := address.NewService(con)
addressResp, err := addressSvc.Create(ctx, *addressLocation,
addressObject)
if err != nil {
log.Printf("Failed to create object: %s", err)
return
}
log.Printf("Address '%s=%s' created", addressResp.Name, *addressResp.
IpNetmask)
// Read Address Objects
addresses, err := addressSvc.List(ctx, *addressLocation, "get", "name
starts-with 'example'", "")
if err != nil {
log.Printf("Failed to list object: %s", err)
} else {
for index, item := range addresses {
```



```

log.Printf("Address %d: '%s'", index, item.Name)
}
}
// Create an Address Group
addressGroupObject := &address_group.Entry{
Name: "example-address-group",
Static: []string{addressResp.Name},
}
var addressGroupLocation *address_group.Location
if ok, _ := con.IsPanorama(); ok {
addressGroupLocation = address_group.NewDeviceGroupLocation()
addressGroupLocation.DeviceGroup.DeviceGroup = "go_device_group"
} else {
addressGroupLocation = address_group.NewSharedLocation()
}
addressGroupApi := address_group.NewService(con)
addressGroupResp, err := addressGroupApi.Create(ctx,
*addressGroupLocation, addressGroupObject)
if err != nil {
log.Printf("Failed to create object: %s", err)
return
}
log.Printf("Address group '%s' created", addressGroupResp.Name)
}

```

Client-side filtering

The pango SDK provides a built-in mechanism to perform client-side filtering using a combination of operators, predefined checks, and groupings using parentheses.

More information can be found at <https://github.com/PaloAltoNetworks/pango/blob/main/filtering/doc.go>.

For example, say you want to filter out all the address objects with the associated Corp tag. You can use the following filter to do that:

```

addressFilterSvc := address.NewService(con)
filtered_addresses, err := addressFilterSvc.List(ctx, *addressLocation,
"get", "tag contains 'Corp'", "")

```

PHP SDK (pan-os-php)

pan-os-php is a PHP library designed to simplify PAN-OS configuration changes, ensuring they are easy to maintain and capable of handling complex scenarios such as rule merging and tracking unused objects.

While pan-os-php serves as an SDK that allows you to extend your PHP application to integrate with PAN-OS-based appliances, it can also be used as a standalone CLI tool to perform predefined tasks (util scripts) against live devices and offline XML configuration files.

Custom scripting

In the following example, we use pan-os-php to assign a security profile to all the security rules in vsys1:

```
<?php
// Load the pan-os-php library
require_once("../lib/pan_php_framework.php");
// input and output files
$source_configuration = "current-config.xml";
$targetVSYS = 'vsys1';
$targetProfile = 'example-profile';
$outputfile = "output.xml";
// Create an instance of a PANConf object
$panc = new PANConf();
$panc->load_from_file($source_configuration);
// Check the existence of vsys
$vsys1 = $panc->findVirtualSystem($targetVSYS);
if( $vsys1 === null )
{
    derr("vsys1 was not found ? Exit\n");
}
// Get the list of rules
$rules = $vsys1->securityRules->rules();
// Iterate through the rules and set the security profile
foreach( $rules as $rule )
{
    PH::print_stdout("- Updating rule '". $rule->name()."' with security
    profile '$targetProfile' ");
}
```

```
$rule->setSecurityProfileGroup($targetProfile);  
}  
// Save updated configuration to a file  
$panc->save_to_file($outputfile);  
// Display statistics  
$vsys1->display_statistics();  
memory_and_gc('end');
```

You can find more examples at <https://github.com/PaloAltoNetworks/pan-os-php/tree/main/examples>.

Beyond custom scripting, pan-os-php also offers a convenient CLI for executing predefined tasks.

CLI usage

pan-os-php is packaged with several predefined utility scripts you can invoke via the CLI without custom scripting. The tool accepts the following arguments:

- **in:** Specify the source of the configuration (API or offline XML config).
- **type:** PAN-OS resource types such as address, rule, or securityprofile.
- **action:** Define the action you want to perform against the PAN-OS resource type specified via type. You can list all the available actions via `pan-os-php type=rule listActions`.
- **filter:** Allows you to filter down the resources before applying the action. You can list all the available filters via `pan-os-php type=rule listFilters`.

The following example illustrates how to enable *log at session start* for rules with the destination zone (to) as dmz and only includes the Webfarms address object:

```
$ pan-os-php type=rule in=api://<firewall hostname> actions=logStart-  
Enable 'filter=(to has dmz) and (dst has.only Webfarms)'
```

While SDKs provide programmatic control, another powerful paradigm for managing configurations is through IaC tools, which focus on declarative state management.

Infrastructure-as-code tools

IaC allows you to design, build, deploy, and manage infrastructure in code instead of clicking through a UI or typing CLI commands. In this section, we will use the terms infrastructure as code, or IaC, and configuration as code interchangeably to refer to PAN-OS configuration management using code.

IaC simplifies configuring a resource by allowing you to define the end state of what you want to be created. From there, your chosen tool or framework, such as Terraform or Ansible, will provision these resources according to the desired state you defined in the code.

Adopting IaC for firewall management offers several significant benefits that streamline operations and enhance security posture:

- **Standardization:** IaC allows you to define and manage best practice configuration blueprints through code, which helps you eliminate the inconsistent configuration that arises from manual processes.
- **Repeatability and reduced human errors:** Codifying configuration allows you to reuse the code to achieve the same outcome repeatedly. For example, if you want to deploy your configuration onto a newly deployed firewall, you can simply run your IaC templates to reproduce the same configuration in a matter of seconds. This also minimizes the risk of errors that can occur during manual configuration tasks.
- **Security and compliance:** Once the configuration is codified, you can assess it programmatically and enable guardrails to detect deviations from the best practices.
- **Version control and auditing:** Embracing software development practices such as version control allows large teams to collaborate on the configuration via code reviews and pull requests and track the changes via commits. Centrally stored code can be treated as the single source of truth. More importantly, the VCS enforces permissions and provides an audit trail. Different classes of users can be granted read and/or write access. Every update to the configuration is recorded in the Git logs. If you want to roll back the configuration changes, you can simply revert to an old commit.

Imperative versus declarative IaC languages

There are two types of IaC languages – imperative and declarative. Consider the history of IaC languages, such as shell or Python scripts. In that case, operators use these languages to execute a set of commands in sequential order to achieve the desired configuration state. These are considered imperative languages, meaning they are essentially ordered, logical lists of commands. In the context of PAN-OS configuration, you must have domain-specific knowledge and follow the process precisely and completely from start to finish, for example, configuring an address object first, followed by address groups, and assigning them to a security rule. IaC written this way requires the user to understand the end result in advance.

On the other hand, declarative tools let you define the desired state you require. By employing providers such as the panos Terraform provider, which knows all of the details and instructions to follow, we do not have to worry about the order of the operations. In a declarative model, no matter how many times the command is executed, you will end up with the same desired state on your firewall. Running the scripts multiple times in an imperative model may produce duplicate configurations or undesired results if not managed with intention.

There is an added benefit of the declarative model, which is security. Older scripting languages contain conditional statements and loops. This often means the result is non-deterministic. In other words, you won't get the same result for every execution. For IaC, non-determinism is actually a problem, primarily because you are unable to compare the result with the code (the source of truth). This is called **drift**. If drift cannot be easily determined, we cannot say whether runtime issues are the product of poor design or malicious intent.

Terraform versus Ansible

Terraform and Ansible are frequently discussed together, as they are two of the most popular tools for automating operations and implementing IaC and configuration/policy as code. Ansible has its roots in configuration management, while Terraform is primarily focused on deploying cloud infrastructure. Despite their distinct origins, both tools can be utilized for a variety of tasks, including those outside their main focus areas.

Some key differences between Ansible and Terraform are as follows:

- Terraform solely relies on its state to calculate the order of operations and execute them, whereas Ansible executes tasks without requiring a comprehensive state reconciliation.
- Ansible runs tasks in the order they are specified in a playbook, while Terraform executes tasks based on the dependency graph it generates.

However, both are capable of operating idempotently and automating PAN-OS configuration.

Regardless of whether you choose Ansible or Terraform for PAN-OS automation, both the PAN-OS Ansible collection and the Terraform provider primarily leverage the XML API for configuration management tasks. Consequently, the **Role-Based Access Control (RBAC)** capabilities are determined by the scopes available within the XML API, emphasizing the need for carefully defined admin roles and permissions.

Ansible for PAN-OS

The PAN-OS Ansible collection is a set of modules available via Ansible Galaxy that can be used to define and manage configuration on both firewalls and Panorama devices.

When creating playbooks with a PAN-OS Ansible collection, you will leverage the local method, which means the operations are executed by the localhost running the playbook as opposed to being executed on the remote machine (firewall or Panorama). In order to do this, Ansible connects with PAN-OS devices over HTTPS using the username/password or API keys:

```
---
- name: Gather system info
  hosts: "panorama"
  # use local mode
  connection: local
  vars:
    # Define a provider block with credentials
    device:
      ip_address: "{{ ip_address }}"
      username: "{{ username }}"
      password: "{{ password }}"
  collections:
    - paloaltonetworks.panos
  tasks:
    - name: Gather facts for device
      paloaltonetworks.panos.panos_facts:
        # Reference the provider block within the task
        provider: "{{ device }}"
```

The PAN-OS Ansible collection provides several key features that make it a powerful choice for network automation:

- **Agentless architecture:** This architecture allows you to manage configuration on PAN-OS devices without having to install agents on the firewalls or Panorama devices, which conforms to the Palo Alto Networks assurance model.
- **YAML-based playbooks:** Ansible uses YAML, a human-readable data serialization language. The YAML syntax is then evaluated and executed sequentially, meaning the automation workflow runs in the order the tasks are defined.

- **Extensive module library:** The PAN-OS Ansible collection supports the configuration of a vast number of features on firewalls and Panorama devices, along with `panos_type_cmd` and `panos_config_element` for configuring any resource using XPath.
- **Idempotent operations:** When configuring PAN-OS resources using one of the modules in the collection, the idempotency is guaranteed when executing the same playbook multiple times without making any changes.
- **Operational tasks:** In addition to configuration, you can install content updates and perform software upgrades using the `panos_software` and `panos_dynamic_updates` modules.

Complementing Ansible's imperative approach, Terraform offers a declarative way to manage PAN-OS configurations, which we will explore next.

PAN-OS Terraform provider

Terraform offers a powerful way to manage firewall configuration through its PAN-OS provider, enabling true IaC practices.



This section refers to v2 of the provider, which is the latest major release of the provider. V2 contains several enhancements and introduces major breaking changes to v1.

The PAN-OS Terraform provider enables true IaC practices for firewall management and includes the following key features:

- **Agentless architecture:** This architecture allows you to manage configuration on PAN-OS devices without having to install agents on the firewalls or Panorama devices, which conforms to the Palo Alto Networks assurance model
- **Declarative configuration:** Terraform uses **HashiCorp Configuration Language (HCL)** to define the desired configuration, which is both human- and machine-friendly.
- **Extensive Terraform resources:** The PAN-OS Terraform provider supports the configuration of a vast number of features on firewalls and Panorama devices.
- **Drift detection and idempotency:** When configuring PAN-OS resources using Terraform, you can guarantee idempotency, which is Terraform's default behavior. When terraform apply is run, by default, Terraform will perform a refresh to detect drift. If a drift is detected, Terraform will calculate the changes required for state parity between the actual configuration on the device and the defined configuration.

- **Multi-config support:** Earlier in this chapter, we discussed multi-config, which helps you perform multiple configuration API requests within a single HTTP request. The Terraform provider leverages this feature to enhance the performance of the operations when configuring some of the resources by batch-processing multiple calls through a single API request. By default, it is set to batch-process 500 configuration API requests and is configurable to suit your needs.



Only use `panos_security_policy` if you want to manage all the rules in the security policy. When `panos_security_policy` is used, the existing rules in the policy will be overwritten by the rules you define in your Terraform configuration. If your requirement is to manage only some of the rules in the policy via Terraform while maintaining the rest of the existing rules on the device, use `panos_security_policy_rules` and control the position of the rules to be placed relative to the existing ones via the `position` attribute.

The following HCL script illustrates how you can configure the Terraform provider and use it to create an address, add it to an address group, and then use it in a security rule:

```
# Define required providers to install
terraform {
  required_version = ">= 1.4.0, < 2.0.0"
  required_providers {
    panos = {
      source = "PaloAltoNetworks/panos"
      version = "2.0.0"
    }
  }
}

# Provider configuration
provider "panos" {
  hostname = "4.151.218.139"
  username = "username"
  password = "password"
  skip_verify_certificate = true
}

# Create an address object in Panorama device group terraform-device-group
resource "panos_address" "example-address" {
```



```
location = {
  device_group = {
    name = "terraform-device-group"
  }
}
name = "example-address"
description = "example-description"
ip_netmask = "10.0.0.1/32"
}
# Create an address group and add the previously created address object
resource "panos_address_group" "example" {
  location = {
    device_group = {
      name = "terraform-device-group"
    }
  }
  name = "example-address-group"
  description = "example address group"
  static = [panos_address.example-address.name]
}
resource "panos_security_policy_rules" "example-policy-rules" {
  location = {
    device_group = {
      name = "terraform-device-group"
    }
  }
  position = {
    where = "first"
  }
  rules = [
    {
      name = "allow-web-browsing"
      action = "allow"
      source_zones = ["any"]
      source_addresses = [panos_address_group.example.name]
      destination_addresses = ["172.0.0.0/8"]
      destination_zones = ["any"]
    }
  ]
}
```

```
    applications = ["web-browsing"]
    services = ["application-default"]
  }
]
}
```

Commits

Commit operations are not part of the provider due to today's lack of finalizer operation capability within the Terraform core. These operations are fundamentally at odds with how Terraform works as a declarative system. In Terraform, the desired state is declared in a configuration file and compared to the currently active state. Operations are performed in any order (often simultaneously) to reach the desired state. When a commit is involved, we're saying a declared resource is in a desired state even though it is not active yet, and a separate declaration makes it active despite that commit resource having no ongoing state of its own. This works in an imperative model such as Ansible but doesn't fit natively in Terraform.

If you want to perform a commit after `terraform apply`, you will either have to make an API call or use one of the preferred SDKs we discussed earlier in this chapter to perform a commit.

Choosing the right automation approach

When it comes to evaluating automation solutions for your organization, it's anything but a simple comparison. The process is full of nuances and considerations that can significantly impact your decision-making. There isn't a single "right" way to automate or one specific tool that fits all needs. Each business has unique requirements that differ from those of other organizations in the same industry. It's important to recognize that just as there is no one-size-fits-all approach to solving a problem, the automation needs of your organization will be distinct.

When selecting an automation approach, consider the following:

- **The scale of deployment:** Larger deployments with 100+ firewalls with many dependencies will need more careful consideration compared to a smaller deployment of fewer than 10 firewalls. Consider how well the automation will scale as your requirements and the deployment size grow.
- **Team skills:** When choosing an SDK or an automation tool, it is crucial to understand your team's current technical skills and select technologies with a lower learning curve.

- **Operations background:** Consider building automation workflows that are not drastically different from how your teams operate firewalls. Creating a seamless end user experience will lead to higher adoption of automation and collaboration in the long run.
- **DevOps maturity:** If you have dedicated DevOps or platform engineering teams maintaining some pipelines, collaborate with them to integrate the automation you build into that existing pipeline for a smoother experience.
- **Integration requirements:** Build automation by leveraging existing tools used within your environment and integrating them with existing platforms, such as ITSMs, to collect input from end users.
- **Use cases:** Break down use cases into smaller problems that you can solve rather than designing a do-it-all automation solution. This will make it easier to troubleshoot and maintain the solution. For example, configuration management, compliance automation, and operational tasks might not all fit into a single automation solution.

Once you've weighed these factors and chosen an approach, adhering to established best practices is crucial for successful and sustainable automation.

Best practices and implementation guidelines

Successfully implementing and maintaining firewall automation requires more than just choosing the right tools; it demands a thoughtful approach grounded in proven best practices. These guidelines span various aspects, from general automation principles to specific security and performance considerations, all aimed at creating robust, scalable, and secure automation workflows. We will first cover general best practices applicable to any automation endeavor.

General automation best practices

When implementing firewall automation, following these best practices ensures robust and maintainable solutions:

- **Version control:** Version control allows you to track file changes and retrieve specific versions when necessary, using commit IDs or tags. It facilitates team collaboration and is crucial for security automation:

```
# Example Git workflow
git checkout -b feature/security-policy-update
# Make changes to automation code
git add .
```

```
git commit -m "Update Internal Access security rule"
git push origin feature/security-policy-update
# Create a pull request for review
```

- **Testing strategy:** As you extend your automation framework, make it a habit to write unit tests and integration tests. Unit tests are aimed at smaller building blocks, such as creating an address object, and help you catch early-stage bugs. Integration tests can be extended to test end-to-end use cases such as creating address objects, creating address groups, assigning address objects to the groups, and associating them with security rules. They help detect more complex issues in unit interaction.
- **Error-handling and logging:** It is important to specify how you want to handle errors and take necessary remediation actions in your code. You may encounter situations where a specific action could fail at the beginning or during an operation. You need to define the appropriate course of action to take programmatically, whether it involves simply logging or alerting of the issue or reverting any changes made up to that point.

Security considerations

When it comes to ensuring the security of your software and infrastructure, adopting best practices in credential management and access control is crucial. The following are the key considerations to help you enhance security and mitigate risks associated with sensitive information and permissions:

- **Credential management:** Avoid hardcoding credentials within your source code and employ methods that support dynamic credential injection during runtime. Most libraries support passing credentials and sensitive information via environment variables. This will ensure that you do not accidentally commit your code with hardcoded secrets into a VCS. You can also encrypt files with credentials using tools such as `ansible-vault` or leverage a credential management system to store your credentials securely.
- **Short-lived credentials:** Set the API key lifetime in your PAN-OS device to the least viable value to reduce the impact of accidental exposure of credentials or compromised CI/CD pipelines.
- **Key rotation:** Implement mechanisms to rotate credentials on a regular basis to limit the impact of a security breach and improve compliance.
- **RBAC:** Use dedicated admin accounts for automation and avoid overly permissive user admin accounts for automation tasks. Follow the least-privilege principle when assigning permission to an admin role.

- **VCS controls:** Use `.gitignore` files to specify the file types/names of the files that you do not want to commit to a remote repo accidentally. Set the correct IAM permission to your repos to manage correct read/write access within your organization. Use techniques to sign your commit messages, adding an extra layer of security to the development process by providing a mechanism to verify the integrity and the origin of the changes made to your automation code.

Performance enhancements

With security measures in place, optimizing the performance of your automation scripts is also key to efficiency, especially at scale:

- **Concurrent API calls:** Limit the number of concurrent API calls to a firewall or Panorama to five. This will ensure no performance impact on the device's management plane.
- **Batch processing:** Build your automation scripts to make use of bulk operations where possible
- **Proxy API calls:** If your script needs to query multiple firewalls, use the target parameter to send the queries directly to the firewalls. This will improve the response times and eliminate the need for Panorama to issue repetitive commands.

Integration patterns

Efficient scripts are foundational, and their value is further amplified when integrated into broader operational workflows and systems:

- **CI/CD:** Embrace continuous integration and deployment practices for firewall automation. Making continuous, small changes, as opposed to making infrequent, large changes, will ensure that errors can be identified quickly and fixed, leading to increased collaboration and faster changes.
- **Pipeline integration:** As you scale out your automation workflows, integrate your scripts into a CI/CD pipeline and move away from manually executing scripts from a localhost. Break down the pipeline into multiple stages, such as validation, testing, and deployment. You can insert an additional manual or automated approval step between these stages to comply with your organizational policies.
- **Orchestration:** Integrate with other systems, such as IPAM and **Configuration Management Database (CMDB)**, to retrieve metadata dynamically as inputs to your scripts and push updates upon task completion or failure.

Summary

Firewall automation has become an essential aspect of modern network security, and the choice of approach – using APIs directly, leveraging SDKs, or implementing IaC – depends on factors such as team expertise, deployment scale, and specific use cases. Successfully automating firewall management depends on choosing tools that match your requirements and expertise, as well as enforcing stringent security controls and testing procedures. No matter which tool you select, it's essential to follow best practices to establish robust, scalable, and secure workflows.

Fundamental to any successful automation strategy is the diligent use of version control for all code, which provides an audit trail and facilitates collaboration. This must be paired with robust error handling, comprehensive logging, and regular validation to ensure continued reliability. It is crucial to plan for scalability and maintainability from the outset, keeping abreast of evolving automation capabilities. As network security continues to evolve, disciplined and well-documented automation will play an increasingly vital role in managing and securing critical infrastructure.

In the next chapter, we will go through the process of setting up your software development environment.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



7

Setting Up Your Software Development Environment

In *Chapter 6*, you explored various options available to firewall administrators for automating PAN-OS configuration, including creating and maintaining security policies. The goal of this chapter is to walk you through setting up the necessary software and obtaining all the code referenced in the book, so you can easily follow along in the upcoming chapters.

For this book, Python is used as the programming language, and the PAN-OS SDK for Python serves as the foundation for all interactions with PAN-OS devices. This SDK has gaps in covering some security policy automation requirements. In such cases, the PAN-OS XML API will be used to fill in these gaps. All policy-building scripts will also be containerized to ensure their portability and compatibility with a wide range of modern enterprise operating systems and platforms.

In this chapter, you will cover the following topics:

- Configuring GitHub, including account creation and obtaining the project source code
- Setting up a Python coding environment based on PyCharm
- Containerization with Docker Desktop, Docker Hub, and GitHub Actions

By the end of this chapter, you'll have a fully working development environment and a local copy of the project source code. You'll also be equipped to adapt the provided scripts to suit your own technical landscape and organizational requirements.

Configuring GitHub

Asking you to write every single line of code for this project from scratch might feel a bit overwhelming. Similarly, including all the code snippets directly in the book isn't ideal, since printed books aren't really meant for that kind of hands-on coding.

You will need access to a suitable **version control system (VCS)** that allows you to clone the book's source code repository and work on your own copy of the code. GitHub and GitLab are the current de facto standards in the VCS market. Both of them allow you to create a free account and are natively supported by PyCharm, the software product we will cover in the next section of this chapter. Let's go ahead and create a free GitHub account unless you already have one.

Account creation

Navigate to <https://github.com> in a browser and click **Sign Up**. Then, enter your email address and create a password and a unique username. The registration wizard will challenge you with a puzzle to prove you're an actual human and ask you to verify the provided email address. You will be taken to the login screen upon successful email verification. You may be required to set up two-factor authentication, which is always a good idea.

GitHub will ask you some personalization questions, which you may choose to skip.

Copying the project source code

Now, as you're logged in, navigate to the book repository at <https://github.com/PacktPublishing/Palo-Alto-Networks-from-Policy-to-Code>, which contains the project's source code. This repository is a fork of my public project at <https://github.com/ngfw-automation/policy-as-a-code>.

A GitHub **fork** is a new repository created by copying an existing "upstream" repository on GitHub. The fork inherits the upstream repository's code, complete commit history, and visibility settings, and it stays linked to the upstream so that changes can be synchronized or proposed back through pull requests.

This establishes an effective collaboration system that keeps you informed about updates from the original project's maintainers and allows you to suggest new features or bug fixes that they might consider for the benefit of the wider community. When you create a fork of a fork, as described here, you're essentially cloning the original project. However, if it ceases to exist for any reason, the publisher's fork becomes the new upstream.

Click on the **Fork** button, as shown in the following screenshot:

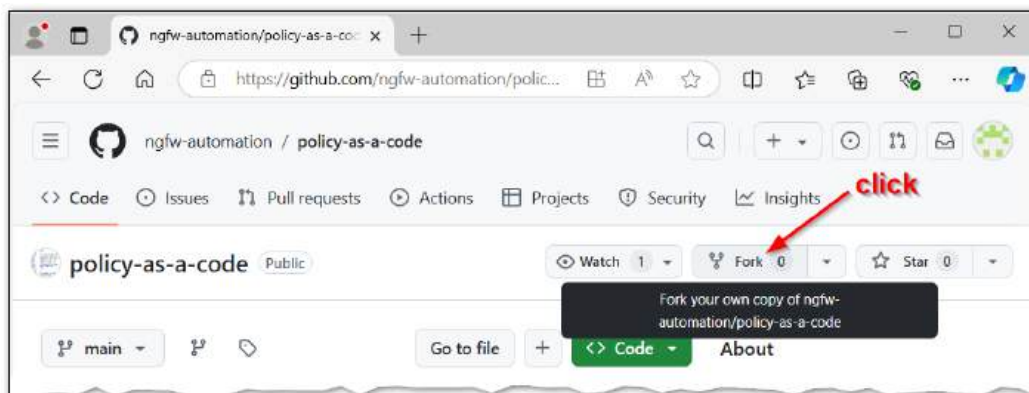


Figure 7.1 – Copying the book’s project repository

The next screen allows you to customize the repository name and description:

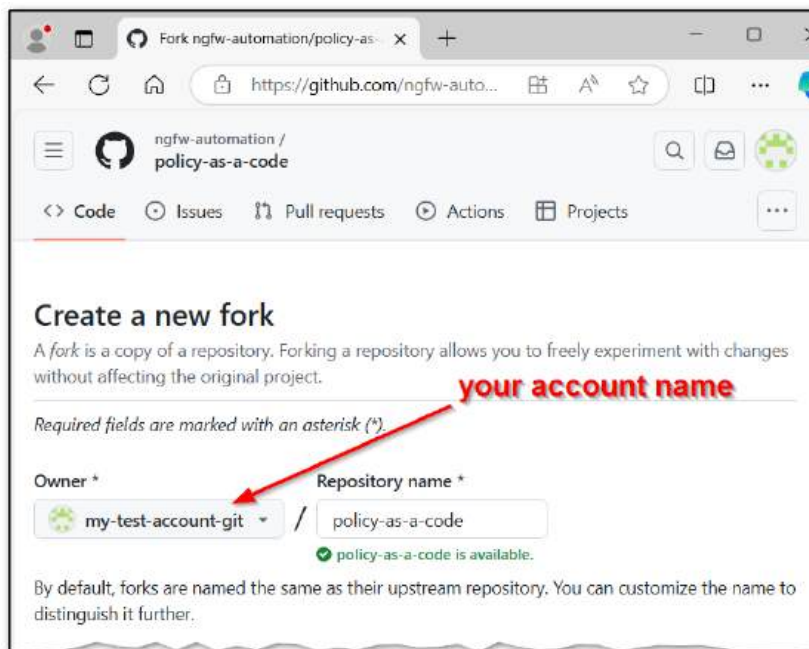


Figure 7.2 – Creation of the book’s repository in your account

Only the main branch will be copied by default, which is a sensible choice. Click on **Create fork** at the bottom of the screen.

GitHub will take a few seconds to copy the original repository to your account.



The fork you just created will have the same visibility setting as the original repository – it will be **public**. You can freely work on it as if it were your own, including making changes, but you cannot alter its visibility. Therefore, **avoid storing any sensitive private information in the fork**. Use the fork solely to track changes in the original project and to contribute to it for the community’s benefit. For handling sensitive information, only use private repositories.

Follow these steps to create a new private repository:

1. Click on **New** in the GitHub dashboard.
2. Choose a repository name, for example, `policy-as-code-private`.
3. Click on the **Private** radio button to change the default **Public** visibility.
4. Click on **Create repository**.

Then, navigate back to the dashboard and open the forked repository. Click on the <> **Code** button and select **Download ZIP**. All files from the repository will be downloaded to your computer in a ZIP archive. Unzip the archive and put these files aside – we’ll need them soon.

Now is an excellent time to get you a suitable environment for software development.

Setting up Python and PyCharm

This book is not a programming textbook by any means. Nevertheless, a brief introduction is in order.

Python is a popular cross-platform language that can be used with any modern operating system, including Windows, macOS, Linux, and others. Many firewall administrators learned and used Python as part of their university/college curriculum, for automation tasks and projects at work, or as a hobby. Even if you have not used it before but have experience with virtually any other programming language, you will quickly master the basics of Python sufficient to achieve the goals of our project here.

Many multi-purpose text editors support Python, including the original editor supplied with the Python interpreter. However, your programming performance will significantly increase if you use a purpose-built **integrated development environment (IDE)**. A good IDE will have numerous features designed to make the programming process easier and more efficient. These features

typically include, but are not limited to, debugging, integration with VCSs, syntax highlighting, smart code navigation, and a range of plugins.

Several excellent IDEs are available for Python. Although many discussions take place daily on various programming forums about the pros and cons of different IDEs, I will recommend just one – PyCharm by JetBrains. You might be accustomed to a different one, which is perfectly fine – stick with it. However, if you’re new to Python IDEs, try PyCharm, and you won’t be disappointed.

As of version 2025.1, PyCharm is a unified product that merges features from the former Professional and Community editions. The core features (similar to the previous Community edition) are free, while advanced capabilities require a paid license. You don’t need the paid features for this project. However, you might find some of them, such as the AI Assistant, valuable enough to justify the subscription cost.

Let’s start by navigating to <https://www.jetbrains.com/pycharm/download/> and downloading the installation file for your OS and computer architecture.

Installation

Run the downloaded file, which will invoke a simple installation wizard. The following screenshot shows the second screen of the wizard, which allows you to choose the destination folder:

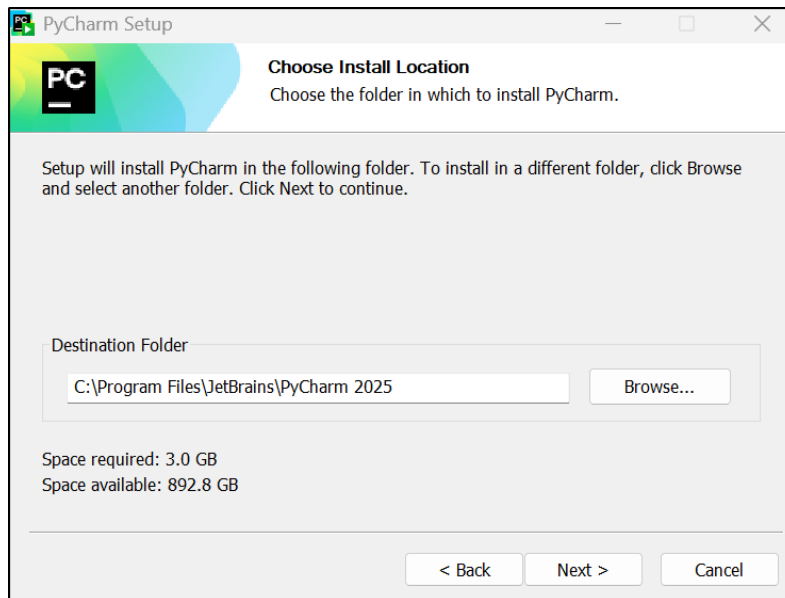


Figure 7.3 – PyCharm installation (destination folder selection)

The screen *after* the one shown in *Figure 7.3* will suggest creating a desktop shortcut, associating PyCharm with the `.py` file extension, updating the `PATH` variable to include PyCharm's `bin` folder, and updating the Windows context menu to open any folder as a PyCharm project.

I suggest you agree to all of those and proceed to the next screen, which will ask for a name for the **Start Menu** folder containing the IDE shortcuts. Let's agree to the default one and click **Next** to proceed with the installation.

Wait for file copying to finish, and if you have chosen to update the `PATH` variable, reboot your computer to complete the installation.

First steps and further setup

PyCharm will ask you to read and accept its User Agreement on the first run. After you've accepted the terms, you will be taken to a welcome screen. The screen has four options to explore:

- **Projects:** This option allows you to create a new project or open an existing one from a disk or a VCS. You can also take a quick product tour, which I highly recommend if you're new to this IDE.
- **Customize:** Here, you can adjust the IDE's visual appearance and a myriad of other settings (hidden behind **All settings...** at the bottom of the screen). You can also configure all this later after you have tasted the environment.
- **Plugins:** This section allows you to install plugins from the JetBrains Marketplace. At the time of writing this book, more than 4,600 third-party plugins were available for the PyCharm Community edition. Our project does not require any compulsory plugins, but you may want to install some to work with CSV and JSON file formats, as both are used in our code.

Warning



Bear in mind that the Marketplace hosts plugins produced not only by JetBrains and other reputable vendors but also by community members. Therefore, there is a heightened risk of encountering malicious code that may expose your computer and network to a supply chain-style cyber-attack. Give preference to reputable vendors and ensure you use up-to-date antivirus software.

- **Learn:** This screen allows you to learn the IDE itself or to program. If you're new to Python, this is a great starting point for studying the language, the IDE, and associated technologies such as Git.

Many of the menu items you select on the welcome screen will offer to automatically install a Python interpreter. Unless you already have Python installed, let's accept this convenient offer. The following screenshot demonstrates what this looks like:

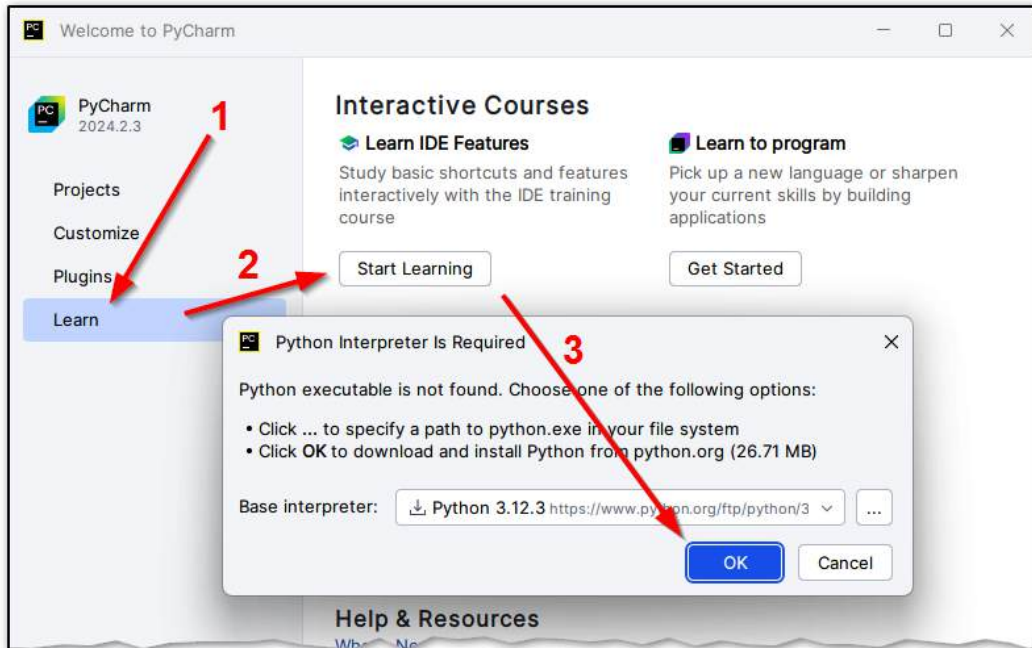


Figure 7.4 – Python installation from within PyCharm

As the screenshot suggests, navigate to **Learn | Start Learning** and click **OK** when prompted to install Python. You may already have it installed. If so, click the button with three dots (...) to locate the installed Python executable.

After installing the language interpreter, you will be taken to a learning project that the IDE will create for you, as shown in Figure 7.5 in the following section of the chapter.

Learning PyCharm

The learning project consists of a series of lessons integrated into the IDE itself. When you select a lesson, the program presents a concise step-by-step guide visually and interactively demonstrating how a particular feature works. The following screenshot shows the topics of the learning project at a high level:



Figure 7.5 – PyCharm learning project

Do not shy away from the seemingly large number of lessons. Each of these 40 lessons is tiny, but you will be better off spending time learning the IDE basics from the onset of your project instead of making mistakes and wasting time down the line.

When you eventually get to the **Git** section, the IDE will notice that you do not have the **Git** tool installed and suggest installing one for you, in the same way it installed Python. The following screenshot demonstrates Git installation:



Figure 7.6 – Git installation

Click on **Download and Install**. The **Git** for Windows package will be downloaded and automatically installed by the IDE.



It is very important that you carefully complete all lessons in the **Git** section of the **Learn IDE Features for Python** project. **Git** is a tool you will use regularly to save your project changes to your local repository and GitHub.

Project creation

Now, you can pull the project from your GitHub repository and start customizing it for your environment.

Start PyCharm and select **Projects | Get from VCS**. Select **GitHub** on the next screen and click **Log In to GitHub**. This will invoke OAuth authorization of the PyCharm IDE with GitHub. Alternatively, you can click on **Use token...** and then on the **Generate** button. The latter option will allow you to fine-tune parameters such as **Token expiration** and **Scope**.

Let's use the former option. It will bring up a preview screen, as shown in the following screenshot:

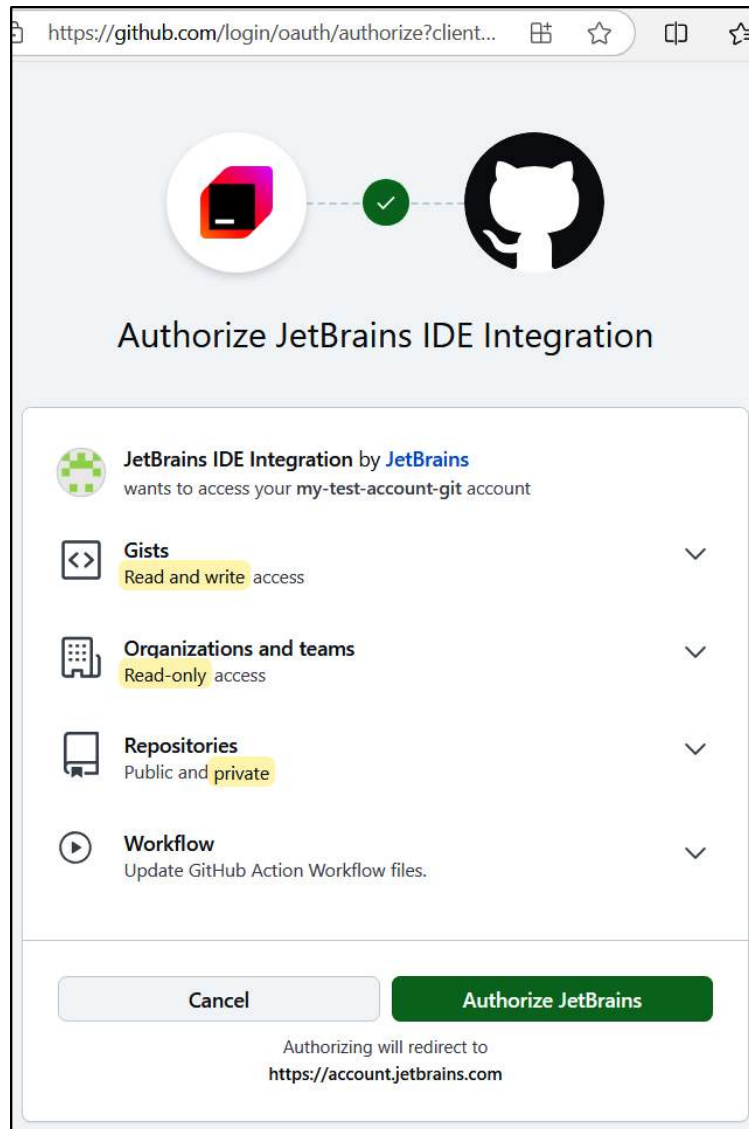


Figure 7.7 – PyCharm authorization with GitHub

Review the scope of authorization and click the **Authorize JetBrains** button. Once authorized, you will see all your repositories in the **GitHub** section of the **Get from Version Control** window, as the following screenshot demonstrates:

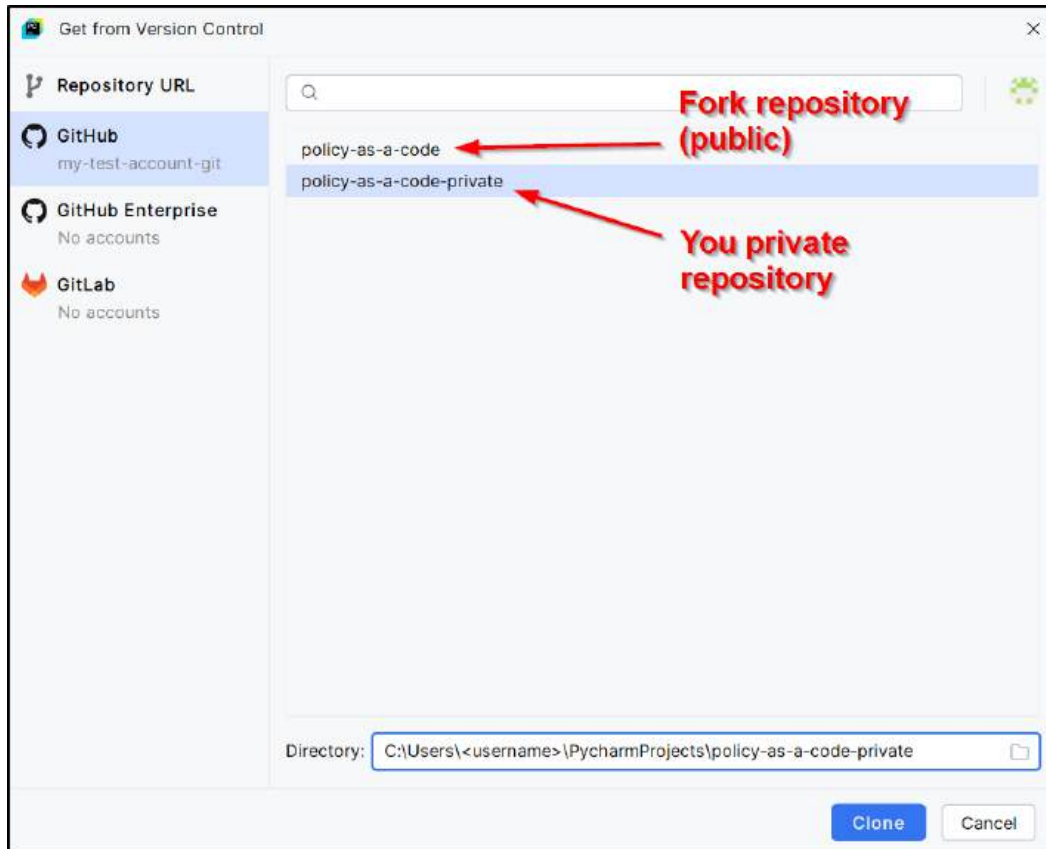


Figure 7.8 – Cloning a GitHub repository

Select the private repository and click the **Clone** button. When cloning a repository from a remote source, PyCharm will prompt you to confirm whether you trust the content. Click the **Trust Project** button.

Open the project repository folder in Windows Explorer and copy all files (which you put aside at the end of the *Copying the project source code* section of this chapter) into this folder. Return to the PyCharm window and press **Alt + O** to reach the **Commit** screen. Select all the files you have just copied, enter a comment in the provided space below, and click on the **Commit and Push...** button. PyCharm will commit the files to your local Git repository and to your private repository on GitHub.

Finally, you now have the source code to work with – congratulations! In the next chapter, we'll examine the code structure. For now, there's one last component to configure, which will enable you to run the project in a container format.

Containerization with Docker

This element is optional, but it is so easy to configure that it would be a shame to miss out on the benefits of containerization: code portability and guaranteed execution on platforms that support Docker. This includes all popular operating systems and public cloud services.

There are many ways to build and run containers, but let's focus on just one for simplicity and to illustrate the concept. You can use GitHub Actions to create the container, then store it on Docker Hub, and run it on Docker Desktop.

Docker Desktop

Docker Desktop is an all-in-one application for creating and running containers. It has a graphical interface and a set of command-line tools. We'll use it to run our project container on our software development machine.

Start by navigating to <https://docs.docker.com/desktop/> and reviewing the system requirements for your OS in the **Install Docker Desktop** section of the guide. Virtually any modern computer can run Docker, and you can even run it in a virtual machine such as VMware Workstation, provided that your hypervisor supports nested virtualization.

Download and install the application.

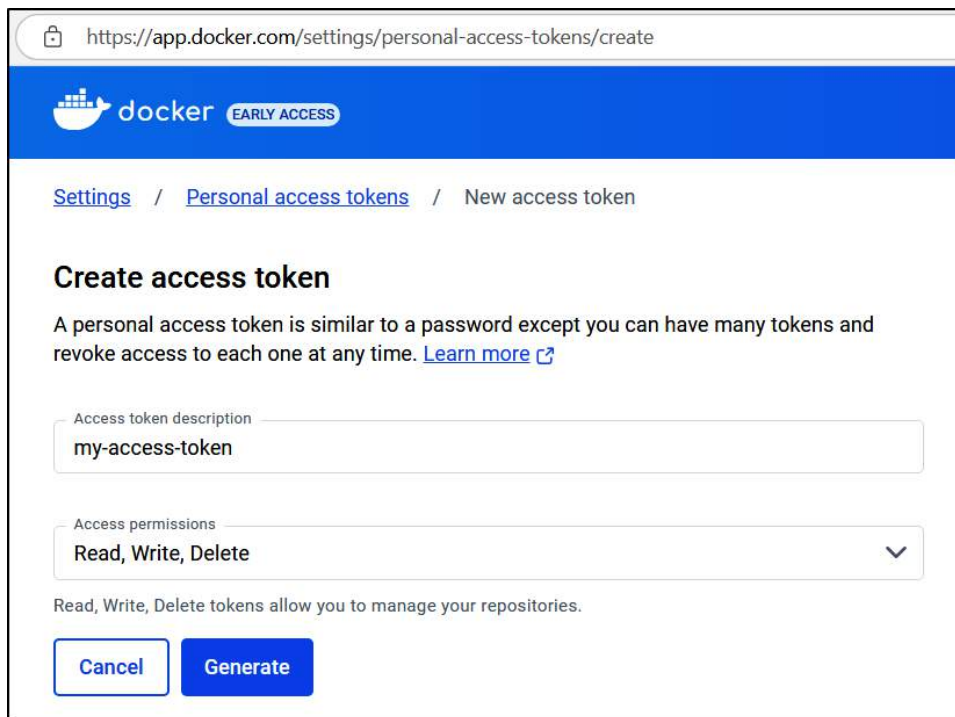
Now that you have it, you need a place that is accessible from GitHub and your machine and that can store containers.

Docker Hub

Let's start by creating a free account on Docker Hub, a go-to container marketplace where we can store our project container.

Navigate to <https://hub.docker.com> and click on the **Sign up** button. You can follow a traditional registration process by specifying your email address and choosing a username and password. Alternatively, you can make your life easier by proceeding with your GitHub account.

Once logged in, navigate to your account settings and create a personal access token, as demonstrated in the following screenshot:



The screenshot shows a web browser window with the URL `https://app.docker.com/settings/personal-access-tokens/create`. The Docker logo and 'EARLY ACCESS' badge are in the top blue header. The breadcrumb trail is 'Settings / Personal access tokens / New access token'. The main heading is 'Create access token'. Below it, a paragraph explains that a personal access token is similar to a password but can have many tokens and be revoked at any time, with a link to 'Learn more'. The form contains two fields: 'Access token description' with the text 'my-access-token' and 'Access permissions' set to 'Read, Write, Delete'. A note below the permissions field states: 'Read, Write, Delete tokens allow you to manage your repositories.' At the bottom are 'Cancel' and 'Generate' buttons.

Figure 7.9 – Creation of a personal access token

Create a meaningful description, make sure the access permissions are set to **Read, Write, Delete**, and click on the **Generate** button.

Copy the generated token to a safe place (i.e., a password manager tool). You can also go ahead and log in to Docker Hub from the CLI by running the following:

```
C:\>docker login -u <your account name>
```

When prompted, enter the token as a password. This will authenticate your Docker environment (including the GUI component of Docker Desktop) to Docker Hub.

Now, you can test your Docker setup by running the `docker run hello-world` command, as shown here:

```
C:\>docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
```

```
c1ec31eb5944: Download complete
Digest:
sha256:91fb4b041da273d5a3273b6d587d62d518300a6ad268b28628f74997b93171b2
Status: Downloaded newer image for hello-world:latest
Hello from Docker!

This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub
    (amd64)
 3. The Docker daemon created a new container from that image which runs
    the executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
    sent it to your terminal.
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.



🔒 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



This output proves that you configured everything correctly.

GitHub Actions

To conclude the chapter, let's briefly discuss how the container will be created.

It will be built by the GitHub service called **GitHub Actions** based on the instructions in the `docker-build.yml` file (stored in the `.github` folder). The service will interpret the commands in this YAML file and invoke a Docker daemon in a special virtual machine called a **runner**. A runner is a virtual machine that executes jobs in GitHub Actions workflows. It can be hosted by GitHub (this is a default configuration), or you can host one for yourself.

The Docker daemon will use the code in the Dockerfile (located in the root of your repository) to create a container and store it in Docker Hub.

If you open this file, you will see a section that defines credentials for Docker Hub:

```
username: ${ secrets.DOCKER_USERNAME }  
password: ${ secrets.DOCKER_PASSWORD }
```

As you can see, these entries are just variable names; the actual values are defined in the **Settings | Secrets and variables** section of your GitHub repository, as shown in the following screenshot:

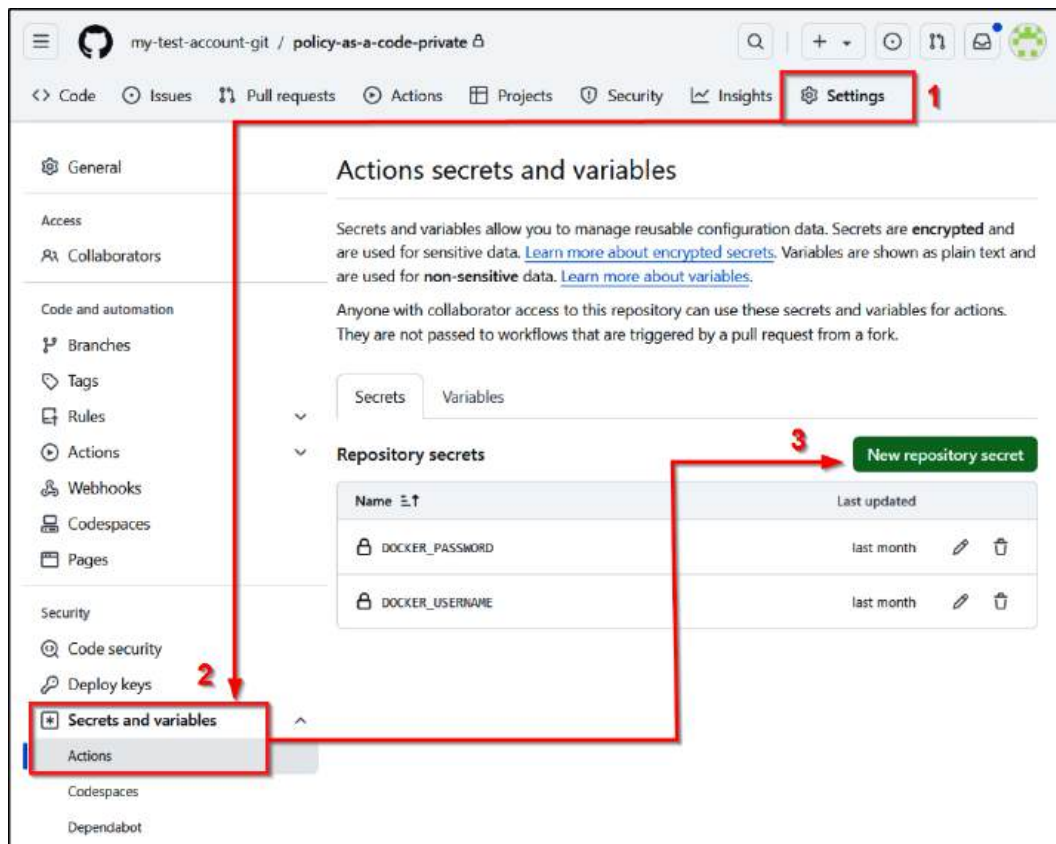


Figure 7.10 – Creating secrets for GitHub Actions

Click on **New repository secret** and enter `DOCKER_USERNAME` as the secret name and your Docker Hub username as a value. Then, repeat the process for `DOCKER_PASSWORD`. Enter the personal access token you created earlier as a value for this secret.

This step concludes the setup.

Summary

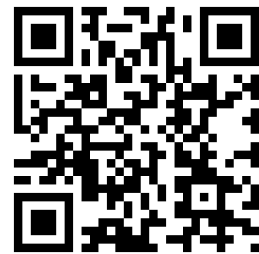
This chapter was brief but packed with activity. You prepared everything needed for practical work on customizing the source code provided with the book. You installed the PyCharm IDE for Python and set up a GitHub account to store the code. You forked the original repository and then created a private one to keep a copy of the code hidden from the public, making it ready for further customization. Additionally, you created an account on Docker Hub to store a containerized version of the code and installed Docker Desktop to run containers.

The next two chapters will focus on coding and troubleshooting. You will review the project structure and implementation details, learn the basics of the SDK and XML API, and modify the code to meet the requirements outlined in *Chapter 2*.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



8

Policy to Code: Foundations

In *Chapters 5* and *7*, you set up your development environment, obtained a copy of the project code, and learned about the core principles that should guide your firewall security policy. Now, it's time to apply that knowledge by exploring the PAN-OS SDK for Python – the framework that will help you turn these principles into an effective firewall security policy.

This chapter provides a foundation for working with the SDK efficiently. You'll become familiar with its core components, understand how data flows through the project, and learn the prerequisites for successful implementation. Additionally, you'll discover important caveats to help you avoid common mistakes. Mastering these basics will prepare you for more advanced topics such as debugging and customization.

This chapter will cover the following topics:

- First steps with the PAN-OS SDK for Python
- Separation of code and data
- Repository structure and key definitions
- Representation of the policy rules
- Firewall policy creation considerations
- Firewall policy creation algorithm

By the end of this chapter, you'll understand how to use the PAN-OS SDK for Python to automate firewall policy management. You'll know how to organize your code and data efficiently, understand the repository's structure, and follow best practices for creating policies. These skills will enable you to develop reliable, maintainable configurations that meet your security needs while paving the way for more advanced customization in *Chapter 9*.

Technical requirements

You must have a lab firewall against which to run the scripts and commands. You will also need the PyCharm IDE and a copy of the project code that you obtained in the previous chapter.

First steps with the PAN-OS SDK for Python

Chapter 6 briefly discussed the PAN-OS XML API and PAN-OS SDK for Python. The latter is much easier to use than the XML API as it provides a good level of abstraction from the complexities of the native API, so we'll use it when possible. However, if a feature we need is not implemented in the SDK, we'll resort to the regular XML API.

In this chapter, I will provide concise snippets of code to illustrate various aspects of the task at hand. To explore them further, refer to the relevant modules and functions of the project code referenced throughout the chapter. The official PAN-OS SDK for Python reference guide at <https://pan-os-python.readthedocs.io/en/latest/reference.html> will also be extremely helpful in understanding the relationships between different SDK classes and available class methods.

Connecting to a firewall or Panorama with the PAN-OS SDK for Python

Now, navigate to **View | Tool Windows | Python Console** in PyCharm and connect to a firewall. Type the following in the console to do so:

```
from panos.firewall import Firewall

fw = Firewall("<panos_address>", "<username>", "<password>")
```

An equivalent initialization for Panorama is as follows:

```
from panos.panorama import Panorama

pn = Panorama("<panos_address>", "<username>", "<password>")
```

Replace the arguments in the double quotes with your own values and press *Enter*. The interpreter will create an instance of the `Firewall` | `Panorama` class and assign it to the `fw` | `pn` variable.

Let's use a firewall in our examples.

If you type the name of the variable we have just created, followed by a dot, PyCharm will show a context menu with all methods, functions, and properties available for this `Firewall` class instance. You may find properties such as `content_version`, `platform`, `serial`, and so on. However,

if you pick any of them and press *Enter*, the output will be empty, as their values are not populated during the initialization of the class instance. To populate the properties with values, you must invoke a method. In this case, it would be `refresh_system_info()`. Once you do so, the values become available, as shown in the following example:

```
Python Console>>> from panos.firewall import Firewall
>>> fw=Firewall("<fw_address>", "<username>", "<password>")
>>> fw.platform
>>> fw.content_version
>>> fw.refresh_system_info()
SystemInfo(version='10.2.13-h5', platform='PA-440', serial='<serial
number>')
>>> fw.platform
'PA-440'
>>> fw.content_version
'8910-9047'
```

This sequence of commands in the Python console demonstrates that the `platform` and `content_version` properties of the `Firewall` object are empty right after the firewall object instantiation. They are populated with values only after you call the `refresh_system_info()` method.



When you create objects and invoke class methods, you can explore their current values in the right-hand side pane of the Python console window.

Let's do something more exciting than viewing your firewall model name.

Working with applications and policies

This section discusses how various SDK classes work together to control firewall applications and security policies. An instance of one class can often be provided as a parameter to a method of another class. Grasping these interactions is essential for efficiently automating policy management tasks.

For example, take a look at the applications your firewall knows about:

```
>>> from panos.predefined import Predefined
>>> predefined_objects = Predefined(fw)
>>> predefined_objects.refreshall_applications()
```

Here, you import the Predefined class from an SDK module of the same name and create an instance of it, having passed the firewall object as an argument. Then, you retrieve all built-in applications by calling the `refreshall_applications()` method.

Now, you can count these applications:

```
>>> len(predefined_objects.application_objects)
5072
```



You will see a different number of applications unless you install the exact content version (8910–9047) specified in the code snippet from the previous section. This number (5072) also includes internal applications not visible in the firewall UI, which are useless for our security policy since they cannot be referenced in policy rules. Therefore, we will disregard them in the upcoming steps.

Let's explore these applications further:

```
>>> all_apps_metadata = []
>>> for obj in predefined_objects.application_objects:
...     app_meta = predefined_objects.application(obj)
...     all_apps_metadata.append(app_meta)
```

In this code snippet, you initialize an empty list called `all_apps_metadata` and populate it with attributes of each predefined application.

If you explore the resultant list in the Python console object browser, you will discover application attributes familiar to you from *Chapter 3* and the PAN-OS application browser. Go ahead and grab all available application subcategories because we will rely heavily on them in our project:

```
>>> list_of_subcats = []
>>> for app in all_apps_metadata:
...     if app.subcategory is not None:
...         list_of_subcats.append(app.subcategory)
>>> list_of_subcats = list(dict.fromkeys(list_of_subcats))
>>> list_of_subcats.sort()
```

In this code, you create an empty list to hold all subcategories and then populate it by iterating through the `all_apps_metadata` list (created in the previous code snippet) and retrieving each application's subcategory attribute. You ignore the internal apps, which all have their subcategory attribute set to `None`.

Then, you deduplicate and sort the list, which gives you all application subcategories known to the firewall and familiar to you from **OBJECTS | Applications** in the PAN-OS administrative user interface. The list is shown here:

```
>>> list_of_subcats
['analytics', 'artificial-intelligence', 'audio-streaming', 'auth-
service', 'content-management', 'customer-service', 'data-privacy',
'database', 'design', 'development', 'email', 'encrypted-tunnel', 'erp-
crm', 'file-sharing', 'gaming', 'general-business', 'hr', 'ics-protocols',
'infrastructure', 'instant-messaging', 'internet-conferencing', 'internet-
utility', 'iot-management', 'ip-protocol', 'it-infrastructure', 'it-
management', 'management', 'marketing', 'medical', 'office-programs',
'photo-video', 'proxy', 'remote-access', 'routing', 'security', 'social-
business', 'social-networking', 'software-development', 'software-update',
'storage-backup', 'supply-chain-logistics', 'voip-video', 'web-posting']
```

Now, let's see what you can do with security policies. The policies module is responsible for this. Let's grab the necessary classes from it:

```
>>> from panos.policies import SecurityRule, Rulebase
```

The `Rulebase` class represents the firewall's ruleset and includes all policy types supported by the SDK: security, decryption, nat, authentication, and application-override.

As you remember from the *Panorama 101* section in *Chapter 5*, Panorama does not have a flat policy structure. In virtue of that, its ruleset is represented by two classes rather than one: `PreRulebase` and `PostRulebase`.

The `SecurityRule` class represents individual security policy rules within a rulebase.

Since we're focusing on a firewall in all examples, let's instantiate the `Rulebase` class and associate it with your firewall object:

```
>>> myrulebase = Rulebase()
>>> fw.add(myrulebase)
```

Once completed, you can retrieve all security rules with the `refreshall()` method and output the result:

```
>>> current_security_rules = SecurityRule.refreshall(myrulebase)
>>> for rule in current_security_rules:
...     print(rule.name)
...
name-resolution
access-to-example.com
access-to-example.net
default-browsing
```

The bottom four lines are the names of the security policy rules of my lab firewall. Your output will be different. You may want to create some dummy security rules in the PAN-OS UI with the preceding names before executing these commands in the Python console. I will continue using this sample policy in the following examples.

Let's create a rule that allows the ping application immediately after the name-resolution rule, shall we?

First, define the actual rule that needs to be created:

```
>>> new_rule = SecurityRule(
...     name="network-troubleshooting",
...     fromzone=["any"],
...     tozone=["any"],
...     application=["ping"],
...     service=["application-default"],
...     action="allow"
... )
```

Then, find the index of the rule of interest (the position at which you want to insert the new rule):

```
>>> position = myrulebase.find_index("access-to-example.com",
SecurityRule)
```

Rule positions start from 0, and therefore, our position is going to be 1.

Insert the rule into the rulebase:

```
>>> myrulebase.insert(position, new_rule)
<SecurityRule network-troubleshooting 0x13aeb785590>
```

Finally, apply the updated ruleset to the firewall:

```
myrulebase.apply()
```



Reminder

So far, all your changes have been applied to the firewall's candidate configuration and will not impact any traffic unless you commit the staged change through the firewall UI or call the `commit()` method on the firewall object.

If you look at the firewall security policy in the PAN-OS graphical user interface, you will see that the new rule has now appeared right after the name-resolution rule.

Now, let's remove a few rules. There are several methods to do this. Although they achieve the same outcome, they vary greatly. Let's examine how each method works.

Performance optimization

Open the firewall configuration log (**MONITOR | Logs | Configuration**) to observe the effect of the API calls you initiate, and then run the following commands in the Python console:

```
>>> myrulebase.find("access-to-example.com", SecurityRule).delete()
>>> myrulebase.find("access-to-example.net", SecurityRule).delete()
>>> myrulebase.find("default-browsing", SecurityRule).delete()
>>> myrulebase.find("network-troubleshooting", SecurityRule).delete()
```

(If necessary, replace the rule names with the ones you have in your test policy.)

If you refresh the log now, you will see four separate delete operations, as shown in the following screenshot:

ADMINIS...	CLIENT	COMMA...	RESULT	CONFIGURATION PATH	FULL PATH
admin	Web	delete	Succeeded	vsys vsys1 rulebase security rules network-troubleshooting	/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/rulebase/security/rules/entry[@name='network-troubleshooting']
admin	Web	delete	Succeeded	vsys vsys1 rulebase security rules default-browsing	/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/rulebase/security/rules/entry[@name='default-browsing']
admin	Web	delete	Succeeded	vsys vsys1 rulebase security rules access-to-example.net	/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/rulebase/security/rules/entry[@name='access-to-example.net']
admin	Web	delete	Succeeded	vsys vsys1 rulebase security rules access-to-example.com	/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/rulebase/security/rules/entry[@name='access-to-example.com']

Figure 8.1 – Multiple delete operations

You ran four commands, each producing a separate API call against the firewall device, which was reflected in the log.



Strictly speaking, the entries in this log correspond to configuration commands, not API calls. Therefore, the multi-config API requests described later in this section will result in a separate log entry for each sub-command included in a single multi-config API request.

Now, roll back the made changes, as demonstrated in the following screenshot:

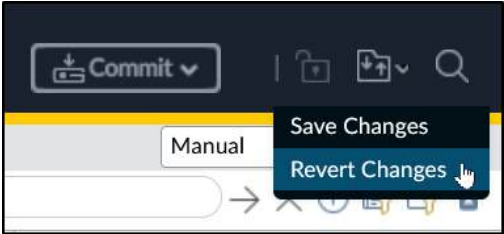


Figure 8.2 – Cancellation of changes

Refresh the in-memory configuration tree in the Python console (so that your code knows that the deleted rules have reappeared in the firewall configuration after you reverted all changes):

```
>>> current_security_rules = SecurityRule.refreshall(myrulebase)
```

Now, delete the same rules with the following commands:

```
>>> myrulebase.remove(myrulebase.find("access-to-example.com",
SecurityRule))
>>> myrulebase.remove(myrulebase.find("access-to-example.net",
SecurityRule))
>>> myrulebase.remove(myrulebase.find("default-browsing", SecurityRule))
>>> myrulebase.remove(myrulebase.find("network-troubleshooting",
SecurityRule))
```

In each command in this code snippet, you search for a `SecurityRule` object with a given name and call the `remove()` method on the `Rulebase` instance, referencing the found object.

Refresh the logs and review the security policy in the firewall UI. There will be no new configuration log entries, and the rules will remain in the security policy because the preceding `remove()` methods only affected the `Rulebase` instance in the computer memory, not the firewall itself.

Now, run the command as follows:

```
>>> myrulebase.apply()
```

The `apply()` method is executed against the PAN-OS device. The rules will now be gone from the policy, and there will be only one record in the configuration log:

ADMINIS...	CLIENT	COMMA...	RESULT	CONFIGURATION PATH	FULL PATH
admin	Web	edit	Succeeded	vsys vsys1 rulebase	/config/devices/ entry[@name='localhost.localdomain']/ vsys/entry[@name='vsys1']/rulebase

Figure 8.3 – Single edit operation substituting multiple delete ones

This demonstrates how you can replace multiple API calls concerning individual lower-level objects (security rules) with a single API call addressing a higher-level object (rulebase).

While this may seem minor in this example, such an optimization can make a significant difference in a production environment. Imagine that you have 600 rules to delete, and the latency of your connection to the PAN-OS device is 100 ms (which is not uncommon in the modern world of remote working and distributed global networks).

Even if your firewall could process each API request instantaneously (which it does not, unfortunately), going through all objects at a rate of 10 requests per second would still take a minute. However, in real-world situations, a speed of just 1–2 requests a second is much more common, meaning you’ll likely wait about 5–10 minutes for this object deletion to finish. Compare this to a few seconds for a single API call (large calls take longer to process), and you will appreciate the difference.

Unfortunately, there are caveats.

While the method I described appears effective on paper and in a simple test against a firewall like the one you just conducted, it may not fully achieve the goal in all scenarios and configurations. The issue is that some parts of the configuration might also be stored in an internal database (often referred to as MongoDB) in addition to the XML configuration file. This is done to enable other advanced PAN-OS management features.

If you reproduce an equivalent sequence of commands against a Panorama and then try to delete security profiles or profile groups used in the deleted security rules, this will likely fail with an error message similar to **Failed to delete Security Profile Group -<profile_group_name>. <profile_group_name> cannot be deleted because of references from:**

The message appears truncated, but it is complete—there won’t be anything after the colon. While not very informative, the error message indicates that the method did not fully remove the rules, and the profiles cannot be deleted because of this. Some metadata related to the deleted policy rules remains in the database, preventing the security profiles from being deleted successfully.

At the same time, unlike deletion, creating multiple rules using this method may work well for relatively small rulesets. In my experience, this method worked perfectly with policies of 300–400 rules. Strange issues may begin to occur once the policy exceeds these numbers. For example, you might see duplicate zones, applications, and services within the same policy rule.

Another important point to remember when working with parent objects is that they may contain child objects of not only the type you’re interested in but also other types. If you don’t account for this, leaving them in the default “empty” state and applying the updated parent object to the PAN-OS device will wipe out all those non-initialized child objects.

For example, in the previous case, we finished with the `rulebase.apply()` command. If there were decryption or NAT policy rules on the firewall, they would also have been unintentionally deleted along with the security rules. This would happen because we did not update our configuration tree with `DecryptionRule.refreshall(rulebase)` and `NatRule.refreshall(rulebase)` commands before applying the entire rulebase object to the firewall.

The bottom line is that you should be very careful with bulk operations. The only guaranteed method for bulk operations is the `multi-config` request, which is very briefly described in the online documentation here: <https://docs.paloaltonetworks.com/pan-os/11-1/pan-os-panorama-api/pan-os-xml-api-request-types/configuration-api/multi-config-request-api>.

When working with the PAN-OS SDK for Python, you can make this type of bulk request using the `multi_config()` method of the `xapi` module.



Ensure you have the latest version of the `pan-device` package. Earlier versions of this module do not have the `multi_config()` method.

The tricky part of this method is that you have to create XML code to pass as an argument. Basically, all you need to do is build a string that follows this pattern:

```
<multi-config>
  <action1 id="action1-id" xpath="xpath1">
    <element-xml></element-xml>
  </action1>
  ...
  <actionN id="actionN-id" xpath="xpathN">
    <element-xml></element-xml>
  </actionN>
</multi-config>
```

While you may start thinking about deep-diving into your firewall XML config or API/CLI debugging to figure out the correct XPath and the element, you do not need to do so. Luckily, the SDK provides two information methods that can help you retrieve these object attributes. First, you must create an object (such as `SecurityRule`) as usual and add it as a child object to a corresponding higher-level object (such as `Rulebase`), eventually traceable to a firewall or Panorama instance. This will ensure that the XPath of the object is complete. Once your object is initialized this way in the configuration tree, you can grab its complete XPath by calling the `xpath()` method. In turn, `element_str()` will give you a byte-string representation of the *element* part of your object definition.

Here is a complete example of creating three address objects using the multi-config approach:

```
from panos.firewall import Firewall
from panos.objects import AddressObject
fw = Firewall('<fw_address>', '<username>', '<password>')

address_objects = [
    AddressObject(name='addr1', type='ip-netmask', value='10.0.0.1'),
    AddressObject(name='addr2', type='ip-netmask', value='10.0.0.2'),
    AddressObject(name='addr3', type='ip-netmask', value='10.0.0.3')
]

action_id = 1
multi_config_xml = '<multi-config>'
for address in address_objects:
    fw.add(address)
    multi_config_xml = multi_config_xml + f'<edit id="{action_id}"
xpath="{address.xpath()}">{address.element_str().decode()}</edit>'
    action_id = action_id + 1
multi_config_xml = multi_config_xml + '</multi-config>'
fw.xapi.multi_config(multi_config_xml)
```

This code snippet initializes a firewall object and creates a list of three address objects. A counter is set up to assign unique IDs to each sub-operation in the multi-config process. Additionally, a string is prepared to hold the XML element for the entire multi-config operation. The code then loops through all address objects, attaching each to the firewall so that the `xpath()` method can generate a complete XPath. Using `xpath()` and `element_str()`, it constructs the XML for each sub-operation. The `decode()` method converts each byte string into a regular string to match the other parts of the `multi_config_xml` variable. With each loop cycle, the ID is incremented by 1, gradually building the `multi_config_xml` string. After the loop finishes, a closing XML tag is added, and the `multi_config()` method from the `xapi` module is called with the complete XML string as its argument. The process is then complete.

The PAN-OS SDK for Python suggests an alternative approach for bulk operations. The following methods are available for bulk operations on objects: `apply_similar()`, `create_similar()`, and `delete_similar()`. These methods bundle all objects of the same type located in the configuration tree at the same XPath and push them all in one big request to the firewall.

For example, you can add all address objects to your Firewall or Panorama DeviceGroup class instance *without* invoking the `apply()` method on the individual addresses and then call the `apply_similar()` method on the last address object, as shown here:

```
for address in address_objects:
    fw.add(address)
address.apply_similar()
```

In this code snippet, you loop through a list of address objects, adding each of them as a child object to the firewall object. This only builds up a configuration tree in computer memory. Then, you take the last address object and invoke its `apply_similar()` method. The method takes all objects of the same type and with the same XPath found in the configuration tree and pushes them to the firewall as individual entries bundled in a single API call.

Here is the XML API request produced by this `apply_similar()` call:

```
xpath=/config/devices/entry[@name='localhost.localdomain']
/vsys/entry[@name='vsys1']
&element=<address>
  <entry name="addr1">
    <ip-netmask>10.0.0.1</ip-netmask>
  </entry>
  <entry name="addr2">
    <ip-netmask>10.0.0.2</ip-netmask>
  </entry>
  <entry name="addr3">
    <ip-netmask>10.0.0.3</ip-netmask>
  </entry>
</address>
&type=config
&action=edit
&key=
```

Figure 8.6 – The `apply_similar()` method translated to an XML API request (decoded form data)

HTTP headers of this request would be identical to the previous multi-config example and, therefore, not shown. If you compare this API request to the multi-config request in *Figure 8.5*, you will see that we have one big `config/edit` command in *Figure 8.6*. The SDK sends all `AddressObject` child elements in the configuration tree in one big element. Only one action (`edit`, for example) is possible. It will be applied to all specified entries.

If the firewall configuration already had 100 other address objects, and we had pulled them all with the `AddressObject.refreshall(fw)` command, then our last for loop through only 3 new address objects, followed by the `apply_similar()` method, would have produced an XML API call with 103 address objects in the `element` parameter of the form. This is not something you would typically want to do.

Now, let's compare the preceding screenshot to the XML API representation of a regular SDK `apply()` call, `fw.add(address_objects[0]).apply()`, pushing a single address object:

```
xpath=/config/devices/entry[@name='localhost.localdomain']
/vsys/entry[@name='vsys1']/address/entry[@name='addr1']
&element=<entry name="addr1">
| <ip-netmask>10.0.0.1</ip-netmask>
</entry>
&type=config
&action=edit
&key=
```

Figure 8.7 – The `apply()` method translated to an XML API request (decoded form data)

As you can see, the difference between the `apply_similar()` (Figure 8.6) and `apply()` (Figure 8.7) methods is only in the number of entries and the `xpath` representation.

Unfortunately, these `apply_similar()` methods are not guaranteed to work on all types of objects. In some cases, Palo Alto Networks revoked support for this type of XML API request for certain types of objects.

In our project, we will primarily utilize `multi-config` XML API requests, opting for individual API calls only when needed. Next, let's quickly review the methods offered in the SDK.

SDK methods

The PAN-OS SDK for Python provides a comprehensive set of methods to interact with your firewall or Panorama device, organized into four distinct groups:

- **Device methods:** Used to retrieve basic system information (e.g., `refresh_system_info()`) and manage connections
- **Navigation methods:** Help locate objects within the configuration tree (e.g., `find_index()` for policy rules)
- **Configuration methods:** Allow you to add, modify, or remove objects from the running configuration (i.e., `add()`, `remove()`, and `apply()`)

- **Information methods:** Provide details about objects, such as their XPath representation (`xpath()`) or XML definition (`element_str()`), which are essential for bulk operations

Understanding these groupings helps streamline policy automation by making it easier to do the following:

- Retrieve system data efficiently (e.g., checking firewall version before applying changes)
- Navigate and modify configurations programmatically (e.g., inserting a rule at a specific position)
- Optimize performance with bulk operations (e.g., using `xpath()` and `element_str()` for multi-config requests)



While the SDK covers the most common automation needs, there are some gaps in its functionality. For example, it lacks direct support for security profiles (such as vulnerability or URL filtering profiles). In these cases, you'll need to use the PAN-OS XML API directly to perform operations that aren't covered by the SDK's higher-level abstractions.

You can find descriptions of commonly used SDK methods in the official documentation on this page: <https://pan-os-python.readthedocs.io/en/latest/useful-methods.html>. While not every method is listed there, these will cover most automation needs. In *Chapter 9*, we'll explore how to supplement SDK functionality with direct XML API calls when necessary.

Now, let's discuss the source data component of our automation project.

Separation of code and data

In theory, you can define every element of the policy right in the source code, as was done in the code snippets in the previous sections of this chapter. In practice, this is not always a good idea. Here are just *some* of the reasons:

- Every time you need to change the data, you must modify the code. This is time-consuming and prone to errors. It's much easier to add a line in a purpose-built CSV file or change a definition of an object in a JSON file than locating and updating a complex definition of a class instance right in the source code, where you must be careful about maintaining the correct Python syntax and taking care not to damage other code in the same file.

- Embedding data in the code does not scale well. While it's easy to define a handful of address objects right in the code, as we did in one of our preceding examples, when their number grows to hundreds and thousands, it starts to look ugly and is challenging to handle in an IDE editor.
- Collaboration and segregation of duties become nearly impossible. While a programmer may need to update a function in the source code, a security analyst may need to add or remove a few address objects in the same file. This will introduce unnecessary friction and delays, not to mention the impossibility of applying a role-based access control that can be done only at the file level in version control systems.
- When the same code is deployed across different environments (for example, development vs. production), hardcoded data makes it challenging to customize behavior for each environment.

This list is by no means complete. Programming books and forums go to great lengths to describe the disadvantages of embedding data in the code. The bottom line is: do not do this except in limited scenarios when carefully weighing all pros and cons and concluding that the benefit of defining that object in the code outweighs the efforts required to place this definition in a separate data file.

In our project, we will follow this basic principle in most scenarios. We will place our business requirements, object definitions, and policies in separate dedicated files. Depending on the type of data, we will use mostly **comma-separated values (CSV)**, **JavaScript Object Notation (JSON)**, and **YAML** file formats.

Let's explore where all components live in the repository file structure.

Repository structure and key definitions

Now that you're familiar with the core components of the PAN-OS SDK for Python and how to work with firewall objects, let's take a closer look at the project repository. In this section, we will explore the organizational structure of the code base, which is designed to mirror the hierarchical layout of the PAN-OS administrative interface. This alignment ensures that anyone familiar with the firewall's configuration menus can intuitively navigate the project files.

Understanding the repository structure is essential because it determines where and how policy-related data is stored. Many folders within the repository correspond to a specific configuration element in PAN-OS, such as objects (addresses, applications, and services), security profiles, or policies. By maintaining this parallel structure, we ensure consistency between the code base and the actual firewall configuration, making it more straightforward to manage, update, and troubleshoot policy definitions.

Additionally, this section will introduce key definitions that form the basis of our automation framework. These include naming conventions, file formats such as CSV and JSON/YAML, and the distinction between static and dynamically generated configurations. Understanding these concepts will help you efficiently locate, modify, and expand policy components while following best practices.

Once you finish this section, you'll see exactly how the repository is organized – where different configurations are stored and why it's built this way for better automation. This will prepare you for the next part: turning those policies into code and using an algorithm to push them out to your firewall.

Let's begin by taking a closer look at the core folder structure of our project:

```
nextgen/  
├─.github/  
├─docs/  
├─export/  
│   └─service-desk/  
├─lib/  
├─misc/  
├─ngfw/  
│   └─device/  
│       └─certificate management/  
│           └─certificates/  
│           └─response pages/  
│   └─objects/  
│       └─addresses/  
│       └─applications/  
│       └─application groups/  
│       └─custom objects/  
│           └─data patterns/  
│           └─spyware/
```


Let's start with the `ngfw` folder and all its subfolders. As you can see, this set of directories is a close replica of the menu structure in the PAN-OS graphical interface. It is a good idea to organize folders this way so that anyone new to the project but familiar with the firewall administrative interface can find their way around.

Each subfolder under the `ngfw` folder contains a data structure describing the corresponding configuration element(s). For example, the definitions of the vulnerability profiles can be found in the `ngfw/objects/security_profiles/vulnerability` folder, and all response page templates are stored in `ngfw/device/response_pages`.

The policy rules are kept in `ngfw/policies`. You can see two folders there: `security` and `decryption`. Each contains subfolders for the PRE and POST sections of the policy, which correspond to the respective sections of the target Panorama device group. When the policy is deployed to a firewall, the rules from these two sections are concatenated: the POST rules are applied *after* the PRE ones, thus forming one continuous ruleset.

The subfolders of the PRE and POST folders are numbered. This numbering is essential here as it dictates the order of policy sections.

Contrary to its counterpart for the decryption policy, the `ngfw/policies/security/POST` folder is just a placeholder with a README file advising that no rules should be created there. This is because our code automatically generates all rules in this section based on the business requirements defined in the `/requirements/categories_app.csv` and `/requirements/categories_url.csv` files.



The CSV format is not always the first choice for software developers. However, in our case, it's a win-win. While code-friendly, the CSV format allows you to easily convert the requirements to a business-friendly Google Sheets or Microsoft Excel spreadsheet and run them by key decision makers in your organization, such as IT, business management, and compliance.

To simplify the presentation, you can hide all columns except `SubCategory`, `Action`, `Approver`, and `Description`. Once the requirements are signed off, convert them back to CSV and feed them to our Python script to generate the policy.

The former file defines what application categories you would like to control. Here is a small extract from this spreadsheet:

	SubCategory	Action	Approver	UserID	Category	Tags	Risk	ExtraApps	Excluded Apps	Description
13	erp-crm	do not manage				[Web App]	1,2,3,4			SaaS and on-premises enterprise resource planning (ERP) and customer relationship management (CRM) systems and software
14	file-sharing	manage	compliance	UG-file-sharing	general-internet, saas	[Web App]	1,2,3,4	google-drive-web		File storage and sharing applications, protocols, and cloud or SaaS services and their functions
15	gaming	do not manage			media		1,2,3			Online games and game-related services
16	general-business	do not manage			business-systems, saas	[Web App]	1,2,3,4		soap	Business related applications that do not fit into other subcategories
17	hr	do not manage					1,2,3			SaaS and on-premises human resources (HR) systems and HR support software
17	ics-protocols	deny								Industrial control and operational technology (OT) communication protocols and applications.

Figure 8.8 – Requirements for application categories

Let’s deconstruct it to understand the purpose each of its components serves. As you can see in this screenshot, every line corresponds to a standard subcategory of applications. Here, you define what you would like the policy to do with this particular set of applications and what additional criteria you would like to apply to narrow down or expand the set. Effectively, each line defines a corresponding application filter and an application group in which the filter will be included, provided the value in the Action column is not deny. If the required action is deny, then we do not need to create any application filters or groups – we’ll simply block all applications from this category unconditionally.

Let’s examine what these manage and do not manage requirements translate to in policy terms.

For example, line #14 (file-sharing) corresponds to an application filter defined as follows:

NAME	SUBCATEGORY	CATEGORY	RISK	TAGS
APF-file-sharing	file-sharing	general-internet	1	Web App
		saas	2	
			3	
			4	

Figure 8.9 – Example application filter created based on requirements

As you can see in the screenshot, it inherited **SUBCATEGORY, CATEGORY, RISK, AND TAGS** values from the respective columns in the CSV spreadsheet. This application filter is subsequently used to create a corresponding application group, shown in the following screenshot:

NAME ^	MEMBERS	APPLICATIONS	FILTERS
APG-file-sharing	2	google-drive-web	APF-file-sharing

Figure 8.10 – Example application group created based on requirements

The screenshot demonstrates that the group includes both the application filter and the google-drive-web application found in the ExtraApps column of the spreadsheet.

Subsequently, the application group itself is used to create two security rules in the POST section of our policy, exactly as we designed in *Chapter 3*. They are shown in the following screenshot:

NAME	ACTION	Source		APPLICATION	SERVICE	URL CATEGORY
		USER				
managed-apps-file-sharing-risky	✔ Allow	 UG-file-sharing		 APG-file-sharing	 service-http	high-risk
					 service-https	medium-risk
managed-apps-file-sharing-regular	✔ Allow	 UG-file-sharing		 APG-file-sharing	 application-default	any

Figure 8.11 – Example security rules generated for a managed application category

As you can see, the UserID value (UG-file-sharing) from the spreadsheet is referenced as the Source User matching criterion of the policy rules.

Finally, the Description column in the spreadsheet is used to populate the eponymous field of both policy rules for the category:

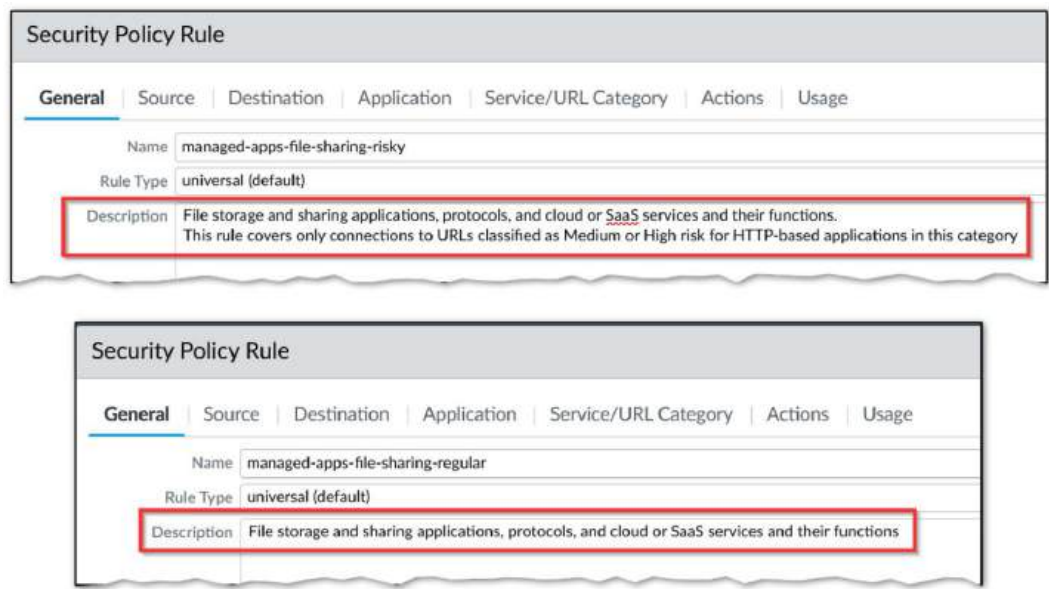


Figure 8.12 – Description field of the security rules for a managed application category

As the screenshots demonstrate, the rule covering “regular” file-sharing traffic has a description identical to what we have defined in the spreadsheet (see *Figure 8.8*). In contrast, its “risky” counterpart has a sentence added to reflect the rule’s additional risk-specific matching criterion.

Now, let’s turn to the requirements set for URL categories. Here is a small fragment of the respective CSV spreadsheet (*categories_url.csv*):

1	Category	Abbreviation	Action	Approver	UserID	Description
43	online-storage-and-backup	online-storage	manage	compliance	UG-file-sharing	Websites that provide online storage of files for free and as a service.
44	parked	parked	deny			Domains registered by individuals, oftentimes later found to be used for credential phishing. These domains may be similar to legitimate domains, for example, pal0alto0netw0rks.com, with the intent of phishing for credentials or personal identify information. Or, they may be domains that an individual purchases rights to in hopes that it may be valuable someday, such as panw.net.

Figure 8.13 – Requirements for URL categories

As you can see in the screenshot, this file is different from its application-oriented equivalent. This spreadsheet defines the Action, Approver, and UserID attributes. Our code will use this information to create two rules, as shown here, for every managed category:

NAME	ACTION	Source		APPLICATION	SERVICE	URL CATEGORY	PROFILE
		USER					
managed-urls-online-storage-and-backup-risky	Allow	UG-file-sharing		APG-web-browsing-risky	application-default	CM-online-storage_high-risk CM-online-storage_med-risk	
managed-urls-online-storage-and-backup-regular	Allow	UG-file-sharing		APG-web-browsing-regular	application-default	online-storage-and-backup	

Figure 8.14 – Example security rules generated for a managed URL category

As you can see in the screenshot and remember from *Chapter 3*, the two rules we create for each category differ by the **URL CATEGORY** field. For the “risky” rule, we create two **Category Match** custom URL categories. Unfortunately, the maximum length of a custom category name is 31 characters. Therefore, when generating a category name, we cannot simply concatenate the names of the standard categories that constitute our custom category. For instance, if we were to concatenate online-storage-and-backup with high-risk, this alone would have given us a 35-character-long string, online-storage-and-backup_high-risk, not to mention the prefix we chose according to our naming convention.

Hence, we need to abbreviate all *extra-long* category names so that their risk-aware variants do not exceed the 31-character limit. The Abbreviation column in the spreadsheet serves this very purpose. It contains a shortened version of the category name so that it can be concatenated with the “risk” categories without violating the system’s limitations.

Along with the two security policy rules we just reviewed, the action assigned to a URL category in the CSV spreadsheet must be reflected in two dynamically generated URL-filtering profiles used in the rules that handle default web browsing. This ensures that when a user tries to access a URL from a blocked or managed category that they do not have permissions for, they will be blocked. On the contrary, we would have the alert action if the category is non-managed.

This approach is demonstrated in the example in the following screenshot:

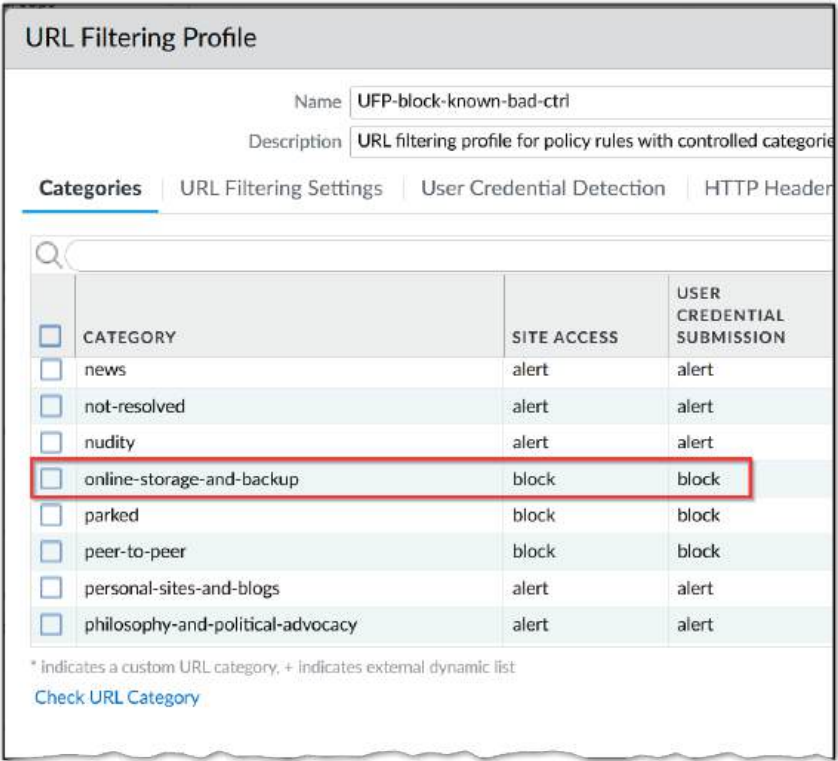


Figure 8.15 – Referencing of a managed category in a URL-filtering profile

The highlighted online-storage-and-backup category was declared managed

Suppose the user does not hit the dedicated AD-based rules for this category (shown in *Figure 8.14*) because they are not members of the specified user group. In that case, they will naturally hit the rule for default browsing to which this profile (*Figure 8.15*) is assigned. The connection will be blocked, and the firewall will display a custom block page.

If you follow along, you might have noticed that we have not yet covered the Approver field in both spreadsheets we discussed.

This field is not directly leveraged in the policy generation code. Instead, we will use it in the input metadata we generate for the service desk system as part of our policy generation process. The metadata will include names of all managed categories and approvers, which will all be used by a corresponding approval workflow in the service desk system. We will store the exported metadata in the export/service-desk folder.

Representation of the policy rules

Earlier in this chapter, we discussed separating code and data. So, how can we ensure a clean presentation of our static policy rules in code?

The CSV file format is not practical – if we try to represent policy rules as a spreadsheet, they become difficult to manage. Try exporting your policy by clicking the **CSV/PDF** button at the bottom of any policy screen in the PAN-OS UI and see how hard it is to read, understand, and make any changes to the exported spreadsheet.

JSON and YAML are the alternative formats that first come to a programmer’s mind. JSON tends to be more machine-friendly, whereas YAML is often seen as more user-oriented. Both are extensively supported across programming languages and automation tools.

The main disadvantage of a “classic” JSON representing a firewall policy in our use case is that it does not support inline comments. Workarounds exist to address this issue: you can use the newer JSON5 specification or create a preprocessor that strips comments before regular JSON processing. YAML isn’t ideal either. Among other issues, the indentation format is prone to mistakes, and debugging can be difficult. What they both have in common is that the support and required configuration in IDEs vary. Some features may work out of the box, while others may require additional setup or a paid version of the IDE, and some may not be supported at all.

Auto-completion is a crucial usability feature when creating a policy rule. Any firewall administrator naturally expects the ability to, for example, select a source zone from a drop-down list in the firewall’s administrative GUI. It would be advantageous to have this supported to some extent in code.

So, there is a “native” and, therefore, highly compatible way with all IDEs to check these options in Python. We can keep our policy rules in very lean Python files. They will contain no code as such, only the definition of data structures representing the policy. We can define each section of the policy as a tuple composed of dictionaries, where each dictionary specifies all elements of a rule.

A tuple in Python is an ordered collection of elements; therefore, it is ideal for representing a firewall policy. Dictionaries, on the other hand, are almost identical to JSON and are perfect for defining policy rules.

We can improve this method of representing policy by introducing defaults for each policy section. When you have a group of rules serving a similar purpose, all rules would naturally share common elements that don't need to be repeatedly declared in every rule. For example, source and destination zones and addresses, the group tag, security profiles, the log forwarding profile, and rule types would all typically be the same across all rules within a policy section.

Here is an example of how a tuple with section defaults can be defined:

```
section_defaults = (
{
    "name":            "MUST-BE-UNIQUE-FOR-EACH-RULE",      # Default rule name
    "type":            "universal",                        # Default rule type (universal|interzone|intrazone)
    "description":     "MUST-BE-POPULATED-FOR-EACH-RULE",   # Default description
    "tag":             group_tags["business-apps"]["name"], # Default tag(s)
    "group_tag":       group_tags["business-apps"]["name"], # Default group tag

    "fromzone":        settings.zone_inside,               # Default source zone(s)
    "source":          settings.source_address,             # Default source address(es)
    "negate_source":    False,                             # Default source negation
    "source_user":     "known-user",                       # Default source user(s)

    "tozone":          settings.zone_outside,               # Default destination zone(s)
    "destination":     "any",                              # Default destination address(es)
    "negate_destination": False,                          # Default destination negation

    "application":     "any",                             # Application(s)
    "service":         "application-default",              # Default service(s)
    "category":        "any",                             # URL categories

    "action":          "allow",                            # Default action

    "group":           "PG-apps-regular",                  # Default profile group
    "virus":           None,                              # Antivirus Security Profile
    "spyware":         None,                              # Anti-Spyware Security Profile
    "vulnerability":   None,                              # Vulnerability Protection Security Profile
    "url_filtering":   None,                              # URL Filtering Security Profile
    "file_blocking":   None,                              # File Blocking Security Profile
    "wildfire_analysis": None,                            # Wildfire Analysis Security Profile
    "data_filtering":  None,                              # Data Filtering Security Profile

    "log_setting":     settings.lfp_default,               # Default Log Forwarding Profile
    "log_start":       False,                             # Do not log at session start by default
    "log_end":         True,                              # Log at session end by default

    "target":          None,                              # "None" is equivalent to "all firewalls"
    "negate_target":   False,                             # Do not negate the target
    "disabled":        False                             # Enable the rule
}
)
```

Figure 8.16 – Section defaults for policy rules

As you can see in the screenshot, every possible attribute of a security rule is defined here. The same approach will apply to other rule types, such as decryption.

We can also reference global constants such as zone and profile names from a global settings module at the root of the repository. The same applies to tags and other attributes we may want to define in a central location.

The following code snippet shows how we import modules:

```
import settings
from ngfw.objects.tags.group_tags import group_tags
from lib.auxiliary.functions import get_source_user_for_category
```

We can import a module as a whole (the settings module in this example) or individual objects defined in the module (the group_tags dictionary and the get_source_user_for_category function). This pattern is present throughout all modules in the project repository.

Here is what the actual rules would look like:

```
section_defaults = (
> {...}
)
# =====
#
# Create security rules below this comment section.
# Define only attributes that differ from the section defaults specified above this comment
#
# =====
section_security_rules = (
    # The Firm's in-house generic trusted web apps
    {
        "name":      'firm-generic-app',
        "group":     'PG-apps-trusted',
        "application": 'APG-web-browsing',
        "service":    ['service-https', 'service-http'],
        "category":   'UCL-firm-generic-app',
>     "description": "...",
    },
    # Legacy apps defined in higher level device groups and provisioned via tagging and an application filter
    {
        "name":      'firm-custom-apps-legacy',
        "group":     'PG-apps-trusted',
>     "application": 'APF-custom-apps-legacy',
        "description": "...",
    },
    # Business apps approved for use by all employees firm-wide regardless of their category
    {
        "name":      'calendly-app',
        "application": 'calendly',
>     "description": "...",
    },
),
```

Figure 8.17 – Security rules defined as a deviation from the section defaults

In this example, the `firm-generic-app` rule overrides the security profile group, application, service, and category fields of the section defaults. The second rule overrides only the security profile group and application attributes, while the third rule defines only application.

Needless to say, the name and description fields must be defined for each rule.

Here is a snippet of code demonstrating how we can ingest the policy defined as described previously:

```
def find_and_import_rules(directory)
    all_rules = [] # Initialize a list to hold all policy rules
    # Traverse the directory structure
    for root, dirs, files in os.walk(directory):
        # Sort directories and files to ensure consistent file order
        dirs.sort()
        files.sort()
        # Take into account only files named "rules.py"
        for file in files:
            if file == "rules.py": # the filename is hardcoded
                file_path = os.path.join(root, file)

                # Normalize path to ensure consistency in slashes
                file_path = os.path.normpath(file_path).replace(os.sep, "/")
                # Create a unique module name based on the file path
                # First, we remove the file extension
                module_name = file_path.replace(".py", "")
                # Second, we substitute all slashes, dots, dashes,
                # and spaces with underscore
                module_name = re.sub(r"[ /.-]", "_", module_name)
                # Third, we load the module from the normalized file path
                try:
                    module = load_module_from_file(module_name, file_path)
                # If there is a syntax error, we display details and exit
            except SyntaxError as e:
                print(f"!!! Syntax error in the file: '{file_path}'")
                print(e)
                sys.exit(1)
```

```

    # If the module loaded successfully, we check if
    # 'section_rules' and 'section_defaults'
    # are present in the imported module

    if hasattr(module, 'section_rules') and hasattr(module, 'section_
defaults'):
        # Finally, we merge defaults with each rule,
        # ensuring not to overwrite existing keys
        updated_rules = [] # List to hold all rules from the module
        folder_name = os.path.basename(root)
        for rule in module.section_rules:
            # Start with a copy of the defaults
            merged_rule = module.section_defaults.copy()
            # Update the merged rule with the actual rule,
            # preserving the rule's original keys
            merged_rule.update(rule)
            updated_rules.append(merged_rule)

        all_rules.extend(updated_rules)
    return all_rules

```

As the inline comments explain, we traverse all subfolders of a given folder, looking for files named `rules.py`. We import these files as Python modules and then look for tuples called `section_defaults` and `section_rules`. The former contains a single dictionary with all attributes of a policy rule, while the latter contains multiple dictionaries, each representing the rule's deviations from the defaults. The resultant list of policy rules is returned to a higher-level function that, in turn, would create the actual policy rules in the form of `SecurityRule` or `DecryptionRule` class instances, which we got familiar with earlier in this chapter.

Now that we have discussed the basic building blocks, API capabilities, and the input data format, let's consider some other essential points and caveats that will help us ensure the successful deployment of our policy.

Firewall policy creation considerations

Fundamentally, the goal of deploying a complete firewall policy via automation is to ensure that the policy objects and logic prescribed in the code are identical to those in the resultant firewall configuration.

For example, when we add a new address object or a couple of security policy rules in the code and run our script, the exact change must happen in the firewall configuration – the address object and the rules must appear in the configuration. Equally important is that they appear in the correct location and order (speaking of the policy rules). When we delete an **External Dynamic List (EDL)** definition in the code and run the script, the corresponding EDL must disappear from the firewall. When our business requirements change (we introduce a new managed URL or application categories) and we run the script, new policy rules, application filters, and groups must be created, and URL-filtering profiles must be updated.

While it sounds relatively simple, it is important to understand the details.

Stateful frameworks such as Terraform have the “luxury” of a state file. One must be maintained for each environment where we want to achieve and maintain the desired final state of a configuration. We do not find this in the realm of traditional programming languages such as Python and in APIs such as the PAN-OS XML API. Therefore, every time we run our policy deployment script, we should perceive the target as a black box. We cannot be sure that the changes we made before were the only changes made to the system. Therefore, the only programmatic way of ensuring the desired behavior of the firewall policy is ensuring that the policy on the target device does not have any objects and settings other than what we have set in there.

What policy elements are already configured on the PAN-OS device by the time we run our policy deployment script? Should we retain them without any changes, delete and recreate them, or establish a delta between what is already on the device and what is in the code? If we choose the latter, we can deal with that delta alone, thus improving the script performance.

And the performance considerations are no joke. On high-end hardware platforms, a policy can have up to 160,000 individual address objects and 80,000 address groups. When we run our script to add/remove/update just a handful of them, it makes no sense to touch thousands of other address objects in the running configuration that are already identical to what we have in the code. We’d better establish the delta and recreate only these objects, improving performance dramatically.

Even though it’s unlikely that a web gateway such as the one we’re building will ever need tens of thousands of address objects in its configuration, such optimization does make a noticeable difference, even if you have just a couple of thousand address objects.

Other types of objects may rarely change; they may be insignificant to the desired policy, or there may not be that many of them anyway. For instance, there can only be 30 custom EDLs regardless of the hardware platform, and it's not a big deal to simply delete all EDLs found in the configuration and recreate the required ones from scratch using the `multi-config` API request every time we run the script – it will only take a few seconds. The same applies to security profiles and groups, service objects and groups, custom application and vulnerability signatures, log forwarding profiles, tags, and so on.

Another critical aspect of the task at hand is the interdependencies. One object can be a component of another object (or even of a few objects of different types). For example, if you have a tag assigned to an address object, you cannot delete the tag without first deleting the address object or unassigning the tag from the address object. The same tag can also be used on, say, a custom application signature. On the other hand, when you want to create a tagged address object or a signature, you cannot reference a tag in their definitions without creating the tag first.

There are many ways to look at it. I have chosen the sequence outlined in the next section, but you can opt for a different algorithm that would still achieve a consistent result and meet your unique requirements.

Firewall policy creation algorithm

This section outlines the step-by-step process for deploying the firewall policy. By following this structured algorithm, you will learn how to programmatically create, modify, and optimize security policies while ensuring consistency with your organizational requirements.

To deepen your understanding of the implementation details and troubleshoot potential issues, it is recommended that you review the source code as you proceed. Pay close attention to the functions invoked in `main.py` and their corresponding modules, as this will help you grasp how each component contributes to the overall policy deployment process.

The algorithm covers key aspects of firewall policy management, including the following:

- **Configuration locking:** Ensuring exclusive access to the target device during policy updates
- **Object management:** Deleting existing objects and recreating them from code to maintain consistency and idempotency
- **Policy rule generation:** Building security and decryption rules based on predefined requirements

- **Performance optimization:** Leveraging bulk operations to minimize API calls and improve efficiency

By the end of this section, you'll understand how to automate firewall policy deployment using an algorithm that achieves idempotence through systematic removal of outdated objects. The algorithm also guarantees reliable deployments by correctly managing interdependencies while enhancing performance with batch operations. The solution maintains necessary flexibility when needed, preventing configuration drift and ensuring consistent results across multiple deployments.



While reading the source code, you can double-click a function name to select it, then press *Ctrl + B*, and PyCharm will navigate to the function definition.

The algorithm should begin by creating an instance of the target PAN-OS device in the code with appropriate credentials. In other words, speaking in PAN-OS SDK for Python terms, you need to create a `Firewall` or `Panorama` object. The credentials can be obtained interactively or from environment variables. The latter is a good choice if you want to set up a fully automated deployment using a CI/CD pipeline. For illustrative purposes, the project included in the book chooses the interactive option.

The `requirements/policy_targets.json` file defines all possible targets for policy deployment. A target is a combination of the PAN-OS device address, device type (Panorama or a firewall), firewall **Virtual System (VSYS)** or Panorama device group name, Panorama template, and, finally, the type of the target environment.

The latter is essentially a string, such as `prod` or `lab` (you can use virtually anything you like), that is used to substitute the `<target_environment>` keyword in predefined places, such as the folder name for custom response pages or EDL URLs. You'll need this feature when you test the policy before it is deployed in your production environment.



If you use PyCharm, you can press *Ctrl + Shift + F* (in Windows) or right-click on the repository's root folder and select the **Find in Files...** option to find where exactly a keyword of interest is referenced.

Now, I suggest you open the `requirements/policy_targets.json` file and specify your test device using the provided example. Save the file and run `main.py`.

All targets are presented as an interactive text menu, and the operator must select one of the preconfigured options. Following the operator's choice, the script outputs the definition of the selected target and a brief description of the forthcoming actions should the operator agree to proceed. This initial dialog is shown in the following screenshot:

```

=====
Running the script with:
  - pan-os-python: v1.12.1
  - pan-python:    v0.25.0
=====

Please select an option (press Enter for default *):
  1. LAB Panorama
  * 2. LAB Firewall

Enter your choice (1-2, Enter for default *): 2

You selected: LAB Firewall
  Target Environment:  lab
  PANOS Address:      192.168.50.103
  Deployment Type:    firewall
  Firewall VSYS:      [vsys1]

The script will initiate the deployment of policy version [1.0.3], dated [17 Feb 2025].
All existing policy rules and associated objects, such as address objects and groups, tags,
profiles, services, EDLS, custom signatures, data patterns etc., will be deleted and recreated
from the ground up as prescribed in the code in this repository.

The dynamic part of the security policy will be generated based on the business
requirements specified in the files as follows:

  - App categories: [requirements/categories_app.csv]
  - URL categories: [requirements/categories_url.csv]

The static part of the policy will be generated based on the rules located in
the folders as follows:

  - [ngfw/policies/security/PRE]
  - [ngfw/policies/decryption/PRE]
  - [ngfw/policies/decryption/POST]

Enter 'YES' to Continue or anything else to Abort script execution: █

```

Figure 8.18 – Deployment target selection

As you can see, the script displays, among other information, the version of the installed pan-os-python and pan-python packages. If the corresponding versions in your system are lower than 1.12.1 and 0.25.0, you must upgrade them. Otherwise, some features may not work as designed.

Then, we ask the operator for credentials that should have the necessary permissions to deploy the policy to the target device.

Once we have the credentials, it's time to initialize the required class instances for our PAN-OS device. This looks as follows for Panorama:

```
pn = Panorama(hostname      = panos_address,
               api_username = admin_username,
               api_password = admin_password)
```

And it's virtually the same for a firewall:

```
fw = Firewall(hostname      = panos_address,
               api_username = admin_username,
               api_password = admin_password,
               vsys="shared")
```

It is a good idea to continue by retrieving all available URL categories and App-ID categories from the target device to ensure that the requirements are still adequate. As you recall from our security policy design described in *Chapter 5*, we must take into account all application and URL categories known to the firewall to ensure consistent coverage of all possible network traffic. Palo Alto Networks constantly improves its products, including releasing new application signatures, application categories, and URL categories; hence, this step is essential as a governing mechanism.

The App-ID categories are obtained via the `refreshall_applications()` PAN-OS SDK for Python method of the `Predefined` class:

```
built_in_apps = Predefined(panos_device)
built_in_apps.refreshall_applications()
```

Unfortunately, this call takes a relatively long time to execute, but there is no good way around this. Once we have the apps, we simply loop through them all, extracting the subcategory attribute. Some apps will have this attribute set to `None`. These are internal apps that cannot be referenced in the policy, so we ignore them.

If your device has the **App-ID Cloud Engine (ACE)** feature licensed, you can use an alternative, much quicker method of retrieving all categories:

```
panos_device.op('show predefined xpath "/predefined/application-type"',
cmd_xml=True)
```

If you do not have the ACE license, the command is still available, but it will give you extra categories that are not really available for use in the policy. This method is commented out in the code. Use it at your discretion.

URL categories are retrieved in a similar fashion:

```
panos_device.op('show predefined xpath "/predefined/pan-url-categories"',
xml=True)
```

Then, we parse the categories specified in the requirements spreadsheets and cross-reference them with what we got from the live device. Missing categories are displayed with a warning message, as shown in the following screenshot:

```
Enter username [admin]:
Enter password:

Proceeding with policy provisioning on the specified firewall...
Retrieving all predefined applications to establish all available application sub-categories...done (43 categories found)
Retrieving all predefined URL-categories...done (85 categories found)
URL categories template file already exists - bypassing its creation...
App categories metadata file found - parsing data...42 categories found.
URL categories metadata file found - parsing data...84 categories found.

>>> WARNING: APP category "customer-service" is not mentioned in the list of requirements
>>> Applications from this category will be handled by the policy as non-sanctioned ones (blocked)

>>> WARNING: URL category "artificial-intelligence" is not mentioned in the list of requirements
>>> URLs from this category will be ALLOWED and NOT LOGGED - make sure this is a desired behaviour

Review the warning(s) above and enter 'OK' to Continue or anything else to Abort script execution:
```

Figure 8.19 – Sanity check for the available URL and APP categories

In this example, the customer-service application category and the artificial-intelligence URL category are missing from the requirements spreadsheets. They must be specified accordingly so that the policy behavior is consistent and predictable.

This concludes the scene setting, and the script can proceed with deploying the policy.

The actual policy creation is a complex process on its own, so it makes sense to place it in a separate module so that all our building blocks are manageable and have a distinguishable function. Let's call it `build_policy.py` and define an eponymous function inside. Let's place the module in the `/lib` folder.



Please open the source code of this function in PyCharm and follow the relevant code fragments as you continue reading this chapter.

The function will accept the following arguments: the `panos_device` object we instantiated in the main module, the name of the target device group or VSYS as a `policy_container`, the name of the template or VSYS as `policy_template`, two lists with `app_categories_requirements` and `url_categories_requirements` that we ingested from the requirements spreadsheets, `current_url_categories` that we got from the device, and `target_environment`. Let's see what we can do with all this.

We start the function by taking the configuration and commit locks on the target device. This is important in an enterprise environment where several administrators may want to make changes or commit configuration all at the same time. This operation consists of two steps:

1. We specify the target for the lock taking:

```
panos_device.op(
    cmd=f"<set><system><setting><target><template><name>{{policy_
template}}</name></template></target></setting></system></set>",
    cmd_xml=False)
```

It will be a VSYS rather than a template when we work with a firewall:

```
panos_device.op(cmd=f"<set><system><setting><target-vsys> {{policy_
template}}</target-vsys></setting></system></set>", cmd_xml=False)
```

2. We acquire the locks:

```
panos_device.op(cmd=f"<request><config-lock><add><comment>Policy
revision {{settings.policy_version}} ({{settings.policy_date}})
rollout</comment></add></config-lock></request>", cmd_xml=False)
panos_device.op(cmd=f"<request><commit-lock><add><comment>Policy
revision {{settings.policy_version}} ({{settings.policy_date}})
rollout</comment></add></commit-lock></request>", cmd_xml=False)
```

This will conclude the lock taking on a firewall. It will have two locks as a result: commit and config.

However, in the Panorama case, we will also have to take locks on the device group:

```
panos_device.op(cmd=f"<set><system><setting><target><device-group>{policy_
container}</device-group></target></setting></system> </set>", cmd_
xml=False)
```

We set a new target and repeat the lock-taking operation. This will result in four locks in total on Panorama: config and commit locks on the template and config and commit locks on the device group.

We continue our algorithm by further narrowing down the target for our policy. In the Panorama case, we need to specify the device group name, and for a firewall, we set the virtual system:

```
if isinstance(panos_device, Panorama):
    # For Panorama the target is a DeviceGroup,
    # and the target_template is a Template
    target = DeviceGroup(policy_container)
    target_template = Template(policy_template)
else:
    # For a firewall, both the target and the
    # target_template must be the same object - a VSYS
    target = Vsys(policy_container)
    target_template = target
    # Associate the DG/VSYS with the Panorama/Firewall object
    panos_device.add(target)
    panos_device.add(target_template)
```

We will use the approach shown here every time we need to account for differences between Panorama and a firewall. The standard `isinstance()` Python function is extremely useful for this. In this example, we initialize `DeviceGroup` and `Template` in the Panorama case or `Vsys` when we deal with a firewall. We attach these objects to the PAN-OS object and proceed with building the policy.

The next object we need to initialize for a firewall is `Rulebase`, and for Panorama, we'll need `RulebasePre` and the `RulebasePost`:

```
if isinstance(panos_device, Panorama):
    rulebase_pre = PreRulebase()
    rulebase_post = PostRulebase()
```

```

    target.add(rulebase_pre)
    target.add(rulebase_post)
else:
    rulebase = Rulebase()
    target.add(rulebase)

```

As I noted in the *Firewall policy creation considerations* section earlier in this chapter, we need to wipe out existing objects of pretty much all types we will create as part of our policy deployment. This will ensure the policy target, be it a Panorama device group or a firewall VSYS, always remains free of redundant objects and that the objects referenced in the policy rules are identical to what we have defined in the code.

We start by retrieving all rules of the types that we're going to (re)create:

```

current_security_rules = SecurityRule.refreshall(rulebase)
current_decryption_rules = DecryptionRule.refreshall(rulebase)

```

This code snippet shows the rule retrieval for a firewall. We'll need to do the same for both RulebasePre and RulebasePost and concatenate the results when dealing with Panorama.

Then, we delete them all using the multi-config method we discussed in this chapter's *Performance optimization* section:

```

action_id = 1
multi_config_xml = '<multi-config>'
for rule in current_security_rules + current_decryption_rules:
    multi_config_xml += f'<delete id="{action_id}" xpath="{rule.xpath()}"></delete>'
    action_id += 1
multi_config_xml += '</multi-config>'
panos_device.xapi.multi_config(multi_config_xml)

```

This is a simplified version of the code to illustrate this step. The real code that you'll find in the `delete_objects()` and `execute_multi_config_api_call()` functions of the `auxiliary_functions.py` module has error handling and diagnostic output.



Our project deals only with security and decryption policies. At the same time, PAN-OS itself and the PAN-OS SDK for Python support a wider range of policies: NAT, QoS, application override, and authentication, to name a few. We will not cover them in the book, but you can do this by following the principles and examples outlined in this chapter.

The policy rules are the main “obstacle” for removing all other objects (address objects and services, custom URL categories, application and threat signatures, EDLs, profiles, and tags) since these objects may be referenced in the policy rules directly or indirectly. Now that the policy rules are out of the way (after the `multi-config delete` operation shown previously), we can proceed with deleting all dependent objects.

We delete and (re)create them from code in the following order:

1. **Delete and (re)create tags:** The deletion operation is allowed to fail on a tag because the tag may be referenced in an object or policy that the script does not touch (NAT, for example). So, we either need to untag those objects first and then delete the tags, or we need to tolerate these potential errors during tag deletion.

Tag creation is equally essential at this stage. Many objects we will be dealing with in the following steps may reference tags. If we do not ensure the required tags already exist when we create those objects, the object creation will fail.

So, I prefer to get tags out of the way at the beginning of the process.

2. **Delete application groups:** As we proceed with deleting existing application groups, it's not enough to simply enumerate them and then delete them all at once in one `multi-config API` call. Suppose you have an application group that another application group is a member of, and the latter is placed earlier in the `multi-config` sub-operations. In that case, the operation will fail because of this cross-dependency.

We must establish all groups with other groups as their members and delete the container groups first. Once the container groups are gone, we can safely delete the rest.

Application groups may also contain custom applications. Thus, the groups must be deleted before the applications are deleted at step 7.

3. **Delete application filters:** Application filters can be application group members. Therefore, they should be deleted *after* the groups.
4. **Delete application tags:** Application tags are a somewhat confusing term; in fact, they are a list of applications tagged by regular tags. This list can be stored in each device group. We must delete these lists before we delete the custom application signatures.
5. **Delete profile groups:** We must delete all profile groups before individual profiles.
6. **Delete individual security profiles:** This includes vulnerability, antivirus, anti-spyware, WildFire analysis, URL filtering, file blocking, data filtering profiles, and data patterns.

Confusingly, the latter is also stored under the `/profiles` XPath, so I am listing data patterns here along with other “regular” profiles.

Let’s include decryption profiles in the batch and remove them in this step as well.

7. **Delete and (re)create log forwarding profiles:** Similarly to the tags’ case, the deletion of log forwarding profiles is allowed to fail because they may be referenced in policies that we do not change as part of our policy deployment.

For example, you may have site-specific device groups nested in the device group holding your global policy. It would only be logical for all site-specific policy rules to reference the same log forwarding profile used in the parent device group. Therefore, we should tolerate the potential failures of these delete operations.

8. Establish which **address objects** and **groups** exist in the device configuration but are missing in the code, and delete them. We also delete objects that exist in both locations but are different in any way.
9. Establish which **address objects** and **groups** exist in the code but are missing in the device configuration, and create them. We also recreate updated objects.



Steps 8 and 9 are examples of *performance optimization* for address objects, which we discussed in the eponymous section earlier in this chapter. We establish a delta between the device’s current configuration and the address objects’ definition in the code, and work with this delta.

10. If **necessary**, delete and re-import custom applications, vulnerability, and spyware signatures.

The necessity of deletion and import may not be evident with these object types. Depending on the complexity and the specifics of firewall management arrangements in your organization, you may have custom signatures defined at a parent device group (or even at the Shared level). If this is the case, the `import` operation of signatures with names identical to those at a higher-level device group will fail because of the name conflict. So, we will set a global flag in the `settings.py` module that will determine the script's logic in this regard.



None of the objects listed in steps 5, 6, and 10 have corresponding PAN-OS SDK for Python classes. Therefore, we cannot enumerate them using the `refreshall()` method and, subsequently, cannot reference them with the `xpath()` method and delete them as we did before. We have to come up with a custom function that is aware of the location of all these objects within the configuration XML tree. This is implemented in the `delete_non_sdk_objects()` function.

Custom signatures are also tricky because there is no “native” XML API method to import them. Here is the workaround algorithm:

- Calculate their Xpath by concatenating the target's `xpath()` and the objects' hard-coded relative XPath: `/application`, `/threats/vulnerability`, and `/threats/spyware`.
- Take each signature (in the XML format that you get by exporting a signature from the PAN-OS admin UI) and strip the outermost tags so that only the `<entry name=<signature name>>...</entry>` fragment remains.
- Use the `set` method to place the XML code of the signature right at the constructed XPath. Combine all signatures you're importing in one `multi-config` API call, where each suboperation looks as follows:

```
f'<set id="{action_id}" xpath="{xpath}"> {signature_xml_definition}</set>'
```

11. Assign application tags. You may want to tag both the standard and custom applications. If you intend to tag custom applications in your environment, make sure the previous step is enabled or the signatures are defined in a parent device group.

12. Create application filters. We create filters for managed application categories, as per the requirements set in the `requirements/categories_app.csv` file. As you remember, this file effectively defines application filter selection criteria for each application subcategory and individual applications to exclude from filters if required.
13. Create application groups. Each managed application category also requires an application group, which we create at this step. The corresponding application filter from the previous step becomes a member here.

We also create custom groups based on static JSON files rather than the core requirements.

14. Import custom response pages. This import operation is another tricky part of the code. There is an import method in the XML API, but it does have an attribute to specify the target device group or VSYS:

```
https://{panos_device.hostname}/api/?type=import&category={page_
type}
```

Similarly to the lock-taking operation, we must specify the import operation target first. This is done as shown here:

```
panos_device.
op(cmd=f"<set><system><setting><target><template>{target_template}</
template></target></setting></system></set>", cmd_xml=False)
```

This code snippet demonstrates the target setting for Panorama import. A firewall equivalent looks as follows:

```
panos_device.op(cmd=f"<set><system><setting><target-vsyz>{target_
vsyz}</target-vsyz></setting></system></set>", cmd_xml=False)
```

Once the target is set, we perform the actual import operation by posting each custom response page as form data to the following:

```
https://{panos_device.hostname}/api/?type=import &category={page_
type}
```

15. Delete and (re)create EDLs. As we create the EDLs, we leverage the `target_environment` constant that we defined in the policy target definition in the `menu.json` file:

```
def create_edls(edl_container, panos_device, target_env)
    edls = parse_metadata_from_csv('EDLs', settings.edls_filename)
    for edl in edls:
```

```

... # some code omitted for brevity
edl_source = edl['Source']
if "<target_environment>" in edl_source:
    edl_source = edl_source.replace("<target_environment>",
target_env)
    edl_object = Edl(name=edl['Name'], edl_type=edl['Type'],
                    repeat=edl['Repeat'], repeat_at=repeat_at,
                    source=edl_source, username=username,
                    password=password, certificate_profile=
                    certificate_profile,
                    description=edl['Description'])
    edl_container.add(edl_object)

```

In this function, we read definitions of all EDLs from a CSV file, and then we create an `Edl` class instance for each entry found in the file. As we do so, we look for the `<target_environment>` string in the **EDL source** field of the EDL definition. If we find this string, we replace it with the `target_env` string that comes from the target environment definition in the `requirements/policy_targets.json` file.

This means an EDL source such as `https://edls.example.local/<target_environment>/edl1.txt` becomes `https://edls.example.local/prod/edl1.txt` or `https://edls.example.local/lab/edl1.txt` depending on the policy target attributes defined in the `policy_targets.json` file. This approach allows us to have a single EDL hosting environment and a single firewall policy, but two distinct sets of the actual EDL files so that we can experiment and test policy behaviour in our lab environment without affecting the production one.

16. Delete and (re)create custom URL categories.
17. Delete and (re)create service groups and services.

The groups must be deleted before the services. Creation should occur the other way around – services first, then the service groups.

18. Create all security profiles: data patterns, data filtering profiles, vulnerability profiles, anti-spyware profiles, file blocking profiles, antivirus profiles, WildFire analysis profiles, and decryption profiles.

We have already deleted all of these objects in steps 5 and 6; therefore, we need to (re) create them now.

19. Create decryption profiles



None of the profiles listed in the previous two steps have corresponding methods in the PAN-OS SDK for Python, which is unfortunate because you cannot have a complete security policy without security profiles. We will need to develop custom functions to construct the required XML code for each profile type and then upload all profiles to the PAN-OS device using the `set` command we previously used for uploading custom signatures.

20. Create security profile groups.

21. Finally, since all dependent objects are in place by this step, we can create the policy rules.

The PRE section of the policy will be created using the method outlined in the *Representation of the policy rules* section of this chapter. The POST section of the policy will be generated based on the requirements spreadsheets.

22. We conclude the script by removing the `commit` and `config` locks and then exporting policy metadata for our service desk system. The metadata will include a complete list of all application and URL categories managed by the policy, along with a description of each category, the associated AD group, and the approver.

Congratulations if you followed the steps and cross-checked them with the code. The first run of the script might have failed, but that's okay; we'll make it work in the next chapter. The key point now is that you've learned the basics of using the PAN-OS SDK for Python and the PAN-OS XML API, and you've understood the main idea behind the policy creation algorithm.

Summary

We started this chapter with simple tasks, connecting to your PAN-OS device and performing basic operations using the SDK. We discussed API performance and looked at potential optimizations, including comparing bulk operations with single API calls.

The importance of separating code from data was emphasized next. The project structure and key definitions were introduced, with decisions made about how to represent policy rules and other configuration objects in code. Various aspects of policy creation were considered, and the chapter concluded by running the main module, which follows a straightforward algorithm for creating policies.

That's a lot to cover in one chapter. The script you ran at the end might have had issues, but this is a normal part of the development process. In the next chapter, we'll look at these challenges by examining the dependencies and prerequisites for successful policy deployment. We'll also explore debugging techniques and ways to fill gaps in SDK functionality using direct XML API calls when needed. These steps will help you improve your implementation to meet specific organizational requirements while making sure your firewall security policies are deployed as intended.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



9

Policy to Code: Advanced

In the previous chapter, you learned the basics of using the PAN-OS SDK for Python and established the foundation of our coding project. Many assumptions were made in that chapter while an effort was made to stay focused and deliver a comprehensive overview of what needs to be done and how to deploy a firewall security policy.

Now, it is the right moment to uncover some of those assumptions and ensure that your policy deployment script actually works and builds the policy as designed. We will begin with a minimally viable product by covering the dependencies and prerequisites. Then, we'll consider some of the customizations that may be necessary to meet the requirements of your organization.

This chapter will cover the following main topics:

- Dependencies and prerequisites
- Various debugging methods
- How to cover SDK gaps with the XML API
- Customization of response pages
- Use of artificial intelligence

Technical requirements

You must have full administrative access to a standalone lab firewall or a Panorama-managed one. Your development environment should be configured as described in *Chapter 7, Setting Up Your Software Development Environment*.

Dependencies and prerequisites

While focusing on the basics, the sequence of operations, and the big picture in the previous chapter, we did not pay much attention to the fine details. However, they do matter, of course. When you want all API calls and the subsequent commit operation to be completed successfully and the firewall policy to process network traffic without a hitch, you must ensure all prerequisites are satisfied. When you cannot satisfy a requirement, you must understand how you can troubleshoot and debug the source code of the program to resolve the issue and move forward.

To ensure you fully understand the reasoning behind all dependencies mentioned in this section of the chapter, let me summarize the functionality of the automation project that comes with the book.

Summary of the project functionality

The script creates a firewall policy that includes both a security and a decryption component for an internet gateway, which serves as a perimeter firewall controlling outbound internet access for an organization.

Here is a high-level diagram of the assumed network setup for which we are building a firewall policy:

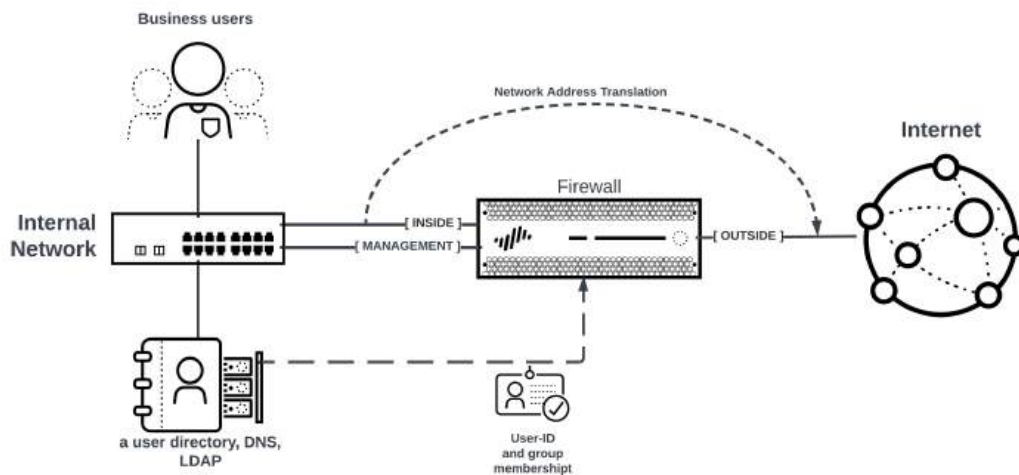


Figure 9.1 – Assumed network setup

This diagram illustrates a typical setup where a Palo Alto Networks firewall serves as a gateway, providing internet access for the company's business users. The internal network includes several directory servers (such as Microsoft Active Directory domain controllers) that also function as LDAP and DNS servers for the internal clients and for the firewall.

If you recall the fictitious example company, **ACME**, introduced at the end of *Chapter 1, Next-Gen Firewall Fundamentals*, this is precisely how each of their sites looks.

Now, let us go back to the code functionality. The `main.py` module, which serves as the project's entry point, depends on two key inputs:

- The target for the policy deployment
- The list of requirements prescribing which categories of applications and URLs to control users' access to

The former can be either a combination of a firewall and a VSYS (a virtual system on this firewall), or a combination of a Panorama appliance, a device group, and a template. All possible targets are defined in the `requirements/policy_targets.json` file. You can have more than one target listed in this file, but each execution of the script will push the policy only to one target at a time.

The latter is provided in the form of two CSV files in the `requirements/` folder: `categories_app.csv` and `categories_url.csv`. Each of these files prescribes the desired policy behaviour for *every* application and URL category known to the firewall.

Each category can be treated in three different ways:

- **Be managed** (dedicated rules will be created for this category; they will allow access only to the user group associated with the category)
- **Be non-managed** (there will be a generic rule allowing access to this category, among other non-managed categories, for all authenticated users)
- **Be denied** (blocked) outright

The policy will try to control access based on application (App-ID) categories and resort to URL filtering when it fails to identify a specific application.



If you have forgotten the principles of the policy operation, revisit *Chapter 3, PAN-OS Security Policy Features: Connection Matching*, where a detailed explanation of the policy principles is provided.

In addition to the security rules generated based on the preceding requirements and pushed to the POST section of the target device group, the deployment script also creates static policy rules prescribed in the respective subfolders of the `ngfw/policies` folder. The rules from these folders are pushed to the PRE section of the target device group. When the target of the script is a standalone firewall, then the effective policy will be identical to that of the PRE section, followed by the POST one in the Panorama case (as if the firewall were placed in the target device group on Panorama).

Prior to pushing the policy rules, the script first deploys all objects and profiles that the rules directly or indirectly depend on. These artifacts include tags, address objects and groups, service objects and groups, custom URL categories, external dynamic lists, security profiles and profile groups, decryption profiles, log forwarding profiles, data patterns, custom vulnerability signatures, custom application signatures, custom data patterns, and response pages. The latter is the only reason why Panorama-based policy targets must include a template in addition to a device group.

To keep the configuration consistent and to prevent a buildup of unused legacy objects, the script deletes existing objects of these types before recreating them from the code definitions. This approach helps achieve idempotence where it matters: the result of running the script multiple times is predictable and aligned with the desired state. Some object types may fail to delete, but that is acceptable and does not compromise the overall process.

It is also important to note that the script only operates on the specific target it is given. Objects residing in other device groups or virtual systems on the PAN-OS device remain untouched and unchanged.

As you may recall from the previous chapter, we walked through the sequence of how a policy is built from the ground up. Keeping that in mind, it should be easier to picture the many moving parts involved here. So, how do you actually make this algorithm work in your own environment?

Let's cover a few essential points in the context of all the preceding.

Zone names

Each policy rule references source and destination zones, among other connection matching criteria. Review the rules' definitions in the subfolders of the `ngfw/policy` folder in the project repository. You will find zones mentioned either in the section's defaults or directly in the rules. They are also referenced in the code that generates the POST section of the policy (`lib/security_policy_post.py`) and in the code defining log forwarding profile match lists (`lib/log_forwarding_profiles.py`).

Specifying all zone names as regular strings in various places would not have been a very elegant solution. It's only logical to define zone names as global constants in the `settings.py` module, making the deployment script easily adaptable to any environment. You will find all zones defined in this file as follows:

```
ZONE_INSIDE    = 'INSIDE'
ZONE_OUTSIDE   = 'OUTSIDE'
```

The former zone corresponds to the part of your network where your end users reside, while the latter represents the internet.

You must ensure that the values of these constants match the names of your respective zones (defined in the **Network | Zones** section of your firewall or Panorama template). Bear in mind that zone names are case-sensitive. Therefore, *Inside* and *INSIDE*, for example, would be considered different zones.

Default internal addresses

Best practices from Palo Alto Networks and other vendors recommend avoiding security policy rules that use “any” for source or *destination* addresses. For destination addresses in *generic* internet gateway rules, this is often unavoidable. In such cases, App-ID signatures and URL filtering usually provide the necessary control instead.

Source addresses, however, are much easier to define, since you already know the internal IP address space of your organization. Just as you would use constants for zone names, you can also use a constant to represent internal addresses.

In this book's project, the entire IP range defined by RFC 1918 for unicast private IP addresses is used. It is represented by the following constant:

```
DEFAULT_INSIDE_ADDRESS = 'AG-internal_network'
```

This constant is subsequently referenced as `settings.DEFAULT_INSIDE_ADDRESS` in the policy rules.

Whatever value you choose here (as long as it's not the keyword `any`) must also exist as either an address object or an address group in the file `ngfw/objects/addresses/address_objects.csv`.

Forward trust certificate

The deployment script will create a decryption policy provided the following flags are set:

```
CREATE_DECRYPTION_POLICY_FIREWALL = True
CREATE_DECRYPTION_POLICY_PANORAMA = True
```

They modify the script's behavior depending on the type of the target device.

This differentiation between Panorama and a firewall may come in handy if you already have a global decryption policy (deployed at the *Shared* level of your production Panorama, for example) and you do not want to change anything in this regard. At the same time, you may want to ensure that your standalone test or production firewalls always get their decryption policy via the deployment script.

The script will create four decryption profiles named according to these four constants defined in the `settings.py` module:

```
DP_DEFAULT          = "DP-default_decryption"  
DP_STRICT           = "DP-strict_decryption"  
DP_COMPATIBLE       = "DP-compatible_decryption"  
DP_NO_DECRYPTION    = "DP-no_decryption"
```

The decryption policy itself is static. It does not depend on the requirements set in the `requirements/` folder. Decryption rules will be created purely based on their definitions in the subfolders of the `ngfw/policies/decryption` folder.

What the script is *not* going to do is create a **Forward Trust Certificate** to be used by the TLS forward proxy subsystem of the firewall.

As explained in the *TLS decryption* section of *Chapter 5*, there are two options for how you can do this. You either issue this certificate with an issuing or root certificate authority deployed on the firewall itself, or you issue it with an issuing CA of your enterprise PKI infrastructure.

The latter is the option recommended by Palo Alto Networks. However, I will explain the former since the book focuses on Palo Alto Networks' technologies. Here is the step-by-step process:

1. First, create a root certification authority. Navigate to **DEVICE | Certificate Management | Certificates | Device Certificates** and click on **Generate**.
2. Specify a certificate name (for example, Firewall Root CA) and a common name. You may want to use a distinguishable name as your common name because it will be visible on the client computers as the root certificate in the certificate chain of every decrypted website. Something such as FIRM's Firewall Root CA would do the job here.
3. Check the **Certificate Authority** box. Leave the **Signed By** drop-box empty.

4. Choose and specify suitable cryptographic settings and optional certificate attributes. The following screenshot shows an example:

Generate Certificate

Certificate Type

☒ Local☐ SCEP

Certificate Name

Firewall Root CA

Common Name

FIRM Root CA

IP or FQDN to appear on the certificate

Signed By

☒ Certificate Authority☐ Block Private Key Export

OCSP Responder

Cryptographic Settings

Algorithm

RSA

Number of Bits

4096

Digest

sha512

Expiration (days)

3650

Certificate Attributes

☐	TYPE	VALUE
☐	Organization = "O" from "Subject" field	FIRM Co.
☐	Department = "OU" from "Subject" field	Information Security

+ Add

- Delete

Generate

Cancel

Figure 9.2 – Generation of a root CA certificate

As you can see in the screenshot, I have chosen the maximum supported RSA key length and hash algorithm.



Ensure you always consult industry best practices when you generate a root certificate for production use. Official publications of **National Institute of Standards and Technology (NIST)** of the U.S. Department of Commerce are always a solid indicator of current industry trends.

- 5. Click **Generate** to complete the certificate creation. Then, open the certificate properties and select the **Trusted Root CA** option as the following screenshot demonstrates:

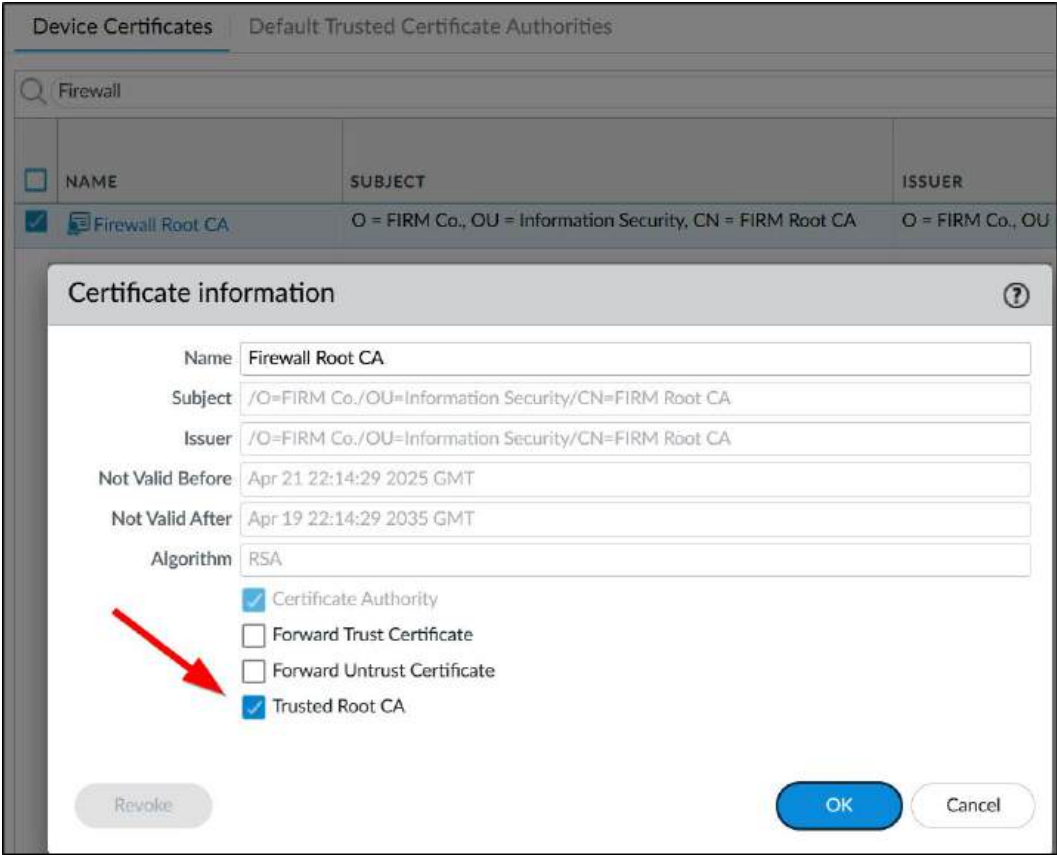


Figure 9.3 – Certificate role selection for the root CA

Once you've generated the self-signed certificate of your root certification authority, you can create the actual forward trust certificate for TLS decryption purposes:

1. Click on **Generate** again. Use the parameters shown in the following screenshot as guidelines:

Generate Certificate

Certificate Type: ☒ Local ☐ SCEP

Certificate Name: TLS Forward Proxy

Common Name: TLS Forward Proxy
IP or FQDN to appear on the certificate

Signed By: Firewall Root CA

☒ Certificate Authority
☐ Block Private Key Export

OCSP Responder:

Cryptographic Settings

Algorithm: RSA
Number of Bits: 2048
Digest: sha256
Expiration (days): 365

Certificate Attributes

TYPE	VALUE
Organization = "O" from "Subject" field	FIRM Co.
Department = "OU" from "Subject" field	Information Security

+ Add - Delete

Generate Cancel

Figure 9.4 – Certificate creation for TLS forward proxy

- 2. Ensure you have selected the root CA created at the previous step as the issuer of this certificate. Also, make sure you enable the **Certificate Authority** option. Click **Generate**.
- 3. Now, open this new certificate and declare it a forward trust certificate as demonstrated in the following screenshot:

Certificate information

Name

TLS Forward Proxy

Subject

/O=FIRM Co./OU=Information Security/CN=TLS Forward Proxy

Issuer

/O=FIRM Co./OU=Information Security/CN=FIRM Root CA

Not Valid Before

Apr 21 22:49:06 2025 GMT

Not Valid After

Apr 21 22:49:06 2026 GMT

Algorithm

RSA

☒ Certificate Authority

☒ Forward Trust Certificate

☐ Forward Untrust Certificate

☐ Trusted Root CA

Revoke

OK

Cancel

Figure 9.5 – Role selection for the TLS forward proxy certificate

- 4. Once you’ve clicked **OK**, you should have two certificates, as shown here:

Device Certificates

Default Trusted Certificate Authorities

FIRM

	NAME	CA	KEY	ALGORITHM	USAGE	STATUS
☐	Firewall Root CA	☒	☒	RSA	Trusted Root CA Certificate	valid
☐	TLS Forward Proxy	☒	☒	RSA	Forward Trust Certificate	valid

Figure 9.6 – Correctly provisioned root and forward trust certificates

This screenshot demonstrates that you now have a hierarchy of two certificates: a root one and a forward trust one.

Now your firewall knows that when an SSL or TLS connection matches a decryption policy of the **SSL Forward Proxy** type with its action set to **Decrypt**, it must decrypt and inspect this connection, having used the forward trust certificate to re-encrypt the client leg of the proxied connection.

This change will take effect as soon as you commit the firewall configuration.



Never do this step in your production environment without prior testing in a lab and a complete understanding of the potential consequences of your actions.

Before enabling the decryption as explained, you must distribute the public key of your **Firewall Root CA** certificate to all clients. Select the certificate and click on **Export Certificate**. Ensure that you *do not* select the **Export Private Key** option, and click **OK**.

Your browser will save the certificate as a file with the CRT extension that will contain the certificate in the Base64 format. This certificate must be installed in the **Trusted Root Certification Authorities** store on all your clients.

Mechanisms and technologies available to distribute enterprise root certificates are beyond the scope of this book. They vary depending on the operating systems and software in use. In infrastructures based on Microsoft products, you should be able to use Microsoft Intune to distribute the certificate to Windows, macOS, iOS/iPadOS, and Android devices. In the case of on-prem Microsoft Active Directory, you can use **Group Policy Objects (GPOs)** to push the root certificate to all endpoints.

Also, bear in mind that some software may not rely on the standard OS certificate stores and use their own certificate bundles.



You *do not* need to distribute the forward trust certificate to your clients.

It is possible that, in your environment, you run the deployment script behind a firewall that also performs TLS inspection. If this is the case, the connections the script makes (for example, to fetch URLs or IP addresses for custom objects) will fail.

This will happen because the HTTPS requests are made using the standard Mozilla certificate bundle located in `misc/mozilla-and-internal-ca-bundle.pem`. You must add the root certificate of your enterprise PKI to this file. There is a placeholder for this purpose right at the beginning of this file.

Network address translation

Any private network using IPv4 that needs internet access must include a device for **network address translation** (NAT). This function can be handled by the firewall, a router, or another upstream device.

However, it's likely that you already have a NAT setup on your firewalls, which you may prefer to keep as is. This is particularly relevant if you manage a multi-site infrastructure with NAT configurations that vary by location, making it challenging to implement a single global policy for several reasons.

The software project that comes with the book *does not* deploy a NAT policy, but it has a placeholder for this functionality should you decide to implement it in your environment.

Find the following code fragment in the `build_policy.py` module:

```
if settings.CREATE_NAT_POLICY:
    print("Staging NAT policy rules:")
    nat_pre, nat_post = create_nat_rules(panos_device)
    policy_rules_pre.extend(nat_pre)
    policy_rules_post.extend(nat_post)
```

Here, we call the `create_nat_rules()` function if the `CREATE_NAT_POLICY` global flag is set to `True`. The function is designed to return two lists of instances of the `NatRule` class (NAT rules, to put it simply). However, it is just a placeholder located in the same module and returning empty lists:

```
def create_nat_rules(panos_device):
    """
    Create NAT policy rules
    - placeholder for future implementation
    """
    # TODO: Implement NAT rule creation logic
    return [], []
```

Use the `create_decryption_rules()` function right next to this placeholder as an example. This function, in turn, calls the `decryption_policy()` function from the `eponymous` module.

To add support for NAT rules, follow these steps:

1. Clone that module and call it `nat_policy.py`.
2. Search and replace the `decryption` string with `nat`, making sure to respect case sensitivity when it appears in code (e.g., the `DecryptionRule` class becomes `NatRule`, and the function name `decryption_policy` becomes `nat_policy`, etc.).
3. Update the class parameters inside the `rules.append(NatRule(...))` section with those specific to the `NatRule` class. The full parameter list is available in the SDK documentation: <https://pan-os-python.readthedocs.io/en/latest/module-policies.html#panos.policies.NatRule>.



You might find this task slightly tedious, so I have completed it for you and placed the prepared module in the `lib/nat-example` folder.

4. Create folders for your NAT rules under `ngfw/policies`, following the pattern used in the project for decryption rules.
5. Define the NAT rules in `rules.py` files within those folders, again following the decryption rule pattern but using only the parameters valid for the `NatRule` class (refer to <https://pandevice.readthedocs.io/en/latest/module-policies.html#panos.policies.NatRule>).
6. Import the new function in the `build_policy.py` module as follows:

```
from lib.nat_policy import nat_policy
```

7. Complete the placeholder function mentioned.

That's it — once these steps are done, your project will support NAT rules. Some debugging may still be required, which we'll cover later in this chapter.

User-ID and Windows domain

It is expected that you already have the User-ID subsystem fully configured and functional in your environment. This technology comes in many different variants, and it would not be practical to cover all of them in this book, which is aimed at helping you automate the security policy.

If you do not have User-ID deployed and need a good starting point, I recommend the book *Mastering Palo Alto Networks* by Tom Piens, from Packt Publishing, as well as the *User-ID* section of the **PAN-OS Administrator's Guide** (<https://docs.paloaltonetworks.com/pan-os/11-1/pan-os-admin/user-id/user-id-overview>).

Typically, in an enterprise setting, you have an integration with on-premise Active Directory or Azure Active Directory (now Entra ID). Our example organization, introduced in *Chapter 2, Navigating Real-World Firewall Management and Cyber Risks*, has this very type of integration. All its firewalls can identify individual users and the groups these users belong to.

The values from the *User-ID* column in the spreadsheets containing business requirements (*categories_app.csv* and *categories_url.csv*) are treated literally as strings, which will be placed by the script in the **Source User** field of the respective security policy rules. Thus, the assumption is that these strings represent user groups recognized by the firewall (typically through LDAP integration with on-premise AD or via Cloud Identity Engine with Azure AD or other identity providers).

The AD groups would usually be prepended with a prefix in the form `<domain_name>\`. Your production environment may have AD integration, while the lab environment is completely isolated from the production network. However, you want to test the firewall policy in the lab before deploying it to production. For this reason, the script determines whether to add the prefix based on the environment type of the policy target.

This is done with two global flags and a constant that defines the domain name:

```
ADD_DOMAIN_PREFIX_FOR_LAB = False
ADD_DOMAIN_PREFIX_FOR_PROD = True
AD_DOMAIN_NAME = 'example_domain'
```

They are subsequently leveraged in the functions that create policy rules:

```
if (target_environment.lower() == "lab" and
    settings.ADD_DOMAIN_PREFIX_FOR_LAB):
    domain_prefix = settings.AD_DOMAIN_NAME + '\\'

elif (target_environment.lower() == "prod" and
      settings.ADD_DOMAIN_PREFIX_FOR_PROD):
    domain_prefix = settings.AD_DOMAIN_NAME + '\\'

else:
    domain_prefix = ''
```

The prefix itself is subsequently added to the `source_user` parameter value of each rule unless it's one of the predefined identity types:

```
if (rule['source_user'] not in ['any', 'known-user',
                                'unknown', 'pre-logon', None] and domain_prefix):
```

```
source_user = domain_prefix + rule['source_user']
else:
    source_user = rule['source_user']
```

You can easily modify this logic based on your requirements.

Now that we have covered all significant dependencies and prerequisites, you should try rerunning `main.py`. Fingers crossed, it will successfully complete the policy deployment to your target.

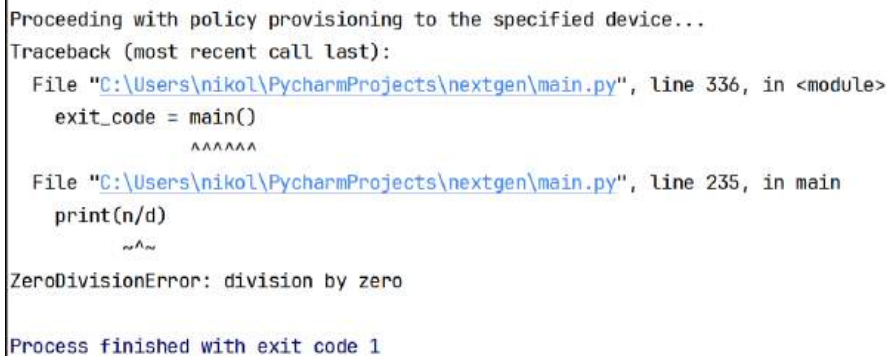
However, if it still doesn't work as expected, a debugging session may be necessary. I'll guide you through that in the next section.

Debugging

As you make changes to the code, you may naturally introduce software errors. While the IDE usually catches syntax mistakes very quickly (literally as you type), algorithmic, logical, and API errors aren't that easy to see.

Rich traceback

When a Python program fails, it produces a so-called traceback referencing the module where the error occurred. Here is an example of the default traceback output:



```
Proceeding with policy provisioning to the specified device...
Traceback (most recent call last):
  File "C:\Users\nikol\PycharmProjects\nextgen\main.py", line 336, in <module>
    exit_code = main()
    ^^^^^^
  File "C:\Users\nikol\PycharmProjects\nextgen\main.py", line 235, in main
    print(n/d)
    ~~~~
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Figure 9.7 – Regular Python traceback

I am sure you will agree it does not look pretty.

In this artificial example, it is quite obvious that the variable `d` is somehow assigned a `0`, leading to the classic division by zero error. However, it may not be as clear when encountering a more complex error. You may want to see the runtime values of all variables and gain more context for the error. This is where the *Rich* module proves to be very handy.

We enable the traceback globally as follows:

```
if settings.RICH_TRACEBACKS:
    from rich.traceback import install
    install(show_locals=settings.RICH_TRACEBACKS_SHOW_VARS)
```

The exception demonstrated in the previous screenshot turns into what is shown in the following screenshot:

```
C:\Users\nikol\PycharmProjects\nextgen\main.py:235 in main
232 |         sys.exit(1)
233 |
234 |         print('\nProceeding with policy provisioning to the specified device...')
235 |         print(n/d)  ← Provides a clear indication of the operator that caused the exception
236 |
237 |         # Check if templates exist for APP and URL categories, and if not - create t
238 |         # NOTE: This is not Panorama Templates !!!

locals
admin_password = '123456789'  ← May reveal sensitive data
admin_username = 'admin'
arguments = {
    'target_environment': 'lab',
    'panos_address': '192.168.50.102',
    'deployment_type': 'firewall',
    'firewall_vsys': 'vsys1',
    'panorama_device_group': '',
    'panorama_template': ''
}
choice = 'yes'
cookie = {'admin_username': 'admin', 'last_menu_choice': 'LAB Firewall 2 (PAN-OS'+9
d = 0  ← You can see the variable value that caused the error right away, without the need to run a full debug in IDE
default_admin_username = 'admin'
deployment_type = 'firewall'
f = <io.TextIOWrapper name='misc/cookie.json' mode='w' encoding='cp1252'>
firewall_vsys = 'vsys1'
kwargs = {}
n = 1
panorama_device_group = ''
panorama_template = ''
panos_address = '192.168.50.102'
panos_device = <Firewall '192.168.50.102' 'shared' at 0xic398aec40>
selected_option = 'LAB Firewall 2 (PAN-OS 11.2)'
start_time = 1745797221.8188723
target_environment = 'lab'

ZeroDivisionError: division by zero

Process finished with exit code 1
```

Figure 9.8 – Rich traceback

The screenshot demonstrates that the output becomes very structured with syntax highlighting and context. By default, the new handler shows three lines above and below the line where the exception occurred. You can also clearly see all local variables of the function.



On the flipside, the option to display local variables (the `RICH_TRACEBACKS_SHOW_VARS` flag in `settings.py`) may inadvertently expose sensitive information such as an admin password or an API key. Use it with caution.

IDE debugging

PyCharm, like other popular Python IDEs, provides robust debugging capabilities. If you have not yet used them and are unsure how, revisit the *Learning PyCharm features* section in *Chapter 7, Setting Up Your Software Development Environment*, which explains how to invoke the built-in learning project. One of the modules in this project explains the debugging workflow of the IDE:

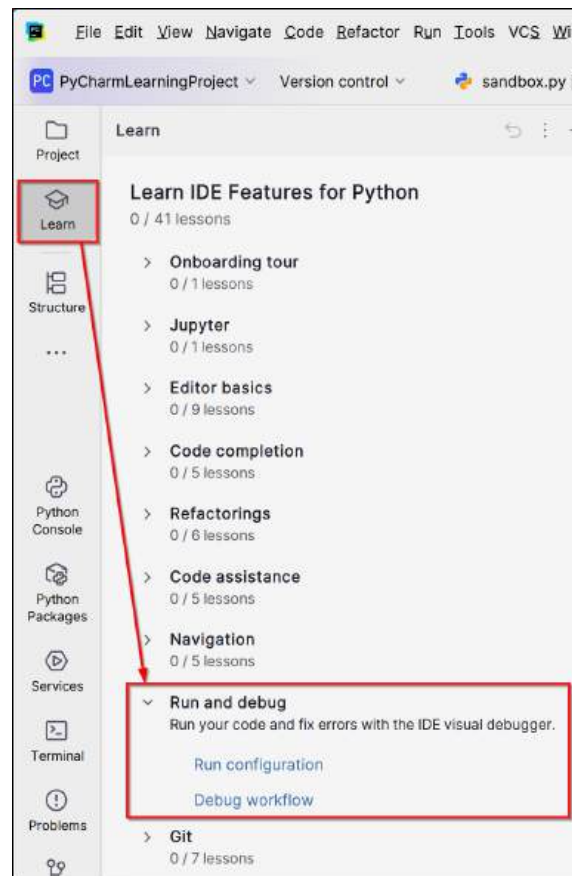


Figure 9.9 – How to learn PyCharm debugging capabilities

Once you have opened the learning project, navigate to the **Learn** tab and run the two lessons from the **Run and debug** section.

I do not have much to add to how the creators of PyCharm explain these features. I found these lessons to be very succinct, clear, and easy to follow. The only tip I believe is worth sharing here, as it may save you a lot of head-scratching, is how to adjust the settings of the output console.

When you run scripts within PyCharm, using its **Run** or **Debug** features, they are executed in a bespoke console. This allows all the robust debugging features you have just learned about to function effectively. As with all good things in life, there is always a downside. This console has limitations.

Depending on your operating system, you may find the script behaving oddly when you enter firewall credentials; it might not accept them at all, leaving you stuck at the password entry dialog. If this is the case, try enabling emulation of the terminal in the PyCharm output console:

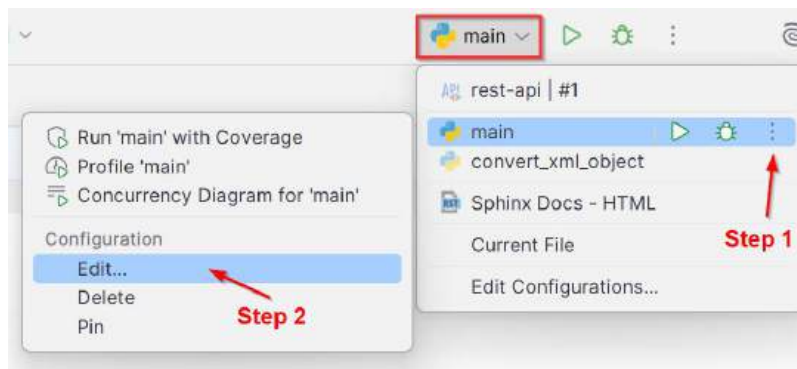


Figure 9.10 – How to edit Run/Debug configuration

Select `main.py` in the drop-down menu in the top-right corner of the PyCharm main window, as the screenshot demonstrates. Navigate to the **Edit...** menu as shown.

A new window will open as depicted in the following screenshot:

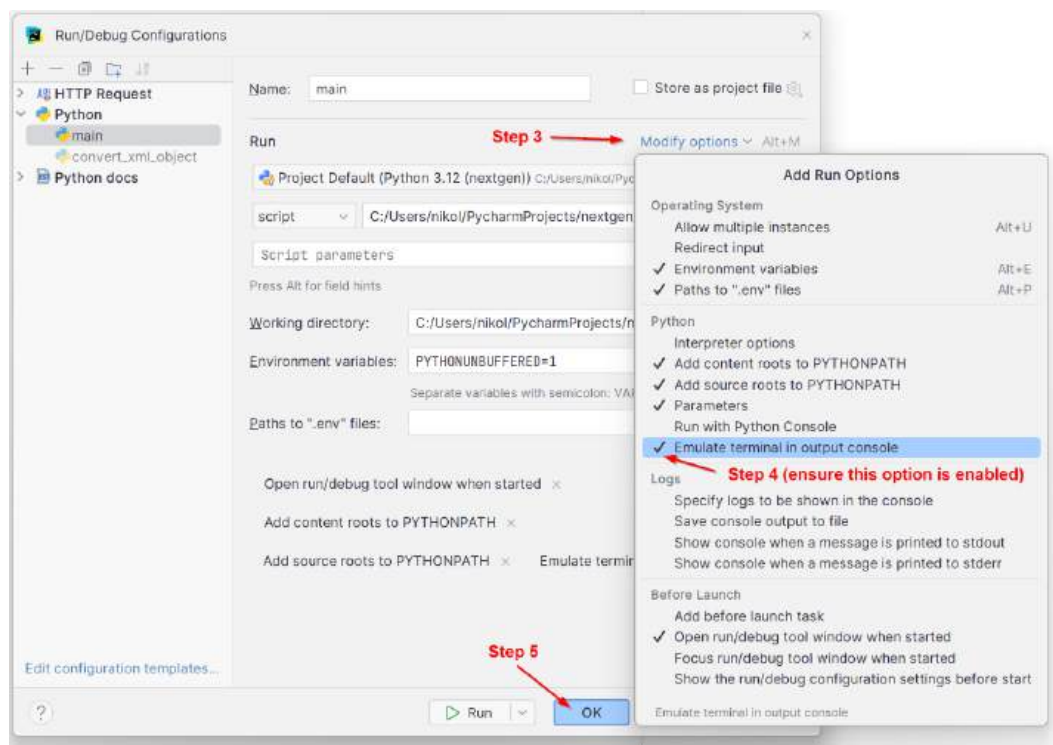


Figure 9.11 – How to enable terminal emulation

Click on the **Modify options** drop-down menu and make sure the **Enable terminal in output console** option is enabled (it is not enabled by default). Click **OK** to save the changes made to the settings.

Now, all your problems with entering passwords should be solved.

File logging versus console logging

Some error messages returned by PAN-OS may be very long. This can be the case with multi-config API calls comprising hundreds or even thousands of suboperations. When a firewall performs such a request, it returns the status of every single multi-config suboperation. Depending on the type of terminal the script is executed in, the output may appear truncated.

You can overcome this potential limitation by duplicating the output of an exception both to the terminal and to a file.

auxiliary_functions.py:

```
try:
    status = panos_device.xapi.multi_config(multi_config_xml)
except PanDeviceXapiError as e:

    console.print('-- XML API error --')
    console.print(e.message)

    with open(settings.API_ERROR_LOG_FILENAME, "w") as f:
        f.write(str(e))
```

In this code snippet, we attempt to execute a multi-config request and catch a potential exception returned by the XML API. We output the exception to the terminal and then to the file defined by the `API_ERROR_LOG_FILENAME` global constant.

View this file (by default it is `logs/api_errors.log`) if the exception is very long (many screens, typically) and appears truncated, with no apparent error in the messages comprising the text of the exception.

Strict transactional multi-config

Multi-config operations, discussed in *Chapter 6, Firewall Automation and Management Choices*, and extensively used throughout our automation project, are always transactional. This means that all suboperations that comprise a multi-config XML API request must succeed. If any of them fail, *all* suboperations in the request are rolled back.

This is great; however, what if there were other pending changes previously made under the same account? This may be the case if an administrator who runs an automation script had previously made another change. Such changes may inadvertently affect the changes made by the automation script.

Certainly, you could navigate to the firewall administrative UI and revert any pending changes every time, prior to running your automation script, but this is suboptimal, to say the least. You'd want to algorithmically ensure that the changes made by the script are the only ones staged for commit. This isn't easy to achieve without extra support from the XML API itself. Luckily, there is a feature for this.

The multi-config request has a parameter called `strict_transactional`.

It makes the multi-config operation even stricter by doing the following:

- Failing if a commit operation is in progress
- Rolling back any pending uncommitted changes made under the same administrative account that performs the multi-config request

The former is not that useful to us because we're already taking a config lock on the target right at the beginning of our sequence of configuration changes (see the algorithm overview in the *Firewall policy creation algorithm* section in *Chapter 6*).

The latter, on the contrary, is a sensible choice. It makes sense to do this only once, when we execute the very first multi-config call.

Let's expand the functionality of the `execute_multi_config_api_call()` function by introducing a module-level variable in the module containing this function. This variable will track the invocation of the strict transactional mode in the multi-config XML calls:

auxiliary_functions.py

```
_first_multi_config_call = True
```

Then, we should reference this variable as a global one in the function itself:

```
def execute_multi_config_api_call(panos_device, multi_config_xml,
    output_message, indentation_level, failure_mode="hard", strict_
    transactional=False):
    global _first_multi_config_call
```

Toggle it to False once executed for the first time:

```
if _first_multi_config_call and settings.MAKE_THE_FIRST_MULTI_CONFIG_
    TRANSACTIONAL:
    strict_transactional = True
    _first_multi_config_call = False
```

The same condition of the `if` statement sets the flag called `strict_transactional`, which is subsequently used in the API call:

```
status = panos_device.xapi.multi_config(multi_config_xml, strict=strict_
    transactional)
```

Thus, the entire process of making calls strict transactional on the first run looks as follows:

1. `_first_multi_config_call` is initialized to `True` when the `auxiliary_functions.py` module is first imported.
2. It remains `True` until the first call to `execute_multi_config_api_call()`.
3. After that first call, the flag is set to `False` and remains `False` for all subsequent calls during the program's execution.

If Python were to reload the module, the variable would be reinitialized to `True`. We don't do this, though — you'd have to explicitly invoke something such as `importlib.reload()` to make Python do so.

Logging of multi-config requests

In addition to logging the raw XAPI exception text that we discussed in the *File logging versus console logging* section, we can take it further by carefully logging the entire XML element of each multi-config call and the PAN-OS response to the call.

Let's initialize a file on the first run of a multi-config call:

`auxiliary_funtions.py`

```
file_mode = "w" if strict_transactional else "a"
with open(settings.API_CALLS_LOG_FILENAME, file_mode) as f:
```

Thus, during the first run, we will overwrite the log file or create it from scratch (the “w” mode), and on subsequent runs, we will open the file in append mode – adding to the existing file.

Then we write a summary of the metadata of the call to the open file:

```
f.write(f"== API Call at {timestamp}==\n")
f.write(f"Strict Transactional:      {strict_transactional}\n")
f.write(f"Associated output message: {output_message}\n")
f.write("XML Content (formatted for readability):\n\n")
```

Also, the XML code of the call itself:

```
dom = xml.dom.minidom.parseString(multi_config_xml)
pretty_xml = dom.toprettyxml(indent="  ")
if pretty_xml.startswith('<?xml'):
    pretty_xml = pretty_xml[pretty_xml.find('>') + 2:].lstrip()
pretty_xml = '\n'.join(line for line in pretty_xml.split('\n') if line.
strip())
f.write(pretty_xml)
```

This code snippet parses the `multi_config_xml` XML string and formats it with proper indentation. It removes the XML declaration line and empty lines and writes the result to the file.

We make our API call as follows:

```
status = panos_device.xapi.multi_config(multi_config_xml, strict=strict_
transactional)
```

Then we parse the response:

```
for response in status.findall('.//response'):
    resp_status = response.get('status')
    code = response.get('code')
    action_id = response.get('id')
    msg = response.find('msg').text if response.find('msg') is not None
    else None
```

This code uses XPath syntax to find all `<response>` elements in the XML response from the PAN-OS device. The `//` means it will search for response elements at any level in the XML tree.

For each response element found, the code extracts several attributes:

- The status attribute
- The response code
- The ID of the action
- The message text if a message element exists

Once we have deconstructed the entire response, we write the details to the log file:

```
f.write(f"\tID: {action_id}, Status: {resp_status}, Code: {code}, Msg:
{msg}\n")
```

All of the preceding results in a nicely formatted log file containing all multi-config requests and their respective PAN-OS responses that the program executes. I recommend opening the resulting log file in an editor capable of understanding XML, such as Notepad++ on Windows.



When using Notepad++, open the file as usual and then navigate to **Language** in the program menu and select **XML**.

The following screenshot illustrates the result you should get:

```

===== API Call at 2025-05-05 19:36:26 =====
Strict Transactional:      False
Associated output message: Deleting all staged objects...
XML Content (formatted for readability):

<multi-config>
  <delete id="1" xpath=
    "/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name=
    'vsys1']/application/entry[@name='APP-windows-conn-check']"/>
  <delete id="2" xpath=
    "/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name=
    'vsys1']/threats/vulnerability/entry[@name='41206']"/>
  <delete id="3" xpath=
    "/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name=
    'vsys1']/threats/spyware/entry[@name='15001']"/>
</multi-config>

--- API Response at 2025-05-05 19:36:27 ---
Result: success (1s 31ms, 3 ops, AVG: 0s 343ms per op)
PAN-OS detailed response:

    ID: 1, Status: success, Code: 20, Msg: command succeeded
    ID: 2, Status: success, Code: 20, Msg: command succeeded
    ID: 3, Status: success, Code: 20, Msg: command succeeded
=====

```

Figure 9.12 – Fragment of the log file with multi-config API calls and responses

The output presented this way makes troubleshooting your automation much easier. It enables you to identify the precise *XPath* and *Element* parts of all multi-config requests, which constitute the vast majority of API calls the program performs.

You can also use this log file (by default, it is `logs/api_calls.log`) to understand the algorithm of the policy deployment better. Run `main.py` after ensuring that the dependencies we discussed earlier in this chapter are satisfied. Let it finish, and then explore this log file, cross-referencing the API calls with messages on the terminal. This will be a great learning experience.

Low-level debugging

The debugging methods outlined earlier are excellent and help pinpoint most of the issues you may encounter. Despite this, there remains a slight chance that you will come across something awkward that requires a deeper look at the actual HTTP calls that your API calls translate to.

Unfortunately, there is no “native” way in the PAN-OS SDK for Python to see the HTTP requests its methods make. The only way to address this is to debug the HTTP calls the Python standard libraries make.

As a very minimum, you can go with the following code.

You begin by importing Python’s standard logging framework, the low-level HTTP protocol client, and the requests library:

main.py

```
import logging
import http.client as http_client
import requests
```

Once these modules are in place, the next three lines turn on progressively higher levels of HTTP visibility:

```
http_client.HTTPConnection.debuglevel = 1
logging.basicConfig(level=logging.DEBUG)
requests.packages.urllib3.add_stderr_logger(level=logging.DEBUG)
```

Setting `HTTPConnection.debuglevel=1` activates the raw, socket-level dumps in the standard library client, outputting every byte of headers and payload as it travels over the wire.

Calling `logging.basicConfig(level=logging.DEBUG)` initializes the root logger at the debug level, ensuring that no debug messages from any library are silently dropped.

Finally, the `add_stderr_logger(level=logging.DEBUG)` function hooks into `urllib3` (the engine under `requests`) and enables all its debug output: request lines, response statuses, connection pool events, retries, and so on.

The book’s project allows this level of debugging by setting the `DEBUG_OUTPUT` global flag to `True`.



Be aware that this level of debugging will output all your credentials, including the username, password, and the generated XML API key, in clear text. The API key will be displayed with each request. Use this capability with great caution.

Unfortunately, the output generated by this method is quite dense and hard to understand. It also fails to provide the complete API response, as it only displays the status and headers. Here is an example of how a typical request would be logged:

```
send:

b'POST /api/ HTTP/1.1\r\nAccept-Encoding: identity\r\nContent-Type:
application/x-www-form-urlencoded\r\nContent-Length: 310\r\nHost:
192.168.50.102:443\r\nUser-Agent: Python-urllib/3.12\r\nConnection: close\
\r\n\r\n'
send:
b'type=op&cmd=%3Cshow%3E%3Cpredefined%3E%3Cxpath%3E%2Fpredefined%2Fpan-
url-categories%3C%2Fxpath%3E%3C%2Fpredefined%3E
%3C%2Fshow%3E&vsys=shared&key=<your_API_key_in_clear_text>'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Mon, 05 May 2025 23:44:58 GMT
header: Content-Type: application/xml; charset=UTF-8
header: Content-Length: 6905
header: Connection: close
header: Cache-Control: no-store, no-cache, must-revalidate
header: Expires: Thu, 19 Nov 1981 08:52:00 GMT
header: Pragma: no-cache
header: Content-Security-Policy: frame-ancestors 'none'
header: Strict-Transport-Security: max-age=86400
header: X-Content-Type-Options: nosniff
header: X-Frame-Options: DENY
Retrieving all predefined URL-categories... complete (85 categories found)
```

You can work around these limitations by piping all requests through a purpose-built local HTTP proxy. Several decent third-party software tools allow you to debug HTTP requests made by your Python code. You can try HTTP Toolkit (<https://httptoolkit.com>), Fiddler (<https://www.telerik.com/fiddler>), or MITMProxy (<https://mitmproxy.org/>), among a few others.

ReplayDB

At this point, you're likely feeling frustrated by the prospect of diving deep into HTTP debugging. However, you're probably not going to need this. What is more likely is encountering something similar to this message:

```
status="error" code="10" id="1" <object_name> cannot be deleted because
of references from:
```

This will be a complete message – nothing will follow the colon. Not quite self-explanatory, is it?

Your first instinct will be to take the object name and use the global search feature in the PAN-OS UI to see where the object is referenced. You may even end up exporting the PAN-OS running config and searching in that very raw XML file – all to no avail. There will be no references that the error message implies.

The first thing you should try is to revert the changes. This can be done in the administrative UI, as shown in the following screenshot:

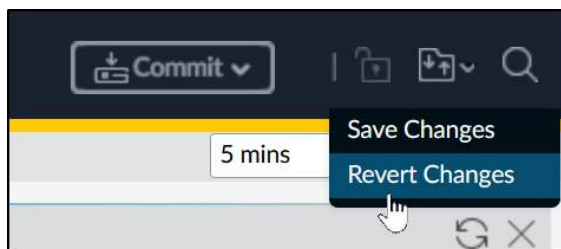


Figure 9.13 – How to revert uncommitted changes

This will take you to the next screen, where you will be able to select what changes to revert: specific admin changes or all changes.

Alternatively, you can use the CLI command:

```
> configure
# revert config partial admin <admin name>
```

In many cases, this will help, but unfortunately, not always.

What inadvertently makes this error possible is a management feature aimed at ensuring the persistence of uncommitted changes. The feature relies on an internal PAN-OS database called ReplayDB running on the MongoDB engine.

If your firewall or Panorama management server unexpectedly restarts, PAN-OS will store all uncommitted configuration changes locally in this database until a successful commit is completed. An unexpected restart could involve either the PAN-OS device or a PAN-OS management process linked to configuration management. This feature eliminates the need to recreate configuration changes after an unforeseen restart.

While this sounds great on paper, the implementation is far from optimal at the time of writing this book. In busy environments with numerous administrators, monitoring software, and automation scripts that continually bombard PAN-OS devices with API requests, and with changes remaining staged (thus staying uncommitted) for hours or days, the database may reach a size at which it cannot cope well with the load.

Run this command on your firewall to see the current size of the database:

```
> show system pancfg-directory-usage | match replay
14M      /opt/pancfg/mgmt/replaydb
```

You're likely observing errors similar to what I described when the size of the database reaches tens of megabytes. If this is the case and regular reverting of pending changes is not helping, clear the *committed* entries from the database by running the following command:

```
> request clean-replay entries committed
```

If you're still observing the error after this, try clearing *all* entries:

```
> request clean-replay entries all
```

Panorama requires a different command to see the database size:

```
> request mongo show databases instance mgmt
```

The output will contain sizes of various databases, with *panreplay* being one of them:

```
...
panreplay
31.25MB
...
```

The commands to trim the database on Panorama will be the same as those on the firewall. Once you have cleared the replay database, the error should disappear. You will also notice better responsiveness of the PAN-OS administrative UI and API performance. This feature, supported by the replay DB, is likely to be enhanced in future versions of PAN-OS. Therefore, you should not need to run the preceding commands very frequently, if at all.

This concludes the section on debugging. You are now equipped to quickly catch any errors and identify their root causes. What we have not yet covered is how to fill the gaps in SDK functionality. As you expand the functionality of your automation scripts, you will almost inevitably encounter configuration artifacts that the PAN-OS SDK for Python does not support. Let me provide you with one of the possible approaches to address this.

Covering SDK gaps with the XML API

It's not uncommon to discover gaps in the functionality provided by object-oriented SDKs, such as the PAN-OS SDK for Python. Sadly, some of these gaps concern vital parts of the firewall policy – you won't be able to complete your policy automation without finding a workaround. The XML API can cover all such gaps, but this requires some effort.

Let's consider one possible approach. Decryption profiles would be a good example. There is no class for this object type in the SDK, and it is also not described in the XML API documentation. Therefore, you'll need to resort to reverse engineering to determine how this object type can be created using the XML API.

For any configuration object type, you will need to know its full XPath and XML definition. You also need to decide on the data format to store the object in a file.

Let's start with establishing the XPath and learning the correct XML code. Begin by creating a decryption profile.



This type of profile is defined in **OBJECTS | Decryption | Decryption Profiles**.

Go ahead and add a new profile right in the PAN-OS administrative interface, clicking on all possible options on every tab of the profile definition.



The following example serves a specific purpose: to identify the XML tags that PAN-OS uses internally (that is, in its configuration file) to represent configuration options visible in the graphical administrative interface. Do not use the demonstrated profile in a production policy.

The following screenshot shows such a profile:

Figure 9.14 – Example decryption profile to generate its XML definition

The screenshot demonstrates that all options are selected. The main point, however, is to change the default state of each option. If one was enabled, then disable it. If one wasn't disabled, then enable it. If you double-click each option, this will also work. Ensure you perform the same steps on all other tabs as well: **SSL Inbound Inspection**, **SSL Protocol Settings**, **No Decryption**, and **SSH Proxy**.

Also, ensure that your profile has a unique name so that no other objects within the entire firewall configuration share the same name; this will make it easier to locate this object. Commit the configuration. Now, you have the full XML definition of your object (with all possible parameters), precisely at the correct XPath, in the running configuration of your PAN-OS device.

You can download the running configuration manually (navigate to **DEVICE | Setup | Operations | Export Named Configuration Snapshot** and select **running-config.xml**), open it in a suitable text editor such as **Notepad++**, and locate the object using the editor's Find command. However, it's not the ideal approach. Even with an XML plugin, it requires some effort to discover the XPath and get it in the exact notation used by the PAN-OS XML API.

I recommend using the **pan-os-php** automation **Swiss knife**, which we briefly discussed in *Chapter 4, PAN-OS Security Policy Features: Connection Processing*. Since you have already got **Docker Desktop** on your workstation (we installed it in *Chapter 7*), it's a breeze to grab the tool as a container:

```
> docker pull swaschkut/pan-os-php:latest
```

Then run it:

```
> docker run -it swaschkut/pan-os-php:latest
ubuntu@7b19ce34a8fa:/share$
```

Now you can invoke the tool with `xpath` specified as the `type` argument. This will allow you to grab the object's definition and XPath in one go:

```
ubuntu@7b19ce34a8fa:/share$ pan-os-php type=xpath in=api://<firewall_
address> display-xmlnode "filter-nameattribute=DP-example_profile_1"
```

The tool will prompt you to enter credentials to connect to the PAN-OS device, establish a connection, retrieve the running configuration, and locate your object.

You should get an output along these lines:

```
* XPATH:
/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@
name='vsys1']/profiles/decryption/entry[@name='DP-example_profile_1']
* VALUE:
<entry name="DP-example_profile_1">
<ssl-forward-proxy>
<block-expired-certificate>yes</block-expired-certificate>
<block-untrusted-issuer>yes</block-untrusted-issuer>
<block-unknown-cert>yes</block-unknown-cert>
<block-timeout-cert>yes</block-timeout-cert>
<restrict-cert-exts>yes</restrict-cert-exts>
<auto-include-altname>yes</auto-include-altname>
<block-unsupported-version>yes</block-unsupported-version>
```

```

<block-unsupported-cipher>yes</block-unsupported-cipher>
<block-client-cert>yes</block-client-cert>
<block-if-no-resource>yes</block-if-no-resource>
<block-tls13-downgrade-no-resource>yes</block-tls13-downgrade-no-
resource>
<strip-alpn>yes</strip-alpn>
</ssl-forward-proxy>
<ssl-inbound-proxy>
  <block-unsupported-version>yes</block-unsupported-version>
  <block-unsupported-cipher>yes</block-unsupported-cipher>
  <block-if-no-resource>yes</block-if-no-resource>
  <block-tls13-downgrade-no-resource>yes</block-tls13-downgrade-no-
resource>
</ssl-inbound-proxy>

  <-- part of the output omitted for brevity -->
</entry>

```

As you can see in the VALUE section of the output, it displays all the options you selected in the UI when creating the profile. If you had not toggled all the check boxes, the corresponding entries would not appear in the profile's XML code, and you would not be able to determine the correct names of the required XML tags. Many of those entries would simply be omitted from the code.

In theory, this XML code is virtually all you need to create the profile via the XML API. However, in practice, it would not be a very human-friendly representation of a configuration object to store in a file.

I suggest you convert this XML to JSON or YAML format as follows (JSON example):

```

import xmltodict
import json
object_as_xml = """
<entry name='DP-exaple_profile_1'>...</entry>
"""
object_as_dictionary = xmltodict.parse(object_as_xml)

```



You may need to install the `xmltodict` module using the `pip install xmltodict` command.

In this code snippet, you import the `xmltodict` and `json` modules and define the source XML definition of the profile as a string (you can copy the `VALUE` part of the `pan-os-php` output here). Then the `object_as_xml` string is converted to `object_as_dictionary` using the `xmltodict.parse()` method. Now that you have a dictionary, you can easily convert it to a JSON string using the `json.dumps()` method:

```
object_as_json = json.dumps(object_as_dictionary, indent=4)
```



I have simplified the process by providing a slightly more advanced version of the preceding code that converts XML to both JSON and YAML formats. You will find it in `misc/convert_xml_object.py`.

This returns a nicely formatted JSON representation of the profile definition that you can save to a file:

```
{
  "entry": {
    "@name": "DP-example_profile_1",
    "ssl-forward-proxy": {
      "block-expired-certificate": "yes",
      "block-untrusted-issuer":    "yes",
      "block-unknown-cert":       "yes",
      ...
      "block-ssh-errors":          "yes"
      "block-if-no-resource":      "yes"
    }
  }
}
```

Given the repository structure introduced in the previous chapter, the most logical place for this profile definition is the `ngfw/objects/decryption/decryption` profile folder.

Now that you have the profile definition in a file, it becomes a trifle to push it to the firewall. You should do this with a multi-config operation so that several profiles can be pushed in one go.

Implement this as a function following the pattern familiar from the previous chapter.

First, import the required modules and methods and declare the function:

```
import os.path
import settings
import json
from xmltodict import parse, unparse

def create_decryption_profiles(profile_container, panos_device):
```

The `profile_container` argument is either a `Vsys` or `DeviceGroup` object. The `panos_device` is either a `Firewall` or a `Panorama` object to which `profile_container` has been added before calling this function.

Start the function by initializing the `multi_config` XML string and a counter, and by printing a diagnostic message:

```
print(f"Staging decryption profile(s):")
action_id = 1
multi_config_xml = "<multi-config>"
```

Enumerate all files in the folder, looking for the ones with the `.json` extension and loading their content into the profile string:

```
for file_name in (os.listdir
                  (settings.decryption_profiles_folder)):
    if file_name.endswith('.json'):
        file_path = (os.path.join
                     (settings.decryption_profiles_folder,
                      file_name))
        if os.path.isfile(file_path):
            with open(file_name, mode='r') as json_file:
                profile = json.load(json_file)
```

If the profile definition has been read successfully, get its name and print it:

```
if profile is not None:
    print(f"\t{profile['entry']['@name']}")
```

Establish the base XPath:

```
obj_xpath = (profile_container.xpath() + "/profiles/decryption")
```

(It will be the same for every decryption profile.)

Convert the profile definition from the dictionary to XML using the `unparse` function and use the returned value as the `Element` part of the operation:

```
obj_element = (unparse(profile,
                        pretty=False,
                        full_document=False))
```

Add the `action_id` counter, `XPath`, and `Element` to the multi-config XML string and repeat the cycle for the next file:

```
multi_config_xml += \
    (f'<set id="{action_id}" '
     f'xpath="{obj_xpath}" '
     f'>{obj_element}</set>')
action_id += 1
```

Once you're out of the loop, you have parsed all JSON files in the folder. Now you can add the closing XML tag:

```
multi_config_xml += '</multi-config>'
```

Execute the API call for all profiles in one go:

```
panos_device.xapi.multi_config(multi_config_xml)
```

This code has been simplified, of course. It conveys the main idea but lacks exception handling. Refer to the `create_non_sdk_objects()` function in the project's `lib/auxiliary_functions.py` module for more comprehensive and universal code that can create all types of profiles.

Now that you have mastered the JSON representation, it's easy to switch to YAML format if you like it more. One of the advantages of YAML is that you can have inline comments, which come in handy with PAN-OS configuration objects whose elements in the XML representation do not always match the naming in the GUI. To switch to YAML, just replace `json` with `yaml` and load with `safe_load` in the code you've written. The only slight downside of YAML usage is that it does not come with Python as a standard module – you will need to install the `PyYaml` module:

```
pip install pyyaml
```

Add this module to the `requirements.txt` file.

XML API calls without XPath

You may have come across API commands that do not actually have an XPath, even though they are often meant to apply to a particular location on a PAN-OS device: a VSYS, a device group, or a template. This applies to importing, exporting, and a few other commands.

If you log in to your PAN-OS device and navigate to `https://<pan-os_device_address>/php/rest/browse.php/import` or `https://<pan-os_device_address>/php/rest/browse.php/export`, you will see the majority of these API calls.

Let's consider one example. This call imports a custom response page:

```
https://<panos_device_address>/api/?type=import&category=file-block-page
```

As you can see, there is no XPath here, so it's unclear how PAN-OS will determine the destination of this import operation. The key is to specify a target VSYS (or, in the Panorama case, a template).

This can be done interactively with a firewall CLI command:

```
> set system setting target-vsys <vsys_name>
```

Or, with a Panorama CLI command:

```
> set system setting target template <template_name>
```

The XML API equivalent for a firewall would be as follows:

```
https://<firewall_address>/api/?type=op&cmd=
<set><system><setting><target-vsys><target_vsys></target-vsys></setting></
system></set>
```

And for Panorama:

```
https://<panorama_address>/api/?type=op&cmd=
<set><system><setting><target><template><target_template></template></
target></setting></system></set>
```

In SDK terms, these calls translate to:

```
# firewall
fw.op(cmd=f"<set><system><setting><target-vsys>{target_vsys}</target-
vsys></setting></system></set>", cmd_xml=False)
```

```
# Panorama
pano.op(cmd=f"<set><system><setting><target><template>{target_template}</template></target></setting></system></set>", cmd_xml=False)
```

These API calls establish the target for import/export operations during the session, and remain in effect until the target is changed again or the API key expires.

The moment you initialize a Firewall or Panorama object and perform your first call, a new admin session is created. This session will remain active for as long as determined by the max session duration (**DEVICE | Setup | Management | Authentication Settings | Max Session Time (mins)**) and the inactivity timeout (**DEVICE | Setup | Management | Authentication Settings | Idle Timeout (min)**).

Therefore, for all import and export operations, you should always set a target first and then perform the necessary operation(s). In the book's project, this approach is utilized to import response pages.

Even though the same targetting approach works fine for configuration and commit locks, the Firewall and *Panorama* classes in the SDK also expose a set of dedicated helpers for locks:

- `add_config_lock()`
- `remove_config_lock()`
- `add_commit_lock()`
- `remove_commit_lock()`
- `remove_all_locks()`
- `config_locked`
- `commit_locked`
- `check_config_locks()`
- `check_commit_locks()`

These methods accept an extra parameter that defines the scope of the lock. On a firewall, this is the vsys. Unfortunately, neither the SDK documentation nor the PAN-OS API browser makes this parameter obvious, as it is only visible in the SDK's source code.

For example, the following SDK call on a firewall:

```
fw.add_commit_lock(comment="Commit message", vsys="vsys1")
```

translates into the following XML API request:

```
https://<firewall_address>/api/?type=op&cmd=<request><commit-lock><add><comment>Commit message</comment></add></commit-lock></request>&vsys=vsys1
```

As a practice, I recommend activating the low-level debugging discussed earlier in this chapter to observe how the SDK generates the underlying API requests.

Response pages and naming conventions

You should now have a good grasp of the practical aspects of using the SDK and XML API, as well as a high-level understanding of the book project's algorithm, its dependencies, prerequisites, and debugging.

Now, we must cover the firewall response pages.

Response pages and Jinja2 templates

As you may remember from *Chapter 3, PAN-OS Security Policy Features: Connection Matching*, which was dedicated to the design of the security policy, response pages play a pivotal role.

You need to customize them with your organization's branding, update text messages, and links.

It would have been a tedious job prone to errors if you had to modify the HTML code of the pages and the embedded JavaScript directly. The book's project adopts a more manageable approach to customizing the response pages by using **Jinja2** templates. This technology allows you to separate the skeleton of the pages from the text on those pages that you would typically want to change.

The way it works is that you have templates (files with the `.j2` extension) in the `ngfw/device/response_pages/<target_environment>/templates` folder that are, in essence, the original HTML code of the response pages with customizable data substituted with variables in the form `{{ variable }}`.

The values of these variables are defined in separate JSON files.

For each environment, there is a global `ngfw/device/response_pages/<target_environment>/shared.json` file defining shared variables applicable to each page:

- Organization name
- Department name
- Service desk base URL (the project uses ServiceNow as an example)

- CSS code defining the page style
- “Small print” added at the bottom of the page

Each page, in turn, has further customizations in `ngfw/device/response` `pages/<target_environment>/configs/<page name>.json`.

They include things such as the following:

- Page title
- Default heading and message
- Unique identifiers for the forms/landing pages related to the service desk workflow managing the page use case (i.e., blocked file download or a certificate error)
- Other parameters

Refer to the project documentation that comes with the book for a more detailed explanation (<https://github.com/PacktPublishing/Palo-Alto-Networks-from-Policy-to-Code>).



Do not underestimate the importance of the firewall response pages, especially in a large organization. The better you design them and the associated service desk workflows, the more time you will save in handling support cases in the long term.

Once you have ensured that the `main.py` script successfully deploys the policy and it actually works as designed, revisit this small section and then leave no stone unturned in deciphering how response pages are generated, and how the text and links to the service desk system are generated.

As you progress to policy testing in the next chapter, you will see how response pages are rendered and understand how they can be modified to better meet the requirements of *your* organization.

Enforcing the naming convention

One of the benefits of a generic programming language, such as Python, is the freedom of introducing virtually any custom logic to suit your unique requirements.

The enforcement of a naming convention is a good application of such flexibility. Let's see how this can be implemented.

Follow along the sequence of invoked functions as the script proceeds with policy deployment. You will find that the `find_and_import_rules()` function from the `lib.auxiliary_functions` module is responsible for the ingestion of static policy rules from the subfolders of the `ngfw/policies` folder.

Find the fragment where the rule is constructed from the rule definition and the section defaults:

auxiliary_functions.py

```
merged_rule.update(rule)
```

At this point, `merged_rule` is a dictionary with all rule components populated. So, we can grab any of them as a string and compare it with the corresponding regular expression to see if there is a match.

Import the `regex` module and start your validation function with a declaration:

auxiliary_functions.py

```
import re
def validate_string_for_compliance(
    string_to_validate, regex_pattern,
    validated_entity_name, message_to_display_if_no_match
):
```

Compile the passed pattern argument into a regex object:

```
pattern = re.compile(regex_pattern)
```

Check whether it matches the string passed for validation. If there is no match, then print a console message accordingly and return `False` (meaning the validation has failed):

```
if not pattern.fullmatch(string_to_validate):
    console.print(f"Validation Error: "
                  f"The {validated_entity_name} "
                  f"'{string_to_validate}' "
                  f"{message_to_display_if_no_match}.")
    return False
```

Otherwise, return `True` (meaning the validation has succeeded):

```
else:
    return True
```

Now, we can create respective global constants:

settings.py

```
PERFORM_VALIDATION_CHECKS    = True

VALIDATE_RULE_NAMES          = True
VALIDATE_RULE_DESCRIPTIONS    = True

VAL_PATTERN_FOR_RULE_NAMES    = r"^[a-z0-9][a-z0-9_-]{0,60}$"
VAL_PATTERN_FOR_RULE_DESCRIPTIONS = r"^.{12,1024}$"
```

Once we have the function and the constants, we can modify the original `find_and_import_rules()` function:

auxiliary_functions.py

```
merged_rule.update(rule) # find this line and insert
                        # the validation code below

if settings.PERFORM_VALIDATION_CHECKS and
   settings.VALIDATE_RULE_NAMES:
    if not validate_string_for_compliance(
        merged_rule['name'],
        settings.VAL_PATTERN_FOR_RULE_NAMES,
        "rule name",
        "is not compliant with the naming convention"):
        list_of_invalid_rules.append(merged_rule['name'])

if settings.PERFORM_VALIDATION_CHECKS and
   settings.VALIDATE_RULE_DESCRIPTIONS:
    if not validate_string_for_compliance(
        merged_rule['description'],
        settings.VAL_PATTERN_FOR_RULE_DESCRIPTIONS,
        "rule description",
        "is not compliant with the naming convention (must be from 12 to
        1024 characters long)":
        list_of_invalid_rules.append(merged_rule['name'])
```


In this code snippet, we verify that the validation checks are necessary. If this is the case, we perform the validation using our new function. Names of the rules that failed validation are appended to `list_of_invalid_rules`, which we also need to initialize earlier, in the very beginning of the function:

```
list_of_invalid_rules = []
```

The list of rules that failed our validation checks can then be output just before we exit the function. I will omit this here for brevity. Refer to the source code for further details.

Instead of validating the data that was input manually in the rule definitions, we can also automatically generate correct object names in the first place. This method does not suit all types of configuration objects, but the service objects, for example, are the perfect candidate to be handled this way.

When we create a service object, we specify the port number or a range of ports, and the protocol (tcp or udp). In the vast majority of cases, this is all we need to have in the service name: a prefix to denote the object type, the protocol name, and the port number or a port range. For example, we can have a service called SVC-tcp-8080 or SVC-udp-3478-3481. This is precisely how it's done in the project code:

service_objects.py

```
if service['Name'] == '' and ',' not in service['Destination Port']:
    service_name = 'SVC-' + service['Protocol'] + '-' +
    service['Destination Port']
```

In this code snippet, we start by checking whether the name was provided or whether there is a comma in the destination port field. This allows us to override autogeneration if we want a non-standard service name, which we can then provide in the Name field. We also use the comma as an indicator of something nonstandard because service objects can include long lists of port numbers. If this is the case, we expect the service name to be provided, so we skip the autogeneration:

```
else:
    service_name = service['Name']
```

On the contrary, when the Name field is empty and there is no comma in the value, it's a regular service object comprising a protocol and a port number, so we generate the service object name by concatenating the standard prefix, the protocol name, and the port value.

Use of artificial intelligence

When I began writing this book, I initially planned to include a tutorial on using **Large Language Models (LLMs)** for PAN-OS automation. At the time, that seemed like it would add real value. But after watching how quickly AI has evolved over the past year (I'm writing this in Summer 2025), I realized that any such tutorial would be outdated before the book even went to print. Instead, I think it's more useful to share some practical observations.

For one, even the most advanced LLMs struggle with PAN-OS. Sometimes they approximate how a feature works, but often they miss important details. The same holds true for APIs and SDKs: these models were never trained on detailed, authoritative data from Palo Alto Networks, so their answers are not something you can rely on without verification. Until Palo Alto releases an official model trained on accurate, PAN-OS-specific material, it's wise to treat any AI-generated output with caution.

That said, some tools are improving. ChatGPT's research mode, for example, produces noticeably better results than its standard output, though errors are still frequent. Claude Sonnet has proven to be a strong assistant for Python development when given clear, detailed prompts. And in the development environment space, JetBrains made a major leap forward with the release of Junie in Q2 2025. By enabling PyCharm to leverage Claude and other leading models—including local options—it turned AI assistance into something genuinely helpful for coding, rather than an occasional frustration.

And a final word of perspective: if you've been in IT or cybersecurity long enough, you'll remember how central the principle of Zero Trust once was. Today, it feels like that principle has been pushed aside, replaced by new acronyms that don't always reflect the same solid foundation.

Do not let this happen in your environment:



Figure 9.15 – LLM tossing Zero Trust out of the window (AI-generated; take a moment to admire the six-finger right hand of the man)



Never upload your production firewall or Panorama configuration to an LLM unless it is a locally hosted model under your complete control. Remember the old saying: there is no cloud; it is just someone else's computer. This applies to AI as well.

Summary

In this chapter, we took a significant step forward from the foundational setup and tackled the real-world considerations necessary to make your firewall policy deployment script truly functional. We began by breaking down the dependencies and prerequisites, including zone definitions, default IP allocations, certificates for TLS decryption, NAT settings, and essential integrations such as User-ID with Windows domains. This section helped create a baseline environment for building and deploying your policy.

From there, we explored debugging techniques – starting with rich tracebacks and IDE tools, then moving on to file versus console logging strategies, and concluding with a look at low-level debugging, transactional configuration behaviors, and issues potentially caused by ReplayDB. Recognizing that the PAN-OS SDK may not address every edge case, we also explored how to fill those gaps using the XML API. The chapter wrapped up by focusing on custom response pages and how to integrate Jinja2 templating while enforcing naming conventions. Finally, we touched on the role of artificial intelligence in assisting you in building a PAN-OS automation.

With a working deployment script in place, the next chapter shifts focus to quality assurance and testing. We'll explore how to validate configurations and build confidence in your automation before it ever reaches production.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



Part 3

Quality Assurance, Testing, and Go-Live

In this final part, you will focus on validating and operationalizing your firewall automation. *Chapter 10* introduces quality assurance methods and testing techniques, showing how to combine automated and manual tests with pilot deployments to ensure that security policies behave as intended. You will also see how to integrate response pages with service desk workflows to streamline feedback and incident handling.

Chapter 11 takes you through the process of going live. It covers migration methodology for zero business impact, preparing and deploying policy changes to test firewalls, analyzing logs to predict policy actions, and executing a controlled cutover. You will also explore strategies for scaling migrations across multiple firewalls and identifying opportunities for further improvements.

This section contains the following chapters:

- *Chapter 10, Quality Assurance and Testing*
- *Chapter 11, Your First Cutover and Next Steps*

10

Quality Assurance, Customization, and Testing

Building on the foundation established in the previous two chapters, where we focused on developing a functional deployment script, the next logical step is to ensure that the firewall policy we created performs as expected. With a working policy in place, we now turn our attention to quality assurance and testing.

We will explore how to test security features, automate incident response, and evaluate user experience, ultimately giving you the confidence to deploy the policy in a production environment. After testing all the policy features manually and with Python scripts, we will discuss exceptions that you will inevitably have to make in your environment, user acceptance testing, and touch on the controversial topic of performance aspects of firewall security controls.



As we move forward with testing various features, I'll be referencing the **non-functional requirements (NFRs)** outlined at the end of *Chapter 2, Navigating Real-World Firewall Management and Cyber Risks*. These references will be visually highlighted just like this paragraph.

This chapter will cover the following main topics:

- Setting a user identity for testing
- Automated tests
- Manual tests
- Automated incident response

- Exceptions
- Response pages and service desk integration
- User acceptance testing
- A note on performance

Technical requirements

You must have a firewall with a functioning policy deployed in the prior two chapters. The following diagram illustrates the network setup for testing:

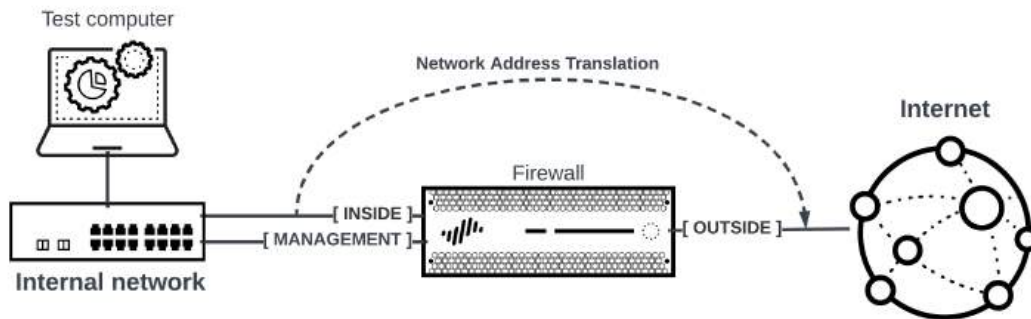


Figure 10.1 – Setup for testing

The tests we will conduct must be performed on a computer located ‘behind’ the firewall, so that all internet-bound traffic passes through the firewall.

All tests outlined in this chapter are compiled into a single script: `testing/policy-test.py`. You can either use it as is or refer to it as sample source code for creating your own tests.

Setting a user identity for testing

First things first. Since a significant portion of our policy is based on user identity, we will need a method to simulate a user triggering a policy action while being a member of the various groups referenced in the policy rules.

Changing a test user’s identity and group membership outside the firewall is highly impractical, especially in a distributed enterprise network. For example, if your firewall is connected to an on-premises **Active Directory (AD)**, a change you make to a user account in the AD may occur on a domain controller different from the one from which the firewall pulls group membership via **Lightweight Directory Access Protocol (LDAP)** integration. The change would need to be replicated from one domain controller to another, and then the firewall would need to pick up

this change during its next scheduled LDAP synchronization. Although there are methods to optimize this process and manually trigger each stage of the change propagation, it's a suboptimal approach, to say the least.

Fortunately, PAN-OS includes a set of API calls that can assist with our testing. Firstly, we can create an IP-to-user mapping. We can also set the group names in which the user is a member. The best part is that neither the user nor the groups need to exist in the local firewall user database or a connected directory. This allows us to specify any identities we want in the firewall policy and then alter the user identity and group membership 'on the fly' as we proceed with policy testing.

All User-ID XML API methods can be invoked as follows:

```
https://<firewall_address>/api/?type=user-id&cmd=<XML message>
```

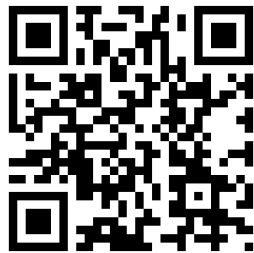
Where the XML message is a combination of login, logout, and groups commands, the commands can be related to different users and groups. Here is an example:

```
<uid-message>
  <version>1.0</version>
  <type>update</type>
  <payload>
    <login>
      <entry name="test_user" ip="<user's IP address>"
        timeout="20"/>
    </login>
    <groups>
      <entry name="group1">
        <members>
          <entry name="test_user"/>
        </members>
      </entry>
      <entry name="group2">
        <members>
          <entry name="test_user"/>
        </members>
      </entry>
    </groups>
  </payload>
</uid-message>
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.



🔑 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Here, the code instructs the firewall to map all traffic from the specified IP address to a user named `test_user`, and to include that user in both `group1` and `group2`. This mapping remains valid for 20 minutes.

Refer to the *Apply User-ID Mapping and Populate Dynamic Groups (API)* section in the *PAN-OS and Panorama API Usage Guide* to explore other relevant methods: https://docs.paloaltonetworks.com/content/dam/techdocs/en_US/pdf/pan-os/11-1/pan-os-panorama-api/pan-os-panorama-api.pdf.

The ability to combine multiple User-ID operations in a single XML message and a corresponding API call makes this approach very powerful, allowing administrators to optimize identity-related operations in busy environments.

The PAN-OS SDK for Python includes a special module for working with User-ID. First of all, it has a straightforward method that allows you to send any preconstructed User-ID XML message. Basically, this is the SDK version of the generic User-ID API call, which we covered earlier. Here's the base algorithm. Go ahead and try it out in the Python console.

1. Import the required modules:

```
from panos.firewall import Firewall
from panos.userid import UserId
from xml.etree import ElementTree as ET
```

2. Instantiate the respective class instances:

```
panos_device = Firewall("<firewall_address>",
                        "<username>", "<password>")
uid = UserId(panos_device)
panos_device.add(uid)
```

3. Construct the required XML message:

```
uid_message = """
    <paste the XML code from the previous page here>
    """
```

4. Send the message to the firewall after converting it to an ElementTree object:

```
uid.send(ET.fromstring(uid_message))
```

You can verify a successful mapping in the firewall CLI as follows:

```
admin@PA-440> show user ip-user-mapping ip <user's IP address>
IP address:    <user's IP address> (vsys1)
User:         test_user
From:         XMLAPI
Idle Timeout: 1165s
Max. TTL:     1165s
HIP Query:    Disabled
Group(s):     test_user(94)
              group1(2147483751)
              group2(2147483773)
```

This example demonstrates that the firewall correctly mapped the required IP address to the test user and included the user in the specified groups.

If you take a look at the firewall traffic log, you'll notice that `test_user` is now listed in the Source User field of all network connections coming from the specified IP address.

For situations where you don't need to combine multiple operations in a single API call, or you prefer not to deal with constructing XML messages, the PAN-OS SDK for Python offers several straightforward methods in its `userid` module.

First, initialize the firewall and the `UserId` class as follows:

```
from panos.firewall import Firewall
from panos.userid import UserId
panos_device = Firewall("<firewall_address>",
                        "<username>",
                        "<password>")
uid = UserId(panos_device)
panos_device.add(uid)
```

Then, you can invoke the following main methods on the `uid` object:

```
uid.get_group_members("<group name>")
uid.set_group("<group name>", "<list of users>")
uid.login("<user>", "<IP address>", timeout=<timeout>)
uid.logout("<user>", "<IP address>")
```

The names of these methods and their parameters are self-explanatory. The SDK also has functions that provide tagging functionality. Refer to <https://pan-os-python.readthedocs.io/en/latest/module-userid.html> for a complete list of available methods.

Automated tests

Now that we know how to generate a user identity with the required group membership, we can start testing the policy. Let's begin by testing the URL filtering functionality.

URL filtering

Testing the URL filtering is as simple as accessing URLs from all the known PAN-OS URL categories. We need to repeat this test, consequently assuming the identity of every user group referenced in the URL filtering requirements. We also need to fetch these URLs, having taken the identity of a user who is not a member of any of these groups. This will allow us to cover all possible scenarios.

Palo Alto Networks provides test web pages for every URL category. Their addresses have the following format:

```
urlfiltering.paloaltonetworks.com/test-<category_name>
```

Each page is available over plain HTTP as well as HTTPS. Here is what these test pages look like:



This is a test page that has been categorized as **web-based-email** by PAN-DB.

Figure 10.2 – Test page for the web-based-email URL category

The preceding screenshot illustrates that this is a “proper” website, which is unfortunate. I mention this because having a test URL that returns a lightweight structured response would have made it much easier to parse the page programmatically. However, we have to work with what we have. Therefore, we need to fetch the whole page and parse the HTML code that comprises it. We must first extract the page title:

```
<title>Palo Alto Networks URL filtering - Test A Site</title>
```

Then, we must extract the header naming the category:

```
<h1>PANDB TEST PAGE: web-based-email</h1>
```

As we process the server response, we must also evaluate the HTTP response code. When the page is retrieved successfully, the response code will be 200. This would mean access to the respective category is allowed for the impersonated user.

If the response code is 503, this means we have likely encountered a firewall block page. We can verify this further by parsing the HTML code of the firewall response page.

Lastly, the initial response code might be 302, which corresponds to the Continue URL-filtering action. This code redirects the browser to a special page hosted on the firewall that shows a warning message or disclaimer, processes the **Continue** button, and lets the user proceed to the *paused* website.

So, the algorithm to determine whether a URL is allowed, blocked, or paused can be simplified to these conditions:

- **Allowed:** HTTP status code 200 with the page title different from **Web Page Blocked**
- **Blocked:** HTTP status code 503 with the page title **Web Page Blocked**
- **Paused:** HTTP status code 200 with the page title **Web Page Blocked**

Depending on the number of user groups we need to verify (which is the same as the number of managed URL categories), the results table can become very large and overflow the console. It's a good idea to save the results as both a CSV file and in HTML format, so they can be easily viewed as a spreadsheet and in a web browser.

For an example implementation of this algorithm, see the `_single_url_test()`, `test_url_filtering()`, and `test_url_filtering_for_all_groups()` functions in the `testing/lib/url_testing.py` module.



Relevance of this test to requirements

This test validates how the policy meets the following part of NFR 3: *“The classification mechanism should resort to URL filtering when a web application does not have an applicable App-ID signature.”* We evaluate how effectively the policy fulfills the requirements for URL categories.

Now, let's move on to another essential component of the policy – DNS Security.

DNS Security

This section validates the PAN-OS DNS Security capability, ensuring that malicious domain queries are detected and blocked.

As you may recall from *Chapter 4, PAN-OS Security Policy Features: Connection Processing*, this feature is configured in the **DNS Policies** tab of anti-spyware profiles. Our testing should demonstrate the following:

- The **DNS-over-HTTPS (DoH)** and **DNS-over-TLS (DoT)** protocols are blocked
- Plain-text DNS is permitted
- FQDNs that correspond to the blocking DNS Security policy rules are blocked



If you've changed the *DNS Security* section of the policy supplied with the book, you might have different expectations for testing. Keep in mind that the PAN-OS threat prevention engine didn't support detecting threats in DoH until PAN-OS 11.0. So, you must not leave a gap in your defense – disable this protocol if you are still on a lower PAN-OS version.

On the other hand, if you're running PAN-OS 11 or higher and allowing DoH, you need to decrypt it. It's also a good idea to block **Encrypted Client Hello (ECH)**. There is an option in the anti-spyware profile that lets you do that.

For DoT, you'll need to decrypt TCP port 853 to inspect the embedded DNS protocol correctly.

In a production enterprise environment, name resolution is provided by dedicated servers or network appliances. Instead of reaching out to a public DNS server on the internet, clients usually resolve all names through these internal servers, creating a more efficient and reliable network experience.

However, in our test environment, we'll assume we only have the computer running the tests. The task at hand boils down to the following steps:

1. Parse a file with all test FQDNs.
2. Iterate through all FQDNs, executing DNS queries using the standard plain text DNS protocol, DoT, and DoH.
3. Output the results.

The dnspython package has all the necessary methods to work with the DNS protocol. The algorithm can be structured as follows:

1. Create a custom DNS resolver that ignores the system DNS configuration because we want to guarantee that all queries we send go through the firewall:

```
>>> import dns.resolver
>>> resolver = dns.resolver.Resolver(configure=False)
```

2. Specify a custom dns_server allowed by the firewall policy (for example, 208.67.222.222 for OpenDNS) and timeout values:

```
>>> resolver.nameservers = ["208.67.222.222"]
>>> resolver.timeout      = 5
>>> resolver.lifetime     = 5
```

3. Disable search domains to avoid automatic domain suffixes:

```
resolver.search = ()
```

4. Next, build and run a DNS query to find A records matching the FQDN:

```
>>> dns_response = resolver.resolve(
...     "www.example.com",
...     rdtype="A",
...     raise_on_no_answer=False)
```

The raise_on_no_answer=False parameter prevents exceptions when no A records exist.



Refer to *Table 4.2* in the *DNS Security algorithm* section of *Chapter 4, PAN-OS Security Policy Features: Connection Processing*, for a complete list of test FQDNs and the DNS Security policy rule names corresponding to these FQDNs.

5. Finally, check whether any A records were found. Extract them and return as a semicolon-separated string:

```
>>> if dns_response.rrset is None:
...     print("No A records found")
... ips = [r.address for r in dns_response]
... print("; ".join(ips))
```

6. This should generate an output similar to the following:

```
81.134.117.51; 81.134.117.49
```

With a bit of extra exception handling, this covers everything we need for a basic DNS resolver that uses the standard plain-text DNS protocol. Refer to the `_resolve_plain_text_dns()` function in the `testing/lig/dns_testing.py` module for the complete source code.

DoT name resolution follows virtually the same algorithm, except that it establishes a TLS connection over TCP port 853 to send the query. Refer to the `_resolve_dns_over_tls()` function in the same module.

DoH is more complex: it switches between a wire/binary mode that transmits the raw DNS packet (the `application/dns-message` MIME type, i.e., the exact bytes you'd send on the wire when using classic plain-text DNS) and a JSON mode that utilizes simple `name=/type=` URL parameters with an `application/dns-json` reply (a human-readable JSON object containing the status and answers). Refer to the `_resolve_dns_over_https()` function for the complete implementation.

The result you get should demonstrate that all malicious FQDNs are *sinkholed*. In other words, malicious FQDNs must be resolved to the IP address defined in the `settings.DNS_SINKHOLE_RESOLVED_ADDRESS` constant and referenced in the **DNS Policies | DNS Sinkhole Settings | Sinkhole IPv4** field of all anti-spyware profiles.

Let's say you try resolving one of the pseudo-malicious FQDNs, such as the following:

```
>>> dns_response = resolver.resolve(
...     "test-malware.testpanw.com",
...     rdtype="A",
...     raise_on_no_answer=False)
```

You should get the sinkhole address:

```
>>> print([r.address for r in dns_response][0])
198.135.184.22
```

This will also produce a corresponding record in the threat log:

THREAT ID/NAME	THREAT CATEGORY	APPLICATION...	RULE	ACTION	SEVERITY
generic:test-malware.testpanw.com	dns-malware	dns-base	name-resolution-secure	sinkhole	medium

Figure 10.3 – DNS sinkhole action in the firewall threat log

The log entry shown in the screenshot proves that the firewall took the sinkhole action to block traffic to the specified FQDN because it was classified as malware.



Relevance of this test to requirements

These DNS tests validate how the policy meets the following part of NFR 8: “*DNS Security should be used to satisfy FR 7.*” The latter, in turn, refers to a functional requirement that reads “*The policy must block known bad sources, destinations, and malware.*” We check how well the policy handles threats identified in DNS queries.

The next important aspect of testing is the App-ID-based section of the policy.

App-ID-based rules

As you very well know, our firewall policy heavily relies on App-ID technology. It manages access to application categories and specific applications.

The issue with testing them is that all application signatures are proprietary; Palo Alto Networks does not disclose the exact traffic patterns that make up one application or another. This means that we cannot generate traffic that would trigger a relevant signature for every single application that PAN-OS is aware of. Even if the vendor disclosed all these signatures, it would be a nightmare to programmatically generate more than 5,000 various traffic patterns to test every application.

Fortunately, the PAN-OS XML API and SDK for Python provide a method for this kind of testing.

It may already be familiar to you with the CLI command as follows:

```
admin@UK-LAB-440-1> test security-policy-match
+ application           Application name
+ category              URL Category name
+ check-hip-mask        check hip mask
+ destination           destination IP address
+ destination-category  Destination Category of Devices
+ destination-model     Destination Model of Devices
+ destination-os        Destination OS of Devices
+ destination-osfamily  Destination OS Family of Devices
+ destination-port      Destination port
+ destination-profile   Destination Profile of Devices
+ destination-vendor    Destination Vendor of Devices
```

```

+ from           from
+ protocol       IP protocol value
+ show-all      show all potential match rules until first allow
rule
+ source         source IP address
+ source-category Source Category of Devices
+ source-model   Source Model of Devices
+ source-os      Source OS of Devices
+ source-osfamily Source OS Family of Devices
+ source-port    Source port
+ source-profile Source Profile of Devices
+ source-user    Source User
+ source-vendor  Source Vendor of Devices
+ to            to
+ uappid         Universal Application Id
|              Pipe through a command
<Enter>       Finish input

```

As you can see, the command supports all possible matching criteria of security policy rules.

The corresponding XML API call for this command is as follows:

```

https://<firewall_address>/api/?type=op&cmd=<test><security-policy-match>
...matching criteria... </security-policy-match></test>

```

You can utilize it through the universal SDK `op()` method, which inherently supports all possible matching criteria of the CLI command. Here is an example:

```

panos_device.op("<test><security-policy-match>
    <source-user>user1</source-user>
    <application>facebook-base</application>
    <destination>1.1.1.1</destination>
    <source>10.0.0.1</source>
    <from>INSIDE</from>
    <to>OUTSIDE</to>
    <protocol>6</protocol>
    <destination-port>443</destination-port>
    <show-all>yes</show-all>
</security-policy-match>
</test>", cmd_xml=False)

```

The method returns an XML object that you would need to parse. The highlighted keywords represent the corresponding CLI command.

Alternatively, you can leverage the purpose-built `test_security_policy_match()` SDK method. It supports all criteria except those specific to the Device-ID, including device category, model, OS, OS family, profile, and vendor of the device. Here’s how the test from the preceding example would look with this method:

```
panos_device.test_security_policy_match(
    application = "facebook-base",
    destination = "1.1.1.1",
    source      = "10.0.0.1",
    from_zone   = "INSIDE",
    to_zone     = "OUTSIDE",
    protocol    = 6,
    port        = 443,
    show_all    = True
)
```

As you can see, this representation is more human-friendly. It also returns the result as a list of dictionaries, where each dictionary describes a matched rule – its name, index, and action:

```
[{'name': 'non-managed-apps-regular',
  'index': 81,
  'action': 'allow'}]
```

This is, of course, a more convenient data structure than XML.

Let’s see how this method works in practice. According to the example policy included with the book, email is considered a managed application category. Therefore, the policy has two security rules corresponding to this category:

	NAME	ACTION	Source			Destination		APPLICATION	SERVICE	URL CATEGORY
			ZONE	ADDRESS	USER	ZONE	ADD...			
48	managed-apps-email-risky	 Allow	 INSIDE	 AG-internal_network	 ug-email	 OUTSIDE	any	 APG-email	 service-http	high-risk
									 service-https	medium-risk
49	managed-apps-email-regular	 Allow	 INSIDE	 AG-internal_network	 ug-email	 OUTSIDE	any	 APG-email	 application...	any

Figure 10.4 – Security rules corresponding to the email application category

Log in to the firewall CLI and ensure there is no current IP-user mapping for a test address:

```
admin@PA-440> show user ip-user-mapping ip 10.0.0.1

No matched records
```

Open the Python console in PyCharm, import the required SDK modules, and initialize a firewall object and a User-ID object:

```
>>> from panos.firewall import Firewall
>>> from panos.userid import UserId
>>> panos_device = Firewall("<firewall_address>", "<username>",
"<password>")
>>> uid = UserId(panos_device)
>>> panos_device.add(uid)
<panos.userid.UserId object at 0x000001D875391750>
```

Now, map an arbitrary user to the chosen IP address:

```
>>> uid.login("user1", "10.0.0.1")
```

Verify in the CLI that the mapping is now active:

```
admin@PA-440> show user ip-user-mapping ip 10.0.0.1
IP address:    10.0.0.1 (vsys1)
User:         user1
From:         XMLAPI
Idle Timeout: 2689s
Max. TTL:     2689s
HIP Query:    Disabled
Group(s):     user1(1)
```

Test the policy match, without specifying the source user:

```
>>> panos_device.test_security_policy_match(
... application="gmail-base",
... category="web-based-email",
... destination="1.1.1.1",
... source="10.0.0.1",
... from_zone="INSIDE",
... to_zone="OUTSIDE",
... protocol=6,
```

```
... port=443,
... show_all=False)
[{'name': 'not-authorized-for-email', 'index': 86, 'action': 'deny'}]
```

The connection matches a denying rule associated with the email App-ID category. This outcome is expected since our test user has not joined any groups yet. Although we did not specify a user, the system would still check whether a user is mapped to the source IP address and utilize this for the policy lookup.

Let's fix that with a command as follows:

```
>>> uid.set_group("ug-email", ["user1"])
```

We can now verify the result:

```
admin@PA-440> show user ip-user-mapping ip 10.0.0.1
IP address:    10.0.0.1 (vsys1)
User:          user1
From:          XMLAPI
Idle Timeout:  2689s
Max. TTL:      2689s
HIP Query:     Disabled
Group(s):      user1(1)
                ug-email(2147483673)
```

As you can see, the group is now mapped to the same IP address. Repeat the same test as before:

```
>>> panos_device.test_security_policy_match(
... application="gmail-base",
... category="web-based-email",
... destination="1.1.1.1",
... source="10.0.0.1",
... from_zone="INSIDE",
... to_zone="OUTSIDE",
... protocol=6,
... port=443,
... show_all=False)
[{'name': 'managed-apps-email-regular', 'index': 48, 'action': 'allow'}]
```

Now, the network connection hits the expected rule, and it is allowed.



Relevance of this test to requirements

This test validates how the policy meets the NFR 2: “*User-ID must be used to control access to restricted resources to satisfy FR 2.*” The latter, in turn, reads as “*The policy should prevent unauthorized access to restricted resources. Resources may include in-house and third-party online applications, websites, and categories thereof.*”

Now, let’s see how the policy handles higher-risk scenarios by testing with a high-risk URL. What if it’s a high-risk URL included in an email application?

This scenario is somewhat artificial since risky URLs are usually embedded into email messages rather than being present on the email provider’s website itself, but this isn’t a concern because we’re testing a universal principle that applies across all application categories. We know this for sure because we understand how all our App-ID category-based policy rules are generated in the code. As long as the test works for one application category, it will work for all of them.

Proceed by repeating the previous command, but replace the URL category with “high-risk”:

```
>>> panos_device.test_security_policy_match(
...   application="gmail-base",
...   category="high-risk",
...   destination="1.1.1.1",
...   source="10.0.0.1",
...   from_zone="INSIDE",
...   to_zone="OUTSIDE",
...   protocol=6,
...   port=443,
...   show_all=False)
[{'name': 'managed-apps-email-risky', 'index': 47, 'action': 'allow'}]
```

Now, the connection has matched the rule above the previously matched one, with a stricter set of security profiles.



Relevance of this test to requirements

The test you have just conducted validates how the policy meets the NFR 9: “*Policy rules must determine the strictness of the applied security profiles and log forwarding profiles based on the risk level of the destination URL (for web-based applications).*”

This is expected policy behavior. For a refresher on how this works, see *Chapter 5, Security Policy Design*.

Let's see how this requirement is enforced. What if we try to use an application that is not part of the *ug-email* application group? Looking at the spreadsheet with application requirements in the `/requirements` folder of our software project, you'll notice that this category only includes applications with risk levels 1–4. Let's pick something riskier. The application called *horde* has a risk level of 5 at the time of writing this book. Let's use it for testing:

```
>>> panos_device.test_security_policy_match(  
... application="horde",  
... category="web-based-email",  
... destination="1.1.1.1",  
... source="10.0.0.1",  
... from_zone="INSIDE",  
... to_zone="OUTSIDE",  
... protocol=6,  
... port=443,  
... show_all=False)  
[{'name': 'non-sanctioned-email', 'index': 98, 'action': 'deny'}]
```

Exactly what was expected. The application didn't meet the custom selection criteria for the email category, so it naturally fell through the rule base to a generic rule that catches *all* email applications with the most generic filter, based solely on the category name.



Relevance of this test to requirements

This test validates how the policy meets the following part of the NFR 3: “*Generic web applications and sites should be classified based on their distinctive characteristics and risk levels. Therefore, they should be described using application filters and controlled at this level of granularity rather than individually.*”

What if we log our user out?

```
>>> uid.logout("user1", "10.0.0.1")
```

Now, there should be no user mapped:

```
admin@PA-440> show user ip-user-mapping ip 10.0.0.1

No matched records
```

However, if we repeat our horde test, we'll get this again:

```
[{'name': 'non-sanctioned-email', 'index': 98, 'action': 'deny'}]
```

It's the same rule. This is despite the fact that the rule has the known-user value specified as the source user. Clearly, there is no user mapped, yet the test still reports this rule as a matching one.

You will observe the same behavior in the CLI and in the firewall GUI (this testing functionality is available in **DEVICE | Troubleshooting | Test Configuration | Security Policy Match**). Unfortunately, this appears to be a limitation of the test command. It always seems to assume that there is a user. It's impossible to specify one of the preset special values: known-user, unknown, any, or pre-login.

You would need to initiate a real network connection that matches a required application signature to observe the policy behavior. Manually perform this task using a browser on your test computer. You will see that your network connection hits the last rule in the policy that applies to everything from anonymous endpoints.

	NAME	ACTION	Source			Destination			APP...	SERVI...	URL CATE...
			ZONE	ADDRESS	USER	ZONE	ADD...				
97	non-authenticated-connections	Deny	INSIDE	AG-internal_network	unknown	OUTSIDE	any	any	any	any	any

Figure 10.5 – The rule matching connections from anonymous endpoints

Refer to the `testing/lib/application_testing.py` module for the complete code that performs interactive application-based testing. Building on the preceding algorithm, the module adds an interactive menu that lets you set all test parameters. It also includes a feature to look up application details before proceeding with the test.

The following screenshot illustrates such a test:

Policy Test Menu

Current values for testing:

Firewall address: 193.308.50.105 (vsys1)

Firewall details: PA-440, sn 8120484324

Firewall version: 10.2.13-h5, 8988-9483

Source IP address: 272.100.31.12

Domain prefix: None

Decryption group: ug-decryption

Mapped user/group: user1/ug-email

Select:

1. Set source IP address
2. Set a domain prefix for all users and groups (optional)
3. Set decryption group
4. Set user and group mapping to the source IP
5. URL filtering test (for currently mapped user & group)
6. URL filtering test (for all groups)
- 7. App-ID test for a single application (for currently mapped user & group)**
8. App-ID test for all known applications (for currently mapped user & group)
9. DNS security test
10. Exit

Note: URL and DNS tests produce real traffic from the specified source IP, while App-ID is a synthetic test.

Choice (1-10): 7

Application name: gmail-base

APPLICATION DETAILS

App name: gmail-base

App description: Gmail is a free, advertising-supported email service provided by Google. Users may access Gmail as secure webmail, as well as via POP3 or IMAP4 protocols.

Default port number(s): tcp/80,443,465,587,993,995

App risk level: 4

Destination IP address ['23.215.0.138']:

IP protocol (tcp/udp) ['tcp']:

Port number (1-65535) ['80']: 443

Testing security policy match...

Test results:

ALLOWED

App name: gmail-base

Rule name: managed-apps-email-regular

Rule index: 48

App description: Gmail is a free, advertising-supported email service provided by Google. Users may access Gmail as secure webmail, as well as via POP3 or IMAP4 protocols.

Default port number(s): tcp/80,443,465,587,993,995

App risk level: 4

Figure 10.6 – Interactive App-ID testing

The menu shown in the preceding screenshot is produced after you run the `testing/policy-test.py` module. It will first ask you for a firewall address and credentials, and then suggest that you provide the source IP address, a test username and group, and a domain prefix (if you used one in the policy).

Once this is in place, selecting option 7 takes you through choosing an application name and all other parameters of the simulated connection:

- **Destination IP address:** The script suggests the address of **example.com**
- **IP protocol:** The script suggests the first IP protocol from the list of protocols specified in the application signature
- **Port number:** The script suggests the first protocol from the list of default application ports as specified in the signature definition, but you can override this

Then, the familiar `test_security_policy_match()` method is executed, and the test results are displayed.

Menu option 8 follows the same principles and uses application-default parameters to iterate through all applications known to the firewall. Results are saved to a CSV spreadsheet.

It's worth noting that iterating through all possible application signatures takes a significant amount of time. With a sub-millisecond latency from a test computer to a PA-440 firewall, evaluating the policy for all ~5,100 applications known to PAN-OS took me roughly half an hour.

Manual tests

Some tests are difficult to automate. However, since you only need to run them once during policy creation, it's not a significant trade-off.

File blocking

File blocking is one of the more challenging types of tests. The primary issue is the requirement for externally hosted storage. Since we're building a web gateway, the storage must be HTTP(S)-based. The simplest solution is to create an Amazon S3 bucket, ensuring that it is open only to your public IP address and your firewall. If you're not familiar with this process, any AI chatbot, such as ChatGPT, can provide you with a clear, step-by-step set of instructions on how to do this manually or with a script.

Create or find files that are supported by PAN-OS and are included in the file blocking profiles (`ngfw/objects/security profiles/file blocking`).

Create a simple script that iterates through all files and attempts to download or upload them from/to the bucket. Check the firewall file blocking logs (**MONITOR | Logs | Data Filtering**) to confirm the desired behavior. Decryption needs to be enabled for this testing method to work.

Creating a custom application signature for your S3 bucket will enable you to simulate a web application from any of the App-ID subcategories specified in the policy – just set the signature attributes to match the criteria set for a category in `requirements/categories_app.csv`. Use this approach to test the POST section of the policy, specifically the rules for managed and non-managed application categories. You can also explicitly include your custom application signature in key security policy rules to test the operation of the respective file blocking profiles.



Relevance of this test to requirements

This test validates how the policy meets the NFR 6: “*File-blocking profiles must be used to satisfy FR 5.*” The latter reads as “*The policy should prevent users from downloading arbitrary software from the internet.*”

The good news is that the policy has only four file blocking profiles. Therefore, it’s sufficient to use the global search in the PAN-OS administrative UI to find all security profile groups that they are included in, as well as the policy rules to which the profile groups are assigned. This will allow you to run only four tests and extrapolate the results to all respective rules.

Data filtering

Data filtering prevents unauthorized exposure of sensitive information by scanning traffic for predefined or custom patterns (e.g., **Personally identifiable information (PII)**, credit card numbers, or files with custom sensitivity labels).

Apply the method described in the *File blocking* section to test the data filtering profile as well. Out of the box, the policy uses only one data filtering profile. It is defined in the `ngfw/objects/security_profiles/data_filtering/DFP-default.yaml` file. The profile, in turn, references custom data patterns defined in the respective YAML files in the `ngfw/objects/custom_objects/data_patterns` folder.

The data patterns reference fictitious **Azure Information Protection (AIP)** sensitivity labels. Update them with the labels and respective GUIDs used in your organization. If you do not use this technology for data classification, feel free to exclude this profile from all security profile groups. This is defined in the `lib/security_profile_groups.py` module.

The data filtering profile does not block anything (**BLOCK THRESHOLD** is set to **0** for all data patterns); it only logs detected patterns using the severity levels that correlate with the sensitivity of each label. For example, files classified as **General** will generate log entries with an *informational* severity level, files classified as **Sensitive** will generate *low* severity log entries, and **Confidential** will produce *medium* severity entries.

You can certainly change this behavior in accordance with your company's policy. Typically, you'll need to create more profiles that, for example, allow uploading and downloading files of all sensitivity levels to your corporate OneDrive, but block uploads of sensitive and confidential files to any other websites.



Relevance of this test to requirements

This test validates how the policy meets the NFR 7: *“Data Filtering profiles and custom Data Patterns differentiated at the Application Filter/Group level should be leveraged to fulfill FR 6.”* The latter reads as *“The policy should be able to identify sensitive company data (files tagged with Microsoft’s Azure Information Protection sensitivity labels) and prevent potential exfiltration of this data via unauthorized file-sharing applications and webmail.”*

As always, decryption is a prerequisite for this security feature to work.

Automated incident response

Automated incident response helps security teams react faster to threats by automatically isolating compromised systems and triggering protective actions. Instead of waiting for manual intervention when malware or command-and-control traffic is detected, the firewall can immediately take steps such as blocking access or alerting administrators. This section walks through how to test these automated responses to ensure they work as expected during real security incidents.



Relevance of this test to requirements

This test validates how the policy meets the NFR 11: *“The policy must use the Built-In Actions mechanism in Log Forwarding Profiles to isolate hosts exhibiting strong indicators of compromise, such as matching threat signatures for command-and-control traffic. HTTP profiles must be used to raise security incidents.”*

Unfortunately, there is no built-in test signature that would allow you to simulate a command-and-control connection. You’ll have to create a custom spyware signature. Let’s go ahead and create one.

The simplest protocol to use here is HTTP (as you can use a browser or cURL to generate traffic). You can look for specific values in HTTP headers and declare a combination of them as a “command-and-control” connection.

The HTTP host header and the URI path are good choices for connection attributes because they can be easily generated by visiting a website in a browser or using a CLI tool such as cURL. Corresponding contexts in the signature creation dialog in PAN-OS UI are http-req-host-header and http-req-uri.

One of the many options available for selecting a website here is to use the vendor itself. You can create a threat signature that identifies connections to `https://www.paloaltonetworks.com/test-command-and-control`. This specific page doesn’t exist, but that doesn’t really matter since the firewall will be matching the client request rather than the server response.

The book project already has this signature created. You can find its definition in the `ngfw/objects/custom/objects/spyware/spyware_15001.xml` file. The signature should also be already uploaded to the firewall as part of the security policy you deployed in the previous two chapters. You will find it in **OBJECTS | Custom Objects | Spyware**, as shown in the following screenshot:

HIP Objects							
HIP Profiles							
External Dynamic Lists							
Custom Objects							
Data Patterns							
Spyware							
	☐	THREAT ID	NAME	SEVER...	DIRECTION	DEFAULT ACTION	COMMENT
	☐	15001	test command and control	high	client2server	reset-client	test spyware signature for command and control traffic detection

Figure 10.7 – Custom spyware signature for command-and-control traffic simulation

This particular signature relies on TLS decryption because it leverages HTTP attributes, both of which are encrypted. Therefore, if you do not enable decryption, this threat signature will not work. You will need to create another one, using a non-encrypted context, such as `ssl-req-chello-sni`, for example.

Once you have confirmed that the signature is in place and you have satisfied all prerequisites, such as decryption, go ahead and open a browser and navigate to the URL declared by the signature as command-and-control (<https://www.paloaltonetworks.com/test-command-and-control>).

Your connection will be blocked, and the browser will display an error message.

This triggers numerous events under the hood. Let's explore them in detail so that you can drill down to a failing step should anything go wrong in your case.

Look at *Figure 10.8*, depicting the events that have taken place after you pressed *Enter* in the browser (match the numbers from the following list to the numbers in the figure):

1. The traffic log shows that the browser established a TCP and a TLS connection to the website. Based on the SNI field of the TLS handshake, the firewall classified the category of the website and matched it to the `non-managed-url-category-regular` rule. The rule had its action set to `allow`. The connection was decrypted. The App-ID engine failed to match it to a specific signature and defaulted to `web-browsing`.
2. The browser sent its HTTP request. It was handled by the respective protocol parser that logged it in the URL log.
3. A copy of the request was also forwarded to the threat prevention engine, which successfully matched the request against our custom signature. Based on the settings in the anti-spyware profile assigned to the rule, the engine chose a blocking action.
4. The session was terminated, which was reflected in the **SESSION END REASON** of the traffic log entry.
5. Once the threat event occurred, it was captured by the log forwarding profile assigned to the rule. The profile had a rule matching a filter that included a pattern in the threat name (`command and control`), severity, action, and the destination zone.
6. All criteria of the filter matched the log entry, and the associated built-in action was applied. The action assigned the `Compromised Host` tag to the source IP address of the matched threat log entry. This is the address of our test computer.

7. The IP address got the tag assigned, which automatically included this address in the dynamic address group, DAG-compromised_hosts.

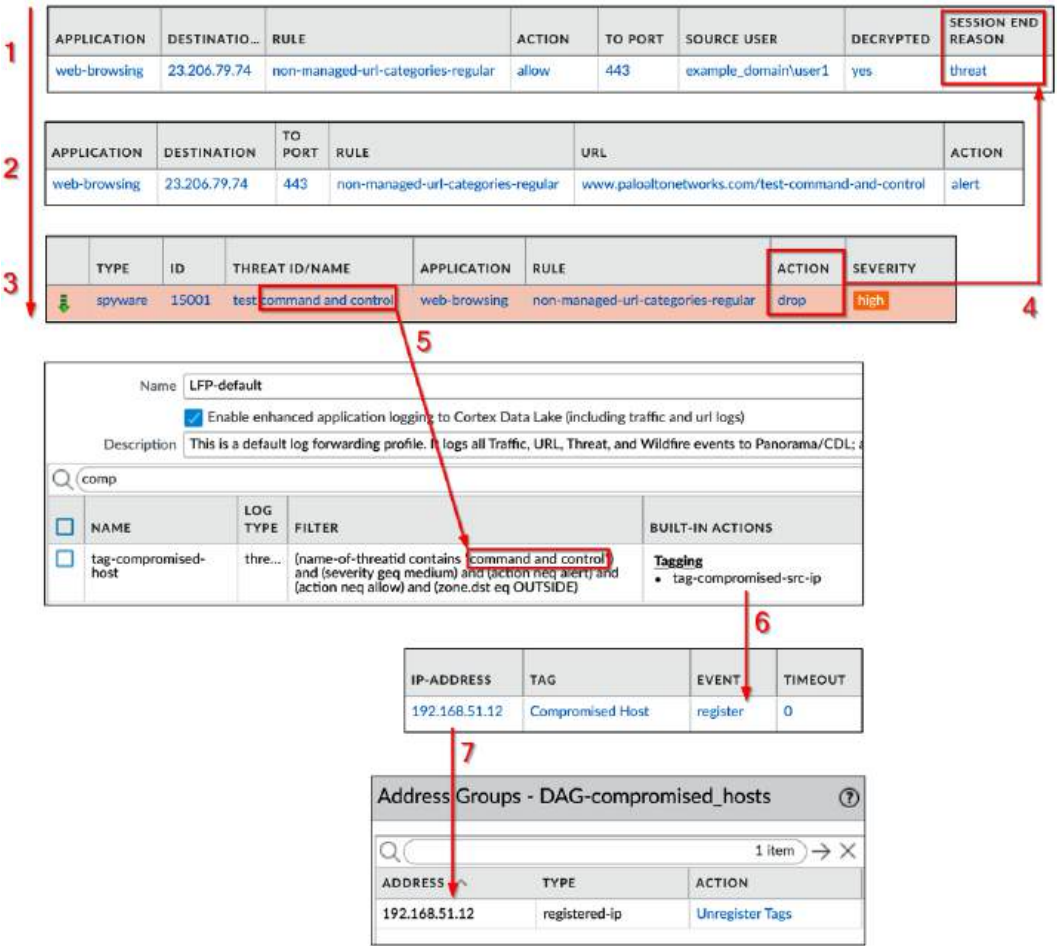


Figure 10.8 – Command-and-control simulation: sequence of events

Now that the compromised computer is tagged, try to navigate to a website. This new connection (as well as all subsequent connections) will match the rule for host isolation in the *Incident Response* section of the policy. The rule is shown in the following screenshot:

	NAME	ACTION	Source		Destination		APPLICATI...	SERVICE	URL CATEGORY
			ADDRESS	USER	ZONE	ADDR...			
22	incident-response-compromised-host-isolation	Deny	DAG-compromised_hosts	any	any	any	any	any	any

Figure 10.9 – Security rule isolating compromised hosts

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



As you can see, the rule matches all connections from the dynamic group to which our test IP address has just been added.

All HTTP(S)-based connections will trigger the generation of the application block response page. The browser will render the page injected by the firewall and execute the embedded JavaScript code.

This code had been generated based on the following definition in the `conditional_blocks` section of the YAML configuration file for the application response page (`ngfw/device/response pages/lab/configs/application-block-page.yaml`):

```
- match: rule
  operator: regex
  value: "(incident-response-compromised-host-
isolation|incident-response-block-ip-source)"
  heading: "Blocked – Device Isolated"
  message: |
    Our security system has identified a critical
    security issue and isolated your computer from the
    Internet to prevent any damage to company data. A
    ticket has been opened and confirmation e-mail
    from the Service Portal has been sent to your
    Inbox for tracking.
```

This is precisely what you should expect to see on the block page. It will look as shown in the following screenshot:

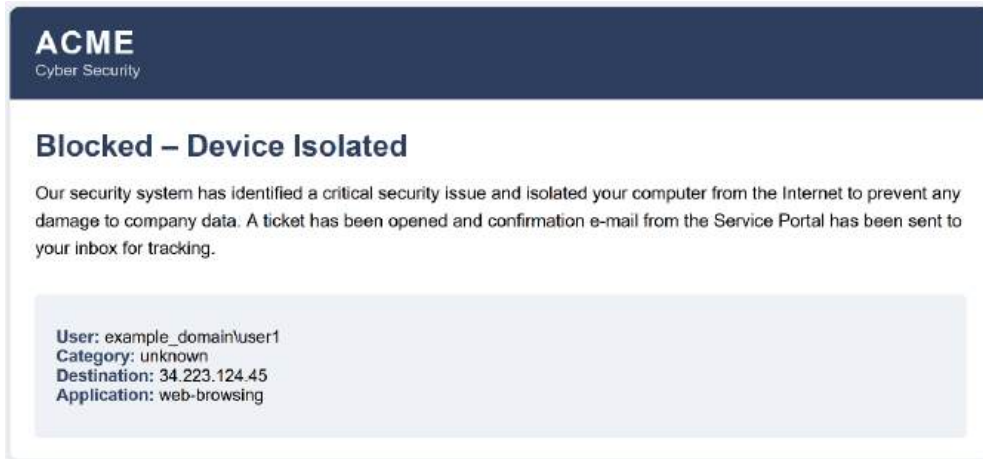


Figure 10.10 – Application block page handling the isolation of a compromised host



Testing with an HTTPS website does not always yield a response page, as its successful injection depends on many factors. I recommend using a plain-text website such as <http://neverssl.com> to ensure the appearance of the firewall response page.

The message states that a service desk ticket has been generated; however, this functionality has not been fully implemented in the script that comes with the book. Incident creation is a function of HTTP profiles. You can create one manually in **DEVICE | Server Profiles | HTTP** and enable its use by setting the following global variables in the `settings.py` module:

```
ENABLE_SIR_GENERATION_FOR_HOST_ISOLATION = True
SERVER_PROFILE_HTTP_SIR_IPS = <profile_name>
```

If you do this, the profile will be included in the log forwarding profile rule we covered in detail in the previous section.

In a large company with a well-established security function, it's likely you already have a SIEM-like system that generates security incidents after proper triage of all relevant indicators of compromise. If that's the case, leave the flag set to its default value of `False`. Update the verbiage of the response page according to your standard operating procedures for security incident response.

Geo-blocking

Geo-blocking restricts access from specific countries to ensure compliance and protect against threats from high-risk areas. This section evaluates how effectively our firewall enforces these geographic restrictions.



Relevance of this test to requirements

This test validates how the policy meets the NFR 9: *“Policy rules with geo-blocking based on a built-in geolocation database must be used to satisfy FR 8.”* The latter reads as *“The policy must block connections to and from geographical territories and countries in accordance with current regulations.”*

Testing geo-blocking is straightforward. You can use one of the free, public Geo-IP databases. Obtain some addresses from the region you want to block, ping those IP addresses from your test computer, and review the firewall logs. Make sure the geo-location of those addresses (the **Destination Country** field in the traffic logs) is correctly identified, and that the packets are blocked by the correct rule.

You can use one of these websites:

- <https://www.ip2location.com/free/visitor-blocker>
- <https://www.ipdeny.com/ipblocks/>

The first resource appeared to contain more complete and up-to-date information when I evaluated them both. I suppose this may change in the future. The following screenshot demonstrates how to get the IP data for your testing purposes. Note that the blocked country shown in this example is used purely for demonstration purposes:

The screenshot shows the IP2LOCATION website interface. On the left, there is a sidebar with a menu under 'Applications' containing: 'Firewall List by Country' (highlighted in blue), 'Firewall List by ASN', 'Firewall List by Search Engine', 'Redirect Visitor by Country', and 'Traceroute Application'. The main content area has the title 'Block Visitors by Country Using Firewall'. Below the title, it asks 'Do you want to block visitors by country?' and provides instructions: 'Select the countries you want to block, IP address version (IPv4 or IPv6), output format and press the "Download" button.' It also lists supported output formats: 'The output formats supported are Apache .htaccess, Linux iptables, CIDR, Netmask, Inverse Netmask, IIS web.config, Cisco ACL, PeerGuardian2, network-object, Cisco bit bucket, Juniper Junos and MikroTik. Please find the details below:'. Below this, there is a 'Download List' section with three dropdown menus: 'Country' (set to 'Afghanistan'), 'IP Version' (set to 'IPv4'), and 'Output Format' (set to 'CIDR'). A red 'DOWNLOAD' button is positioned below these dropdowns.

Figure 10.11 – Obtaining geo-IP data for a country

As you can see, all you need to do is navigate to the right section of the site, select the desired countries and output format, and download the file.

Exceptions

Two types of exceptions can be made with the firewall policy that comes with the book: manual and automatic. The manual exceptions can be made by updating the **External Dynamic List (EDL)** files referenced in the break-glass.

Ensure you're satisfied with the frequency of EDL updates (defined in `ngfw/objects/external_dynamic_lists/edls.csv`). You may want to modify the suggested schedules. Instead of setting an EDL to be checked for updates every five minutes (which may put unnecessary strain on the management plane of low-end platforms such as the PA-800 series), you may find it more convenient and practical to manually force an EDL refresh when required via respective CLI commands or API calls.

The policy uses the built-in action mechanism in the log forwarding profile not only to isolate compromised hosts but also to make auto-exceptions for failed TLS decryption attempts.

The respective rule is defined as shown in the following screenshot:

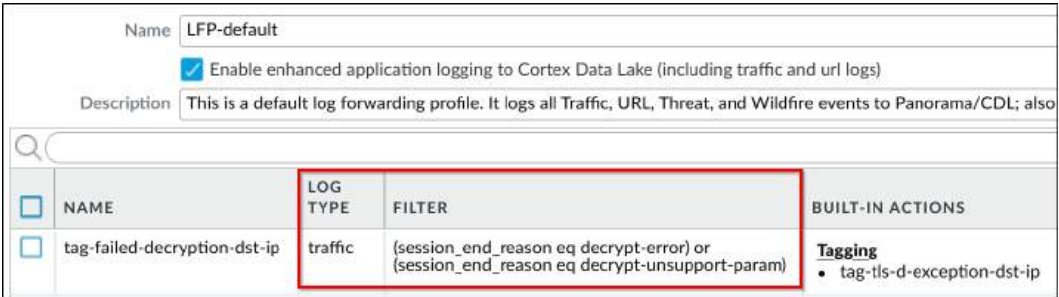


Figure 10.12 – A log forwarding profile rule that defines criteria for automatic decryption exceptions

As the preceding screenshot demonstrates, the tagging action is triggered by a failed TLS handshake during the decryption process. You can simulate this by navigating to `https://badssl.com` from the test computer and clicking on one of the deliberately misconfigured endpoints provided by this site.

Cross-reference the suggested options with the parameters you have configured in the decryption profile and choose a suitable one that would violate, say, the allowed SSL protocol settings. It is recommended to set a short **Time To Live (TTL)** for the tagging action during testing so that the policy lets you try out different scenarios quickly. Another option is to manually remove the tag each time the exception mechanism triggers, and you need to re-enable decryption for the next test.

Response pages and service desk integration

As emphasized many times throughout the book, the firewall response pages are the only visible interface between a user and the firewall policy. It's the first line of technical support that business users have to interact with. Therefore, each block page must be clear and concise, conveying a message understandable by users of varying technical literacy levels, and suggesting a clear path for problem resolution.

The layout of all response pages is defined in the base Jinja template file `ngfw/device/response/pages/<environment>/templates/`

`base.html.j2`. It is referenced in the individual templates covering each of the pages located in the same folder. The templates define the core structure but not the actual verbiage of the messages. The latter are defined in the corresponding YAML files in the adjacent `ngfw/device/response/pages/<environment>/configs` folder and in the `ngfw/device/response/pages/<environment>/shared.yaml` file.

The `shared.yaml` file specifies the company's name, the base URL for the service desk system, and the visual style of the page through embedded **Cascading Style Sheets (CSS)**. ServiceNow is used as an example of the service desk system. You can use any other ticketing system as long as it supports parameterized invocation of forms. In other words, it should be able to accept a link such as `https://<base URL>?form=<form_id>&parameter1=<parameter value>&.....parameterN=<parameter value>`, directing the user to the specified interactive form, prepopulating the relevant form fields with the parameter values, and invoking a workflow that leads to the incident being resolved successfully. You need to work with the team responsible for your service desk system and support workflows to ensure this functionality is in place.

By reviewing the YAML configuration files, you can identify all the use cases that require a service desk form and a corresponding workflow to be created.

Start by reviewing the `shared.yaml` file and updating the base details there: specify your organization and department name, the service desk base URL, and update the color scheme to match your branding style.

Then, open each of the YAML files and review the verbiage. As a rule of thumb, the number of entries in the `conditional_blocks` section of each page will determine the number of use cases this particular response page handles. For example, the application block page has six such blocks, along with a default one that triggers when none of the conditional ones match. It does not necessarily mean that you need six forms for the application blocks alone. Work with your service desk consultants to see whether the forms can be made intelligent enough to adjust based on the parameters passed in the link – this will simplify overall design.

The following is a short extract from the application block page configuration file:

```
page_id: application_block_page
title: "Web Application Blocked"
# ...default message is defined here (not shown)...
servicedesk_form_id:
  non_authorized_user:      a11111111111111111111111111111111
  non_sanctioned_app:      b22222222222222222222222222222222
  blocked_app_category:    c3333333333333333333333333333333
  non_authenticated_user:  d4444444444444444444444444444444
  unknown:                 e5555555555555555555555555555555
  misclassified_app_or_url: f6666666666666666666666666666666

conditional_blocks:
- match: rule
  operator: starts_with
  value: not-authorized-for-
  heading: "Approval required"
  message: |
    This website falls under the "{category}" category
    and needs approval. If you need access, please
    <a href='${servicedesk_base}
    ${servicedesk_form_id.non_authorized_user}&url=
    ${url}&category=${category}&details=${app}&rule=
    ${rule}&time=${timenow.toISOString()}'>
    submit a request</a>.
# ...other conditions follow here (not shown)...
- match: rule
  operator: includes
  value: non-authenticated-
```



```

heading: "Not authenticated"
message: |
    We couldn't verify your identity, which is required
    to access this site. Please <a href='
    ${servicedesk_base}
    ${servicedesk_form_id.non_authenticated_user}
    &url=${url}&category=${category}&details=${app}
    &rule=${rule}&time=${timenow.toISOString()}'>
    contact IT Support</a>.
# ... other use cases follow ...

```

In the first conditional block, you will recognize the mechanism the policy uses to identify the category of the blocked application, while the other one defines the behavior of the page when the identity of the user is unknown.

The following figure shows an example of the page invoked when the user is not authorized to access an application from a managed category:

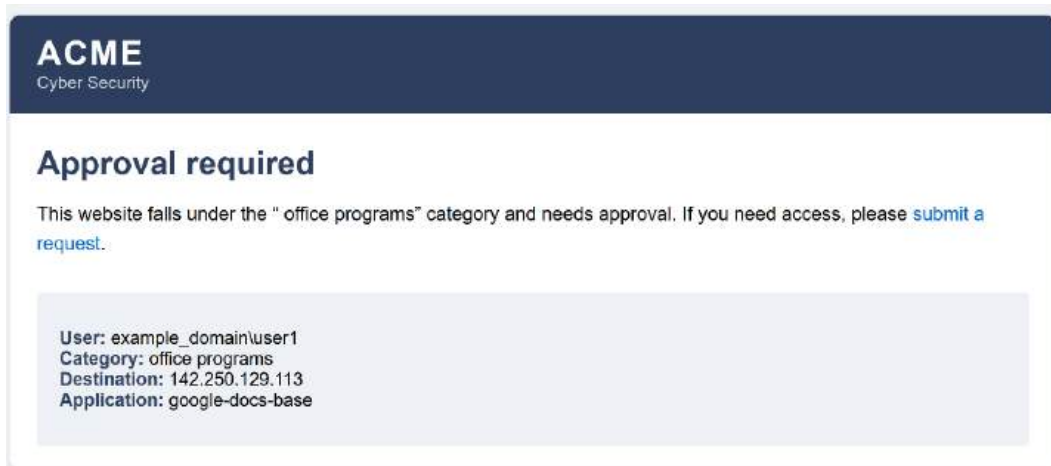


Figure 10.13 – Application block page blocking unauthorized access

And here is the page generated based on the other conditional block shown previously:

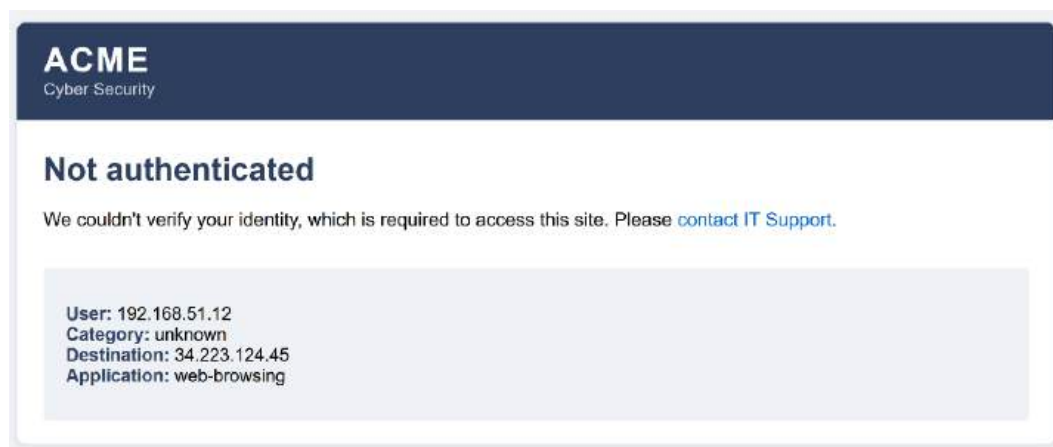


Figure 10.14 – Application response page managing the case of when the source user is unknown

Now, let's discuss the next stage of the quality assurance process.

User acceptance testing

Before rolling out the security policy, **user acceptance testing (UAT)** helps confirm that it's effective for real employees, not just in a controlled lab environment. Whenever possible, assemble a pilot group of users from diverse departments, such as finance, engineering, sales, and IT support. Route all their internet traffic through the lab firewall with the new policy in place. Aim for at least a whole business week of testing to ensure they can complete their regular tasks.

Prerequisites for UAT

Discuss with your networking team the best way to direct traffic to the lab firewall. If you're the networking team, consider setting up a new VLAN and configuring the lab firewall as the default gateway for that VLAN. This is a simple solution that works well for relatively straightforward networks. However, if you're managing a complex network setup, you have the expertise to choose the approach that works best for your specific environment.

Needless to say, the lab firewall must be integrated with the production User-ID infrastructure. The simplest way to do this is through User-ID redistribution. The user groups referenced in the new policy must be created in the user directory, and pilot users should be included in the respective groups to retain their current level of access.

Be aware that the latter process can be quite involved. What you're doing with the pilot group of users is essentially a scaled-down version of a full site migration. You'll need to do almost the same prep work as if you were migrating the entire office. I'll outline a possible methodology in the next chapter, so please read it before starting your pilot. This methodology could even allow you to switch to the new policy without running a traditional pilot project. While I wouldn't recommend it if you're new to this, it's definitely an option. In some cases, with limited resources, UAT might not be feasible. If that's your situation, skim through to the end of this chapter and read the next one.

For now, if you're feeling up to a challenge, concentrate on grasping how to approach pilot testing.

Objectives for the pilot

First, let's discuss what exactly the pilot testing is expected to achieve.

Application coverage

Pilot users should run every critical service your company depends on, including Microsoft 365, GitHub, Zoom, CRM, video conferencing, and ad-hoc web browsing. I am referring to the essentials of our fictitious ACME Inc. here, but you know your list of must-haves. Encourage your pilot users to test edge cases such as large file uploads in OneDrive, code pushes to private GitHub repos, long Zoom breakout sessions, and high-resolution screen shares.

The finance team's end-of-period reporting season might be the ideal stress test for the new policy, yet it's the most stressful time for you as a firewall engineer. Make sure you do **not** put their entire department on the test VLAN. Be open, polite, professional, and reassuring with department managers and team leaders. Convey that what you do is a business requirement, not just a nice-to-have.

Be ready to activate those EDL-driven "break-glass" rules for your essential business-critical apps.

Experience the block pages

When the policy denies or challenges traffic, users will see custom response pages that we tested earlier in this chapter. Ask the pilot group to note the following:

- Is the reason for the block clear?
- Are the next steps obvious (e.g., **Click here to request an exception**)?
- Do support links open the correct form in the service desk portal and pre-populate useful context?

Workflow validation

Track how many tickets the pilot raises, how quickly the support team handles them, and whether the escalation path is effective. A flawless security policy that overwhelms the help desk still fails UAT.

The apparent unobtrusiveness of your new policy and smooth migration will be your main success criteria. Now, let's run through the key focus area during the test.

Running the test

Here's a straightforward approach you can follow for this pilot testing.

1. **Communicate expectations:** Send a brief kickoff email explaining why UAT matters, how long it lasts, and what feedback you need. Include a screenshot of a sample block page and an associated service desk form so users know what to look for.
2. **Log and monitor:** Keep your eyes on the logs. Some errors aren't visible to the end users, but you do not want them to build up. Ideally, stream your logs to a robust logging/SIEM system and build dashboards. This gives you quantitative data to correlate with user feedback. If you do not have a specialist team to help develop proper reporting, the built-in **Automation Command Center (ACC)** dashboards and custom reports still provide sufficient information.
3. **Collect feedback daily:** Use a short survey or a Slack/Teams channel for quick check-ins. Capture application issues, confusing block messages, and general performance impressions.
4. **Iterate fast:** Implement policy adjustments overnight and retest the following day. Decryption exceptions should be handled immediately via EDLs. Fast cycles help keep users engaged and problems contained.

Success criteria

Now, let's discuss the key success criteria of the pilot:

- Not a single critical application failure occurred during the pilot period. If you cause a brief outage for your pilot users, extend the test to reassure them and demonstrate system stability.
- Block pages receive high scores for clarity in the user survey, typically scoring 4 or higher out of 5.

- Support tickets get resolved within the **Service Level Agreement (SLA)**, and no surge in escalations.
- Performance within planned bounds (no noticeable slowdown versus production internet breakout).

Well-run UAT not only validates the firewall policy but also builds trust. When the broader rollout occurs, these pilot users become advocates who reassure their peers that the new controls protect the business without hindering their work.

A note on performance

Performance often comes up in discussions when firewall engineers push for a more secure and feature-rich enterprise policy. Let me give you some points on this topic so you can win these small debates.

Today, more than 95 percent of enterprise traffic is encrypted. Enabling every security control - including full TLS decryption - is therefore not a luxury, but a necessity. Without decryption, the firewall cannot inspect the very traffic it is meant to protect.

That said, you should expect a real performance cost: typically a 40–60 percent reduction in raw throughput once threat prevention, URL filtering, DNS Security, DLP, and decryption are all active. This trade-off is normal, and the alternative is flying blind.

With that in mind, remember two key points:

- Palo Alto's single-pass engine reduces, but does not eliminate, the performance cost. Larger appliances such as the PA-3400 and PA-5400 families absorb the computational load of decryption and signature matching far better than branch-class models. Select hardware for the *enabled* feature set – never for a marketing benchmark measured with half the services disabled.
- All vendors pay essentially the same *tax*. Whether it is Fortinet's ASICs, Cisco's FP chips, or Check Point's Hyperscale blades, performance drops when every packet is decrypted and inspected. Test with your own traffic mix and treat any glossy datasheet figure as a best-case scenario.

The following practical tips will help you effectively evaluate and deploy your next-generation firewall.

- When evaluating a firewall – perhaps by testing it with the very policy included in this book – always use real TLS traffic with decryption enabled. TLS 1.3 handshakes and complex, transaction-heavy SaaS applications place a much heavier load on the device than the synthetic traffic commonly used in laboratory benchmarks.
- If you need to create “no-decrypt” exceptions (for example, for highly sensitive banking applications), keep them to a minimum and document the associated risk.
- Plan for timely upgrades. Signature databases expand, new threats emerge, additional PAN-OS features appear, and both your business and the wider Internet continue to grow – all of which increase the processing demand on the firewall.
- Remember why you purchased the firewall – to provide security, not to showcase bandwidth figures. Enable the necessary features, size the platform correctly, and let the device do its job.

The key takeaway is that security features really show their worth when they’re activated. Make sure to size your firewall appropriately and test its performance in real-world scenarios to ensure that everything runs smoothly.

Summary

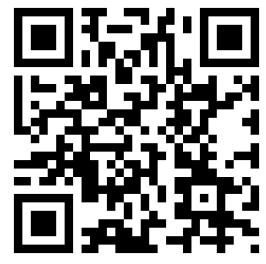
Now that you’ve verified and tested the complete security policy, including decryption, advanced threat prevention, and incident response hooks, it’s time to move forward. We’ve validated every rule against real traffic, handled the inevitable exceptions, and measured the end user experience, so you can rest assured there won’t be any nasty surprises in production.

With this assurance in hand, it’s time to move from the lab to the live network. *Chapter 11, Your First Cutover and Next Steps*, guides you through a zero-impact migration methodology and shows how to build on this success for continuous improvement across your environment. Let’s flip the switch safely, smoothly, and with confidence.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



11

Your First Cutover and Next Steps

Now that you've thoroughly tested the policy, it's time to roll it out to production. You may or may not have had the chance to run a full-scale pilot deployment. Either way, the next challenge is to implement the new controls without disrupting business processes.

This chapter introduces a repeatable cutover methodology that minimizes disruption to business operations. It provides guidance on pre-migration reporting and automation, along with clear success criteria. The discussion then focuses on migration steps and identifying areas for incremental improvement.

By the end of the chapter, you will possess a complete playbook for executing a controlled migration, preserving the gains achieved throughout this book, and planning future enhancements. This chapter will cover the following topics:

- Migration methodology for zero business impact
- Automating preparation tasks
- What you achieved
- Opportunities for further improvement

Migration methodology for zero business impact

No business wants its operations interrupted by an IT system change, which can be frustrating for all involved. Therefore, the key success factor for firewall migrations is minimizing disruption. Without careful planning, such migrations can significantly impact business activities and result in overly permissive security rules that are difficult to remove later.

The ultimate goal is to implement the new security policy that was designed and automated in the earlier chapters of the book.

Let's start by reviewing what needs to be done to apply the new policy to a firewall that is already running a completely different legacy policy. Once we understand this, we'll discuss how to minimize any disruptions.



This chapter primarily discusses Panorama-based infrastructure, a more enterprise-focused solution than standalone firewalls. When relevant, remarks highlight the specifics of working with standalone firewalls. Additionally, **Strata Cloud Manager (SCM)** is actively promoted by the vendor and is thus gaining increasing popularity. Refer to the online documentation on book's GitHub repository for recommendations on integrating SCM support.

Policy formation

To illustrate the technical aspects of the migration, I will use the infrastructure of our reference company, ACME, as an example. Refer to the *Example company* section of *Chapter 2, Navigating Real-World Firewall Management and Cyber Risks*, to remind yourself what it looks like.

The following Panorama device group hierarchy reflects the firewall infrastructure of the organization:

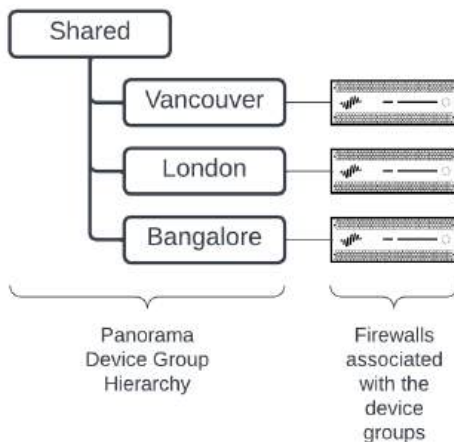


Figure 11.1 – Initial hierarchy of the Panorama device groups

As you can see, it consists of three device groups—one for each physical location. Each site has either a single firewall or an **High Availability (HA)** pair of firewalls assigned to its respective device group.

To start the migration staging, a new device group must be created to host the organization's new global firewall policy. I'll refer to it as *NextGen* in this example. This group should be a direct child of the *Shared* group and include three sub-groups, one for each site to be migrated. From a technical standpoint, the site migration from the legacy production policy to the new global policy includes three key steps:

1. Deploy the new policy to the *NextGen* device group.
2. Copy any critical legacy rules that need to be kept from the old site-level device group to the new one (for example, from *Vancouver* to *Vancouver-new*, as shown in *Figure 11.2*). Some rules might also need to be copied from the *Shared* level; I will clarify this in a moment.
3. Reassign the firewall from the old device group to the new group.

The following figure illustrates these high-level steps:

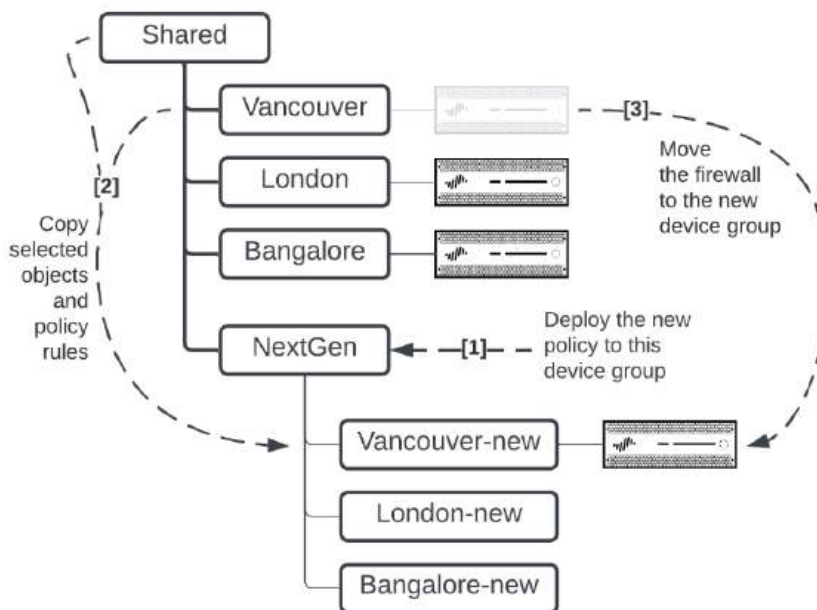


Figure 11.2 – High-level site migration steps

The *Shared* device group may also contain some policy rules. If rules are present in its *PRE* section, they might override key parts of the new policy, potentially invalidating the policy logic. Conversely, rules in the *Shared POST* section would definitely be overshadowed by the *NextGen* policy rules (*Figure 11.3* explains why).

Therefore, all these shared rules should be deactivated for the migrated firewall. The rules would still be in place, but would only apply to other firewalls, not the one being migrated. Such selective deactivation can be achieved by removing the migrated firewall as a target. This can be done either manually, by going to the **Target** tab of each policy rule and deselecting the firewall, or through a script.

The following figure illustrates what the effective policy would look like for a firewall placed in the *Vancouver-new* device group. It also indicates parts of the policy that require modification.

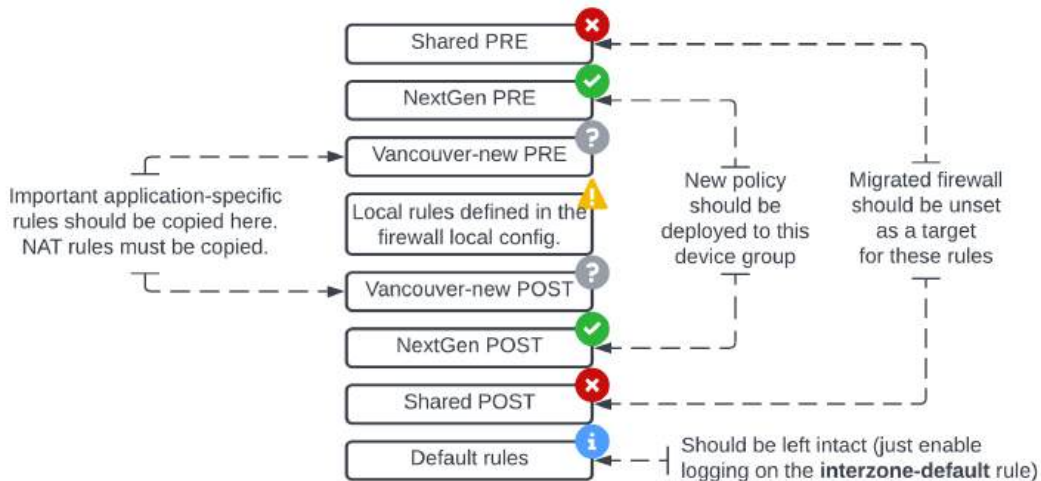


Figure 11.3 – Effective firewall policy and essential modifications

As shown, the effective policy consists of the following groups of rules (listed from top to bottom in this specific order): *Shared PRE* rules, *NextGen PRE* rules, *Vancouver-new PRE* rules, rules defined in the firewall local configuration, *Vancouver-new POST* rules, *NextGen POST* rules, *Shared POST* rules, and finally, *Default rules*.

This is entirely in accordance with the principles discussed in the *Panorama 101 / Policy hierarchy* section of *Chapter 5, Security Policy Design*.

Functionally, the *NextGen* device group supplies the core set of rules, forming the foundation of the policy. As you recall from *Chapter 5, Security Policy Design*, its *PRE* section contains static, hardcoded rules crucial for infrastructure and key business applications. Consequently, the legacy rules you choose to clone to the new site-level device group should follow suit. Only those rules that support vital business applications, not general web browsing, should be copied. We will explore how to prepare and automate this copying in the next section of the chapter: *Automating preparation tasks*.

There is one more component involved in forming the firewall policy. As you may recall, certain parts of the new policy, such as the response pages, need to be deployed using a template. You will have to specify it as one of the components of the policy target definition in the `requirement/policy_targets.json` file. For the migration stage of the project, this template must be added to the template stack of each firewall being migrated. It should be placed on top of the firewall template stack during the migration so that the objects and settings in this template override the corresponding settings in the other templates in the stack.

This will ensure that the response pages generated by the deployment script are the effective response pages applied to the running configuration of the migrated firewall. While this process doesn't necessarily need to be automated, it must be timed correctly. The moment the updated template stack is pushed to the firewall determines when end users will begin receiving the new response pages.

Now, let's focus on automating the handling of legacy policy objects that you need to copy to the new policy to ensure a smooth migration and continuous business operations.

Automating preparation tasks

The legacy ruleset could contain thousands of rules, making it impractical and labor-intensive to analyze and process each one individually.

One great way to make this work easier is by taking advantage of the helpful tools included in the `pan-os-php` framework. The primary advantage of these utilities is their zero-coding approach and extensive out-of-the-box versatility.

To illustrate this, we'll first look at the utilities themselves, which provide ready-made solutions for common migration tasks. You'll see how they can quickly handle use cases such as identifying unused rules, tagging high-risk configurations, and cleaning up objects before migration. Later, we'll contrast this approach with custom Python scripting for situations where you need more tailored automation.

PAN-OS-PHP utilities

The main steps to get started with the pan-os-php framework are straightforward:

1. Follow the installation steps outlined in the *Containerization with Docker* section of *Chapter 7, Setting Up Your Software Development Environment*.
2. (Optional) Once you have Docker fully functional, export the running configuration from your PAN-OS device and save it to a dedicated folder:

```
PS C:\Tools\PaloConfig> ls
Directory: C:\Tools\PaloConfig
Mode                LastWriteTime         Length Name
----                -
-a----           06/07/2025 18:46         600023 running-config.xml
```

3. Get and run pan-os-php packaged as a Docker container:

```
PS C:\Tools\PaloConfig> docker run --name panosphp --rm -v ${PWD}:/share -it swaschkut/pan-os-php:latest
ubuntu@76e42356e433:/share$
```

This single command will download the latest version of the container image and execute it.

4. The /share folder in the container will be mapped to the folder you were in when you invoked the container. Any files created by pan-os-php will be immediately available to you in the host operating system, and vice versa:

```
ubuntu@76e42356e433:/share$ ls -l
total 588
-rwxrwxrwx 1 root 600023 Jul 6 17:46 running-config.xml
```

Now you can run pan-os-php. Its minimum set of parameters is as follows:

```
pan-os-php in=[filename]||[api://IP]||[api://serial@IP] type=<UTIL>
```

The required in parameter determines the target of all actions. The tool can connect to a firewall or Panorama directly through the XML API, route your firewall API calls via Panorama, or use an offline configuration file.

The latter approach is practical for two reasons. First, it offers rapid execution of all operations. Second, it allows many migration changes to be prepared beforehand. This helps shorten the actual change window, as many operations are pre-staged in the configuration file and only need to be imported and applied to the PAN-OS device.

The type parameter refers to one of the 75(!) utility scripts included in the package, such as address, rule, service, and tag, among others.

Run `pan-os-php` without any parameters to see the complete list of available utilities. Adding the `help` keyword after the utility name will provide a built-in help page for the respective utility:

```
pan-os-php in=running-config.xml type=rule help
```

Tools designed to operate objects, such as those mentioned previously, use **filters** to define the scope of **actions**. Available filters and actions vary depending on the utility. You can see them by using the `ListFilters` and `ListActions` parameters, as in this example:

```
pan-os-php in=running-config.xml type=rule ListActions
pan-os-php in=running-config.xml type=rule ListFilters
```

There are too many actions and filters available to list here, so I will provide just a few example use cases to give you an idea of the capabilities of `pan-os-php` in the context of migrating a firewall from one device group to another.



The tags referenced in the following commands, such as `unused-rule`, `any-app`, and `any-svc`, must already exist on Panorama. Create them in **Objects | Tags** before you run the respective commands. When running the commands, make sure to type them as a single line. They are split into multiple lines only for readability.

Use case 1

Identify all unused security rules on the *Vancouver* firewall and tag them all on Panorama.

Usually, there is little to no value in keeping legacy policy rules that have never been used. Here is how you can identify such rules first:

```
pan-os-php in=api://<firewall>
  type=rule
  ruleType=security
  actions=exportToExcel:vancouver-unused-rules.xlsx
  'filter=(rule is.unused.fast)'
```

In this command, the `rule is.unused.fast` filter identifies all security policy rules that have never been used since the creation of the rules or since the usage statistics have been reset, whichever is later. Once identified, all these rules are exported to an Excel spreadsheet.

The command will output detailed information about each rule, while you only need the rule name for the next step. Open this spreadsheet and copy the column with rule names into a new text file, with one rule name per line. Name the file `vancouver-unused-rules.txt`. Then, run the following command against Panorama:

```
pan-os-php in=api://<panorama>
  type=rule
  ruleType=security
  location=Vancouver
  actions=tag-Add:unused-rule
  'filter=(name is.in.file vancouver-unused-rules.txt)'
```

Note the distinction between where we identify unused rules (the actual firewall that processes network traffic) and where we eventually tag these rules (Panorama). This is typically unnecessary because Panorama collects rule usage statistics from all managed firewalls. However, there may be instances when this isn't the case due to communication gaps, Panorama migration to a new appliance, or software bugs that prevent Panorama from receiving accurate statistics from the firewall.

Use case 2

Identify and tag all rules that have any in the application, service, or category field, and tag them accordingly. Also, tag rules that reference high-risk applications.

This command identifies and tags all policy rules that do not use the application field as a matching criterion, effectively converting them into legacy Layer 4 rules:

```
pan-os-php type=rule in=<panorama>
  location=Vancouver
  actions=tag-Add:any-app
  'filter=(app is.any)'
```

The next command identifies and tags all rules where the destination port (service) is not specified:

```
pan-os-php type=rule in=<panorama>
  location=Vancouver
  actions=tag-Add:any-svc
  'filter=(service is.any)'
```

The following command identifies and tags all rules where the URL category is not used as a matching criterion:

```
pan-os-php type=rule in=<panorama>
  location=Vancouver
  actions=tag-Add:any-cat
  'filter=(url.category is.any)'
```

This command identifies and tags policy rules that reference high and very high-risk applications:

```
pan-os-php type=rule in=<panorama>
  location=Vancouver
  actions=tag-Add:high-risk
  'filter=((app risk.recursive.is 4)
    or (app risk.recursive.is 5))'
```

Use case 3

Identify and delete all unused addresses, services, and security profiles.

These commands are straightforward. They remove objects identified as unused, meaning they are not referenced in any rules:

```
pan-os-php type=address in=<panorama>
  location=Vancouver
  actions=delete
  'filter=(object is.unused)'
pan-os-php type=service in=<panorama>
  location=Vancouver
  actions=delete
  'filter=(object is.unused)'
pan-os-php type=securityprofile in=<panorama>
  location=Vancouver
  actions=delete
  'filter=(object is.unused)'
```

It's advisable to run these commands initially with the actions parameter set to display to see what will be deleted.

Use case 4

Remove the -1 suffix added to rule names after their manual cloning in the Panorama UI.

When cloning objects in the PAN-OS administrative GUI, the system always appends a -1 suffix to the new object's name. While this is useful for avoiding name conflicts within the same device group, it becomes unnecessary and problematic when objects are copied across different, unrelated device groups, as there's no risk of duplicate names.

Since this system behavior cannot be changed, you may find the following command very helpful:

```
pan-os-php type=rule in=<panorama>
  location=Vancouver
  actions=name-removeSuffix:-1
  'filter=(name regex "-1$")'
```

As you can see, the filter uses a regular expression that matches all rule names ending with -1. You can easily modify this command to address the same issue with other object types, such as addresses or services.

Use case 5

Remove the *Vancouver* firewall as a target from all rules in the *Shared* device group.

The following command removes the device when it's explicitly in the target list:

```
pan-os-php type=rule in=<panorama>
  location=shared
  actions=target-Remove-Device:<firewall_serial>
  'filter=(target has <firewall_serial>)'
```

However, if firewalls are not explicitly specified as targets, the default setting should be used, which is any. If this is the case, we should set the firewall of interest as the target and then negate this field. This will ensure these rules apply to all firewalls except the one specified, effectively excluding it—aligning with our goal:

```
pan-os-php type=rule in=<panorama> location=shared
  actions=target-Add-Device:<firewall_serial>,target-Negate-Set:yes
  'filter=(target is.any)'
```

Notice how multiple actions can be applied to objects matching the same filter. In this example, the firewall serial number is added first, then the target field is negated, effectively meaning that the firewall is excluded as a target for the rule.

These examples show that the `pan-os-php` utilities allow you to automate many useful tasks that would be hard to perform manually in the Panorama administrative UI. Additionally, no programming skills are needed for this type of automation.

Custom Python scripting

If you prefer to spend time writing your own code instead of figuring out the right mix of third-party tools and their command-line parameters, you can definitely do this with Python.

The main principles are exactly as discussed earlier in *Chapter 8, Policy to Code: Foundations*. You need to import the necessary classes and initialize your PAN-OS device (I will use Panorama in the following examples):

```
from panos.panorama import DeviceGroup, Panorama
from panos.policies import PreRulebase, PostRulebase
from panos.policies import SecurityRule
panos_device = Panorama("<panorama>", "<username>", "<password>")
```

Then narrow down to the target device group:

```
dg = DeviceGroup("<device_group_name>")
panos_device.add(dg)
```

Initialize the pre and post rule bases, associate them with the device group, and retrieve the security policy rules:

```
rulebase_pre = PreRulebase()
rulebase_post = PostRulebase()

dg.add(rulebase_pre)
dg.add(rulebase_post)

current_rules_pre = SecurityRule.refreshall(rulebase_pre)
current_rules_post = SecurityRule.refreshall(rulebase_post)

current_rules = current_rules_pre + current_rules_post
```

Once you have got the rules and rule bases, you can, for example, do the following:

- Get the rule hit count for all the rules:

```
rulebase_pre.opstate.hit_count.refresh("security", rules=current_
rules_pre)
rulebase_post.opstate.hit_count.refresh("security", rules=current_
rules_post)
```

- Identify which rules are unused (have zero hits):

```
unused_rules = []
for r in current_rules:
    if r.opstate.hit_count is not None:
        if r.opstate.hit_count.hit_count == 0:
            unused_rules.append(r)
```

These examples are clear and easy to understand. However, moving a rule from one device group to another can be challenging. Even the PAN-OS administrative UI cannot perform this task unless all objects referenced by the rule are already available in the destination. While single, precise operations – such as identifying unused rules through the opstate helper – fit neatly into short scripts, copying rules between device groups is more complex. When you move a rule, you also inherit its entire network of dependencies (addresses, services, applications, and URL categories), along with scoping constraints (*Shared* vs. device group), tag prerequisites, and policy-ordering considerations. The algorithm must then address three main challenges:

- Identifying what to copy
- Resolving and recreating dependencies in the correct order
- Placing the final rule accurately – without disrupting existing configurations

Copying rules across device groups involves many moving parts, and the underlying implementation is more involved than it may appear at first glance. To make this manageable, the book includes a sample script (`copy-rules.py` in the `/migration` folder) along with docstrings and inline comments that explain the algorithm in detail.

At a high level, the workflow is as follows: connect to Panorama, locate the source and destination device groups, select rules to copy (by tag), and then process each rule in turn.

Before creating anything, the algorithm checks whether referenced objects already exist, searching first in *Shared* (Panorama root) and then in the destination group to respect inheritance. That search order matters: if an object is already defined in *Shared*, you reuse it; you never delete or

overwrite *Shared* objects even when an overwrite is requested, preserving global intent. Conflict handling is explicit and idempotent (either skip existing objects or replace them), while a dry-run mode shows what would happen without making any changes.

Dependency resolution is handled from the bottom up. For each rule, the script traverses every referenced object family: addresses (*AddressObject*, *AddressGroup*, and *Ed1*), services (*ServiceObject* and *ServiceGroup*), applications (*ApplicationObject*, *ApplicationGroup*, and *ApplicationFilter*), and custom URL categories (*CustomUrlCategory*). Groups can contain other groups or members, so the algorithm recursively ensures that members are created before their parents. Built-ins such as *any* and *application-default* are recognized and skipped, and fields that might be strings or lists are normalized to prevent edge-case bugs. Since PAN-OS requires tags to exist before they are referenced, the algorithm also pre-creates any missing tags for both objects and the rule itself.

Only after all dependencies are in place does the script clone the rule. It preserves attributes, guarantees all required tags exist in the destination device group, and then creates the rule at a specific position: top, bottom, or before/after a named rule. This positioning step is critical to avoid changing the evaluation order in the resulting policy.

Finally, the implementation is designed to be both defensive and extensible. It logs issues and proceeds when encountering missing members whenever possible, prevents redundant work through existence checks, and separates policy choices such as skip versus overwrite and dry-run versus apply. While it primarily targets common dependency sets, extending it for additional features such as schedules, log-forwarding profiles, or security profile groups is relatively straightforward.

In practice, these additional features typically do not need the scaling that automation offers or require careful review on a case-by-case basis anyway. For instance, creating a single global log forwarding profile and assigning it to all rules is usually better than managing multiple profiles across different device groups, which could lead to inconsistencies in logging. Conversely, legacy security profiles and profile groups often need manual review and consolidation, as maintaining many profiles is generally counterproductive and can weaken your firewall's security posture. It's best to identify which profiles can be safely deprecated in favor of those used in most rules within the new policy.

Now, let's clarify what specifically you should keep from the old policy and why it's important to do so.

What needs to be preserved

When reviewing which rules to preserve from your legacy policy, it helps to distinguish between those that are essential for maintaining connectivity and business functionality and those that should be avoided because they can weaken security or conflict with the new policy model. The following guidelines highlight the key rules you should keep and those you should exclude or redesign during migration.

Rules to keep are as follows:

- Keep your NAT rules; otherwise, internet access will be broken.
- Keep your QoS, authentication, and policy-based forwarding rules if you have any.
- Keep some or all of your decryption rules if you already have them in the old policy.
- Preserve security rules with specific destinations (where the destination address is not set to any).
- Preserve rules with custom URL categories set as one of the matching criteria.
- Preserve rules with custom application signatures that leverage deep packet inspection.
- Retain bespoke rules that provision access to complex SaaS applications.
- Also, keep rules with custom **External Dynamic Lists (EDLs)** unless their functionality overlaps with what is defined in the new policy. For example, if you have an EDL blocking malicious destinations, it is helpful to cross-reference this with the incident response section in the new policy and choose one or the other. Ensure that you migrate the respective EDL content.

Rules to avoid are as follows:

- Do not keep rules with destinations defined as any. Such rules should be retained only if they leverage one or a combination of the following:
 - Custom URL categories or URL EDLs as additional matching criteria
 - Custom application signatures
- Avoid rules with generic protocol-like application signatures unless they have additional matching criteria such as specific source and destination addresses (for example, a rule allowing zoom-base to any destination is OK, but a rule allowing websocket or ssl to any destination is not OK because these two application signatures describe generic protocols rather than specific commercial applications).

- Do not retain rules that utilize standard URL categories. The new policy will manage these categories according to your definitions in the CSV spreadsheets with business requirements.
- Avoid keeping rules that use application filters based on standard App-ID subcategories. The new policy will manage these categories based on your definitions in the CSV spreadsheets with business requirements. However, if you have rules that use application filters based solely on tagging (whether custom or vendor-provided, such as **G Suite**), ensure that you keep them.
- Do not keep application override rules. Take time to create new rule(s) provisioning access to respective resources by other means (for example, use a custom URL category or a custom signature that leverages deep packet inspection).

After you have chosen which policy rules to keep and successfully cloned them to the new device using one of the methods from the *Automating preparation tasks* section of this chapter, ensure that they are all tagged with the Site-specific applications group tag. The deployment script will have created this tag for you.

With these preparation steps complete, the next phase is to validate the new policy in a safe environment before pushing it to production.

Deployment to a test firewall

To validate the new policy safely, assign a test firewall (such as a lab device) to the new device group (*Vancouver-new* in the ACME's case) and push the new policy to it. Troubleshoot and resolve any potential commit errors as they arise.

Once the commit succeeds, log in to the test firewall directly, navigate to **POLICIES | Security**, and switch from the default view to the **Rulebase by Groups** view, as shown in the following screenshot:

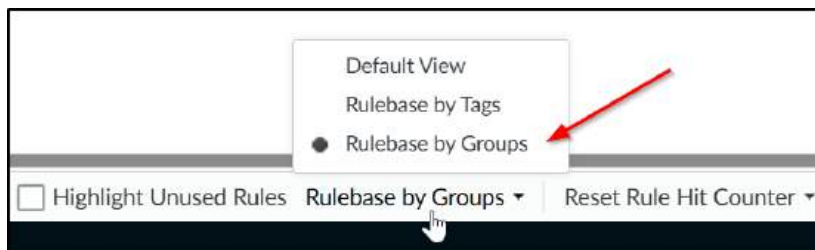


Figure 11.4 – Rulebase by Group view of the policy rules

This should provide you with a clear, structured view of your new policy. It should look similar to the example shown in the following screenshot:

Security	
NAT	
QoS	
Policy Based Forwarding	
Decryption	
Tunnel Inspection	
Application Override	
Authentication	
DoS Protection	
SD-WAN	
DNS Security (4)	1-4
Infrastructure essentials (11)	5-15
Break-glass (3)	16-18
Incident response (4)	19-22
Block lists (4)	23-26
Infrastructure apps (7)	27-33
Business apps (11)	34-44
Site-specific applications (1)	45
Managed APP categories (10)	46-55
Managed URL categories (23)	56-78
Default web browsing (6)	79-84
Block for non-authorized access (5)	85-89
Block for non-sanctioned apps (37)	90-126
Block for non-compliant and anonymous (1)	127

Figure 11.5 – Policy rules grouped by their function

So, what you now have is your entire policy, except for the rules, manually migrated from the legacy policy, which is defined as code. The policy is deployed to a test firewall. All application filters are populated with real applications based on the latest content update from the vendor.

You’re very close to migrating a real production firewall, but not quite there yet. The final step is to clarify what will happen to the usual network traffic of *all* users who currently rely on the legacy firewall policy.

If ensuring uninterrupted user work is your primary goal during migration (which it should be), then the next step is to assess which applications and URLs employees have been successfully accessing under the legacy firewall policy and how the new policy will manage the same traffic.

Once you understand this, you will be able to choose the best action based on your organization’s specific needs. Analyzing the traffic logs generated by the legacy policy will be the key factor in helping you determine the right course of action.

Analyzing logs and predicting policy actions

Fortunately, you have a few key advantages with your new firewall setup. First, the security policy is well structured, enabling you to review any application’s attributes and precisely identify which security rule it matches based on the policy creation criteria. Second, the API methods from the previous chapter enable you to verify whether a network connection will be allowed or blocked, and which policy rule will take that action.

Having a test firewall with a policy that comprises the core policy built by the deployment script and the legacy rules that you decided to retain will make this API-based testing possible.

The only thing you need for this analysis is high-quality traffic and URL-filtering logs, which would let you see the actual current (pre-cutover) usage of all applications and URLs on a per-user basis.

You may already have such logs. Let’s define *high-quality* to understand this. The logging configuration of your production firewall scheduled for migration should have the following characteristics:

- Only custom and built-in content/behavior URL categories are logged in the **CATEGORY** field of the URL filtering log records
- Risk-related URL categories are not logged in the **CATEGORY** field (they would still be logged in the **URL CATEGORY LIST** field)
- All security policy rules are set to **Log at Session End**
- All security policy rules must share the same log forwarding profile to ensure consistent logging across all rules

The following is an example of how good URL log entries should look:

URL	CATEGORY	URL CATEGORY LIST
chatgpt.com/	artificial-intelligence	artificial-intelligence,AI-conversational-assistant,low-risk
clients3.google.com/generate_204	search-engines	search-engines,low-risk
mtalk.google.com/	internet-communications-and-telephony	internet-communications-and-telephony,low-risk
urlfiltering.paloaltonetworks.com/tes...	computer-and-internet-info	UCL-palo-alto-dependencies,computer-and-internet-info,medium-risk

Figure 11.6 – Example of quality URL filtering logging

The preceding screenshot illustrates that the **CATEGORY** field should contain only standard content/behavior-related categories or custom ones, but not the risk-related ones. You can achieve this sort of logging by following the guidelines outlined in the *Logging* section of *Chapter 5, Security Policy Design*.

Your legacy security policy may have thousands of rules, numerous log forwarding profiles, and inconsistent types of session logging (**Log at Session Start**, **Log at Session End**, or both), making it challenging to correct manually. The quickest fix for this would be to use the rules utility that comes with the PAN-OS-PHP framework:

```
pan-os-php type=rule in=<panorama|firewall>  
  location=<device-group|vsys>  
  actions=logEnd-Enable,logStart-Disable,logSetting-set:<lfp_name>
```

This command resets the logging settings for all security rules. Since the `filter` parameter isn't specified, the actions will be applied to all security rules at the specified location.

After configuring logging in your legacy policy as outlined previously, make sure to gather logs over an adequate period. The longer the collection, the better your data quality. Usually, one month suffices, but you can choose a different duration based on how often traffic patterns repeat in your organization, as far as you know. Remember, the log collection period might be constrained by your Panorama log collectors' or firewalls' storage capacity.

Once you've accumulated enough logs, the next step is to prepare two custom reports in **MONITOR | Manage Custom Reports**: one that summarizes traffic by application (App-ID) and another that summarizes traffic by URL category. These reports will form the dataset for analyzing how your users' activity maps to the new security policy.

Traffic report based on App-ID

The first step is to generate a traffic report that shows which applications your users access most frequently. This report will serve as input for the analysis script, helping you test how those applications will behave under the new policy.

Follow these steps to create this report:

1. On Panorama, go to **MONITOR | Manage Custom Reports** and set the configuration scope to the device group where your production firewall is located or to the *Shared* device group.
2. Click on **+ Add** to create a new custom report.
3. Give the report a name and select **Panorama Traffic Log** as the database.
4. Select the desired timeframe.
5. Select **Sort By Count** and **Group By Source User**.

6. Select the source columns as follows:

- **Application**
- **App Container**
- **App Sub Category**
- **IP Protocol**
- **Destination Port**
- **Source Address** (optional)

The latter field should only be included if your users' IP addresses rarely change. If your environment causes users' IP addresses to change constantly, then do not include this field.

7. Create the following query in the **Query Builder** field of the form:

```
((!(addr.dst in '10.0.0.0/8') and
  !(addr.dst in '192.168.0.0/16') and
  !(addr.dst in '172.16.0.0/12')) and
((addr.src in '10.0.0.0/8') or
 (addr.src in '192.168.0.0/16') or
 (addr.src in '172.16.0.0/12')) and
(srcuser neq '') and (session_end_reason neq threat) and (action eq
allow) and (bytes geq 50000) and
(( device_name eq 'firewall-to-be-migrated-1' ) or
 ( device_name eq 'firewall-to-be-migrated-2' ))
```

This is a universal filter that matches traffic from private IP addresses (as defined by RFC 1918) to public IP addresses, intended to match all internet-bound traffic from your end users. Alternatively, you can also use source and destination zone names, which will simplify this query.

The filter allows only traffic from authenticated users, which is routed only through the firewall(s) that are due for migration. The filter also disregards low-volume applications that generated less than 50 KB of traffic during the reporting period. You may want to increase this threshold in reports covering longer time periods.

Run the report and export the result to a CSV file. The resulting data should look as follows:

Source User	Application	App Container	App Sub Category	IP Protocol	Destination Port	Bytes
example_domain\adam.glover	outlook-web-online	ms-office365	email	tcp	443	1318827009
example_domain\adam.glover	web-browsing		internet-utility	tcp	443	1080479956
example_domain\adam.glover	sharepoint-online	ms-office365	social-business	tcp	443	550708667
example_domain\adam.glover	ssl		encrypted-tunnel	tcp	443	472767733
example_domain\adam.glover	websocket		internet-utility	tcp	443	371675312
example_domain\adam.glover	zoom-meeting	zoom	internet-conferencing	tcp	443	306906488
example_domain\adam.glover	boxnet-base	boxnet	file-sharing	tcp	443	267202978
example_domain\adam.glover	ms-teams	ms-office365	office-programs	tcp	443	160549348
example_domain\adam.glover	google-base		internet-utility	tcp	443	72065839
example_domain\adam.glover	zoom-base	zoom	internet-conferencing	tcp	443	68872740
example_domain\adam.glover	ms-powerpoint-online	ms-office365	office-programs	tcp	443	49951539
example_domain\adam.glover	ms-onedrive-business	ms-office365	file-sharing	tcp	443	35950597
example_domain\adam.glover	boxnet-uploading	boxnet	file-sharing	tcp	443	14985224
example_domain\adam.glover	windows-push-notifications		internet-utility	tcp	443	5498524
example_domain\adam.glover	youtube-base	youtube	photo-video	tcp	443	5030325

Figure 11.7 – App-ID-based pre-migration traffic report

Report based on URL filtering

Next, create a report based on URL filtering to capture the categories of websites your users access. Along with the App-ID report, this will give you the dataset needed for testing policy decisions with the analysis script.

Follow these steps to create this report:

1. On Panorama, go to **MONITOR | Manage Custom Reports** and set the configuration scope to the device group where your production firewall is located or to the *Shared* device group.
2. Click on **+ Add** to create a new custom report.
3. Give the report a name and select **Panorama URL Log** as the database.
4. Select the desired timeframe.
5. Select **Sort By Count** and **Group By Source User**.
6. Select the source columns as follows:
 - **Application**
 - **Category**
 - **IP Protocol**
 - **Destination Port**
 - **Source Address** (optional; see the comment in the previous section)

7. Create the following query in the **Query Builder** field of the form:

```
((!(addr.dst in '10.0.0.0/8') and
  !(addr.dst in '192.168.0.0/16') and
  !(addr.dst in '172.16.0.0/12')) and
  ((addr.src in '10.0.0.0/8') or
  (addr.src in '192.168.0.0/16') or
  (addr.src in '172.16.0.0/12')))) and
  (action eq alert) and (user.src neq '') and
  ((app eq web-browsing) or (app eq ssl) or (app eq google-base) or
  (app eq google-app-engine) or (app eq soap) or (app eq websocket))
and
  (( device_name eq 'firewall-to-be-migrated-1' ) or
  ( device_name eq 'firewall-to-be-migrated-2' ))
```

This filter is very similar to the one in the App-ID report, except that it must include only the apps that our new policy allows to provide generic URL filtering. This list of apps must match the applications defined as the APG-web-browsing application group in the `ngfw/objects/application groups/app_groups.json` file.

Run the report and export the result to a CSV file. The resulting data should look as follows:

Source User	Application	Category	IP Protocol	Destination Port	Count
example_domain\test_user	ssl	computer-and-internet-info	tcp	443	1068
example_domain\user1	web-browsing	computer-and-internet-info	tcp	443	909
example_domain\test_user	web-browsing	computer-and-internet-info	tcp	443	728
example_domain\test_user	web-browsing	computer-and-internet-info	tcp	80	705
example_domain\user1	ssl	computer-and-internet-info	tcp	443	598
example_domain\test_user	ssl	computer-and-internet-info	tcp	8883	559
example_domain\user1	ssl	computer-and-internet-info	tcp	8883	487
example_domain\user1	google-base	search-engines	tcp	443	422
example_domain\user1	web-browsing	web-based-email	tcp	80	307
example_domain\user1	websocket	computer-and-internet-info	tcp	443	264

Figure 11.8 – URL-based pre-migration traffic report

You can take this report a step further by also including the **URL Domain** field, which becomes available if you choose the **Panorama URL Summary** database. However, the practical value of this field is relatively low, as a URL cannot be used in the `test_security_policy_match()` SDK method that we discussed in the previous chapter; only the URL category can.

With both reports created, you can now feed them into the analysis script to see how real traffic will map to the new policy. The only caveat you need to bear in mind is that the method implements only the connection-to-rule matching logic but does not take into account the security profiles assigned to the matched rule, including the URL filtering profile.

In other words, let's say you run something like this:

```
>>> from panos.firewall import Firewall
>>> fw = Firewall("<fw_address>", "<user>", "<password>")

>>> fw.test_security_policy_match(source="10.1.1.1",
destination="1.1.1.1", category="malware", protocol="6", port="443",
application="web-browsing")
```

You will get a result as follows:

```
[{'name': 'non-managed-url-categories-regular', 'index': 82, 'action':
'allow'}]
```

This shows you only what rule the connection will match and the action specified in this rule, but not the action specified in the URL filtering profile assigned to the rule. Therefore, you need to add extra logic to identify the URL filtering profile linked to the rule, and then consult the profile configuration to determine the action applied to URLs in the specified category.

The book project includes a script that implements this logic: `migration/traffic-log-analysis.py`.

The analysis outlined previously will help you predict what will happen once your users' regular traffic encounters the new policy.

For example, if a user tries to access an application or URL from a managed category, they will be blocked unless they are part of the relevant user group. To prevent this, you can add the user to the appropriate group *before* the migration, ensuring a seamless transition.

Similarly, you may discover an application that does not match either of the application groups or application filters in the new policy. If you do nothing, the application will be blocked by the new policy as non-sanctioned. Therefore, you need to either expand the selection criteria of the respective managed category, create a dedicated rule allowing this application organization-wide, or keep it blocked as non-sanctioned – the choice is yours.

The most effective approach in commercial organizations is to follow the main success criterion of the migration—ensuring it is non-disruptive. Allow everything that is non-malicious by nature and does not violate your organization’s acceptable use policy and compliance requirements. If you choose to block something that was previously permitted, inform the affected users in advance. This helps prevent surprises and reduces the chance of unnecessary service desk calls on migration day.

An application you discover during the log analysis phase can be an unwanted example of shadow IT. Still, it’s generally a bad idea to handle this issue with a “big bang” approach during policy migration. You can address such applications later because the new policy provides a powerful mechanism to distinguish between sanctioned and non-sanctioned applications, and you can incorporate the change in their status into a service desk workflow, following ITIL best practices.

The migration day

It’s advisable to schedule the migration outside of working hours, ideally on a weekend. However, exceptions exist for 24/7 businesses where “out of hours” does not apply. In such cases, you can still switch to the new policy without any downtime – just allocate extra time for testing and planning the cutover.

Follow your usual change control procedures. Communicate the change to your team and the IT support team. Beyond simply notifying them, work with IT support ahead of time by establishing clear rules of engagement for the migration window. This should include defining which systems and user groups to monitor proactively, which alerts or anomalies require immediate escalation, and how end user issues should be logged and reported.

The IT support team should also prepare a list of tests to confirm that business-critical applications and workflows continue to function as expected. As the firewall engineer, review and validate this list, since you best understand the scope of the change and its potential impact on traffic flows. Arranging a test computer accessible remotely will further help validate connectivity once the new policy is active.

As a precaution, perform a full configuration backup of the migrated firewalls and Panorama, including the firewall device state. Declare a change freeze for the firewall scheduled for migration. Move it to the target device group in advance, assign the new template to its template stack, commit this change to Panorama, and perform commit validation, as shown in the following screenshot:

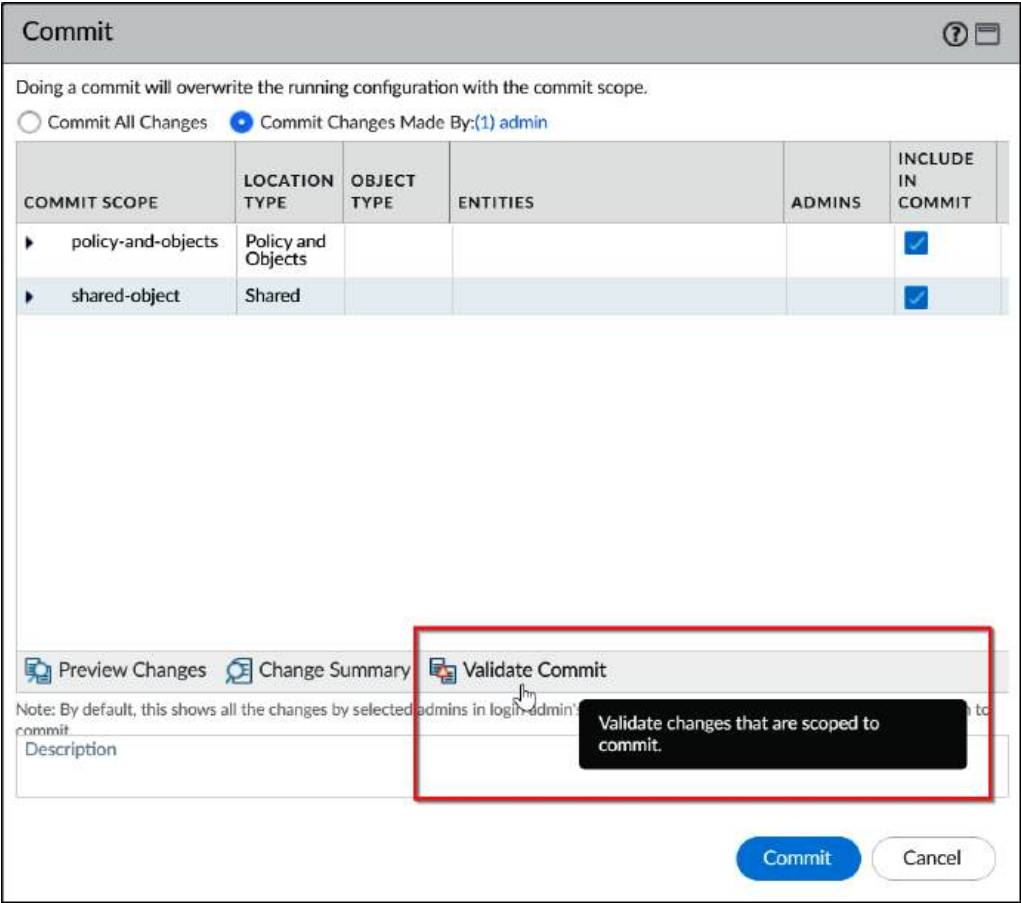


Figure 11.9 – How to validate pending changes

Troubleshoot any potential issues to ensure successful validation. Once you’re in your change window, you can push the policy to the active firewall of the migrated HA pair (provided you have a high-availability active-passive cluster).

After the commit succeeds, run basic connectivity tests from your test computer. If nothing is working, something is likely wrong. In a 24/7 environment, the fastest way to roll back is to fail over to the second HA firewall, which still has the previous policy.

Once you’ve done this and confirmed that connectivity has been restored, take your time to review the running configuration of the first firewall, examine the traffic logs, and troubleshoot the issue. After the problem appears resolved, fail over HA again and retest the policy on the first firewall.

Give it some time. Watch the number of active sessions and traffic logs. The former should remain roughly the same as before the migration, and the latter should not show an excessive number of blocked connections.

Check in with the IT support team. Confirm that everything is quiet, their test cases have succeeded, and no complaints have been raised by end users. Give it a little more time. Have a slow cup of coffee. Enjoy watching the hit counts grow on your beautiful new policy rules. Explore the ACC dashboards and the threat logs for any apparent anomalies.

Now, you can commit the change to the second firewall in the HA pair.

Migration of the subsequent firewalls

As you migrate more sites, you're very likely to discover that the site-specific policy rules you create as part of the preparation and migration work start repeating those you have already made for the sites migrated before.

Merge such rules and convert them to code, including all referenced objects. Ensure each coded rule has a unique name that differs from the respective legacy rule. Deploy the updated global policy, and then delete the old rules. This will ensure zero downtime. Provided that all your company locations are similar in terms of the work employees typically do, you will have to create fewer new policy rules for each subsequent migration.

Eventually, all your policy rules should be defined as code.

What you have achieved

You've reached the end of the book and successfully escaped your legacy policy. You should truly be proud of yourself. This is a major achievement, and not many organizations have accomplished what you have. Let's take a moment to review your work and celebrate your success.

You have implemented a comprehensive enterprise firewall security policy. It includes many features, but here are a few to highlight:

- **Enterprise-wide firewall policy:** A single, centrally managed rule set governs all corporate locations, allowing for site-specific variations
- **Controlled change management:** All changes are version-controlled; proposed revisions can be tested in a dedicated laboratory environment
- **Application and web classification:** Traffic is evaluated by standard and custom categories, with each application designated as sanctioned or unsanctioned

- **Geolocation and regulatory enforcement:** Connections to or from restricted countries and territories are blocked, and access to sensitive application categories is limited in accordance with current legal requirements
- **Risk-adaptive controls and logging:** More stringent enforcement and detailed logging are automatically applied to traffic assessed as higher risk
- **Integrated threat prevention and host containment:** Known malicious sources, malware downloads, and exploit attempts are blocked; compromised hosts are automatically isolated, and incidents are logged
- **Operational resilience and transparency:** Critical business services remain available, users receive clear explanations when access is denied, and administrative overhead is kept to a minimum

Having accomplished this major milestone, you're now in a strong position to pursue further improvements that will expand your policy's capabilities and keep it aligned with future needs.

Opportunities for further improvement

With the policy code stored in a Git repository and generated automatically from a well-defined set of requirements, many avenues for enhancement remain:

- *Add a CLI interface* to complement the interactive deployment UI.
- *Integrate with CI/CD* so policy updates are applied automatically to all targets once a pull request is approved (e.g., via GitHub Actions).
- *Formalize shadow IT management* by tagging each application signature in selected App-ID categories as Sanctioned during the application onboarding process. Do this category by category as resources allow.
- *Implement an access recertification program* to review periodically and reapprove/revoke users' access to managed application and URL categories.
- *Containerize the policy engine* and expose it through an always-on workload with a **Model Context Protocol (MCP)** server frontend, enabling natural language interaction with the policy and its integration into an agentic AI ecosystem.

Beyond these examples, the design is open-ended: the policy can evolve alongside the organization's security requirements without losing structure or auditability.

Summary

This chapter provided a complete guide for migrating your Palo Alto Networks firewall policy into production with minimal disruption to business operations. You’ve learned about a structured method for staging and deploying a new global policy in a Panorama-based setup, using a new device group to centralize management. Key steps included reassigning firewalls, managing legacy rules, and integrating templates to ensure critical applications run smoothly. Automation plays a vital role here – tools like the `pan-os-php` framework and custom Python scripts help streamline rule tagging, rule copying, dependency management, log analysis, and other tasks. This preparation helps you anticipate how the new policy will handle existing traffic, ensuring a smooth rollout.

As the final chapter, this summarizes your journey from firewall policy fundamentals to a fully coded, enterprise-grade security framework. You’ve broken free from legacy constraints, creating a modern policy that is version-controlled, risk-aware, and adaptable. Through detailed log analysis and pre-migration reports, user traffic transitions seamlessly, supported by clear testing and communication to prevent interruptions. By retaining essential rules and automating routine tasks, you’ve built a robust policy that balances security with usability, ready to evolve with your organization’s needs.

Moving forward, the suggestions for improvement offer ways to build on this foundation. Integrating with CI/CD pipelines, managing shadow IT more formally, or adding AI-driven interfaces can further enhance your setup. This book has guided you in transforming firewall management into a code-driven discipline, aligned with DevOps and compliance standards. Celebrate this achievement – a scalable, transparent policy that strengthens your organization’s security while enabling future growth and innovation.

Unlock this book’s exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name.

Note: Keep your purchase invoice ready before you start.



12

Unlock Your Book's Exclusive Benefits

Your copy of this book comes with the following exclusive benefits:

-  Next-gen Packt Reader
-  AI assistant (beta)
-  DRM-free PDF/ePub downloads

Use the following guide to unlock them if you haven't already. The process takes just a few minutes and needs to be done only once.

How to unlock these benefits in three easy steps

Step 1

Have your purchase invoice for this book ready, as you'll need it in *Step 3*. If you received a physical invoice, scan it on your phone and have it ready as either a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.

Note: Did you buy this book directly from Packt? You don't need an invoice. After completing Step 2, you can jump straight to your exclusive content.

Step 2

Scan this QR code or go to packtpub.com/unlock.



On the page that opens (which will look similar to Figure X.1 if you're on desktop), search for this book by name. Make sure you select the correct edition.

A screenshot of the Packt unlock landing page on a desktop. The page features a navigation bar with the Packt logo, a search bar, and links for Subscription, Shopping Cart, and User Profile. Below the navigation bar is a section titled "Discover and unlock your book's exclusive benefits" with a subtext: "Bought a Packt book? Your purchase may come with free bonus benefits designed to maximise your learning. Discover and unlock them here". A progress bar shows three steps: "Discover Benefits" (active), "Sign Up/In", and "Upload Invoice". The main content area is titled "1. Discover your book's exclusive benefits" and contains a search bar labeled "Search by title or ISBN" and a "CONTINUE TO STEP 2" button. Below this are two expandable sections: "2. Login or sign up for free" and "3. Upload your invoice and unlock". A "Need Help?" link is visible in the top right corner of the main content area.

Figure 12.1 - Packt unlock landing page on desktop

Step 3

Once you've selected your book, sign in to your Packt account or create a new one for free. Once you're logged in, upload your invoice. It can be in PDF, PNG, or JPG format and must be no larger than 10 MB. Follow the rest of the instructions on the screen to complete the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. The following QR code will take you to the help page directly:



Note: If you are still facing issues, reach out to customer@packt.com.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

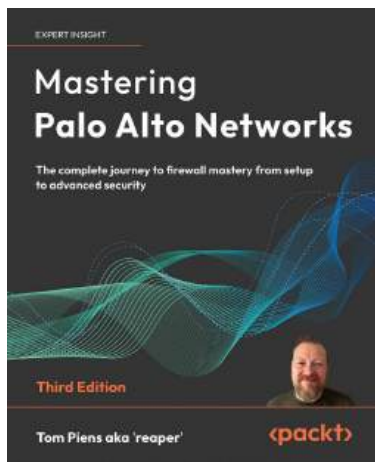
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

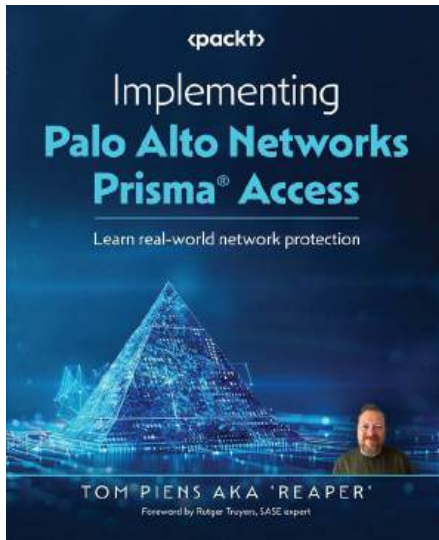


Mastering Palo Alto Networks - Third Edition

Tom Piens aka 'reaper'

ISBN: 978-1-83664-481-1

- Set up and configure Palo Alto firewalls from scratch
- Manage firewall policies for secure network traffic
- Implement VPNs and remote access solutions
- Optimize firewall performance and security settings
- Use threat prevention and traffic filtering features
- Troubleshoot common firewall issues effectively



Implementing Palo Alto Networks Prisma® Access

Tom Piens Aka 'Reaper'

ISBN: 978-1-83508-100-6

- Configure and deploy the service infrastructure and understand its importance
- Investigate the use cases of secure web gateway and how to deploy them
- Gain an understanding of how BGP works inside and outside Prisma Access
- Design and implement data center connections via service connections
- Get to grips with BGP configuration, secure web gateway (explicit proxy), and APIs
- Explore multi tenancy and advanced configuration and how to monitor Prisma Access
- Leverage user identification and integration with Active Directory and AAD via the Cloud Identity Engine

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Palo Alto Networks from Policy to Code*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

actions 369

Action tab 108

Active Directory (AD) 324

Address Groups 80, 81

American Registry for Internet Numbers (ARIN) 82

Anti-Spyware profile 110

DNS exceptions 116

DNS Policies section 112-115

DNS Security algorithm 116-119

Inline Cloud Analysis 116

signature exceptions 112

signature policies 110, 111

Antivirus profile 107

Action tab 107, 108

Anti-Spyware profile 110

signature exceptions 108, 109

wildFire Inline ML 109

API browser 186

APIs

authentication 184

for modern automation 184

PAN-OS REST API 189

PAN-OS XML API 184

scopes 184

XML, versus REST API 191

App-ID 36

Cloud Engine (ACE) feature 90

Application Layer Gateways (ALGs) 37

Applications and Threats 82

application shift 52

application tags 93

artificial intelligence

using 317

asymmetric encryption technology 168

authoritative DNS servers 29

automated incident response 345-351

geo-blocking 351, 352

automated tests 328

App-ID-based rules 334-343

DNS Security 331-333

URL filtering 329, 330

auxiliary functions 129

QoS Marking 129, 130

schedule 129

server response inspection 130

Azure Information

Protection (AIP) 64, 128, 345

B

bloom filter 123

Border Gateway Protocol (BGP) 40

C

Cascading Style Sheets (CSS) 354

Category Match 149

Certificate Authority (CA) 168

Certificate Revocation Lists (CRLs) 169

Certificate Signing Request (CSR) 168

Classless Inter-Domain Routing (CIDR)
notation 7

CLI-based management 182, 183

CLI debug 186

client-to-server (c2s) flow 40

comma-separated values (CSV) 245

**Common Vulnerabilities and
Exposures (CVE)** 119

connection matching 75, 76

application 88, 90

destination zone 77

external dynamic lists 84

regions 82-84

rule types 77

source and destination addresses 78, 79

source and destination address
group 80-82

source and destination port numbers 87

source user 88

source zone 77, 78

core policy principles 142

first iteration 145, 146

policy terminology 144, 145

second iteration 147-149

third iteration 150-155

traffic identification and
classification 142, 143

custom Python scripting 373-375

cybersecurity challenges

versus cybersecurity risks 62

cybersecurity risks

compliance requirements 64

data exfiltration 65

denial of service (DoS) attack 64

distributed denial of service (DDoS)
attack 64

sensitive data 64

unauthorized access, to restricted
resource 62

unauthorized external access, to network 63

versus cybersecurity challenges 62

D

Data Filtering profile 127, 128

debugging 289

file logging, versus console logging 293

IDE debugging 291-293

low-level debugging 298-300

multi-config requests logging 296-298

replayDB 300-303

rich traceback 289-291

strict transactional multi-config 294, 295

decryption policy 173-175

default internal addresses 279

**Defense Advanced Research Projects Agency
(DARPA)** 25

device groups 136, 137

creating 138

DNS-over-HTTPS (DoH) protocol 331

DNS-over-TLS (DoT) protocol 331

DNS records 29

DNS zone 29

Docker 222

 containerization with 222

Docker Desktop 222

Docker Hub 222, 223

Domain Name System (DNS) 28

drift 200

Dynamic IP and Port (DIPP) 18

E

enterprise firewall security policy

 features 387

Ethernet layer 2 interfaces 13

Ethernet layer 3 15

exceptions 352

 automatic exceptions 352

 manual exceptions 352

External Dynamic Lists

 (EDLs) 84-87, 166, 167

F

File Blocking profile 125

File Transfer Protocol (FTP) 37

filters 369

firewall automation

 automation approach, selecting 205, 206

 best practices 206

 general automation best practices 206, 207

 implementation guidelines 206

 integration patterns 208

 performance enhancements 208

 security considerations 207

firewall management

 API-driven automation 182

 CLI-based management 182, 183

 configuration as code 182

 evolution 182

 full automation 183

 intent-based networking 182

 web interface (GUI) management 182

firewall policy creation

 algorithm 260-273

 considerations 258-260

firewall response pages 312, 313

firewalls

 managing 58-61

flow key 40

Forwarding Information Base (FIB) 78

forward trust certificate 279-285

full automation

 journey 183

Fully Qualified Domain Name (FQDN) 79

functional requirement (FRs) 69

 example 66-68

G

GitHub

 account creation 212

 configuring 212

 project source code, copying 212-214

GitHub Actions 225, 226

GitHub fork 212, 213

google-app-engine application 151

Go SDK (pango) 193

 client-side filtering 196

 installing 194

 packages 194

Graphical User Interface (GUI) 191

Group Policy Objects (GPOs) 285

G Suite 377

H

HashiCorp Configuration

Language (HCL) 202

Host Information Profiles (HIPs) 76

host layers 6

I

infrastructure-as-code tools 198

benefits 199

imperative, versus declarative
languages 199

PAN-OS Ansible 201, 202

PAN-OS Terraform provider 202, 203

Terraform, versus Ansible 200

integrated development

environment (IDE) 214

intermediate certificate 168

Internet Assigned Numbers

Authority (IANA) 10

Internet Control Message Protocol (ICMP) 10

Internet Corporation for Assigned Names
and Numbers (ICANN) 28

Internet Protocol (IP) 6

Internet Service Providers (ISPs) 17

IP header 8

IT service desk 156

interface 156-160

J

JavaScript Object Notation (JSON) 245

Jinja2 templates

using 312

K

key definitions 246-253

L

leaf certificate 168

legacy custom application signatures 175

legacy policy objects

automating 367

legacy policy rules 175

load balancing 31-35

log forwarding profile 102

Built-in Actions 104, 105

log forwarding profile match list 102-104

logging 176, 177

log settings, matching connections 101

log forwarding profile 102

M

MAC address 13

machine learning (ML) models 109

Mail Transfer Agent (MTA) 110

manual tests 343

data filtering 344, 345

file blocking 343

matching connections

action setting 100, 101

log settings 101

processing 100

profile setting 106, 107

media layers 6

migration day 385-387

migration methodology

for zero business impact 363

legacy policy objects, automating 367

- logs, analyzing 379, 380
- of subsequent firewalls 387
- policy actions, predicting 379, 380
- policy formation 364-367
- rules, avoiding 376-378
- rules, preserving 376
- test firewall, assigning to device group 377

Model Context Protocol (MCP) 388**multi-config requests 187**

- example request/response 188, 189
- multi-config XML document format 187
- transactional support 188

N**name resolution 28, 29****name resolution algorithm 29, 30****naming convention 161-163**

- enforcing 313-316

Network Address Translation (NAT) 18, 78, 286, 287**networking 101 4**

- load balancing 31-35
- name resolution 28
- Open Systems Interconnection (OSI) model 4-6

network layer, OSI model 6, 7

- interfaces 13-15
- IPv4 addresses, types 16, 17
- IPv4 header 8-12
- IPv6 protocol 18
- network address translation 17
- zones 16

network mask 7**next-gen firewalls 36**

- example network connection 37, 38
- packet flow - App-ID 39-52

- packet flow - content identification and scanning 53
- stateful firewalls 37

non-functional requirements (NFRs) 66, 70, 71**O****Online Certificate Status Protocol (OCSP) 169****Open Shortest Path First (OSPF) 40****Open Systems Interconnection (OSI) model 4-6**

- application layer 25-27
- network layer 6, 7
- transport layer 18

Operational Technology (OT) traffic 154**P****packets 7****Panorama 186**

- API calls, proxying with 186, 187

PAN-OS 10**PAN-OS Ansible collection 201**

- features 201, 202

pan-os-php framework 367

- use cases 369-372
- utilities 368, 369

PAN-OS REST API 189

- example REST API request/response 190, 191
- REST API URL 190

PAN-OS SDK for Python 230

- applications, working with 231-235
- firewall or Panorama, connecting to 230, 231
- methods 243, 244

performance optimization 235-243

policies, working with 231-235

PAN-OS security policy

application categories and
subcategories 90, 91

application characteristics and risks 91

components 74, 75

connection matching features 75, 76

PAN-OS Terraform provider

commit operations 205

features 202, 203

PAN-OS XML API 184

calls, proxying via Panorama 186

example XML API request/response 185

multi-config requests 187

parameters 185

XPath, finding 186

partial hash 109

PHP SDK (pan-os-php) 197

CLI usage 198

custom scripting 197, 198

pilot testing

application coverage 358

block pages 358

objectives 358

running 359

success criteria 359

workflow validation 359

Platform-as-a-Service (PaaS) 151

policy formation 364-367

policy hierarchy 139-141

effective firewall policy 140

policy rules

representation 254-258

profile setting, matching

connections 106, 107

Anti-Spyware profile 110

Antivirus profile 107

Data Filtering profile 127, 128

File Blocking profile 125

URL Filtering profile 120

Vulnerability Protection profile 119

WildFire Analysis profile 126

project functionality 276-278

protocol 6

Public Key Infrastructure (PKI) 167-170

PyCharm 215

installation 215, 216

learning project 218, 219

project creation 219-221

setting up 216, 217

Python 214

Python SDK 191

direct, versus proxy firewall connection 192

object-oriented framework 191

VSYS awareness 193

Q

quality of service (QoS) marking 129

R

Read-Only Domain Controller (RODC) 123

region 82

repository structure 245-253

Réseaux IP Européens Network Coordination
Centre (RIPE NCC) 82

resolver 29

response pages 131, 156

limitations 133

- service desk integration 354-357
- types 131, 132
- using 156-161
- variables 132

REST API URL

- base path 190
- resource URI 190

Role-Based Access Control (RBAC) 200**root CA 168****Root Certificate Store 169****rule type 77**

- interzone type 77
- intrazone type 77
- universal type 77

runner 225**S****SDK gaps**

- covering, with XML API 303-309

Secure Shell (SSH) protocol 172**session key 170****Simple Object Access Protocol (SOAP) 151****six-tuple 39****software development kits (SDKs) 191**

- Go SDK 193, 194
- PHP SDK 197
- Python SDK 191

SSH Proxy 172**SSL Forward Proxy 170, 171****SSL Inbound Inspection 172****Strata Cloud Manager (SCM) 364****symmetric decryption 170****symmetric encryption 170****T****tagging 163, 164****templates 136, 141****template stacks 141, 142****three-way handshake 19****Time To Live (TTL) value 29****TLS decryption 167**

- decryption policy 173-175
- SSH Proxy 172
- SSL Forward Proxy 170, 171
- SSL Inbound Inspection 172

traffic report

- generating, based on App-ID 380, 381
- generating, based on URL filtering 382-385

Transmission Control Protocol (TCP) 10, 19-23**transport layer, OSI model 18**

- TCP 19-23
- UDP 23-25

Transport Layer Security (TLS) 167-170**U****URL category 93**

- custom URL categories 95-97
- predefined URL categories 94, 95

URL Filtering profile 120

- categories 121, 122
- settings 122
- user credential detection 123, 124

user acceptance testing (UAT) 357

- pilot testing, objectives 358
- prerequisites 357, 358

User Datagram Protocol (UDP) 10, 23-25

User-ID and Windows domain 287, 288

user identity 166

setting up, for testing 324-328

V

version control system (VCS) 212

virtual router 15

Virtual Systems (VSYS) 193

Virtual Wire (v-wire) mode 12

VLAN (Virtual Local Area Network) 13

interface 14

Vulnerability Protection profile 119

exceptions 120

rules 119

W

WebSocket 151

web UI debug 186

WildFire Analysis profile 126

windows 19

X

x509 certificate 168

XML API calls

without XPath 310, 311

XML API request 185

example 185

XPath

identifying methods 186

Z

zone 16, 29

record 29

zone names 278, 279

