Network Programmability and Automation Fundamentals

KHALED ABUELENAIN, CCIE® No. 27401
JEFF DOYLE, CCIE® No. 1919
ANTON KARNELIUK, CCIE® No. 49412
VINIT JAIN, CCIE® No. 22854

ciscopress.com

Network Programmability and Automation Fundamentals

Khaled Abuelenain, CCIE No. 27401
Jeff Doyle, CCIE No. 1919
Anton Karneliuk, CCIE No. 49412
Vinit Jain, CCIE No. 22854

Cisco Press
Hoboken, New Jersey

Network Programmability and Automation Fundamentals

Published by:
Cisco Press

ScoutAutomatedPrintCode

Library of Congress Control Number: 2020922839

ISBN-13: 978-1-58714-514-8
ISBN-10: 1-58714-514-6

Warning and Disclaimer

This book is designed to provide information about network programmability and automation. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an “as is” basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the author and are not necessarily those of Cisco Systems, Inc.

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Cisco Press or Cisco Systems, Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book or otherwise alter it to better suit your needs, you can contact us through email at feedback@ciscopress.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Editor-in-Chief: Mark Taub

Director, ITP Product Management: Brett Bartow

Alliances Manager, Cisco Press: Arezou Gol

Managing Editor: Sandra Schroeder

Development Editor: Ellie C. Bru

Project Editor: Mandie Frank

Copy Editor: Kitty Wilson

Technical Editors: Jeff Tantsura, Viktor Osipchuk

Editorial Assistant: Cindy Teeters

Designer: Chuti Prasertsith

Composition: codeMantra

Indexer: Ken Johnson

Proofreader: Abigail Bass

Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV Amsterdam,
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at **www.cisco.com/go/offices.**

Credits

Figure/Text Selection	Attribution/Credit
"HTTP is not designed to be a transport protocol. It is a transfer protocol in which the messages reflect the semantics of the Web architecture by performing actions on resources through the transfer and manipulation of representations of those resources [Section 6.5.2]"	© Roy Thomas Fielding, 2000
"This specification [HTTP/2.0] is an alternative to, but does not obsolete, the HTTP/1.1 message syntax. HTTP's existing semantics remain unchanged."	Hypertext Transfer Protocol Version 2
"a sequence of octets, along with representation metadata describing those octets that constitutes a record of the state of the resource at the time when the representation is generated."	Uniform Resource Identifier (URI): Generic Syntax, Copyright © The Internet Society (2005)
"While RFC 2396, section 1.2, attempts to address the distinction between URIs, URLs and URNs, it has not been successful in clearing up the confusion."	IETF (Internet Engineering Task Force). Architectural Principles of Uniform Resource Name Resolution, ed. K. Sollins. 1998
1. Device state metrics; 2. Data from shared services such as DDI (DNS, DHCP and IPAM) and Active Directory; 3. Network flows from sources such as NetFlow; and 4. Configuration data normalized into key value pairs.	Shamus McGillicudy, "A Network Source of Truth Promotes Trust in Network Automation," Enterprise management Associates, May 2020
"Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale".	©2020 Agile Alliance
"allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery."	Copyright (c) 2020 IETF
"the most important security protocol on the internet"	Copyright (c) 2020 IETF
"there was more TLS 1.3 use in the first five months after RFC 8446 was published than in the first five years after the last version of TLS was published as an RFC"	Copyright (c) 2020 IETF

Figure 1-6	CI/CD	©2021 Red Hat, Inc. www.redhat.com
Figure 6-2	Accessing a Django Application in a Web Browser	© 2005-2021 Django Software Foundation
Figure 7-1	The SupportApache-small.png Image, as It Appears in a Web Browser	Copyright © 2020 The Apache Software Foundation
Figure 7-6	The Postman Interface	©2020 Postman, Inc.
Figure 7-7	A GET Request Using Postman	©2020 Postman, Inc.
Figure 7-8	Viewing the Response Headers in Postman	©2020 Postman, Inc.
Figure 13-1	GitHub Website with YANG Modules	© 2020 GitHub, Inc.
Figure 17-3	Comparing the Body of the Response in the Developer Sandbox and Postman	Screenshot of Comparing the body of the response in the Developer Sandbox and Postman ©2020 Postman, Inc.
Figure 18-3	Converting a username:password Tuple to Base64 Format	Screenshot of Converting username:password tuple to Base64 format © Cisco systems
Figure 18-4	Arista YANG Modules	Screenshot of Arista YANG modules © 2020 GitHub, Inc.
Figure 19-3	Available Ansible Module Categories	Copyright © 2020 Red Hat, Inc.

About the Authors

Khaled Abuelenain, CCIE No. 27401 (R&S, SP), is currently the Consulting Director at Acuative, a Cisco Managed Services Master Partner. Khaled has spent the past 18 years designing, implementing, operating, and automating networks and clouds. He specializes in service provider technologies, SD-WAN, data center technologies, programmability, automation, and cloud architectures. Khaled is especially interested in Linux and OpenStack.

Khaled is a contributing author of the best-selling Cisco Press book *Routing TCP/IP, Volume II*, 2nd edition, by Jeff Doyle. He also blogs frequently on network programmability and automation on blogs.cisco.com. Khaled is also a member of the DevNet500 group, being one of the first 500 individuals in the world to become DevNet certified.

Khaled lives in Riyadh, Saudi Arabia, and when not working or writing, he likes to run marathons and skydive. He can be reached at kabuelenain@gmail.com, on Twitter at @kabuelenain or on LinkedIn at linkedin.com/in/kabuelenain.

Jeff Doyle, CCIE No. 1919, is a Member of Technical Staff at Apstra. Specializing in IP routing protocols, complex BGP policy, SDN/NFV, data center fabrics, IBN, EVPN, MPLS, and IPv6, Jeff has designed or assisted in the design of large-scale IP and IPv6 service provider networks in 26 countries over 6 continents.

Jeff is the author of *CCIE Professional Development: Routing TCP/IP, Volumes I and II* and *OSPF and IS-IS: Choosing an IGP for Large-Scale Networks*; a co-author of *Software-Defined Networking: Anatomy of OpenFlow*; and an editor and contributing author of *Juniper Networks Routers: The Complete Reference*. Jeff is currently writing *CCIE Professional Development: Switching TCP/IP*. He also writes for Forbes and blogs for both *Network World* and *Network Computing*. Jeff is one of the founders of the Rocky Mountain IPv6 Task Force, is an IPv6 Forum Fellow, and serves on the executive board of the Colorado chapter of the Internet Society (ISOC).

Anton Karneliuk, CCIE No. 49412 (R&S, SP), is a Network Engineer and Manager at THG Hosting, responsible for the development, operation, and automation of networks in numerous data centers across the globe and the international backbone. Prior to joining THG, Anton was a team lead in Vodafone Group Network Engineering and Delivery, focusing on introduction of SDN and NFV projects in Germany. Anton has 15 years of extensive experience in design, rollout, operation, and optimization of large-scale service providers and converged networks, focusing on IP/MPLS, BGP, network security, and data center Clos fabrics built using EVPN/VXLAN. He also has several years of full-stack software development experience for network management and automation.

Anton holds a B.S. in telecommunications and an M.S. in information security from Belarusian State University of Informatics and Radio Electronics. You can find him actively blogging about network automation and running online training at Karneliuk.com. Anton lives with his wife in London.

Vinit Jain, CCIE No. 22854 (R&S, SP, Security & DC), is a Network Development Engineer at Amazon, managing the Amazon network backbone operations team. Previously, he worked as a technical leader with the Cisco Technical Assistance Center (TAC), providing escalation support in routing and data center technologies. Vinit is a speaker at various networking forums, including Cisco Live! events. He has co-authored several Cisco Press titles, such as *Troubleshooting BGP*, and *Troubleshooting Cisco Nexus Switches and NX-OS*, *LISP Network Deployment and Troubleshooting,* and has authored and co-authored several video courses, including *BGP Troubleshooting*, the *CCNP DCCOR Complete* video course, and the *CCNP ENCOR Complete* video course. In addition to his CCIEs, Vinit holds multiple certifications related to programming and databases. Vinit graduated from Delhi University in mathematics and earned a master's in information technology from Kuvempu University in India. Vinit can be found on Twitter as @VinuGenie.

About the Technical Reviewers

Jeff Tantsura, CCIE No. 11416 (R&S), has been in the networking space for over 25 years and has authored and contributed to many RFCs and patents and worked in both service provider and vendor environments.

He is co-chair of IETF Routing Working Group, chartered to work on new network architectures and technologies, including protocol-independent YANG models and next-generation routing protocols. He is also the co-chair of the RIFT (Routing in Fat Trees) Working Group, chartered to work on a new routing protocol that specifically addresses fat tree topologies typically seen in the data center environment.

Jeff serves on the Internet Architecture Board (IAB). His focus has been on 5G transport and integration with RAN, IoT, MEC, low-latency networking, and data modeling. He's also a board member of San Francisco Bay Area ISOC chapter.

Jeff is Head of Networking Strategy at Apstra, a leader in intent networking, where he defines networking strategy and technologies.

Jeff also holds the certification Ericsson Certified Expert IP Networking.

Jeff lives in Palo Alto, California, with his wife and youngest child.

Viktor Osipchuk, CCIE No. 38256 (R&S, SP), is a Senior Network Engineer at Google, focusing on automation and improving one of the largest production networks in the world. Before joining Google, Viktor spent time at DigitalOcean and Equinix, helping to architect and run their worldwide infrastructures. Viktor spent many years at Cisco, supporting customers and focusing on automation, telemetry, data models, and APIs for large-scale web and service provider deployments. Viktor has around 15 years of diverse network experience, an M.S. in telecommunications, and associated industry certifications.

Dedications

Khaled Abuelenain: To my mother, the dearest person to my heart, who invested all the years of her life so I can be who I am today. I owe you more than any words can express. To my father, my role model, who always led by example and showed me the real meaning of work ethic. Nothing I do or say will ever be enough to thank you both.

And to the love of my life, my soulmate, and my better half, Mai, for letting me work and write while you take care of, literally, everything else. This book would not have happened if not for your phenomenal support, patience and love. I will forever be grateful for the blessing of having you in my life.

Jeff Doyle: I would like to dedicate this book to my large and growing herd of grandchildren: Claire, Samuel, Caroline, Elsie, and Amelia. While they are far too young to comprehend or care about the contents of this book, perhaps someday they will look at it and appreciate that Grampa is more than a nice old man and itinerant babysitter.

Anton Karneliuk: I dedicate this book to my family, which has tremendously supported me during the writing process. First of all, many thanks to my amazing wife, Julia, who took on the huge burden of sorting out many things for our lives, allowing me to concentrate on the book. You acted as a navigation star during this journey, and you are my beauty. I'd also like to thank my parents and brother for me helping me form the habit of working hard and completing the tasks I've committed to, no matter how badly I want to drop them.

Vinit Jain: I would like to dedicate this book to the woman who has been a great influence and inspiration in my life: Sonal Sethia (*Sonpari*). You are one of the most brilliant, talented, courageous, and humble people I have ever known. You have always inspired me to push myself beyond what I thought I was capable of. You have been there for me during difficult times and believed in me when even I did not. You are my rock. This is a small token of my appreciation, gratitude, and love for you. I am really glad to have found my best friend in you and know that I will always be there for you.

Acknowledgments

Khaled: First and foremost, I would like to thank Jeff Doyle, my co-author, mentor, and friend, for getting me started with writing, and for his continuous assistance and guidance. Jeff has played a fundamental role in my professional life as well as in the lives of many other network engineers; he probably doesn't realize the magnitude of this role! Despite all that he has done for this industry and the network engineering community, Jeff remains one of the most humble and amiable human beings I have ever come across. Thank you, Jeff, I owe you a lot!

I am grateful to Anton and Vinit for agreeing to work with me on this project. It has been challenging at times, but it has been seriously fun most of the time.

I would also like to thank Jeff Tantsura and Viktor Osipchuk for their thorough technical reviews and feedback. I bothered Viktor very frequently with discussions and questions over email, and never once did he fail to reply and add a ton of value!

I especially want to thank Brett Bartow and Eleanor Bru for their immense support and phenomenal patience. And I'm grateful to Mandie Frank, Kitty Wilson, and everyone else at Cisco Press who worked hard to get this book out to the light. Such an amazing team.

Jeff Doyle: I would like to express my thanks to my friend Khaled Abuelenain for bringing me into this project, and thanks to Anton and Vinit for letting me be a part of their excellent work. Thanks also to Brett Bartow and everyone at Pearson, whom I've worked with for many years and continue to tell everyone who will listen that this is the best publishing team any technical writer could hope to work for. Finally, thanks to my wife Sara who, as always, puts up with my obsessiveness. When she sees me sitting and staring into nothingness she knows there's writing going on in my head.

Anton: Special thanks to Schalk Van Der Merwe, CTO, and Andrew Mutty, CIO, at The Hut Group for believing in me and giving me freedom and responsibility to implement my automation ideas in a high-scale data center environment. Thanks to all my brothers-in-arms from The Hut Group hosting networks for constantly sharing with me ideas about what use cases to focus on for automation. I want to thank my previous manager in Vodafone Group, Tamas Almasi, who supported me during my initial steps in network automation and helped me create an appropriate mindset during numerous testbeds and proofs of concept. Last but not least, I'm very grateful to Khaled Abuelenain for his invitation to co-author this book and the whole author and technical reviewer team; it was a pleasure to work with you.

Vinit: A special thanks to Khaled for asking me to co-author this book and for being amazingly patient and supportive of me as I faced challenges during this project. I would like to thank Jeff Doyle and Anton Karneliuk for their amazing collaboration on this project. I learned a lot from all of you guys and look forward to working with all of you in the future.

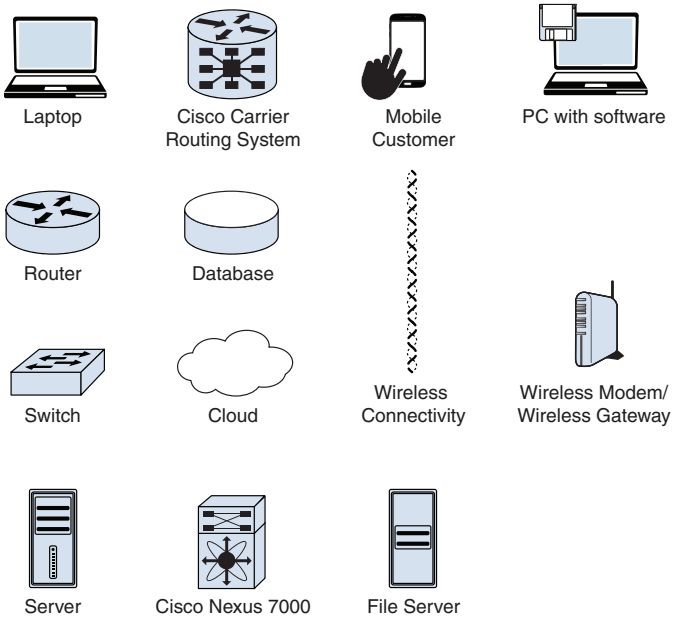
I would also like to thank our technical reviewers, Jeff Tantsura and Viktor Osipchuk, and our editor, Eleanor Bru, for your in-depth verification of the content and insightful input to make this project a successful one.

This project wouldn't have been possible without the support of Brett Bartow and other members of the editorial team.

Contents at a Glance

Contents

Icons Used in This Book



Command Syntax Conventions

The conventions used to present command syntax in this book are the same conventions used in Cisco's Command Reference. The Command Reference describes these conventions as follows:

- **Boldface** indicates commands and keywords that are entered literally as shown. In actual configuration examples and output (not general command syntax), boldface indicates commands that are manually input by the user (such as a **show** command).
- *Italics* indicate arguments for which you supply actual values.
- Vertical bars (|) separate alternative, mutually exclusive elements.
- Square brackets [ ] indicate optional elements.
- Braces { } indicate a required choice.
- Braces within brackets [{ }] indicate a required choice within an optional element.

Note This book covers multiple operating systems, and in each example, icons and router names indicate the OS that is being used. IOS and IOS XE use router names like R1 and R2 and are referenced by the IOS router icon. IOS XR routers use router names like XR1 and XR2 are referenced by the IOS XR router icon.

Introduction

For more than three decades, network management has been entirely based on the command-line interface (CLI) and legacy protocols such as SNMP. These protocols and methods are severely limited. The CLI, for example, is vendor specific, lacks a unified data hierarchy (sometimes even for platforms from the same vendor), and was designed primarily as a human interface. SNMP suffers major scaling problems, is not fit for writing configuration to devices, and overall, is very complex to implement and customize.

In essence, automation aims at offloading as much work from humans as possible and delegating that work to machines. But with the aforementioned legacy interfaces and protocols, machine-to-machine communication is neither effective nor efficient; and at times, close to impossible.

Moreover, device configuration and operational data have traditionally lacked a proper hierarchy and failed to follow a data model. In addition, network management workflows have always been far from mature, compared to software development workflows in terms of versioning, collaboration, testing, and automated deployments.

Enter network programmability. Programmability revolves around programmable interfaces, commonly referred to as application programming interfaces (APIs). APIs are interfaces that are designed primarily to be used for machine-to-machine communication. A Python program accessing a network router to retrieve or push configuration, without human intervention, is an example of a machine-to-machine interaction. Contrast this with the CLI, where a human needs to manually enter commands on a device and then visually inspect the output.

Network equipment vendors (for both physical and virtual equipment) are placing ever-increasing emphasis on the importance of managing their equipment using programmable interfaces, and Cisco is at the forefront of this new world. This new approach to managing a network provides several benefits over legacy methods, including the following:

- Normalizing the interface for interaction with network platforms by abstracting communication with these platforms and breaking the dependency of this communication on specific network OS scripting languages (for example, NX-OS, IOS XR, and Junos OS)
- Providing new methods of interacting with network platforms and, in the process, enabling and aligning with new technologies and architectures, such as SDN, NFV, and cloud
- Harnessing the power of programming to automate manual tasks and perform repetitive tasks efficiently
- Enabling rapid infrastructure and service deployment by using workflows for service provisioning
- Increasing the reliability of the network configuration process by leveraging error checking, validation, and rollback and minimizing human involvement in the configuration process

- Using common software development tools and techniques for network configuration management, such as software development methodologies, versioning, staging, collaboration, testing, and continuous integration/continuous delivery

This book covers all the major programmable interfaces used in the market today for network management. The book discusses the protocols, tools, techniques, and technologies on which network programmability is based. Programming, operating systems, and APIs are not new technologies. However, programmable interfaces on network platforms, and using these programmable interfaces to fully operate and maintain a network, along with the culture accompanying these new methods and protocols, may be (relatively) new. This book explains, in detail, all the major components of this new ecosystem.

Goals and Methods of This Book

This is a "fundamentals" book aimed at transitioning network engineers from a legacy network-based mindset to a software-based (and associated technologies) mindset. A book covering fundamentals generally struggles to cover as many subjects as possible with just enough detail. The fine balance between breadth and depth is challenging, but this book handles this challenge very well.

This book introduces the emerging network programmability and automation ecosystem based on programmable interfaces. It covers each protocol individually, in some significant detail, using the relevant RFCs as guiding documents. Protocol workflows, messages, and other protocol nuances tend to be dry, and at times boring, so to keep things interesting, practical examples are given wherever possible and relevant. You, the reader, can follow and implement these examples on your machine, which can be as simple as a Linux virtual machine with Python Version 3.x installed, and free tools to work with APIs, such as Postman and cURL. This book makes heavy use of the Cisco DevNet sandboxes, so in the majority of cases, you do not need a home lab to test and experiment with physical equipment.

A whole section of the book is dedicated to putting the knowledge and skills learned throughout the book to good use. One chapter covers programming Cisco platforms and another covers programming non-Cisco platforms. A third chapter in that same section is dedicated exclusively to Ansible. This book provides an abundance of hands-on practice.

The last chapter provides a way forward, discussing tools and technologies that you might want to explore after you are done with this book.

Who This Book Is For

This book is meant for the following individuals and roles, among others:

- Network architects and engineers who want to integrate programmability into their network designs
- NOC engineers monitoring and operating programmable networks or those who rely on network management systems that utilize programmability protocols

- Network engineers designing, implementing, and deploying new network services
- Software engineers or programmers developing applications for network management systems
- Network and software engineers working with networks or systems involving SDN, NFV, or cloud technologies
- Network engineers pursuing their Cisco DevNet certifications

Whether you are an expert network engineer with no prior programming experience or knowledge, or a software engineer looking to utilize your expertise in the network automation domain, after reading this book, you will fully understand the most commonly used protocols, tools, technologies, and techniques related to the subject, and you will be capable of effectively using the newly learned material to design, implement, and operate full-fledged programmable networks and the associated network automation systems.

How This Book Is Organized

This book covers the information you need to transition from having a focus on networking technology to focusing on software and network programmability. This book covers six main focus areas:

- Operating systems: Linux
- Software development: Python
- Transport: HTTP, REST, and SSH
- Encoding: XML, JSON, and YAML
- Modeling: YANG
- Protocols: NETCONF, RESTCONF, gRPC, and service provider programmability
- Practical programmability: Cisco platforms, non-Cisco platforms, and Ansible

Each chapter in this book either explicitly covers one of these focus areas or prepares you for one of them. Special consideration has been given to the ordering of topics to minimize forward referencing. Following an introduction to the programmability landscape, Linux is covered first because to get anything done in network programmability, you will almost always find yourself working with Linux. The book next covers Python because the vast majority of the rest of the book includes coverage of Python in the context of working with various protocols. The following chapters present an organic flow of topics: transport, encoding, modeling, and the protocols that build on all the previous sections. For example, understanding NETCONF requires you to understand SSH, XML, and YANG, and understanding RESTCONF requires that you understand HTTP, XML/JSON, and YANG. Both NETCONF and RESTCONF require knowledge of Python, most likely running on a Linux machine.

How This Book Is Structured

The book is organized into nine parts, described in the following sections.

PART I, "Introduction"

Chapter 1, "The Network Programmability and Automation Ecosystem": This chapter introduces the concepts and defines the terms that are necessary to understand the protocols and technologies covered in the following chapters. It also introduces the network programmability stack and explores the different components of the stack that constitute a typical network programmability and automation toolbox.

PART II, "Linux"

Chapter 2, "Linux Fundamentals": Linux is the predominant operating system used for running software for network programmability and automation. Linux is also the underlying operating system for the vast majority of network device software, such as IOS XR, NX-OS, and Cumulus Linux. Therefore, to be able to effectively work with programmable devices, it is of paramount importance to master the fundamentals of Linux. This chapter introduces Linux, including its architecture and boot process, and covers the basics of working with Linux through the Bash shell, such as working with files and directories, redirecting input and output, performing system maintenance, and installing software.

Chapter 3, "Linux Storage, Security, and Networks": This chapter builds on Chapter 2 and covers more advanced Linux topics. It starts with storage on Linux systems and the Linux Logical Volume Manager. It then covers Linux user, group, file, and system security. Finally, it explains three different methods to manage networking in Linux; the **ip** utility, the NetworkManager service, and network configuration files.

Chapter 4, "Linux Scripting": This chapter builds on Chapters 2 and 3 and covers Linux scripting using the Bash shell. The chapter introduces the **grep**, **awk**, and **sed** utilities and covers the syntax and semantics of Bash scripting. The chapter covers comments, input and output, variables and arrays, expansion, operations and comparisons, how to execute system commands from a Bash script, conditional statements, loops, and functions. It also touches on the Expect programming language.

PART III, "Python"

Chapter 5, "Python Fundamentals": This chapter assumes no prior knowledge of programming and starts with an introduction to programming, covering some very important software and computer science concepts, including algorithms and object-oriented programming. It also discusses why programming is a foundational skill for learning network programmability and covers the fundamentals of the Python programming language,

including installing Python Version 3.x, executing Python programs, input and output, data types, data structures, operators, conditional statements, loops, and functions.

Chapter 6, "Python Applications": This chapter builds on Chapter 5 and covers the application of Python to different domains. The chapter illustrates the use of Python for creating web applications using Django and Flask, for network programmability using NAPALM and Nornir, and for orchestration and machine learning. The chapter also covers some very important tools and protocols used in software development in general, such as Git, containers, Docker and virtual environments.

PART IV, "Transport"

Chapter 7, "HTTP and REST": This is one of the most important chapters in this book. It introduces the HTTP protocol and the REST architectural framework, as well as the relationship between them. This chapter covers HTTP connections based on TCP. It also covers the anatomy of HTTP messages and dives into the details of HTTP request methods and response status codes. It also provides a comprehensive explanation of the most common header fields. The chapter discusses the syntax rules that govern the use of URIs and then walks through working with HTTP, using tools such as Postman, cURL, and Python libraries, such as the **requests** library.

Chapter 8, "Advanced HTTP": Building on Chapter 7, this chapter moves to more advanced HTTP topics, including HTTP authentication and how state can be maintained over HTTP connections by using cookies. This chapter provides a primer on cryptography for engineers who know nothing on the subject and builds on that to cover TLS, and HTTP over TLS (aka HTTPS). It also provides a glimpse into HTTP/2 and HTTP/3, and the enhancements introduced by these newer versions of HTTP.

Chapter 9, "SSH": Despite being a rather traditional protocol, SSH is still an integral component of the programmability stack. SSH is still one of the most widely used protocols, and having a firm understanding of the protocol is crucial. This chapter discusses the three sub-protocols that constitute SSH and cover the lifecycle of an SSH connection: the SSH Transport Layer Protocol, User Authentication Protocol, and Connection Protocol. It also discusses how to set up SSH on Linux systems as well as how to work with SSH on the three major network operating system: IOS XR, IOS XE, and NX-OS. Finally, it covers SFTP, which is a version of FTP based on SSH.

PART V, "Encoding"

Chapter 10, "XML": This chapter covers XML, the first of three encoding protocols covered in this book. XML is the oldest of the three protocols and is probably the most sophisticated. This chapter describes the general structure of an XML document as well as XML elements, attributes, comments, and namespaces. It also covers advanced XML topics such as creating document templates using DTD and XML-based schemas using XSD, and it compares the two. This chapter also covers XPath, XSLT, and working with XML using Python.

Chapter 11, "JSON": JSON is less sophisticated, newer, and more human-readable than XML, and it is therefore a little more popular that XML. This chapter covers JSON data formats and data types, as well as the general format of a JSON-encoded document. The chapter also covers JSON Schema Definition (JSD) for data validation and how JSD coexists with YANG.

Chapter 12, "YAML": YAML is frequently described as a superset of JSON. YAML is slightly more human-readable than JSON, but data encoded in YAML tends to be significantly lengthier than its JSON-encoded counterpart. YAML is a very popular encoding format and is required for effective use of tools such as Ansible. This chapter covers the differences between XML, JSON, and YAML and discusses the structure of a YAML document. It also explains collections, scalers, tags, and anchors. Finally, the chapter discusses working with YAML in Python.

PART VI, "Modeling"

Chapter 13, "YANG": At the heart of the new paradigm of network programmability is data modeling. This is a very important chapter that covers both generic modeling and the YANG modeling language. This chapter starts with a data modeling primer, explaining what a data model is and why it is important to have data models. Then it explains the structure of a data model. This chapter describes the different node types in YANG and their place in a data model hierarchy. It also delves into more advanced topics, such as augmentations and deviations in YANG. It describes the difference between open-standard and vendor-specific YANG models and where to get each type. Finally, the chapter covers a number of tools for working with YANG modules, including **pyang** and **pyangbind**.

PART VII, "Protocols"

Chapter 14, "NETCONF and RESTCONF": NETCONF was the first protocol developed to replace SNMP. RESTCONF was developed later and is commonly referred to as the RESTful version of NETCONF. Building on earlier chapters, this chapter takes a deep dive into both NETCONF and RESTCONF. The chapter covers the protocol architecture as well as the transport, message, operations, and content layers of each of the two protocols. It also covers working with these protocols using Python.

Chapter 15, "gRPC, Protobuf, and gNMI": The gRPC protocol was initially developed by Google for network programmability that borrows its operational concepts from the communications models of distributed applications. This chapter provides an overview of the motivation that drove the development of gRPC. It covers the communication flow of gRPC and protocol buffers (Protobuf) used to serialize data for gRPC communications. The chapter also shows how to work with gRPC using Python. The chapter then takes a deep dive into gNMI, a gRPC-based specification. Finally, the chapter shows how gRPC and gNMI are used to manage a Cisco IOS XE device.

Chapter 16, "Service Provider Programmability": Service providers face unique challenges due to the typical scale of their operations and the stringent KPIs that must be imposed on their networks, especially given the heated race to adopt 5G and associated technologies. This chapter discusses how such challenges influence the programmability and automation in service provider networks and provides in-depth coverage of Segment Routing, BGP-LS, and PCEP.

PART VIII, "Programmability Applications"

Chapter 17, "Programming Cisco Platforms": This chapter explores the programmability capabilities of several Cisco platforms, covering a wide range of technology domains. In addition, this chapter provides several practical examples and makes heavy use of Cisco's DevNet sandboxes. This chapter covers the programmability of IOS XE, IOS XR, NX-OS, Meraki, DNA Center, and Cisco's collaboration platforms, with a use case covering Webex Teams.

Chapter 18, "Programming Non-Cisco Platforms": This chapter covers the programmability of a number of non-Cisco platforms, such as the Cumulus Linux and Arista EOS platforms. This chapter shows that the knowledge and skills gained in the previous chapters are truly vendor neutral and global. In addition, this chapter shows that programmability using APIs does in fact abstract network configuration and management and breaks the dependency on vendor-specific CLIs.

Chapter 19, "Ansible": This chapter covers a very popular tool that has become synonymous with network automation: Ansible. As a matter of fact, Ansible is used in the application and compute automation domains as well. Ansible is a very simple, yet extremely powerful, automation tool that provides a not-so-steep learning curve, and hence a quick and effective entry point into network automation. This is quite a lengthy chapter that takes you from zero to hero in Ansible.

PART IX, "Looking Ahead"

Chapter 20, "Looking Ahead": This chapter builds on the foundation covered in the preceding chapters and discusses more advanced technologies and tools that you might want to explore to further your knowledge and skills related to network programmability and automation.

This page intentionally left blank

Chapter 3

Linux Storage, Security, and Networks

Chapter 2, “Linux Fundamentals,” covers Linux basics, and by now, you should be familiar with the Linux environment and feel comfortable performing general system maintenance tasks. This chapter takes you a step further in your Linux journey and covers storage, security, and networking.

Linux Storage

Many network engineers struggle with concepts such as what mounting a volume means and the relationship between physical and logical volumes. This section covers everything you need to know about storage to effectively manage a Linux-based environment, whether it is your development environment or the underlying Linux system on which a network operating system is based, such as IOS XR and NX-OS.

Physical Storage

The /dev directory contains device files, which are special files used to access the hardware on a system. A program trying to access a device uses a device file as an interface to the device driver of that device. Writing data to a device file is the same as sending data to the device represented by that device file, and reading data from a device file is the same as receiving data from that device. For example, writing data to the printer device file prints this data, and reading data from the device file of a hard disk partition is the same as reading data from that partition on the disk.

Example 3-1 shows the output of the **ls -l** command for the /dev directory. Notice that, unlike other directories, the first bit of the file permissions is one of five characters:

- **-** for regular files
- **d** for directories

- **l** for links
- **c** for character device files
- **b** for block device files

You learned about the first three of these bits in Chapter 2, and the other two are covered here.

Example 3-1 *Contents of the /dev Directory*

```
[netdev@server1 dev]$ ls -l
total 0
-rw-r--r--. 1 root    root           0 Aug 10 00:28 any_regular_file
crw-r--r--. 1 root    root     10, 235 Aug 10 00:19 autofs
drwxr-xr-x. 2 root    root         140 Aug 10 00:18 block
drwxr-xr-x. 2 root    root          60 Aug 10 00:18 bsg
drwxr-xr-x. 3 root    root          60 Aug 10 00:19 bus
drwxr-xr-x. 2 root    root        2940 Aug 10 00:20 char
drwxr-xr-x. 2 root    root          80 Aug 10 00:18 cl
crw-------. 1 root    root      5,   1 Aug 10 00:20 console
lrwxrwxrwx. 1 root    root          11 Aug 10 00:18 core -> /proc/kcore
drwxr-xr-x. 6 root    root         120 Aug 10 00:19 cpu
crw-------. 1 root    root     10,  62 Aug 10 00:19 cpu_dma_latency
drwxr-xr-x. 6 root    root         120 Aug 10 00:18 disk
brw-rw----. 1 root    disk    253,   0 Aug 10 00:19 dm-0
brw-rw----. 1 root    disk    253,   1 Aug 10 00:19 dm-1

--------- OUTPUT TRUNCATED FOR BREVITY ---------
```

Character device files provide *unbuffered* access to hardware. This means that what is written to the file is transmitted to the hardware device right away, byte by byte. The same applies to read operations. Think of data sent to the device file of an audio output device or data read from the device file representing your keyboard. This data should not be buffered.

On the other hand, block device files provide *buffered* access; that is, data written to a device file is buffered by the kernel before it is passed on to the hardware device. The same applies to read operations. Think of data written to or read from a partition on your hard disk. This is typically done in data blocks, not individual bytes.

However, note that the *device file* type (as seen in the /dev directory) is not necessarily the same as the *device* type. Storage devices such as hard disks are block devices, which means that data is read from and written to the device in fixed-size blocks. Although this may sound counterintuitive, block devices may be accessed using character device files on some operating systems, such as BSD. This is not the case with Linux, where block

devices are always associated with block device files. The difference between block devices and block device files is sometimes a source of confusion.

The first step in analyzing a storage and file system is getting to know the hard disks. Each hard disk and partition has a corresponding device file in the /dev directory. By listing the contents of this directory, you find the sda file for the first hard disk, and, if installed, sdb for the second hard disk, sdc for the third hard disk, and so forth. Partitions are named after the hard disk that the partition belongs to, with the partition number appended to the name. For example, the *first* partition on the *second* hard disk is named sdb1. The hard disk naming convention follows the configuration in the /lib/udev/rules.d/60-persistent-storage.rules file, and the configuration is per hard disk type (ATA, USB, SCSI, SATA, and so on). Example 3-2 lists the relevant files in the /dev directory on a CentOS 7 distro. As you can see, this system has two hard disks. The first hard disk is named sda and has two partitions—sda1 and sda2—and the second is named sdb and has three partitions—sdb1, sdb2, and sdb3.

Example 3-2 *Hard Disks and Partitions in the /dev Directory*

```
[root@localhost ~]# ls -l /dev | grep sd
brw-rw----. 1 root    disk      8,   0 Jun  8 04:55 sda
brw-rw----. 1 root    disk      8,   1 Jun  8 04:55 sda1
brw-rw----. 1 root    disk      8,   2 Jun  8 04:55 sda2
brw-rw----. 1 root    disk      8,  16 Jun  8 04:55 sdb
brw-rw----. 1 root    disk      8,  17 Jun  8 04:55 sdb1
brw-rw----. 1 root    disk      8,  18 Jun  8 04:55 sdb2
brw-rw----. 1 root    disk      8,  19 Jun  8 04:55 sdb3
```

Notice the letter b at the beginning of each line of the output in Example 3-2. This indicates a block device file. A character device file would have the letter c instead.

The command **fdisk -l** lists all the disks and partitions on a system, along with some useful details. Example 3-3 shows the output of this command for the same system as in Example 3-2.

Example 3-3 *Using the **fdisk -l** Command to Get Hard Disk and Partition Details*

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 26.8 GB, 26843545600 bytes, 52428800 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x000b4fba
```

```
   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *        2048     2099199     1048576   83  Linux
/dev/sda2         2099200    52428799    25164800   8e  Linux LVM

Disk /dev/sdb: 107.4 GB, 107374182400 bytes, 209715200 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x149c8964

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1            2048    41945087    20971520   83  Linux
/dev/sdb2        41945088    83888127    20971520   83  Linux
/dev/sdb3        83888128   115345407    15728640   83  Linux

Disk /dev/mapper/centos-root: 23.1 GB, 23081254912 bytes, 45080576 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk /dev/mapper/centos-swap: 2684 MB, 2684354560 bytes, 5242880 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
[root@localhost ~]#
```

In addition to physical disks /dev/sda and /dev/sdb and their respective partitions, the command output in Example 3-3 lists two other disks: /dev/mapper/centos-root and /dev/mapper/centos-swap. These are two logical volumes. (Logical volumes are discussed in detail in the next section.) Notice that there is an asterisk (*) under the title Boot for partition /dev/sda1. As you may have guessed, this indicates that this is the partition on which the boot sector resides, containing the boot loader. The boot loader is the software that will eventually load the kernel image into memory during the system boot process, as you have read in Section "The Linux Boot Process" in Chapter 2.

In addition to displaying existing partition details, **fdisk** can create new partitions and delete existing ones. For example, after a third hard disk, sdc, is added to the system, the **fdisk** utility can be used to create two partitions, sdc1 and sdc2, as shown in Example 3-4.

Example 3-4 *Creating New Hard Disk Partitions by Using the* ***fdisk*** *Utility*

```
! Current status of the sdc hard disk: no partitions exist
[root@localhost ~]# fdisk -l /dev/sdc

Disk /dev/sdc: 21.5 GB, 21474836480 bytes, 41943040 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

! Using fdisk to create two new partitions on sdc
[root@localhost ~]# fdisk /dev/sdc
Welcome to fdisk (util-linux 2.23.2).

Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table
Building a new DOS disklabel with disk identifier 0x4cd00767.

Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   g   create a new empty GPT partition table
   G   create an IRIX (SGI) partition table
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)

Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
```

```
Select (default p): p
Partition number (1-4, default 1):
First sector (2048-41943039, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-41943039, default 41943039): +5G
Partition 1 of type Linux and of size 5 GiB is set

Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p):
Using default response p
Partition number (2-4, default 2):
First sector (10487808-41943039, default 10487808):
Using default value 10487808
Last sector, +sectors or +size{K,M,G} (10487808-41943039, default 41943039):
Using default value 41943039
Partition 2 of type Linux and of size 15 GiB is set

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.

! Status after creating the two new partitions sdc1 and sdc2
[root@localhost ~]# fdisk -l /dev/sdc

Disk /dev/sdc: 21.5 GB, 21474836480 bytes, 41943040 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x4cd00767

   Device Boot      Start         End      Blocks   Id  System
/dev/sdc1            2048    10487807     5242880   83  Linux
/dev/sdc2        10487808    41943039    15727616   83  Linux3
[root@localhost ~]#
```

The interactive dialogue of the **fdisk** utility is self-explanatory. After the **fdisk /dev/sdc** command is issued, you can enter **m** to see all available options. You can enter **n** to start the new partition dialogue. Note the different methods to specify the size of the

partition. If you go with the default option (by simply pressing Enter), the command uses all the remaining space on the disk to create that particular partition.

Before a hard disk partition can be used to store data, the partition needs to be formatted; that is, a file system has to be created. (File systems are discussed in some detail in Chapter 2.) At the time of writing, the two most common file systems used on Linux are ext4 and xfs. A partition is formatted using the **mkfs** utility. In Example 3-5, the sdc1 partition is formatted to use the ext4 file system, and sdc2 is formatted to use the xfs file system.

Example 3-5 *Creating File Systems by Using the* ***mkfs*** *Command*

```
[root@localhost ~]# mkfs -t ext4 /dev/sdc1
mke2fs 1.42.9 (28-Dec-2013)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
327680 inodes, 1310720 blocks
65536 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=1342177280
40 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
      32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done

[root@localhost ~]# mkfs -t xfs /dev/sdc2
meta-data=/dev/sdc2              isize=512    agcount=4, agsize=982976 blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=1        finobt=0, sparse=0
data     =                       bsize=4096   blocks=3931904, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=1
log      =internal log           bsize=4096   blocks=2560, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
[root@localhost ~]#
```

To specify a file system type, you use **mkfs** with the **-t** option. Keep in mind that the command output depends on the file system type used with the command.

The final step toward making a partition usable is to mount that partition or file system. Mounting is usually an ambiguous concept to engineers who are new to Linux. As discussed in Chapter 2, the Linux file hierarchy always starts at the root directory, represented by /, and branches down. For a file system to be accessible, it has to be *mounted* to a *mount point*—that is, attached (mounted) to the file hierarchy at a specific path in that hierarchy (mount point). The mount point is the path in the file hierarchy that the file system is attached to and through which the contents of that file system can be accessed. For example, mounting the /dev/sdc1 partition to the /Operations directory maps the content of /dev/sdc1 to, and makes it accessible through, the /Operations directory, for both read and write operations. Example 3-6 shows the /Operations directory being created and the sdc1 partition being mounted to it.

Example 3-6 *Mounting /dev/sdc1 to /Operations*

```
[root@localhost ~]# mkdir /Operations
[root@localhost ~]# mount /dev/sdc1 /Operations
```

To display all the mounted file systems, you use the **df** command, as shown in Example 3-7. The option **-h** displays the file system sizes in human-readable format.

Example 3-7 *Output of the df -h Command*

```
[root@localhost ~]# df -h
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   22G  5.3G   17G  25% /
devtmpfs                 3.9G     0  3.9G   0% /dev
tmpfs                    3.9G     0  3.9G   0% /dev/shm
tmpfs                    3.9G  9.4M  3.9G   1% /run
tmpfs                    3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/sda1               1014M  333M  682M  33% /boot
tmpfs                    783M   32K  783M   1% /run/user/1000
/dev/sdc1                4.8G   20M  4.6G   1% /Operations
[root@localhost ~]#
```

Each row in the output in Example 3-7 is a separate file system. The entry /dev/mapper/centos-root is a logical volume (and is discussed in detail in the next section). The following few entries are tmpfs file systems, which are temporary file systems created in memory (not on disk) for cache-like operations due to the high speed of RAM, as compared to the low speed of hard disks. An entry exists in the list for partition /dev/sda1 that is mounted to directory /boot. Then the entry at the bottom is for /dev/sdc1 that was mounted to directory /Operations as shown in Example 3-6.

To unmount the /dev/sdc1 file system, you use the **umount /dev/sdc1** command. You can also use the mount point, in which case the command is **umount /Operations**. Note that the command is **umount**, not *unmount*. Adding the letter *n* is a very common error.

The mounting done by using the **mount** command is not persistent. In other words, once the system is rebooted, the volumes mounted using the **mount** command are no longer mounted. For persistent mounting, an entry needs to be added to the /etc/fstab file. Example 3-8 shows the contents of the /etc/fstab file after the entry for /dev/sdc1 is added.

Example 3-8 *Editing the /etc/fstab File for Persistent Mounting*

```
! Adding an entry for /etc/sdc1 using the echo command
[root@localhost ~]# echo "/dev/sdc1 /Operations    ext4 defaults 0 0" >> /etc/fstab

! After adding an entry for /etc/sdc1
[root@localhost ~]# cat /etc/fstab
#
# /etc/fstab
# Created by anaconda on Sat May 26 04:28:54 2018
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/centos-root /             xfs     defaults        0 0
UUID=dfe65618-19ab-458d-b5e3-dafdb59b4e68 /boot      xfs    defaults    0 0
/dev/mapper/centos-swap swap             swap    defaults        0 0
/dev/sdc1 /Operations            ext4    defaults    0 0

! Command mount -a immediately mounts all file systems listed in fstab
[root@localhost ~]# mount -a
```

The command **mount -a** immediately mounts all file systems listed in /etc/fstab.

The /etc/fstab file has one entry for each file system that is to be mounted at system boot. It is important to understand the entries in the /etc/fstab file because this is the file that defines what file systems a system will have mounted right after it boots and the options that each of these file systems will be mounted with. Each line has the following fields:

- The first field can be either the file system path, the universal unique identifier (UUID), or the label. You can learn the UUID (and type) of all file systems by using the command **blkid**. You can show the label by using the command **tune2fs -l** {*file_system*} for ext2/3/4 file systems or **xfs_admin -l** {*file_system*} for xfs file systems. Using the file system path, which is /dev/sdc1 in this case, is pretty straightforward. However, when a system has tens of hard disks installed, it would be wiser to use the partition UUIDs. A UUID is a unique number that identifies a partition. The UUID does not change if the hard disk containing the partition is moved to another system, and hence it is universal. Using a UUID eliminates the possibility of errors in the /etc/fstab file.

- The second field is the file system mount point, which is /Operations in this case.
- The third field is the file system type, which is ext4 in this example.
- The fourth field is the mounting options. In this example, defaults indicates that the default mounting options will be used. You can also add non-default mounting options such as acl for ACL support. You add options in a comma-separated list.
- The fifth field indicates which file systems are to be backed up by the **dump** utility. The zero value in this case indicates that this file system will not be automatically backed up.
- The sixth field is used by the **fsck** utility to determine whether to check the health of the file system. The **fsck** utility checks file systems with a nonzero value in this field, in order, starting with the file system that has the value one. A zero in this field tells the **fsck** utility not to check that file system.

fdisk is not the only Linux utility available to manipulate disk partitions. Two other popular utilities for disk partitioning are **gdisk** and **parted**. You can use the man pages for these utilities to explore them and use a non-production environment (ideally a virtual machine) to experiment with using them. You may run into a distro that has one of them implemented but not the other. The more utilities you are familiar with, the better.

Logical Volume Manager

Linux generally uses the concept of logical volumes to provide storage to users. Logical volumes abstract the storage that is available to a user from the actual physical disks. Logical volumes on Linux are managed by system software called *Logical Volume Manager* (*LVM*). LVM operates by grouping physical disks or disk partitions, each referred to as a *physical volume* (*PV*), such as /dev/sda or /dev/sdb2, into a *volume group* (*VG*). LVM then manages a VG as one pool of storage that is split by the LVM into one or more *logical volumes* (*LVs*). Figure 3-1 illustrates these concepts.

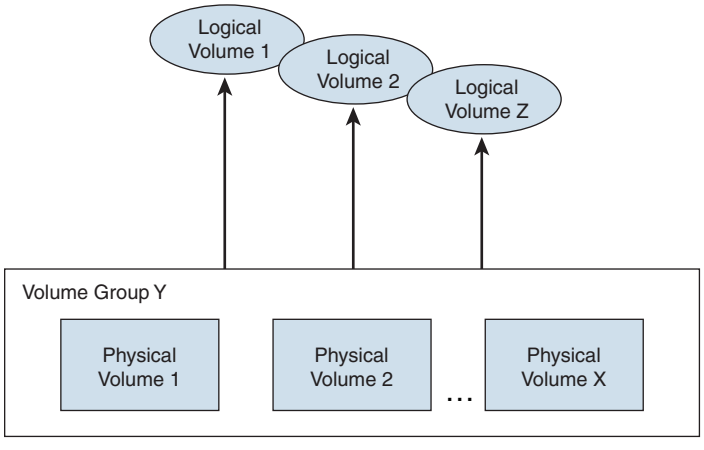


Figure 3-1 *Physical Volumes, Volume Groups, and Logical Volumes*

To better understand the concept of logical volumes, keep in mind the following:

- The different PVs that constitute a VG do not have to be equal in size.
- The different PVs that constitute a VG may be different disks, or different partitions on the same disk, or different partitions on different disks.
- Two different partitions on the *same* disk may be members of two different VGs.
- The LVs that are created from a VG do not correlate to the PVs that constitute the VG in either size or number.

Using LVs created by LVM provides several advantages over using physical storage directly. The most significant benefit is the disassociation between user data and specific physical storage volumes. From a capacity perspective, capacity can be added to and removed from a logical volume without having to repartition a physical disk to create a bigger or smaller partition, and a file system is not limited by the size of the physical disk that it resides on. From a performance perspective, data may be striped across several physical volumes (for added throughput) transparently from the user. These are just a few of the advantages.

The following steps are involved in creating a logical volume that is ready to use:

Step 1. Using the command **pvcreate** {*physical_disk/partition*}, label the physical volumes that will constitute the volume group as LVM physical volumes.

Step 2. Using the command **vgcreate** {*vg_name*} {*pv1*} {*pv2*} .. {*pvN*}, create the VG by using the physical volumes *pv1*, *pv2*,…*pvN*.

Step 3. Using the command **lvcreate -n** {*lv_name*} **-L** {*lv_size*} {*vg_name*}, create the logical volume named *lv_name* from the volume group named *vg_name*.

Step 4. Create the file system of choice on the new logical volume by using the **mkfs** command, exactly as you would on a physical partition.

Step 5. Mount the new file system by using the **mount** command exactly as you would mount a file system created on a physical partition.

In Example 3-9, two disks, sdb and sdc, are each divided into two partitions as follows:

- **sdb1**: 12 GB
- **sdb2**: 8 GB
- **sdc1**: 15 GB
- **sdc2**: 10 GB

Example 3-9 *The Four Partitions That Will Be Used to Create a Volume Group*

```
[root@server1 ~]# fdisk -l | grep -E sd[b,c]
Disk /dev/sdc: 26.8 GB, 26843545600 bytes, 52428800 sectors
/dev/sdc1             2048    31459327    15728640   83  Linux
/dev/sdc2         31459328    52428799    10484736   83  Linux
Disk /dev/sdb: 21.5 GB, 21474836480 bytes, 41943040 sectors
/dev/sdb1             2048    25167871    12582912   83  Linux
/dev/sdb2         25167872    41943039     8387584   83  Linux
[root@server1 ~]#
```

After each of the four partitions is labeled as a PV, all four partitions are added to the VG VGNetProg, which has a total capacity of 40 GB. The volume group capacity is then used to create two logical volumes—LVNetAutom with a capacity of 10 GB and LVNetDev with a capacity of 30 GB—as shown in Example 3-10.

Example 3-10 *Creating Physical Volumes, Volume Groups, and Logical Volumes*

```
! Label the physical volumes
[root@server1 ~]# pvcreate /dev/sdb1 /dev/sdb2 /dev/sdc1 /dev/sdc2
  Physical volume "/dev/sdb1" successfully created.
  Physical volume "/dev/sdb2" successfully created.
  Physical volume "/dev/sdc1" successfully created.
  Physical volume "/dev/sdc2" successfully created.

! Create the volume group
[root@server1 ~]# vgcreate VGNetProg /dev/sdb1 /dev/sdb2 /dev/sdc1 /dev/sdc2
  Volume group "VGNetProg" successfully created

! Create the two logical volumes
[root@server1 ~]# lvcreate -n LVNetAutom -L 10G VGNetProg
  Logical volume "LVNetAutom" created.
[root@server1 ~]# lvcreate -n LVNetDev -L 30G VGNetProg
  Logical volume "LVNetDev" created.
[root@server1 ~]#
```

Example 3-11 shows the **pvdisplay** command being used to display the details of the physical volumes.

Example 3-11 *Displaying Physical Volume Details by Using the* ***pvdisplay*** *Command*

```
[root@server1 ~]# pvdisplay /dev/sdb1
  --- Physical volume ---
  PV Name               /dev/sdb1
  VG Name               VGNetProg
  PV Size               12.00 GiB / not usable 4.00 MiB
  Allocatable           yes
  PE Size               4.00 MiB
  Total PE              3071
  Free PE               511
  Allocated PE          2560
  PV UUID               dPYPj6-Wv1i-iX7H-3iH0-oCnE-OzkA-2LcJlx
[root@server1 ~]# pvdisplay /dev/sdb2
  --- Physical volume ---
  PV Name               /dev/sdb2
  VG Name               VGNetProg
  PV Size               <8.00 GiB / not usable 3.00 MiB
  Allocatable           yes
  PE Size               4.00 MiB
  Total PE              2047
  Free PE               765
  Allocated PE          1282
  PV UUID               ftOYQo-a19G-0Gs6-01ir-i6M5-Yj1N-TRREDR
[root@server1 ~]# pvdisplay /dev/sdc1
  --- Physical volume ---
  PV Name               /dev/sdc1
  VG Name               VGNetProg
  PV Size               15.00 GiB / not usable 4.00 MiB
  Allocatable           yes (but full)
  PE Size               4.00 MiB
  Total PE              3839
  Free PE               0
  Allocated PE          3839
  PV UUID               DYW0TD-vXGl-8Ssr-BCcy-SLQQ-mkfi-rvQFVd
[root@server1 ~]# pvdisplay /dev/sdc2
  --- Physical volume ---
  PV Name               /dev/sdc2
  VG Name               VGNetProg
  PV Size               <10.00 GiB / not usable 3.00 MiB
  Allocatable           yes (but full)
  PE Size               4.00 MiB
  Total PE              2559
  Free PE               0
  Allocated PE          2559
  PV UUID               n1snhx-aevL-X5ay-la43-ljlo-83uC-LIkIT7
[root@server1 ~]#
```

Example 3-12 shows the **vgdisplay** command being used to display the volume group that has been created.

Example 3-12 *Displaying Volume Group Details by Using the* ***vgdisplay*** *Command*

```
[root@server1 ~]# vgdisplay VGNetProg
  --- Volume group ---
  VG Name               VGNetProg
  System ID
  Format                lvm2
  Metadata Areas        4
  Metadata Sequence No  3
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               0
  Max PV                0
  Cur PV                4
  Act PV                4
  VG Size               44.98 GiB
  PE Size               4.00 MiB
  Total PE              11516
  Alloc PE / Size       10240 / 40.00 GiB
  Free  PE / Size       1276 / 4.98 GiB
  VG UUID               PSi3RJ-9lkc-lZFE-oCVA-RaXC-HDh5-K0VuV3
[root@server1 ~]#
```

Example 3-13 shows the **lvdisplay** command being used to display the logical volumes that have been created. A logical volume is addressed using its full path in the /dev directory, as shown in the example.

Example 3-13 *Displaying Logical Volume Details by Using the* ***lvdisplay*** *Command*

```
[root@server1 ~]# lvdisplay /dev/VGNetProg/LVNetAutom
  --- Logical volume ---
  LV Path                /dev/VGNetProg/LVNetAutom
  LV Name                LVNetAutom
  VG Name                VGNetProg
  LV UUID                Y09QdN-J8Fw-s3Nb-RB84-bBPs-1USv-tzMfAw
  LV Write Access        read/write
  LV Creation host, time server1, 2018-08-05 21:57:42 +0300
  LV Status              available
  # open                 0
  LV Size                10.00 GiB
```

```
  Current LE             2560
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     8192
  Block device           253:2
[root@server1 ~]# lvdisplay /dev/VGNetProg/LVNetDev
  --- Logical volume ---
  LV Path                /dev/VGNetProg/LVNetDev
  LV Name                LVNetDev
  VG Name                VGNetProg
  LV UUID                Z9VRTv-CUe6-uSa8-S821-jGY5-ymKh-zsKfHZ
  LV Write Access        read/write
  LV Creation host, time server1, 2018-08-05 21:58:17 +0300
  LV Status              available
  # open                 0
  LV Size                30.00 GiB
  Current LE             7680
  Segments               3
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     8192
  Block device           253:3
[root@server1 ~]#
```

Note that you can issue the **pvdisplay**, **vgdisplay**, and **lvdisplay** commands without any arguments to display all physical volumes, all volume groups, and all logical volumes, respectively, that are configured on the system.

To delete a physical volume, volume group, or logical volume, you use the commands **pvremove**, **vgremove**, or **lvremove**, respectively.

After logical volumes are created, you use the **mkfs** command to format the LVNetAutom LV as an ext4 file system and the LVNetDev LV as an xfs file system, as shown in Example 3-14.

Example 3-14 *Creating File Systems on the new Logical Volumes by Using the* ***mkfs*** *Command*

```
[root@server1 ~]# mkfs -t ext4 /dev/VGNetProg/LVNetAutom
mke2fs 1.42.9 (28-Dec-2013)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
```

```
655360 inodes, 2621440 blocks
131072 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=2151677952
80 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
[root@server1 ~]# mkfs -t xfs /dev/VGNetProg/LVNetDev
meta-data=/dev/VGNetProg/LVNetDev isize=512    agcount=4, agsize=1966080 blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=1        finobt=0, sparse=0
data     =                       bsize=4096   blocks=7864320, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=1
log      =internal log           bsize=4096   blocks=3840, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
[root@server1 ~]#
```

Finally, in Example 3-15, both logical volumes are mounted, which means they are usable for storing and retrieving data.

Example 3-15 *Mounting Both Logical Volumes by Using the* **mount** *Command*

```
[root@server1 ~]# mkdir /Automation
[root@server1 ~]# mkdir /Development
[root@server1 ~]# ls /
Automation  dev          hd3   lib64  opt          root  srv  usr
bin         Development  home  media  proc         run   sys  var
boot        etc          lib   mnt    Programming  sbin  tmp

[root@server1 ~]# mount /dev/VGNetProg/LVNetAutom /Automation
[root@server1 ~]# mount /dev/VGNetProg/LVNetDev /Development/
[root@server1 ~]# df -h
Filesystem                      Size  Used Avail Use% Mounted on
/dev/mapper/centos-root          44G  6.7G   38G  16% /
devtmpfs                        3.9G     0  3.9G   0% /dev
```

```
tmpfs                                3.9G     0  3.9G   0% /dev/shm
tmpfs                                3.9G  8.8M  3.9G   1% /run
tmpfs                                3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/sda1                           1014M  233M  782M  23% /boot
tmpfs                                783M   20K  783M   1% /run/user/1001
/dev/mapper/VGNetProg-LVNetAutom     9.8G   37M  9.2G   1% /Automation
/dev/mapper/VGNetProg-LVNetDev        30G   33M   30G   1% /Development
[root@server1 ~]#
```

Of course, the mounting done in Example 3-15 is not persistent. To mount both logical volumes during system boot, two entries need to be added to the /etc/fstab file—one entry for each LV.

You may have noticed in the output of the **df -h** command in Example 3-15 that each LV appears as a subdirectory to the directory /dev/mapper. The *device mapper* is a kernel space driver that provides the generic function of creating mappings between different storage volumes. The term *generic* is used here because the mapper is not particularly aware of the constructs used by LVM to implement logical volumes. LVM uses the device mapper to create the mappings between a volume group and its constituent logical volumes, without the device mapper explicitly knowing that the latter is a logical volume (rather than a physical one).

The examples in this section show only the very basic functionality of LVM—that is, creating the basic building blocks for having and using logical volumes on a system. However, the real power of LVM becomes clear when you use advanced features such as increasing or decreasing the size of a logical volume, without having to delete the volume and re-create it, or the several options for high availability of logical volumes. Red Hat has a 147-page document titled "Logical Volume Manager Administration" on managing logical volumes. You can check out the document for RHEL 8 at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_logical_volumes/index.

Linux Security

Linux security is a massive and complex topic so it is important to establish the intended scope of this section early on. The purpose of this section is two-fold. The first purpose is to familiarize you with basic Linux security operations that would enable you to effectively manage your development environment without being stumped. For example, you can't execute a script unless your user on the system has the privileges to execute that script, based on the script's file permissions and your group memberships. The second purpose of this section is to show you how to accomplish a minimal level of hardening for your development environment. Using an unsecured device to run scripts that access network devices—and possibly push configuration to those devices—is not a wise thing to do. Accordingly, this section covers user, group, file, and directory security, including access control lists. This chapter also covers the Linux firewall.

User and Group Management

Linux is a multiuser operating system, which means that more than one user can access a single Linux system at a time.

For a user to access a Linux system, the user's account must be configured on the system. The user will then have a username and user ID (UID). A user with an account on the system is a member of one or more groups. Each group has a group name and a group ID (GID). By default, when a user is created on the system, a new group is also created; it has the same name as the username, and this becomes the primary group of the user. A user typically has a password, and each group also has a password.

Each user has a *home* directory that contains that user's files. One way that Linux maintains user segregation and security is by maintaining permissions on files and directories and allowing users with the appropriate authorization level to set those permissions. File permissions are classified into permissions for the owner of the file, the group of the file, and everyone else. The root user and any other user with root privileges can access all resources on the system, including other users' files and directories. The root user and users with root privileges are members of a group named wheel.

You can find user information by using the command **id** {*username*}, as shown in Example 3-16 for user NetProg.

Example 3-16 *Getting User Information by Using the* **id** *Command*

```
[root@localhost ~]# id NetProg
uid=1001(NetProg) gid=1002(NetProg) groups=1002(NetProg),10(wheel)
[root@localhost ~]#
```

User NetProg's UID is 1001. The output in Example 3-16 shows that the user's default (primary) group has the same name as the username. User NetProg in the example is also a member of the wheel group and therefore has root privileges that can be invoked by using the **sudo** {*command*} command, where *command* requires root privileges to be executed. The number to the left of each group name is the group ID.

User information is also stored in the /etc/passwd file, and group information is stored in the /etc/group file. Hashed user passwords are stored in the file /etc/shadow, and hashed group passwords are stored in the file /etc/gshadow. Example 3-17 displays the last five entries of each of the files.

Example 3-17 *Last Five Entries from the /etc/passwd, /etc/group, /etc/shadow, and /etc/gshadow Files*

```
! Sample entries from the /etc/passwd file
[netdev@server1 ~]$ tail -n 5 /etc/passwd
netdev:x:1000:1000:Network Developer:/home/netdev:/bin/bash
vboxadd:x:970:1::/var/run/vboxadd:/bin/false
cockpit-wsinstance:x:969:969:User for cockpit-ws instances:/nonexisting:/sbin/
  nologin
```

```
flatpak:x:968:968:User for flatpak system helper:/:/sbin/nologin
rngd:x:967:967:Random Number Generator Daemon:/var/lib/rngd:/sbin/nologin
[netdev@server1 ~]$

! Sample entries from the /etc/group file
[netdev@server1 ~]$ tail -n 5 /etc/group
netdev:x:1000:
vboxsf:x:970:
cockpit-wsinstance:x:969:
flatpak:x:968:
rngd:x:967
[netdev@server1 ~]$

! file /etc/shadow requires root privileges to be read
[netdev@server1 ~]$ tail -n 5 /etc/shadow
tail: cannot open '/etc/shadow' for reading: Permission denied
[netdev@server1 ~]$

! Sample entries from the /etc/shadow file
[netdev@server1 ~]$ sudo tail -n 5 /etc/shadow
[sudo] password for netdev:
netdev:$6$.JUG9NvdC/NzqiYq$zpCkMR3eENFgk906PjFVLJ526qFRI9L2n13rFApiyPS0lgb2F1CTjJvc1
  dqvvE3XV91q2fK.p3hvlEYtKciD2.:18489:0:99999:7:::
vboxadd:!!:18473::::::
cockpit-wsinstance:!!:18473::::::
flatpak:!!:18473::::::
rngd:!!:18473::::::
[netdev@server1 ~]$

! file /etc/gshadow requires root privileges to be read
[netdev@server1 ~]$ tail -n 5 /etc/gshadow
tail: cannot open '/etc/gshadow' for reading: Permission denied
[netdev@server1 ~]$

! Sample entries from the /etc/gshadow file
[netdev@server1 ~]$ sudo tail -n 5 /etc/gshadow
netdev:!::
vboxsf:!::
cockpit-wsinstance:!::
flatpak:!::
rngd:!::
[netdev@server1 ~]$
```

Each line in the /etc/passwd file is a record containing the information for one user account. Each record is formatted as follows: *username*:x:*user_id*:*primary_group_id*:*user_extra_information*:*user_home_directory*:*user_default_shell*.

The /etc/passwd and /etc/group files can be read by any user on the system but can only be edited by a user with root privileges. For this reason, as a security measure, the second field in the record, which historically contained the user password hash, now shows only the letter x. The user password hashes are now maintained in the /etc/shadow file, which can only be read by users with root privileges. The same arrangement is true for the /etc/group and the /etc/gshadow files. Whenever a user does not have a password, the x is omitted. Two consecutive colons in any record indicate missing information for the respective field.

Each line in the /etc/group file is a record containing information for one group. Each record is formatted as follows: *groupname*:x:*group_id*:*group_members*. The last field is a comma-separated list of non-default users in the group. For example, the record for the netdev group shows all users who are members of the group netdev except the user netdev itself.

Each line in the /etc/shadow file is a record containing the password information for one user. Each record is formatted as follows: *username*:*password_hash*:*last_changed*:*min*:*max*:*warn*:*expired*:*disabled*:*reserved*.

The field *last_changed* is the number of days between January 1, 1970, and the date the password was last changed. The field *min* is the minimum number of days to wait before the password can be changed. The value 0 indicates that it may be changed at any time. The field *max* is the number of days after which the password must be changed. The value 99999 means that the user can keep the same password practically forever. The field *warn* is the number of days to send a warning to the user prior to the password expiring. The field *expired* is the number of days after the password expires before the account should be disabled. The field *disabled* is the number of days since January 1, 1970, that an account has been disabled. The last field is reserved.

Finally, each line in the /etc/gshadow file is a record that contains the password information for one group. Each record is formatted as follows: *groupname*:*group_password_hash*:*group_admins*:*group_members*. The *group_password_hash* field contains an exclamation symbol (**!**) if no user is allowed to access the group by using the **newgrp** command. (This command is covered later in this section.)

You use the command **useradd** {*username*} to create a new user, and the command **passwd** {*username*} to set or change the password for a user. After switching to user root by using the **su** command in Example 3-18, the **id NetDev** command is used to verify that user NetDev does not already exist. The new user NetDev is then created by issuing the command **useradd NetDev**.

Next, the example shows the **su** command being used to attempt to log in as user NetDev. Notice that although a password was requested, no password will actually work. This is because, by default, when a new user is created, a password entry is created in

the /etc/shadow file, but until this password is actually set by using the **passwd** command, you cannot log in as the user because the *default* password hash in the shadow file is an invalid hash. The example shows the password being removed altogether with the command **passwd -d NetDev**. Only at this point are you able to log in without getting a password prompt. The password is then set using the command **passwd NetDev**, and a warning is displayed because the password entered was **Cisco123**. Once the password is set, it is possible to log in as the user in question. Note that creating a user also creates a home directory—in this case /home/NetDev—as shown in the output of the **pwd** command. The files /etc/passwd, /etc/group, and /etc/shadow are also updated to reflect the new user details, as shown in the example.

Example 3-18 *Creating a New User and Setting the Password*

```
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:26:29 +03 2018 on pts/1
[root@localhost ~]#

! Verify whether the user NetDev exists
[root@localhost ~]# id NetDev
id: NetDev: no such user
[root@localhost ~]#

! Add user NetDev and log in to it
[root@localhost ~]# useradd NetDev
[root@localhost ~]# exit
Logout
[NetProg@localhost ~]$

! Authentication will fail due to invalid "default" hash
[NetProg@localhost ~]$ su NetDev
Password:
su: Authentication failure
[NetProg@localhost ~]$

! Switch back to user root and remove the password
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:27:07 +03 2018 on pts/1
[root@localhost ~]# passwd -d NetDev
Removing password for user NetDev.
passwd: Success
[root@localhost ~]# exit
logout
```

```
[NetProg@localhost ~]$ su NetDev
[NetDev@localhost NetProg]$ exit
Exit
[NetProg@localhost ~]$

! Switch to user root and set the password manually then test
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:28:12 +03 2018 on pts/1
[root@localhost ~]# passwd NetDev
Changing password for user NetDev.
New password:
BAD PASSWORD: The password fails the dictionary check - it is based on a dictionary
  word
Retype new password:
passwd: all authentication tokens updated successfully.
[root@localhost ~]# exit
logout
[NetProg@localhost ~]$ su NetDev
Password:
[NetDev@localhost NetProg]$

! Check the home directory and other details for user NetDev
[NetDev@localhost NetProg]$ cd
[NetDev@localhost ~]$ pwd
/home/NetDev
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=1003(NetDev) groups=1003(NetDev)
[NetDev@localhost ~]$ tail -n 1 /etc/passwd
NetDev:x:1002:1003::/home/NetDev:/bin/bash
[NetDev@localhost ~]$ tail -n 1 /etc/group
NetDev:x:1003:
[NetDev@localhost ~]$

! Switch to user root and check file /etc/shadow
[NetDev@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:50:37 +03 2018 on pts/0
[root@localhost ~]# tail -n 1 /etc/shadow
NetDev:$6$y27JA0id$i8Wze1ShSptxy5wRS8f7fOkPeeAezo2cayDl/
  sqikRkYp2VseEXNrzwqDQXqvMeAqzMs2Jd./jj5fm05PK.Wi/:17636:0:99999:7:::
[root@localhost ~]# exit
logout
[NetDev@localhost ~]$
```

A user can change her own password by simply typing **passwd** without any arguments. The user is then prompted to enter the current password and then the new password and then to confirm the new password.

To delete a user, you use the command **userdel** {*username*}. This command deletes the user from the system; to delete that user's home directory and print spool as well, you use the option **-r** with the command. You use the option **-f** to force the delete action even if the user is still logged in.

You can add groups separately from users by using the command **groupadd** {*group_name*}. You can use the option **-g** to set the GID manually instead of allowing automatic assignment of the next available GID. You delete groups by using the command **groupdel** {*group_name*}. Example 3-19 shows how to create a new group called engineers and set its GID to 1111.

Example 3-19 *Creating a New Group engineers*

```
[root@localhost ~]# tail -n 2 /etc/group
NetProg:x:1002:
NetDev:x:1003:
[root@localhost ~]# groupadd -g 1111 engineers
[root@localhost ~]# tail -n 3 /etc/group
NetProg:x:1002:
NetDev:x:1003:
engineers:x:1111:
[root@localhost ~]#
```

To delete a group, you use the command **groupdel {***group_name***}**. You change a group's details by using the command **groupmod**. The command **groupmod -g** {*new_gid*} {*group_name*} changes the group gid to *new_gid*, and the command **groupmod -n** {*new_name*} {*old_name*} changes the group's name from *old_name* to *new_name*. Finally, you change the group password by using the command **gpasswd** {*group_name*}. In Example 3-20, the group engineers is changed to NetDevOps, and its GID is changed to 2222. Then its password is modified to Cisco123.

Example 3-20 *Modifying Group Details*

```
[root@localhost ~]# tail -n -1 /etc/group
engineers:x:1111:
[root@localhost ~]#

! Change the group name to NetDevOps
[root@localhost ~]# groupmod -n NetDevOps engineers
[root@localhost ~]# tail -n -1 /etc/group
NetDevOps:x:1111:
[root@localhost ~]#
```

```
! Change the gid to 2222
[root@localhost ~]# groupmod -g 2222 NetDevOps
[root@localhost ~]# tail -n -1 /etc/group
NetDevOps:x:2222:
[root@localhost ~]#

! Change the group password to Cisco123
[root@localhost ~]# gpasswd NetDevOps
Changing the password for group NetDevOps
New Password:
Re-enter new password:
[root@localhost ~]#
```

A user has one primary group and one or more secondary groups. A user's primary group is the group that the user is placed in when logging in. You modify user group membership by using the command **usermod**. To change a user's primary group, you use the syntax **usermod -g** {*primary_group*} {*username*}. To change a user's secondary group, you use the syntax **usermod -G** {*secondary_group*} {*username*}; note that this command removes all secondary group memberships for this user and adds the group *secondary_group* specified in the command. To add a user to a secondary group while maintaining his current group memberships, you use the syntax **usermod -aG** {*new_secondary_group*} {*username*}. To lock a user account, you use the option **-L** with the **usermod** command, and to unlock an account, you use the **-U** option with this command. Example 3-21 shows how to change the primary group of user NetDev from NetDev to NetOps and add the wheel group to the list of secondary groups to give the user root privileges through the **sudo** command.

Example 3-21 *Modifying User Details*

```
[root@localhost ~]# id NetDev
uid=1002(NetDev) gid=1003(NetDev) groups=1003(NetDev)
[root@localhost ~]# usermod -g NetOps NetDev
[root@localhost ~]# id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps)
[root@localhost ~]# usermod -aG wheel NetDev
[root@localhost ~]# id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)
[root@localhost ~]#
```

Notice that when the **-g** option is used to change the primary group, the secondary group is also changed. This is because user NetDev was only a member of a single group, NetDev, and that group was both the user's primary group and secondary group. When the primary and secondary groups are different, the **-g** option changes only the primary group of the user.

File Security Management

Chapter 2 describes the output of the **ls -l** command and introduces file permissions, also known as the file mode bits. This section builds on that introduction and expands on how to manage access to files and directories by modifying their permissions. It also discusses changing the file owner (user) and group. Keep in mind that in Linux, everything is represented by a file. Therefore, the concepts discussed here have a wider scope than what seems to be obvious. Also, whenever a reference is made to a file, the same concept applies to a directory, unless explicitly stated otherwise.

Example 3-22 shows the output of **ls -l** for the NetProg home directory.

Example 3-22 *Output of the ls -l Command*

```
[NetProg@localhost ~]$ ls -l
total 0
drwxr-xr-x. 2 NetProg NetProg  40 Apr  9 09:41 Desktop
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Documents
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Downloads
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Music
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Pictures
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Public
drwxrwxr-x. 2 NetProg NetProg 183 Apr  7 22:53 Scripts
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Templates
-rw-rw-r--. 1 NetProg NetProg   0 Apr  9 17:51 Testfile.txt
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Videos
[NetProg@localhost ~]$
```

Here is a quick recap on the file permissions: The very first bit indicates whether this is a file (-), a directory (d), or a link (l). Then the following 3 bits define the permissions for the file owner. By default, the owner is the user who created the file. The following 3 bits define the permissions for the users who are members of the file group. By default, this is the primary group of the user who created the file. The last 3 bits define the permissions for everyone else, referred to as *other*. The letter r stands for read permission, w for write permission, and x for execute permission.

The dot right after the mode bits indicates that this file has an SELinux context. SELinux is a kernel security module that defines the access rights of every user, application, process, and file on the system. SELinux then governs the interactions of these entities using a security policy, where an entity referred to as a subject attempts to access another entity referred to as an object. SELinux is an important component of Linux security but is beyond the scope of this book. When a file or a directory has a + symbol in place of the dot (.), it means the file has an access control list (ACL) applied to it. ACLs, which are covered later in this chapter, provide more granular access control to files on a per-user basis.

The output of the **ls -l** command also displays the file owner (more formally referred to as user) and the file group.

File permissions can be represented (and modified) by either using *symbolic* notation or *octal* notation.

Symbolic notation is the type of notation described so far, where user, group, and others are represented by u, g, and o, respectively, and the access permissions are write, read, and execute, represented by w, r, and x, respectively. The following syntax is used to set the file permissions: **chmod** [**u=**{*permissions*}][**,g=**{*permissions*}][**,o=**{*permissions*}] {*file_name*}.

Example 3-23 shows how to modify the file permissions for file TestFile.txt to the following:

- **User:** Read, write, and execute
- **Group:** Read and write
- **Other:** No access

Example 3-23 *Setting File Permissions by Using Symbolic Notation*

```
! Current file permissions
[NetProg@localhost ~]$ ls -l Testfile.txt
-rw-rw-r--. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$

! Change the file permissions as listed
[NetProg@localhost ~]$ chmod u=rwx,g=rw,o= Testfile.txt

! New file permissions
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxrw----. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$
```

Notice that in order to remove all permissions for one of the categories, you just leave the right side of the **=** symbol blank.

One of the challenges with the symbolic notation syntax as used in Example 3-23 is that you have to know beforehand what permissions the file already has and make sure to align the current permissions with the new permissions you are trying to set. For example, if a file already has read and write permissions set for the file group and you would like to add the execute permission, you have to know this fact prior to the change, and then you need to make sure you do not delete the already existing write or read permissions while setting the execute permission. In order to just add or remove permissions for

a specific category, without explicitly knowing or considering the existing permissions, you replace the **=** symbol in the previous syntax with either a **+** or a **-** symbol, as follows: **chmod** [**u**[**+**|**-**]{*permissions*}][,**g**[**+**|**-**]{*permissions*}][,**o**[**+**|**-**]{*permissions*}] {*file_name*}.

In Example 3-24 the permissions for the file TestFile.txt are modified as follows:

- **User**: Unchanged
- **Group**: Write permission removed and execute permission added
- **Other**: Execute permission added

Notice that when using this syntax, you do not need to know what permissions the file already has. You only need to consider the changes that you want to implement.

Example 3-24 *Adding and Removing File Permissions by Using Symbolic Notation*

```
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxrw----. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$ chmod g-w,g+x,o+x Testfile.txt
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxr-x--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$
```

Notice that you can mix the **+** and **-** symbols in the same command and for the same category, as shown in Example 3-24 for the file group, where **g-w** is used to remove the write permission for the group, and **g+x** is used to add the execute permission for the group.

When a certain permission has to be granted or revoked from all categories, the letter a is used to collectively mean u, g, and o. The letter a in this case stands for *all*. The letter a may be dropped altogether, and the command then applies to all categories. For example, the command **chmod +w Example.py** adds the write permission to all categories for the file Example.py.

Octal notation, on the other hand, uses the following syntax: **chmod** {*user_permission*} {*group_permission*}{*other_permission*} {*file_name*}. The *user*, *group*, and *other* categories are represented by their positions in the command. The permission granted to each category is represented as a numeric value that is equal to the summation of each permission's individual value. To elaborate, note the following permission values:

- Read=4
- Write=2
- Execute=1

To set the read permission only, you need to use the value 4; for write permission only, you use the value 2; and for execute permission only, you use the value 1. To set all permissions for any category, you need to use 4+2+1=7. To set the read and write permissions only, you need to use 4+2=6, and so forth. Example 3-25 illustrates this concept and uses octal notation to set the read, write, and execute permissions for both user and group, and set only the execute permission for the category other for file Testfile.txt.

Example 3-25 *Setting File Permissions Using Octal Notation*

```
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxr-x--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$ chmod 771 Testfile.txt
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxrwx--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$
```

The number **7** in each of the first two positions in the command **chmod 771 Testfile.txt** represents the sum of 4, 2, and 1 and is used to set all permissions for user and group. The number **1** in the last position sets the execute only permission for other.

While octal notation looks snappier than symbolic notation, it does not provide the option of adding or removing permissions without considering the existing file permissions, as provided by the **+** and **-** symbols used with symbolic notation.

Besides modifying file and directory permissions, you can control access to a file or directory by changing the file's user and/or group through the **chown** command. The command syntax is **chown** {*user*}:{*group*} {*file*}. Example 3-26 shows how to change the *user* and *group* of file TestFile.txt to NetDev and networks, respectively.

Example 3-26 *Changing File User and Group by Using the* ***chown*** *Command*

```
[root@localhost ~]# ls -l /home/NetProg/Testfile.txt
-rwxrwx--x. 1 NetProg NetProg 0 Apr  9 17:51 /home/NetProg/Testfile.txt
[root@localhost ~]# chown NetDev:networks /home/NetProg/Testfile.txt
[root@localhost ~]# ls -l /home/NetProg/Testfile.txt
-rwxrwx--x. 1 NetDev networks 0 Apr  9 17:51 /home/NetProg/Testfile.txt
[root@localhost ~]#
```

You use the **-R** option (which stands for *recursive*) with both the **chmod** and the **chown** commands if the operation is being performed on a directory, and you want the changes to also be made to all subdirectories and files in that directory.

By default, any file or directory created by a user is assigned to the primary group of that user. For example, if user NetDev is in the NetOps group, any file created by user NetDev has NetDev as the file user and NetOps as the file group. You can change this

default behavior by either using the **sg** command when creating the file or by logging in to another group by using the command **newgrp**. If that other group is one of the user's secondary groups, no password is required. If that other group is not one of the user's secondary groups, the user is prompted for a password.

Example 3-27 shows the default behavior when creating a file. In this case, a new file named NewFile is created by user NetDev. As expected, the file user is NetDev, and the file group is NetOps.

Example 3-27 *Default User and Group of a Newly Created File*

```
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)
[NetDev@localhost ~]$ touch NewFile
[NetDev@localhost ~]$ ls -l NewFile
-rw-r--r--. 1 NetDev NetOps 0 Apr 17 00:59 NewFile
[NetDev@localhost ~]$
```

Example 3-28 shows how to use the **sg** command to create file NewFile_1 but under the group networks.

Example 3-28 *Using the sg Command to Create a File Under a Different Group*

```
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)
[NetDev@localhost ~]$ sg networks 'touch NewFile_1'
Password:
[NetDev@localhost ~]$ ls -l NewFile_1
-rw-r--r--. 1 NetDev networks 0 Apr 17 01:03 NewFile_1
[NetDev@localhost ~]$
```

Notice that the command **touch** {*file_name*}, which itself is an argument to the **sg** command, has to be enclosed in quotes because it is a multi-word command. Notice also that because the user NetDev is not a member in the networks group, as you can see from the output of the **id** command, the user is prompted for the group password, which was set earlier by using the command **gpasswd networks**.

Alternatively, the user can log in to another group by using the command **newgrp** and create a file or directory under that group. Example 3-29 shows the user NetProg logging in to group systems and not being prompted for a password since this is one of NetProg's secondary groups. When the file NewFile_2 is created, the user of the file is NetProg, and the group is systems, not NetProg.

Example 3-29 *Using the* ***newgrp*** *Command to Log In to a Different Group*

```
[NetProg@localhost ~]$ id NetProg
uid=1001(NetProg) gid=1002(NetProg) groups=1002(NetProg),10(wheel),2224(systems)
[NetProg@localhost ~]$ newgrp systems
[NetProg@localhost ~]$ touch NewFile_2
[NetProg@localhost ~]$ ls -l NewFile_2
-rw-r--r--. 1 NetProg systems 0 Apr 17 01:15 NewFile_2
[NetProg@localhost ~]$
```

Access Control Lists

So far in this chapter, you have seen how to set file and directory access permissions for either user, or collectively for group, or other. What if you want to set those permissions individually for a specific user who is not the file owner or for a group of users who belong to a group other than the file group? File mode bits do not help in such situations. Using the file mode bits, the only user whose permissions can be changed individually is the file or directory owner (user) and the only group of users whose permissions can be changed collectively are the users who are members of the file or directory group.

Access control lists (*ACLs*) provide more granular control over file and directory access. ACLs allow a system administrator (or any other user who has root privileges) to set file and directory permissions for any user or group on the system.

Before you can configure ACLs, three prerequisites must be met:

- The kernel must support ACLs for the file system type on which ACLs will be applied.
- The file system on which ACLs will be used must be mounted with the ACL option.
- The ACL package must be installed.

Most common distros today—including CentOS 7 and Red Hat Enterprise Linux (RHEL) 7 and later versions—have these prerequisites configured by default, and you do not need to do any further configuration.

If you are running a different distro or an older version of CentOS, you can check the first prerequisite by using either the **findmnt** or **blkid** command to determine the file system type on your system. The command **findmnt** works only if the file system has been mounted, and **blkid** works whether it is mounted or not. Then you need to inspect the kernel configuration file /boot/conf-<*version.architecture*> to determine whether ACLs have been enabled for this file system type. Example 3-30 shows the relevant output for the file system on the sda1 partition.

Example 3-30 *ACL Support for the sda1 File System*

```
[root@server1 ~]# findmnt /dev/sda1
TARGET SOURCE    FSTYPE OPTIONS
/boot  /dev/sda1 xfs    rw,relatime,seclabel,attr2,inode64,noquota
[root@server1 ~]# cat /boot/config-3.10.0-693.el7.x86_64 | grep ACL
CONFIG_EXT4_FS_POSIX_ACL=y
CONFIG_XFS_POSIX_ACL=y
CONFIG_BTRFS_FS_POSIX_ACL=y
CONFIG_FS_POSIX_ACL=y
CONFIG_GENERIC_ACL=y
CONFIG_TMPFS_POSIX_ACL=y
CONFIG_NFS_V3_ACL=y
CONFIG_NFSD_V2_ACL=y
CONFIG_NFSD_V3_ACL=y
CONFIG_NFS_ACL_SUPPORT=m
CONFIG_CEPH_FS_POSIX_ACL=y
CONFIG_CIFS_ACL=y
[root@server1 ~]#
```

The kernel configuration file lists different configuration options, each followed by an **=** symbol and then the letter y, n, or m. The letter y means that this option (module) was configured as part of the kernel when the kernel was first compiled. In this example, CONFIG_XFS_POSIX_ACL=y means that the kernel supports ACLs for the xfs file system. The letter n indicates that this module was not compiled into the kernel, and the letter m means that this module was compiled as a loadable kernel module (introduced in Chapter 2).

The second prerequisite is that the partition on which the ACLs will be used has to be mounted with the ACL option. By default, on ext3/4 and xfs file systems, ACL support is enabled. In older CentOS versions and other distros where the ACL option is not enabled by default, the file system can be mounted with the ACL option by using the syntax **mount -o acl** {*partition*} {*mount_point*}. On the other hand, if the ACL option is enabled by default, and you want to disable ACL support while mounting the file system, you can use the **noacl** option with the **mount** command. As discussed in the previous section, mounting using the **mount** command is non-persistent. For persistent mounting with the ACL option, you can add an entry to the /dev/fstab file (or amend an existing entry) and add the **acl** option (right after the **defaults** keyword). The /dev/fstab file is discussed in detail earlier in this chapter.

Finally, by using the **yum info acl** command, you can confirm whether the ACL package has been installed. The **yum** command is covered in detail in Chapter 2.

When ACL support has been established, you can use the command **getfacl** {*filename*|*directory*} to display the ACL configuration for a file or directory. Example 3-31 shows the output of the **getfacl** command for the directory /Programming and then for the file NewFile.txt.

Example 3-31 *Output of the getfacl Command*

```
[root@localhost /]# ls -ld Programming
drwxr-xr-x. 2 root root 25 Jun  9 05:46 Programming
[root@localhost /]# ls -l Programming
total 0
-rw-r--r--. 1 root root 0 Jun  9 05:46 NewFile.txt
[root@localhost /]# getfacl Programming
# file: Programming
# owner: root
# group: root
user::rwx
group::r-x
other::r-x
[root@localhost /]# getfacl Programming/NewFile.txt
# file: Programming/NewFile.txt
# owner: root
# group: root
user::rw-
group::r--
other::r—
[root@localhost /]#
```

As you can see from the output in Example 3-31, both the directory and file are owned by the user root, and the group of both is also root. So far, there is no additional information provided by the **getfacl** command beyond what is already displayed by **ls -l**; the format is the only difference.

For the file NewFile.txt, the user NetProg is not the file owner and is not a member of the file group. As per the permissions for *other*, the user NetProg should be able to only read the file but not write to it or execute it. In Example 3-32, the user NetProg attempts to write to the file NewFile.txt by using the **echo** command, but a "Permission denied" error message is displayed. The **setfacl -m u:NetProg:rw /Programming/Newfile.txt** command grants write permission to the user NetProg. When the write operation is attempted again, it is successful due to the new elevated permissions.

Example 3-32 *Changing the Permissions for the User NetProg by Using setfacl*

```
! Echo(write) operation fails since NetProg has no write permissions
[NetProg@localhost /]$ echo "This is a write test" > /Programming/NewFile.txt
bash: /Programming/NewFile.txt: Permission denied

! Grant user NetProg write permission (requires root permissions)
[NetProg@localhost /]$ su
Password:
```

```
[root@localhost /]# setfacl -m u:NetProg:rw /Programming/NewFile.txt
[root@localhost /]# getfacl /Programming/NewFile.txt
getfacl: Removing leading '/' from absolute path names
# file: Programming/NewFile.txt
# owner: root
# group: root
user::rw-
user:NetProg:rw-
group::r--
mask::rw-
other::r--

! Write operation now successful
[root@localhost /]# su NetProg
[NetProg@localhost /]$ echo "This is a write test" > /Programming/NewFile.txt
[NetProg@localhost /]$ cat /Programming/NewFile.txt
This is a write test
[NetProg@localhost /]$ ls -l /Programming/NewFile.txt
-rw-rw-r--+ 1 root root 21 Jun  9 07:24 /Programming/NewFile.txt
[NetProg@localhost /]$
```

Notice the **+** symbol that now replaces the dot to the right of file permission bits at the end of Example 3-32. This indicates that an ACL has been applied to this file. The new write permission has been granted to the user NetProg only, and not to any other user. This was done without amending the file permissions for the user, group, or other categories. It was also done without modifying the group memberships of the user NetProg. The same permission could also be applied to a group instead of an individual user. The level of granularity provided by ACLs should be clear by now.

The **setfacl** command used in Example 3-32 was issued with the option **-m**, which is short for *modify* and is used to apply a new ACL or modify an existing ACL. To remove an ACL, you use the option **-x** instead of **-m**; the remainder of the command remains the same, except that the ACL in the command is an existing ACL that is now being removed.

In Example 3-32 you can see the three-field argument **u:NetProg:rw**. When setting an ACL for a user, the first field is u, as in the example. For a group, the first field would be g, and for other, the first field would be o. The second field is the user or group name, which is NetProg in this example. If the ACL is for other, this field remains empty. The third field is the permissions you wish to grant to the user or group.

Finally, after the three-field argument is the name of the directory or file to which the ACL is applied. Note that whether a full path or only a relative path is required depends on the current working directory relative to the location of the file or directory to which the ACL is being applied. The same rules apply here as with any other Linux command that operates on a file or directory.

Therefore, the general syntax of the **setfacl** command to add, modify, or remove an ACL is **setfacl** {**-ml-x**} {**u|g|o**}:{*username*|*group*}:{*permissions*} {*file*|*directory*}. To remove all ACL entries applied to a file, you use the option **-b** followed by the filename, omitting the three-field argument.

In Example 3-32, notice the text mask::rw- in the output of the **getfacl** command, after the ACL has been applied. The mask provides one more level of control over the permissions granted by the ACL. Say that after granting several users different permissions to a file, you decide to remove a specific permission, such as the write permission, from *all* named users. The ACL mask then comes in handy. The permissions in the mask override the permissions for all named users and the file group. For example, if the mask permissions are r-x and the user NetProg has been granted rwx permissions, that user's effective permissions are r-x after the mask is set. The effective mask permissions are applied using the command **setfacl -m m:**{*permissions*} {*filename*}. In Example 3-33, the user NetProg has permissions rw-, and so does the mask. The mask is modified to r--. Notice the effective permissions that appear on the right side of the output of the **getfacl** command after the mask has been modified. After you remove the write permission from the mask, NetProg's write attempt to the file fails.

Example 3-33 *Changing the Mask Permissions by Using setfacl*

```
! Set the effective rights mask
[root@localhost /]# setfacl -m m:r /Programming/NewFile.txt
[root@localhost /]# getfacl /Programming/NewFile.txt
getfacl: Removing leading '/' from absolute path names
# file: Programming/NewFile.txt
# owner: root
# group: root
user::rw-
user:NetProg:rw-        #effective:r--
group::r--
mask::r--
other::r--

! Write operation to file by user NetProg now fails
[root@localhost /]# su NetProg
[NetProg@localhost /]$ echo "Testing mask permissions" > /Programming/NewFile.txt
bash: /Programming/NewFile.txt: Permission denied
[NetProg@localhost /]$
```

When ACLs are applied to directories, by default, these ACLs are not inherited by files and subdirectories in that directory. In order to achieve inheritance, the option **-R** has to

be added to the same **setfacl** command used earlier. In Example 3-34, an ACL setting rwx permissions for the user NetProg is applied to the directory Programming. Attempting to write to file NewFile.txt under the directory by user NetProg fails because the write permission has not been inherited by the file.

Example 3-34 *ACLs Are Not Inherited by Default by Subdirectories and Files Under a Directory*

```
! Apply an acl to the /Programming directory
[root@localhost ~]# setfacl -m u:NetProg:rwx /Programming
[root@localhost ~]# getfacl /Programming
getfacl: Removing leading '/' from absolute path names
# file: Programming
# owner: root
# group: root
user::rwx
user:NetProg:rwx
group::r-x
mask::rwx
other::r-x

! The acl is not applied to NewFile.txt under the directory
[root@localhost ~]# getfacl /Programming/NewFile.txt
getfacl: Removing leading '/' from absolute path names
# file: Programming/NewFile.txt
# owner: root
# group: root
user::rw-
group::r--
other::r--

! And the write operation fails as expected
[root@localhost ~]# su - NetProg
[NetProg@localhost ~]$ echo "This is a write test" > /Programming/NewFile.txt
bash: /Programming/NewFile.txt: Permission denied
[NetProg@localhost ~]$
```

After the ACL has been removed and then reapplied in Example 3-35 using the **-R** option, the user NetProg can write to the file successfully. The **getfacl** command also shows that the ACL has been applied to the file as if the **setfacl** command had been applied to the file directly.

Example 3-35 *ACL Inheritance by Subdirectories and Files Under a Directory Using the -R Option*

```
! Clear the acl from the /Programming directory
[root@localhost ~]# setfacl -b /Programming

! Apply the acl to directory /Programming using the -R option
[root@localhost ~]# setfacl -R -m u:NetProg:rwx /Programming
[root@localhost ~]# getfacl /Programming
getfacl: Removing leading '/' from absolute path names
# file: Programming
# owner: root
# group: root
user::rwx
user:NetProg:rwx
group::r-x
mask::rwx
other::r-x

! The acl is inherited by the file NewFile.txt
[root@localhost ~]# getfacl /Programming/NewFile.txt
getfacl: Removing leading '/' from absolute path names
# file: Programming/NewFile.txt
# owner: root
# group: root
user::rw-
user:NetProg:rwx
group::r--
mask::rw-
other::r--

! And the write operation is successful
[root@localhost ~]# su - NetProg
[NetProg@localhost ~]$ echo "This is to test inheritance" > /Programming/NewFile.txt
[NetProg@localhost ~]$ cat /Programming/NewFile.txt
This is to test inheritance
[NetProg@localhost ~]$
```

It is important to remember that the ACL applied to a directory and inherited by all subdirectories and files will *not* be applied to any files created *after* the ACL has been applied. Only the files that existed before the ACL was applied will be affected.

The ACLs described so far are called access ACLs. Another type of ACLs, called default ACLs, may be used with directories (only) if the requirement is that all files and subdirectories, when created, should inherit the parent directory ACLs. The syntax for applying a default ACL is **setfacl -m d:**{**u**|**g**|**o**}:{*username*|*group*}:{*permissions*} {*directory*}. Try to

experiment with default ACLs and note how newly created files inherit the directory ACL without your having to explicitly issue the **setfacl** command after the file or subdirectory has been created.

The same concepts discussed previously for a single user apply to a group when you set the ACL for a group of users other than the file or directory group by using the letter **g** along with the group name in the **setfacl** command instead of a **u** with the username.

In addition to using the **setfacl** command to set permissions for a specific user or group, you can use this command to set permissions for the file user, group, or other categories, similar to what can be accomplished using the **chmod** command as shown in the previous section. Note that if the **setfacl** command is used to apply an ACL to a file or directory, it is recommended that you *not* use **chmod**.

When a file or directory is copied or moved, ACLs are moved along with the file or directory.

Linux System Security

CentOS 7 and later versions come with a default built-in firewall service named **firewalld**. This service functions in a similar manner to a regular firewall in terms of providing *security zones* with different *trust levels*. Each zone constitutes a group of permit/deny rules for incoming traffic. Each physical interface on the server is bound to one of the firewall zones. However, **firewalld** provides only a subset of the services provided by a full-fledged firewall.

You can check the status of the **firewalld** service and start, stop, enable, and disable the service just as you would any other service on Linux by using the **systemctl** command. Example 3-36 shows the status of the **firewalld** service: In this example, you can see that it is active and enabled.

Example 3-36 *The firewalld Service Status*

```
[NetProg@localhost ~]$ systemctl status firewalld
● firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled; vendor
     preset: enabled)
   Active: active (running) since Sat 2018-04-21 21:37:06 +03; 30min ago
     Docs: man:firewalld(1)
 Main PID: 787 (firewalld)
   CGroup: /system.slice/firewalld.service
           └─787 /usr/bin/python -Es /usr/sbin/firewalld --nofork --nopid

Apr 21 21:37:05 localhost.localdomain systemd[1]: Starting firewalld - dynamic
  firewall daemon...
Apr 21 21:37:06 localhost.localdomain systemd[1]: Started firewalld - dynamic
  firewall daemon.

--------- OUTPUT TRUNCATED FOR BREVITY ---------
```

The **firewalld** service has a set of zones created by default when the service is first installed; these zones are sometimes referred to as the *base* or *predefined* zones. Custom zones can also be created and deleted. However, base zones cannot be deleted. One zone is designated as the default zone and is the zone to which all interfaces are bound, by default, unless the interface is explicitly moved to another zone. By default, the default zone is the public zone. Each zone has a set of rules attached to it and a list of interfaces bound to it. Rules and interfaces can be added to or removed from a zone.

Example 3-37 shows how to list the base zones of **firewalld** by using the command **firewall-cmd --get-zones** and how to identify the default zone by using the command **firewall-cmd --get-default-zone**.

Example 3-37 *Listing the Base and Default Zones of a Firewall*

```
[root@localhost ~]# firewall-cmd --get-zones
block dmz drop external home internal public trusted work
[root@localhost ~]# firewall-cmd --get-default-zone
public
[root@localhost ~]#
```

You can change the default zone by using the command **firewall-cmd --set-default-zone=**{*zone_name*}.

You can list the details of a zone by using the command **firewall-cmd --list-all --zone=**{*zone_name*}, as shown in Example 3-38. To list the details of the default zone, you omit the **--zone=**{*zone_name*} option.

Example 3-38 *Listing Zone Details*

```
[root@localhost ~]# firewall-cmd --list-all --zone=internal
internal
  target: default
  icmp-block-inversion: no
  interfaces:
  sources:
  services: ssh mdns samba-client dhcpv6-client
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:

[root@localhost ~]# firewall-cmd --list-all
public (active)
```

```
  target: default
  icmp-block-inversion: no
  interfaces: enp0s3 enp0s9 enp0s10 enp0s8
  sources:
  services: ssh dhcpv6-client
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
[root@localhost ~]#
```

Example 3-39 shows how to add rules to the zone dmz to permit specific incoming traffic on interfaces bound to this zone. The first rule added permits traffic from the source IP address 10.10.1.0/24 by using a *source-based* rule. Then BGP traffic on TCP port 179 is permitted by using a *port-based* rule. HTTP service is then permitted by defining a *service-based* rule. Finally, interface enp0s9 is removed from the public zone and bound to the dmz zone. Notice how the rules appear when the details of the zone are listed at the end of the example.

Example 3-39 *Adding Rules to Zone* ***dmz***

```
[root@localhost ~]# firewall-cmd --list-all --zone=dmz
dmz
  target: default
  icmp-block-inversion: no
  interfaces:
  sources:
  services: ssh
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:

[root@localhost ~]# firewall-cmd --zone=dmz --add-source=10.10.1.0/24
success
[root@localhost ~]# firewall-cmd --zone=dmz --add-port=179/tcp
success
```

```
[root@localhost ~]# firewall-cmd --zone=dmz --add-service=http
success
[root@localhost ~]# firewall-cmd --zone=dmz --add-interface=enp0s9
The interface is under control of NetworkManager, setting zone to 'dmz'.
success
[root@localhost ~]# firewall-cmd --zone=dmz --list-all
dmz (active)
  target: default
  icmp-block-inversion: no
  interfaces: enp0s9
  sources: 10.10.1.0/24
  services: ssh http
  ports: 179/tcp
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
[root@localhost ~]#
```

Note that in order to remove a rule, instead of using the **--add** option, you use the **--remove** option. For example, to remove the rule for TCP port 179, you use the command **firewall-cmd --zone=dmz --remove-port=179/tcp**.

Much like running and startup configurations on routers and switches, **firewalld** supports both runtime and permanent configurations. A *runtime configuration* is not persistent and is lost after a reload. A *permanent configuration* is persistent but takes effect only after a reload when the configuration has been changed. Any configuration commands that have been executed are reflected in the runtime configuration. To make a configuration permanent, you use the option **--permanent** with the command. You reload the **firewalld** service by using the command **firewall-cmd --reload**.

Linux Networking

Linux provides several methods for managing network devices and interfaces on a system. Usually, a system administrator can accomplish the same task using several different methods. A network device or an interface is managed by the kernel, and each method accesses the Linux kernel via a different path. There are three popular methods for managing Linux networking:

- Using the command-line **ip** utility
- Using the NetworkManager service
- Using network configuration files

This section covers these three methods listed. It should be fairly easy to use the help resources on your Linux distro, such as the man and info pages, to learn about any utility not covered here.

Note Keep in mind that some commands and utilities for managing Linux networking, such as **ifconfig**, **netstat**, **arp**, and **route**, are considered legacy utilities. These utilities have not been updated for years and have been deprecated on some distros but are still available on others. Even if any of these commands are available in the distro you are using, we do not recommend using them; instead, use the methods described in this section. Basically, the way legacy utilities function, particularly how these utilities speak with the kernel, is not very efficient. You will probably run into these legacy utilities at some point while working on Linux. For example, at the time of this writing, all four legacy utilities mentioned here are still supported on the Bash shells exposed by IOS XR and NX-OS.

The ip Utility

ip is a command-line utility that is part of the **iproute2** group of utilities. It is invoked using the command **ip** [*options*] {*object*} {*action*}. This syntax is quite intuitive in that the *action* in the command indicates what action you would want to apply to an *object*. For example, the command **ip link show** applies the action **show** to the object **link**. As you may have guessed, this command displays the state of all network interfaces (links) on the system, as shown in Example 3-40. To limit the output to one specific interface, you can add **dev** {*intf*} to the end of the command, as also shown in the example.

Note The man pages for the **ip** command refer to the *action* part in the previous syntax as *command*. We took the liberty to call it *action* in the upcoming few paragraphs in order to avoid the obvious confusion that will result from calling it *command*.

Example 3-40 *Output of the Command* ***ip link show***

```
[NetProg@localhost ~]$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
  qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
  DEFAULT qlen 1000
    link/ether 08:00:27:a7:32:f7 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
  DEFAULT qlen 1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
  DEFAULT qlen 1000
    link/ether 08:00:27:b4:ce:55 brd ff:ff:ff:ff:ff:ff
```

```
5: enp0s10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
  mode DEFAULT qlen 1000
    link/ether 08:00:27:48:59:02 brd ff:ff:ff:ff:ff:ff
6: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
  mode DEFAULT qlen 1000
    link/ether 52:54:00:ea:c5:d4 brd ff:ff:ff:ff:ff:ff
7: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 state
  DOWN mode DEFAULT qlen 1000
    link/ether 52:54:00:ea:c5:d4 brd ff:ff:ff:ff:ff:ff
[NetProg@localhost ~]$ ip link show dev enp0s3
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
  DEFAULT qlen 1000
    link/ether 08:00:27:a7:32:f7 brd ff:ff:ff:ff:ff:ff
[NetProg@localhost ~]$
```

Table 3-1 lists some of the objects that are commonly used with the **ip** command.

Table 3-1 *Objects That Are Commonly Used with the* ***ip*** *Command*

Object	Description
address	IPv4 or IPv6 protocol address
link	Network interface
route	Routing table entry
maddress	Multicast address
neigh	ARP entry

As of this writing, there are 19 objects that can be acted upon by using the **ip** command. A full list of objects can be found in the man pages for the **ip** command. Objects can be written in full or in abbreviated form, such as **address** or **addr**. The actions that can be used with the **ip** command are limited to three options listed in Table 3-2.

Table 3-2 *Actions That Can Be Used with the* ***ip*** *Command*

Action	Description
add	Adds the object
delete	Deletes the object
show (or **list**)	Displays information about the object

The keyword **show** or **list** can be dropped from a command, and the command will still be interpreted as a **show** action. For example, the command **ip link show** is equivalent to just **ip link**.

The **ip addr** command lists all interfaces on the system, each with its IP address information, and the **ip maddr** command displays the multicast information for each and every interface. The **ip neigh** command displays the ARP table. The ARP table consists of a list of neighbors on each interface on the local network. The examples in this section show how to use these **show** commands.

You can bring an interface on Linux up or down by using the command **ip link set** {*intf*} {**up|down**}. The **set** action is only applicable to the link object and therefore was not listed in Table 3-2. Example 3-41 shows how to bring interface enp0s8 down and then up again. Note that changing networking configuration on Linux, including toggling an interface's state, requires root privileges. The **show** commands, however, do not. To keep Example 3-41 short and avoid the frequent password prompt, all commands in the example are issued by the root user. However, running commands as root in general is *not* a recommended practice. On a production network, make sure to avoid logging in as root. It is best practice to log in with your regular user account and use the **sudo** command whenever a command requires root privileges to execute, as explained in Chapter 2.

Example 3-41 *Toggling Interface State*

```
[root@localhost ~]# ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
  DEFAULT qlen 1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
[root@localhost ~]# ip link set enp0s8 down
[root@localhost ~]# ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN mode DEFAULT
  qlen 1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
[root@localhost ~]# ip link set enp0s8 up
[root@localhost ~]# ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
  DEFAULT qlen 1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
[root@localhost ~]#
```

You can add an IP address to an interface by using the command **ip addr add** {*IP_address*} **dev** {*intf*}. By replacing the action **add** with **del**, you remove the IP address. In Example 3-42, IP address 10.1.0.10/24 is added to interface enp0s8, and then the original IP address, 10.1.0.1/24, is removed. The **ip addr show dev enp0s8** command is used to inspect the interface IP address before and after the change.

Example 3-42 *Adding and Removing IP Addresses on Interfaces*

```
[root@localhost ~]# ip addr show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
  1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
    inet 10.1.0.1/24 brd 10.1.0.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::8b8:d663:847f:79d9/64 scope link
       valid_lft forever preferred_lft forever
[root@localhost ~]# ip addr add 10.1.0.10/24 dev enp0s8
[root@localhost ~]# ip addr show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
  1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
    inet 10.1.0.1/24 brd 10.1.0.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet 10.1.0.10/24 scope global secondary enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::8b8:d663:847f:79d9/64 scope link
       valid_lft forever preferred_lft forever
[root@localhost ~]# ip addr del 10.1.0.1/24 dev enp0s8
[root@localhost ~]# ip addr show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
  1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
    inet 10.1.0.10/24 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::8b8:d663:847f:79d9/64 scope link
       valid_lft forever preferred_lft forever
[root@localhost ~]#
```

Notice that the IP address 10.1.0.10/24 is added as a secondary address, as long as another IP address is configured on the interface. When the original IP address is removed, the new IP address becomes the primary address.

Notice the mtu value in the output of the **ip addr show** command in Example 3-42. By default the mtu is set to 1500 bytes. To change that value, you use the command **ip link set** {*intf*} **mtu** {*mtu_value*}.

A very useful feature that any network engineer would truly appreciate is interface promiscuous mode. By default, when an Ethernet frame is received on an interface, that frame is passed on to the upper layers for processing only if the destination MAC address of the frame matches the MAC address of the interface (or if the destination MAC address is a broadcast address). If the MAC addresses do not match, the frame is ignored. This renders packet sniffing applications such as Wireshark and features such as port mirroring unusable. In promiscuous mode, an interface accepts any and all incoming

packets, whether the packets are addressed to that interface or not. You can enable promiscuous mode by using the command **ip link set** {*intf*} **promisc on**.

In the routing table, the list of routes on the system can be displayed by using the command **ip route**. Example 3-43 shows that the routing table is empty when no IP addresses are configured on any of the interfaces. When the IP address 10.2.0.30/24 is configured on interface enp0s3, one entry, corresponding to that interface, is added to the routing table.

Example 3-43 *Viewing a Routing Table by Using the* ***ip route*** *Command*

```
[NetProg@server4 ~]$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
  1000
    link/ether 08:00:27:2c:61:d0 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::a00:27ff:fe2c:61d0/64 scope link
       valid_lft forever preferred_lft forever
[NetProg@server4 ~]$ ip route

[NetProg@server4 ~]$ sudo ip addr add 10.2.0.30/24 dev enp0s3
[sudo] password for NetProg:
[NetProg@server4 ~]$ ip addr show dev enp0s3
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
  1000
    link/ether 08:00:27:2c:61:d0 brd ff:ff:ff:ff:ff:ff
    inet 10.2.0.30/24 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe2c:61d0/64 scope link
       valid_lft forever preferred_lft forever
[NetProg@server4 ~]$ ip route
10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30
[NetProg@server4 ~]$
```

Routing tables on Linux systems are very similar to routing tables on routers. In fact, a Linux server could easily function as a router. In order to display routing table functionality in Linux, server1 in the topology in Figure 3-2 is used as a router to route traffic between server2 and server3. server2 is connected to network 10.1.0.0/24, and server3 is connected to network 10.2.0.0/24. All three servers are configured such that server1 routes between the two networks, and eventually server2 should be able to ping server3.

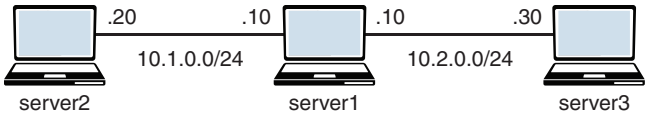


Figure 3-2 *Server1 Configured to Route Between server2 and server3, Each on a Different Subnet*

IP addressing needs to be configured first. server1 is configured with IP addresses ending with .10, server2 with an IP address ending in .20, and server3 with an IP address ending in .30, as shown in Example 3-44.

Example 3-44 *Configuring IP Addresses on the Interfaces Connecting The Three Servers*

```
! server1
[root@server1 ~]# ip addr add 10.1.0.10/24 dev enp0s8
[root@server1 ~]# ip addr add 10.2.0.10/24 dev enp0s9
[root@server1 ~]# ip addr show enp0s8 | grep "inet "
    inet 10.1.0.10/24 scope global enp0s8
[root@server1 ~]# ip addr show enp0s9 | grep "inet "
    inet 10.2.0.10/24 scope global enp0s9
[root@server1 ~]#

! server2
[root@server2 ~]# ip addr add 10.1.0.20/24 dev enp0s3
[root@server2 ~]# ip addr show enp0s3 | grep "inet "
    inet 10.1.0.20/24 scope global enp0s3
[root@server2 ~]#

! server3
[root@server3 ~]# ip addr add 10.2.0.30/24 dev enp0s3
[root@server3 ~]# ip addr show dev enp0s3 | grep "inet "
    inet 10.2.0.30/24 scope global enp0s3
[root@server3 ~]#
```

A ping to the directly connected server is successful on all three servers. However, when server2 attempts to ping server3, the ping fails, as shown in Example 3-45.

Example 3-45 *Pinging the Directly Connected Interfaces Is Successful but Pinging server3 From server2 Is Not*

```
! Pinging the directly connected interfaces

! server2 to server1
[root@server2 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.796 ms

--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.796/0.796/0.796/0.000 ms
[root@server2 ~]#

! server3 to server1
[root@server3 ~]# ping -c 1 10.2.0.10
PING 10.2.0.10 (10.2.0.10) 56(84) bytes of data.
64 bytes from 10.2.0.10: icmp_seq=1 ttl=64 time=1.13 ms

--- 10.2.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.139/1.139/1.139/0.000 ms
[root@server3 ~]#

! Pinging server2 to server3 and vice versa is not successful

! server2 to subnet 10.2.0.0/24
[root@server2 ~]# ping 10.2.0.10
connect: Network is unreachable
[root@server2 ~]# ping 10.2.0.30
connect: Network is unreachable
[root@server2 ~]#

! server3 to subnet 10.1.0.0/24
[root@server3 ~]# ping -c 1 10.1.0.10
connect: Network is unreachable
[root@server3 ~]# ping -c 1 10.1.0.20
connect: Network is unreachable
[root@server3 ~]#
```

You are probably very familiar with the **ping** command. **ping** works on Linux exactly as it does on network devices: by sending one or more ICMP packets to the destination and either receiving an ICMP reply if the ping is successful (one reply per packet sent)

or receiving an ICMP unreachable packet or no response at all if the ping is not. The command in Example 3-45 uses the **-c 1** option to send a single ICMP packet, which is enough to test the reachability of the destination.

Example 3-46 shows how to use the command **ip route add 10.2.0.0/24 via 10.1.0.10** on server2 and the command **ip route add 10.1.0.0/24 via 10.2.0.10** on server3 to add routes to the routing tables of each server. The general syntax for adding a route to the routing table is **ip route add** {*destination*}{*/mask*} **via** {*nexthop*}. The routes instruct each server to use server1 as the next hop to reach the remote network. After the routes are added, server2 and server3 are able to ping server1's interface on the remote network, but they are still not able to ping each other.

Example 3-46 *Adding Routing Table Entries for Remote Subnets on server2 and server3. server2 and server3 Can Ping the Remote Subnets on server1, But Still Cannot Ping Each Other*

```
! server2
[root@server2 ~]# ip route add 10.2.0.0/24 via 10.1.0.10
[root@server2 ~]# ip route
10.1.0.0/24 dev enp0s3 proto kernel scope link src 10.1.0.20
10.2.0.0/24 via 10.1.0.10 dev enp0s3
[root@server2 ~]# ping -c 1 10.2.0.10
PING 10.2.0.10 (10.2.0.10) 56(84) bytes of data.
64 bytes from 10.2.0.10: icmp_seq=1 ttl=64 time=0.822 ms

--- 10.2.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.822/0.822/0.822/0.000 ms
[root@server2 ~]# ping -c 1 10.2.0.30
PING 10.2.0.30 (10.2.0.30) 56(84) bytes of data.

--- 10.2.0.30 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
[root@server2 ~]#

! server3
[root@server3 ~]# ip route add 10.1.0.0/24 via 10.2.0.10
[root@server3 ~]# ip route
10.1.0.0/24 via 10.2.0.10 dev enp0s3
10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30
[root@server3 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.865 ms
```

```
--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.865/0.865/0.865/0.000 ms
[root@server3 ~]# ping -c 1 10.1.0.20
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.

--- 10.1.0.20 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
[root@server3 ~]#
```

Forwarding between the interfaces on server1 is disabled by default for security reasons. Therefore, the remaining step is to enable forwarding in the kernel of server1 by toggling the default value of 0 in file /proc/sys/net/ipv4/ip_forward to 1 by using either the command **echo 1 > /proc/sys/net/ipv4/ip_forward** or the command **/sbin/sysctl -w net.ipv4.ip_forwad=1**. After either command is used, forwarding is enabled, and both servers can ping each other successfully, as shown in Example 3-47.

Example 3-47 *Enabling Routing on Server1 Resulting in Successful ping Between server2 and server3*

```
! Enabling routing on server1
[root@server1 ~]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@server1 ~]# cat /proc/sys/net/ipv4/ip_forward
1

! server2 to server3 ping is successful
[root@server2 ~]# ping -c 1 10.2.0.30
PING 10.2.0.30 (10.2.0.30) 56(84) bytes of data.
64 bytes from 10.2.0.30: icmp_seq=1 ttl=63 time=0.953 ms
--- 10.2.0.30 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.953/0.953/0.953/0.000 ms
[root@server2 ~]#

! server3 to server2 ping is successful
[root@server3 ~]# ping -c 1 10.1.0.20
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=63 time=1.39 ms

--- 10.1.0.20 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.394/1.394/1.394/0.000 ms
[root@server3 ~]#
```

Note that two commands to achieve the same result are mentioned here. The first method gets the job done by editing a file, and the second gets the same job done by using the command **sysctl**. Which one you should use depends on several factors, the first of which is personal preference. Another issue is whether you know which file in the /proc/sys/ directory contains the kernel setting (sometimes referred to as a *kernel tunable*) that you need to change. If you do not know the file, you can simply use the **sysctl** command to target the parameter directly, regardless of where it is located. You can list all kernel tunables by using the command **/sbni/sysctl -a**.

Note *You* in this case does not necessarily have to literally mean you. It may refer to the automation script or tool that you are using to get the job done. Using a particular tool to amend a file may be more efficient than issuing a command; the reverse may be the case for another tool. Always choose the method that is most efficient and effective for your specific environment.

To remove a routing table entry, you use the syntax **ip route delete** {*destination*}{*/mask*} **via** {*nexthop*}. You can also have routes point to exit interfaces rather than next hops by using the syntax **ip route add** {*destination*}{*/mask*} **dev** {*intf*}. You can add a default route by using the syntax **ip route add default via** {*next_hop*} **dev** {*intf*}.

One final note on the **ip** utility is that any configuration performed using the commands discussed in this section is not persistent. Any changes to the configuration disappear after a system reboot. Persistent configuration is discussed in the following sections.

The NetworkManager Service

NetworkManager is the default network management service on several Linux distros, including Red Hat and Fedora. Because NetworkManager is a service, you can check its status, and you can start, stop, enable, or disable it as you can any other service on Linux by using the **systemctl** command. For example, the command **systemctl status network-manager** displays the current status of the service. To poll NetworkManager for information or push configuration to it, you can use one of several user interfaces:

- **Graphical user interfaces (GUIs):** There are two main graphical user interface tools that interact with NetworkManager. The first is the Network Control Center, which is accessible via the Settings menu. The Settings window has an icon labeled Network that opens the network control center, which provides basic network configuration. The other GUI tool is the Connection Editor and is used to configure more advanced settings. You can start the Connection Editor from the terminal by entering the command **nm-connection-editor**.
- **NetworkManager command-line interface (nmcli):** The NetworkManager CLI is a command-line utility that you can use to control NetworkManager. You can use this interface to NetworkManager via the **nmcli** command in the Bash shell.

- **NetworkManager text user interface (nmtui):** Similar to the interface used to configure a computer's BIOS settings or old DOS-based programs, the **nmtui** provides an interface to NetworkManager that displays graphics in text mode. You start the text user interface by issuing the **nmtui** command in the shell.
- **API:** NetworkManager provides an API that can be used by applications for programmatic access to NetworkManager.

Because the majority of automation is typically performed through CLI tools (and API calls) and not the GUI, this section cover NetworkManager configuration via the **nmcli** interface.

NetworkManager deals with objects called connections. A *connection* is a representation of a link to the outside world and may represent, for example, a wired connection, a wireless connection, or a VPN connection. To display the current status of all network connections on a system, use the command **nmcli con show**, as shown in Example 3-48.

Example 3-48 *Listing All Connections on a System*

```
[root@server1 ~]# nmcli con show
NAME                UUID                                  TYPE            DEVICE
Wired connection 1  d8323782-5cf2-3afc-abcd-e603605ac4f8  802-3-ethernet  --
Wired connection 2  669fefb4-bc57-3d19-b83b-2b2125e0036b  802-3-ethernet  --
[root@server1 ~]#
```

The output in Example 3-48 indicates that there are two connections, named Wired connection 1 and Wired connection 2. These connections are not bound (applied) to any interfaces, as indicated by the -- in the last column. Both connections are of type Ethernet. A connection is uniquely identified by its universally unique identifier (UUID). Although not shown in the command output, a connection can either be active or inactive. To activate an inactive connection, you use the command **nmcli con up** {*connection_name*}. To deactivate a connection, you replace the keyword **up** with the keyword **down**.

Each connection is known as a *connection profile* and contains several attributes or properties that you can set. These properties are known as *settings*. Connection profile settings are created and then applied to a device or device type. Settings are represented in a dot notation. For example, a connection's IPv4 addresses are represented by the setting **ipv4.addresses**. To drill down on the details for a specific connection and list its settings and their values, you can use the command **nmcli con show** {*connection_name*}. Example 3-49 lists the connection profile settings for Wired connection 1. The output is truncated due to the length of the list. A full list of settings and their meanings can be found in the man pages for the **nmcli** command.

Note Use of the terms "master" and "slave" is ONLY in association with the official terminology used in industry specifications and standards, and in no way diminishes Pearson's commitment to promoting diversity, equity, and inclusion, and challenging, countering and/or combating bias and stereotyping in the global population of the learners we serve.

Example 3-49 *Connection Attributes for Wired Connection 1*

```
[root@server1 ~]# nmcli con show "Wired connection 1"
connection.id:                          Wired connection 1
connection.uuid:                        d8323782-5cf2-3afc-abcd-e603605ac4f8
connection.stable-id:                   --
connection.interface-name:              --
connection.type:                        802-3-ethernet
connection.autoconnect:                 yes
connection.autoconnect-priority:        -999
connection.autoconnect-retries:         -1 (default)
connection.timestamp:                   1525512827
connection.read-only:                   no
connection.permissions:                 --
connection.zone:                        --
connection.master:                      --
connection.slave-type:                  --
connection.autoconnect-slaves:          -1 (default)
connection.secondaries:                 --
connection.gateway-ping-timeout:        0
connection.metered:                     unknown
connection.lldp:                        -1 (default)
802-3-ethernet.port:                    --
802-3-ethernet.speed:                   0
802-3-ethernet.duplex:                  --
802-3-ethernet.auto-negotiate:          no
802-3-ethernet.mac-address:             08:00:27:83:40:75
802-3-ethernet.cloned-mac-address:      --
802-3-ethernet.generate-mac-address-mask:--
802-3-ethernet.mac-address-blacklist:   --
802-3-ethernet.mtu:                     auto
802-3-ethernet.s390-subchannels:        --
802-3-ethernet.s390-nettype:            --
802-3-ethernet.s390-options:            --
802-3-ethernet.wake-on-lan:             1 (default)
802-3-ethernet.wake-on-lan-password:    --
ipv4.method:                            auto
ipv4.dns:                               --
ipv4.dns-search:                        --
ipv4.dns-options:                       (default)
ipv4.dns-priority:                      0
ipv4.addresses:                         --
ipv4.gateway:                           --
ipv4.routes:                            --

--------- OUTPUT TRUNCATED FOR BREVITY ---------
```

To list the devices (aka interfaces) on the system and the status of each one, you use the command **nmcli dev status** for all devices or the command **nmcli dev show** {*device_name*} for a specific device, as shown in Example 3-50.

Example 3-50 *Device Status Using the* ***nmcli dev status*** *and* ***nmcli dev show*** *Commands*

```
[root@server1 ~]# nmcli dev status
DEVICE  TYPE      STATE         CONNECTION
enp0s8  ethernet  disconnected  --
enp0s9  ethernet  disconnected  --
lo      loopback  unmanaged     --
[root@server1 ~]# nmcli dev show enp0s8
GENERAL.DEVICE:                         enp0s8
GENERAL.TYPE:                           ethernet
GENERAL.HWADDR:                         08:00:27:83:40:75
GENERAL.MTU:                            1500
GENERAL.STATE:                          30 (disconnected)
GENERAL.CONNECTION:                     --
GENERAL.CON-PATH:                       --
WIRED-PROPERTIES.CARRIER:               on
[root@server1 ~]#
```

As you can see from the outputs in Examples 3-49 and 3-50, connections and devices are mutually exclusive. A connection profile may or may not be applied to a device after it is created.

In Example 3-51, both of the wired connections are deleted, and one new connection named NetDev_1 is created. NetDev_1 is of type ethernet and is applied to device enp0s8. Connections are deleted using the command **nmcli con del** {*connection_name*}. You create new connections and configure their settings by using the command **nmcli con add** {*connection_name*} {*setting*} {*value*}. In Example 3-51, the type, ifname, ip4, and gw4 settings are set to Ethernet, enp0s8, 10.1.0.10/24, and 10.1.0.254, respectively. Note that in this command, *setting* can either be entered in the full dot format or in abbreviated format. For example, the IP address can be set using either **ip4** or **ipv4.address**.

Example 3-51 *Deleting and Creating Connections*

```
[root@server1 ~]# nmcli con show
NAME                UUID                                  TYPE            DEVICE
Wired connection 1  d8323782-5cf2-3afc-abcd-e603605ac4f8  802-3-ethernet  --
Wired connection 2  669fefb4-bc57-3d19-b83b-2b2125e0036b  802-3-ethernet  --
[root@server1 ~]# nmcli con del "Wired connection 1"
Connection 'Wired connection 1' (d8323782-5cf2-3afc-abcd-e603605ac4f8) successfully
  deleted.
[root@server1 ~]# nmcli con del "Wired connection 2"
```

```
Connection 'Wired connection 2' (669fefb4-bc57-3d19-b83b-2b2125e0036b) successfully
  deleted.
[root@server1 ~]# nmcli con show
NAME  UUID  TYPE  DEVICE
[root@server1 ~]# nmcli con add con-name NetDev_1 type ethernet ifname enp0s8 ip4
  10.1.0.10/24 gw4 10.1.0.254
Connection 'NetDev_1' (a8ac9116-697a-4a0a-85a2-63428d6e75a3) successfully added.
[root@server1 ~]# nmcli con show
NAME      UUID                                  TYPE            DEVICE
NetDev_1  a8ac9116-697a-4a0a-85a2-63428d6e75a3  802-3-ethernet  enp0s8
[root@server1 ~]# nmcli con show --active
NAME      UUID                                  TYPE            DEVICE
NetDev_1  a8ac9116-697a-4a0a-85a2-63428d6e75a3  802-3-ethernet  enp0s8
[root@server1 ~]# nmcli dev status
DEVICE  TYPE      STATE         CONNECTION
enp0s8  ethernet  connected     NetDev_1
enp0s9  ethernet  disconnected  --
lo      loopback  unmanaged     --
[root@server1 ~]# nmcli dev show enp0s8
GENERAL.DEVICE:                         enp0s8
GENERAL.TYPE:                           ethernet
GENERAL.HWADDR:                         08:00:27:83:40:75
GENERAL.MTU:                            1500
GENERAL.STATE:                          100 (connected)
GENERAL.CONNECTION:                     NetDev_1
GENERAL.CON-PATH:                       /org/freedesktop/NetworkManager/ActiveCon-
  nection/359
WIRED-PROPERTIES.CARRIER:               on
IP4.ADDRESS[1]:                         10.1.0.10/24
IP4.GATEWAY:                            10.1.0.254
IP6.ADDRESS[1]:                         fe80::8c1f:4c4a:51a5:6423/64
IP6.GATEWAY:                            --
[root@server1 ~]# ping 10.1.0.20 -c 3
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=64 time=0.604 ms
64 bytes from 10.1.0.20: icmp_seq=2 ttl=64 time=0.602 ms
64 bytes from 10.1.0.20: icmp_seq=3 ttl=64 time=0.732 ms

--- 10.1.0.20 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 0.602/0.646/0.732/0.060 ms
[root@server1 ~]#
```

Notice that once a connection has been created and the device enp0s8 has been bound to it (all in the same command), the connection and device both come up, and that results in the device successfully pinging server2 on the other end of the link.

After a connection is created, you can modify its settings by using the command **nmcli con mod** {*connection_name*} {*setting*} {*value*}. When modifying a setting, the full dot format is required in the command. If the shorthand format is used, the new value in the command may be added to the existing value of the setting. For example, if the shorthand format is used to modify the IP address, the new IP address in the command is added to the device as a secondary IP address. On the other hand, if the full dot format is used, the IP address in the command replaces the IP address configured on the device. Example 3-52 shows how to modify the IP address of device enp0s8 to 10.1.0.100/24.

Example 3-52 *Deleting and Creating Connections*

```
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr
ipv4.addresses:                         10.1.0.10/24
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.10/24
[root@server1 ~]# nmcli con mod NetDev_1 ip4 10.1.0.100/24

! The new IP address is added as a secondary address due to the shorthand format
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr
ipv4.addresses:                         10.1.0.10/24, 10.1.0.100/24

! The new IP address is not reflected to the device enp0s8
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.10/24

[root@server1 ~]# nmcli con up NetDev_1
Connection successfully activated (D-Bus active path: /org/freedesktop/
  NetworkManager/ActiveConnection/366)

! After resetting the con, the new IP address now is reflected to the device
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.10/24
IP4.ADDRESS[2]:                         10.1.0.100/24

! Using the full dot format will replace the old IP address with the new one
[root@server1 ~]# nmcli con mod NetDev_1 ipv4.address 10.1.0.100/24
[root@server1 ~]# nmcli con up NetDev_1
Connection successfully activated (D-Bus active path: /org/freedesktop/
  NetworkManager/ActiveConnection/367)
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr
ipv4.addresses:                         10.1.0.100/24
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.100/24
[root@server1 ~]#
```

Note that each time a change is made to a connection using **nmcli**, the connection needs to be reactivated in order for the changes to be reflected to the device.

Adding routes using **nmcli** is different than adding routes using the **ip** utility in that when using **nmcli**, routes are added per interface and not globally. You add routes by using the syntax **nmcli con mod** {*intf*} **+ipv4.routes** {*destination*} **ipv4.gateway** {*next_hop*}. Therefore, to accomplish the same task that was done earlier by using the **ip** utility (to add a route on server2 to direct traffic destined for network 10.2.0.0/24 using the next hop 10.1.0.10 on server1), you use the following command: **nmcli con mod enp0s3 +ipv4.routes 10.2.0.0/24 ipv4.gateway 10.1.0.10**.

Unlike with the **ip** utility, changes made through **nmcli** are, by default, persistent and will survive a system reload.

It is important to understand the difference between the **ip** utility and NetworkManager. The **ip** utility is a program. When you use the **ip** command, you run this program, which makes a system call to the kernel, either to retrieve information or configure a component of the Linux networking system.

On the other hand, NetworkManager is a system daemon. It is software that runs (lurks) in the background, by default, and oversees the operation of the Linux network system. NetworkManager may be used to configure components of the network or to retrieve information about the network by using a variety of methods discussed earlier in this section—one of them being **nmcli**.

The nuances of how the **ip** utility interacts with NetworkManager are not discussed in detail here. All you need to know for now is that changes to the network that are made via the **ip** utility are detected and preserved by NetworkManager. There is no conflict between them. As mentioned at the very beginning of this section, different software on Linux can achieve the same result via different communication channels with the kernel. However, any software that needs access to the network will eventually have to go through the kernel.

Network Scripts and Configuration Files

The third method for configuring network devices and interfaces is to modify network scripts and configuration files directly. Different files in Linux control different components of the networking ecosystem, and editing these files was the only way to configure networking on Linux before NetworkManager was developed. Configuration files and scripts can still be used instead of, or in addition to, NetworkManager.

On Linux distros in the Red Hat family, configuration files for network interfaces are located in the /etc/sysconfig/network-scripts directory, and each interface configuration file is named ifcfg-**<***intf_name***>**. The first script that is executed on system bootup is /etc/init.d/network. When the system boots up, this script reads through all interface configuration files whose names start with ifcfg. Example 3-53 shows the ifcfg file for the enp0s8 interface.

Example 3-53 *Interface Configuration File for Interface enp0s8*

```
[root@server1 network-scripts]# cat ifcfg-enp0s8
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=dhcp
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=NetDev_1
UUID=a8ac9116-697a-4a0a-85a2-63428d6e75a3
DEVICE=enp0s8
ONBOOT=yes
[root@server1 network-scripts]#
```

The filename just needs to be prefixed with ifcfg. The network script simply scans the directory and reads any file whose name has this prefix. Therefore, you can safely assume that the configuration file is for the interface *or* connection. However, while the filename has to start with ifcfg, there is general consensus that the value in the DEVICE field (interface) should follow the ifcfg prefix.

The TYPE field in the file indicates the connection type, which is **Ethernet** in this case. The BOOTPROTO field is set to **dhcp**, which means the connection gets an IP address via DHCP. If a static IP address is required on the interface, then dhcp is replaced with none. The interface associated with this configuration is also shown in the DEVICE field (enp0s8 in this case), and the ONBOOT field indicates that this connection is to be brought up at system bootup. When a static IP address is required on the interface, the fields IPADDR, PREFIX, and GATEWAY and their respective values are added to the file.

When **ONBOOT=yes** is set, the /etc/init.d/network script checks whether this interface is managed by NetworkManager. If it is and the connection has already been activated, no further action is taken. If the connection has not been activated, the script requests NetworkManager to activate the connection. In case the connection is not managed by NetworkManager, the network script activates the connection by running another script, /usr/sbin/ifup. The ifup script checks the field TYPE in the ifcfg file, and based on that, it calls *another* type-specific script. For example, if the type of the connection is Ethernet, the ifup-eth script is called. Linux requires type-specific scripts because different connection types require different configuration parameters. For example, the concept of an SSID (wireless network name) does not exist for an Ethernet connection. Similarly, to bring down an interface for an unmanaged interface, the ifdown script is called. The vast majority of interface types are managed by NetworkManager by default, unless the line **NM_CONTROLLED=no** has been added to the ifcfg file.

While the recommended method for configuring interfaces is to use the **nmcli** utility, as discussed in the previous section, you can also configure interfaces by editing the corresponding ifcfg file.

Static routes configured on a system have configuration files named **route-<***intf_name***>** in the same directory as the interface configuration files. As you have probably guessed, the name has to be prefixed with **route**. However, the **-<***intf_name***>** is just a naming convention, and the file may have any name as long as the prefix **route** is there. The routing entries in the file may have one of two formats:

- The **ip** command arguments format:

  {*destination*}/{*mask*} via {*next_hop*} [dev *interface*]

  With this format, specifying the interface using [**dev** *interface*] is optional.

- The network/netmask directives format:

  ADDRESS{*N*}:{*destination*}

  NETMASK{*N*}:{*netmask*}

  GATEWAY{*N*}:{*next_hop*}

  where *N* is the routing table entry starting with 0 and incrementing by 1 for each entry, without skipping any values. In other words, if the routing table has four entries, the entries are numbered from 0 to 3.

Going back to the network of three servers in Figure 3-2, where server1 is required to route between server2 on subnet 10.1.0.0/24 and server3 on subnet 10.2.0.0/24: the static routes previously configured in order to route between the servers are deleted, after which the ping from server2 to server3 fails, as shown in Example 3-54.

Example 3-54 *Ping Fails Due To Lack of Static Routes on server2 and server3*

```
! No routes in routing table of server2 to remote subnet 10.2.0.0/24
[root@server2 ~]# ip route
10.1.0.0/24 dev enp0s3 proto kernel scope link src 10.1.0.20 metric 100
[root@server2 ~]#

! Ping to the directly connected interface on server1 is successful
[root@server2 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.828 ms

--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.828/0.828/0.828/0.000 ms
[root@server2 ~]#
```

```
! Ping to server3 on subnet 10.2.0.0/24 is not successful
[root@server2 ~]# ping -c 1 10.2.0.30
connect: Network is unreachable
[root@server2 ~]#

! No routes in routing table of server3 to remote subnet 10.1.0.0/24
[root@server3 ~]# ip route
10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30 metric 100
[root@server3 ~]#

! Ping to the directly connected interface on server1 is successful
[root@server3 ~]# ping -c 1 10.2.0.10
PING 10.2.0.10 (10.2.0.10) 56(84) bytes of data.
64 bytes from 10.2.0.10: icmp_seq=1 ttl=64 time=0.780 ms

--- 10.2.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.780/0.780/0.780/0.000 ms
[root@server3 ~]#

! Ping to server2 on subnet 10.1.0.0/24 is not successful
[root@server3 ~]# ping -c 1 10.1.0.20
connect: Network is unreachable
[root@server3 ~]#
```

The file route-enp0s3 is created under the directory /etc/sysconfig/network-scripts/ on both servers. A routing entry is added to the routing configuration file on server2 by using the **ip** command arguments format, and a routing entry is added to the file on server3 by using the network/netmask directives format, as shown in Example 3-55.

Example 3-55 *Routing Configuration Files Added on Both server2 and server3*

```
! server2

! No routing configuration files in the directory
[root@server2 ~]# cd /etc/sysconfig/network-scripts/
[root@server2 network-scripts]# ls -l | grep "route"
[root@server2 network-scripts]#

! Create the file route-enp0s3 and populate it with a route to the remote subnet
  10.2.0.0/24 using the IP Command Arguments format
[root@server2 network-scripts]# touch route-enp0s3
[root@server2 network-scripts]# echo "10.2.0.0/24 via 10.1.0.10" >> route-enp0s3
[root@server2 network-scripts]# ls -l | grep " route"
-rw-r--r--. 1 root root    26 Aug 17 15:52 route-enp0s3
```

```
[root@server2 network-scripts]# cat route-enp0s3
10.2.0.0/24 via 10.1.0.10
[root@server2 network-scripts]#

! Restart the network service and check the routing table
[root@server2 network-scripts]# systemctl restart network
[root@server2 network-scripts]# ip route
10.1.0.0/24 dev enp0s3 proto kernel scope link src 10.1.0.20 metric 100
10.2.0.0/24 via 10.1.0.10 dev enp0s3 proto static metric 100
[root@server2 network-scripts]#

! server3

! No routing configuration files in the directory
[root@server3 ~]# cd /etc/sysconfig/network-scripts/
[root@server3 network-scripts]# ls -l | grep " route"

! Create the file route-enp0s3 and populate it with a route to the remote subnet
  10.1.0.0/24 using the Network/Netmask Directives format
[root@server3 network-scripts]# touch route-enp0s3
[root@server3 network-scripts]# echo "ADDRESS0=10.1.0.0" >> route-enp0s3
[root@server3 network-scripts]# echo "NETMASK0=255.255.255.0" >> route-enp0s3
[root@server3 network-scripts]# echo "GATEWAY0=10.2.0.10" >> route-enp0s3
[root@server3 network-scripts]# ls -l | grep " route"
-rw-r--r--. 1 root root    60 Aug 17 16:04 route-enp0s3
[root@server3 network-scripts]# cat route-enp0s3
ADDRESS0=10.1.0.0
NETMASK0=255.255.255.0
GATEWAY0=10.2.0.10
[root@server3 network-scripts]#

! Restart the network service and check the routing table
[root@server3 network-scripts]# systemctl restart network
[root@server3 network-scripts]# ip route
10.1.0.0/24 via 10.2.0.10 dev enp0s3 proto static metric 100
10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30 metric 100
[root@server3 network-scripts]#
```

The ping test is now successful, and server2 can reach server3, as shown in Example 3-56.

Example 3-56 *Ping from server2 to server3 and Vice Versa Is Successful Now*

```
[root@server2 network-scripts]# ping -c 1 10.2.0.30
PING 10.2.0.30 (10.2.0.30) 56(84) bytes of data.
64 bytes from 10.2.0.30: icmp_seq=1 ttl=63 time=2.11 ms

--- 10.2.0.30 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.119/2.119/2.119/0.000 ms
[root@server2 network-scripts]#

[root@server3 network-scripts]# ping -c 1 10.1.0.20
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=63 time=1.58 ms

--- 10.1.0.20 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.585/1.585/1.585/0.000 ms
[root@server3 network-scripts]#
```

The network script is run as a service and, like any other service, can be controlled by using the command **systemctl** {**start|stop|restart|status**} **network**. To enable/disable the network service at startup, you use the command **chkconfig network** {**on|off**}. Keep in mind that after a configuration file is changed, the network service has to be restarted for the changes to take effect. It goes without saying that any configuration done via amending the network configuration files is persistent and will remain intact after a system reload.

Network Services: DNS

Domain Name System (*DNS*) is a hierarchical naming system used on the Internet and some private networks to assign domain names to resources on the network. Domain names tend to be easier to remember than IP addresses. Using domain names provides the additional capability to resolve a domain name to multiple IP addresses for purposes such as high availability or routing user traffic based on the geographically closest server.

DNS uses the concept of a *resolver*, commonly referred to as a *DNS server*, which is a server or a database that contains mappings between domain names and the information related to each of those domain names, such as the IP addresses. These mappings are called *records*. DNS is hierarchical and distributed. The majority of DNS servers maintain records for only some domain names and then initiate queries to other DNS servers for the rest of the domain names, for which it does not maintain records.

Performing a DNS query means sending a request to a DNS server to resolve the domain name and return the data associated with that domain name. To resolve a domain name on Linux to its corresponding information, including its IP address, you use the **dig** command, which stands for *domain information groper*. Example 3-57 shows **dig** being used to resolve google.com to its public IP address. The public IP address received from the DNS response is highlighted in the example.

Example 3-57 *Using the* ***dig*** *Command to Resolve google.com*

```
[root@server1 ~]# dig google.com

; <<>> DiG 9.9.4-RedHat-9.9.4-51.el7_4.2 <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38879
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             264     IN      A       216.58.207.14

;; Query time: 31 msec
;; SERVER: 192.168.8.1#53(192.168.8.1)
;; WHEN: Fri Aug 17 17:16:06 +03 2018
;; MSG SIZE  rcvd: 55
[root@server1 ~]#
```

In Example 3-57, the DNS server used for the name resolution is 192.168.8.1. The IP address of this DNS server is configured in the /etc/resolv.conf file, shown in Example 3-58. To configure other DNS servers, you list each server's IP address on a new line in this file.

Example 3-58 *List of DNS Servers in the /etc/resolv.conf File*

```
[root@server1 ~]# cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 192.168.8.1
[root@server1 ~]#
```

Manual DNS entries are configured in the /etc/hosts file. If an entry for a domain name is found in that file, the DNS servers are not consulted for resolution. There is one caveat, though: The **dig** command still requests the name resolution from the DNS server configured in /etc/resolv.conf. However, the **ping** command and also the web browsers on the system use the hosts file, and, therefore, use the manual entry there. Try to add a manual entry for google.com in the hosts file, pointing to an IP address that is not reachable and then try to use **dig**, use **ping**, and browse to google.com and notice how each of these behave differently.

Summary

This chapter takes Linux administration a step further and covers storage, security, and networking. It discusses the following topics:

- Partitioning, formatting, and managing physical storage
- Creating physical volumes, volume groups, and logical volumes using LVM
- User and group security management
- File security management, including permission bits and ACLs
- Linux system security, including the Linux firewall
- Managing Linux networking by using the **ip** utility
- Managing Linux networking by using the NetworkManager CLI (**nmcli**)
- Managing Linux networking via network scripts and configuration files
- Network services such as DNS

Chapter 4, "Linux Scripting," builds on this chapter and covers Linux scripting, which is one big step towards automation.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y - Z