

O'REILLY®

Identity Security for Software Development

Best Practices That Every Developer Must Know

Compliments of



CYBERARK®
A PALO ALTO NETWORKS COMPANY



John Walsh & Uzi Ailon

Identity Security for Software Development

Maintaining secrets, credentials, and machine identities in secure ways is an important, though often overlooked, aspect of secure software development. DevOps security often addresses vulnerabilities, but it neglects broader discussions like authentication, authorization, and access control, potentially leaving the door open for breaches. That's where an identity security strategy integrated in your code, infrastructure, and environments from day one can help.

In this practical book, authors John Walsh and Uzi Ailon provide conceptual frameworks, technology overviews, and useful code snippets to bridge the gap between development, IT, and security to integrate robust identity security across apps, CI/CD pipelines, Kubernetes and cloud native, hybrid and multicloud, process automation, IoT, and more. You'll learn:

- What developers need to know about managing secrets and identity to build safer apps
- What machine identities, secrets, and credentials are—and how to secure them
- How to secure the software supply chain to build safer apps
- How identity security fits into modern software development practices

John Walsh is a software security expert with over 15 years of experience. He is known for clearly communicating the complexities of securing the software supply chain.

Uzi Ailon is a seasoned software engineering professional focused on the business objectives of Fortune 500 companies and the unique challenges enterprises face in balancing innovation with security.

SECURITY

ISBN: 978-1-098-15804-0



9 781098 158040

O'REILLY[®]



CYBERARK[®]
THE IDENTITY SECURITY COMPANY[®]

YOUR IDENTITY SECURITY JOURNEY STARTS HERE

Helping You Build and Deploy Code Securely - Without Sacrificing Velocity

Explore our Developer Resources Now



developer.cyberark.com

Identity Security for Software Development

Best Practices That Every Developer Must Know

John Walsh and Uzi Ailon

O'REILLY®

Identity Security for Software Development

by John Walsh and Uzi Ailon

Copyright © 2025 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Simina Calin
Development Editor: Gary O'Brien
Production Editor: Katherine Tozer
Copyeditor: Doug McNair
Proofreader: Kim Cofer

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2025: First Edition

Revision History for the First Edition

2025-05-06: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098158033> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Identity Security for Software Development*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and CyberArk®. See our [statement of editorial independence](#).

This book may include trademarks and tradenames of third parties. All such trademarks and tradenames are the property of their respective owners. The use of these trademarks and tradenames does not imply any affiliation with or endorsement by such parties.

978-1-098-15804-0

[LSI]

Table of Contents

Preface.....	ix
1. What You Need to Know About Identity Security.....	1
Why Identity Security Matters	1
Zero Trust Identity Security	2
The Modern Enterprise Makes Identity Security Difficult	4
Machine Identity Challenges	5
Common Attack Types	7
Identity Security Concepts	9
Summary	11
2. Secure Coding Practices for Identity Security.....	13
The Zero Trust Model	13
Best Practices	14
General Secure Coding Best Practices	14
Identity Security Best Practices	19
Understanding Security Standards	26
CWE	26
OWASP	26
NIST	27
PCI DSS	27
HIPAA	27
Summary	28
3. Authentication and Authorization.....	29
AuthN	30
Username and Password	30
MFA	31

Token-Based AuthN	31
Biometric AuthN	32
Common AuthN Techniques	32
Risk Analysis During AuthN	38
AuthN Best Practices	40
AuthZ	40
OAuth 2.0	41
JWT	43
Access Control Lists	44
Extensible Access Control Markup Language	45
Summary	46
4. Overview of Identity and Access Management Solutions and Protocols.....	47
Core Components of IAM	48
Identity Management	48
Access Management	49
AuthN	50
AuthZ	50
Identity Governance and Administration	52
Why Does IAM Matter?	53
Consumer Applications	54
Enterprise Applications	54
How IAM Relates to OWASP	55
Identity Lifecycle Management	57
Identity Provisioning and Deprovisioning	57
Role Management	57
Workflow and Approval Processes	58
Identity Synchronization and Reconciliation	58
IAM Architecture Models	59
Centralized IAM Model	59
Decentralized IAM Model	60
Hybrid IAM Model	61
IAM in Cloud Environments	62
Key Standards in IAM	63
SAML: Centralized AuthN for SSO	63
OAuth 2.0: Securing Third-Party Access to Resources	64
OIDC: Adding Identity to OAuth 2.0	64
LDAP: Centralized Directory Services	65
System for Cross-Domain Identity Management:	
Simplifying User Provisioning	66
FIDO: Passwordless AuthN for Strong Security	66
Emerging Trends in IAM	66

AI in IAM	67
Blockchain in IAM	67
Summary	68
5. Secrets Management.....	69
Why Does Secrets Management Matter?	71
Principles of Secrets Management	73
Principle #1: Encryption	73
Principle #2: Access Control	74
Principle #3: Monitoring and Auditing	74
Principle #4: Compliance	75
Principle #5: Testing	76
Principle #6: Automation	76
Principle #7: Centralization	77
Secrets Management in Code	78
Secrets Management Tools and Frameworks	79
Setting Up a Secrets Policy	80
Creating a Secrets Store by Using ESO	81
Setting a Secret	82
Retrieving a Secret	82
Summary	83
6. Cloud Security and Cloud Native Considerations.....	85
Background on Cloud Computing and Security	86
Watch Out for Security Misconfigurations	88
Cloud Native Versus Lift and Shift	90
The Four Cs of Cloud Native Security	91
How Different CSPs Structure Resources	93
Guiding Principles for Securing Cloud Applications	96
Comparing IAM Services	98
Permissions	98
Access	100
Switching Between CSPs	101
Summary	103
7. Securing Kubernetes.....	105
How Kubernetes Works	107
Kubernetes Security Challenges	108
Secrets Management in Kubernetes	109
Best Practices for Kubernetes Security	112
Use Service Accounts Diligently	112
Leverage the Kubernetes RBAC Framework	113

Securing Your Pods and Containers	116
Ensure Network Security	123
Identity Management on Kubernetes Using SPIFFE	127
Using cert-manager for TLS Identity Security Automation	129
Using Service Meshes to Secure Clusters	132
Identity Management in Service Meshes	133
AuthZ in Service Meshes	133
Trading Off Performance and Security	135
Service Mesh Best Practices	135
Summary	137
8. Security Automation	139
Why Does Security Automation Matter?	140
Types of Security Automation	141
What Security Processes Can Be Automated?	142
Security Automation at Work	143
Security Automation Tools	144
Security Automation Best Practices	148
Infrastructure as Code	149
Policy as Code	151
Secure Automation Tools	153
Security Automation with AI	153
Example: Security Automation with Ansible and Conjur Open Source	154
Summary	157
9. CI/CD Pipeline Security and Software Supply Chains	159
Why Does Securing Your CI/CD Pipeline Matter?	161
Securing Secrets in a CI/CD Pipeline	161
A Jenkins Example	161
What OWASP Says About CI/CD Security	162
Foundations of CI/CD Security	166
Source Code Threats	167
Hardening the Build Process	170
Managing Artifacts	171
Integrating Security Testing into CI/CD Pipelines	172
Monitoring and Incident Response	175
Provenance, Attestations, and Digital Signatures	176
Summary	179
Index	181

Preface

If you're a developer, an engineer, or someone working alongside development teams, chances are you're already making identity security decisions—whether you realize it or not. The rise of cloud services, automation, and AI-driven tools has woven identity security into the fabric of modern development workflows. Developers are provisioning cloud resources, integrating APIs, managing service accounts, and handling secrets—often without considering the security implications.

While AI and automation accelerate software development—making it easier than ever to generate code, configure infrastructure, and deploy applications at scale—they also introduce new risks. Embedding credentials in generated scripts, configuring access policies for AI-assisted deployments, and managing machine identities across automated pipelines are all identity-related decisions that can either strengthen or weaken security. The challenge is that many developers don't always realize they're making security-critical choices until something goes wrong.

This book is here to help you *know what you don't know* about identity security. It's designed for developers and the *dev adjacent*—professionals who are code literate or at least not afraid of diving into some code. Rather than turning you into a security expert overnight, this book will uncover the gotchas, pitfalls, and hidden risks that could leave your applications exposed.

But this book isn't just for developers. Security teams can use it to better understand how developers work, the challenges they face, and why identity security decisions so often fall into their hands. Too often, security and development teams operate in silos, struggling to collaborate effectively. Security wants to reduce risk, and developers want to ship software quickly—but these goals don't have to be at odds. By understanding identity security from a developer's perspective, security teams can offer more relevant guidance, start meaningful conversations, and work together with development teams to build secure software *without slowing things down*.

Identity security isn't just about users logging in. Machines, applications, services, and workloads all have identities, and securing them is just as critical as securing human access. Attackers know this—API keys, cloud credentials, and machine identities are prime targets for breaches because they often provide a direct path to sensitive systems. Yet, many development teams don't fully recognize these risks—or only think about them when something goes wrong.

The problem isn't that developers don't care about security. It's that identity security has often been treated as someone else's job. But as software development accelerates, identity-related decisions are increasingly landing in the hands of developers—whether they're prepared for it or not.

This book won't drown you in abstract security theory. Instead, it provides a broad and practical perspective on securing identity in software development. It will give you a solid foundation and help you integrate security into your development workflows without slowing you down.

By the time you finish, you won't just have a better understanding of identity security—you'll also recognize the risks you didn't even know were there. Whether you're a developer trying to secure your applications or a security professional looking to engage with developers more effectively, this book will help you bridge the gap.

The lines between developers, IT/operations, and security blur every day. We have a shared responsibility to safeguard the integrity, confidentiality, and availability of the systems we create. This book, *Identity Security for Software Development*, is your guide to developing software with security at its core and identity built into the fabric of its code.

Our Approach

We wrote this book to equip developers, engineers, and security teams with the knowledge and tools they need to secure identities from the first line of code to the last push to production. Identity security isn't just another security checkbox—it's a fundamental part of building resilient, secure applications.

Our goal is to provide you with the knowledge, strategies, and practical techniques you need to embed security into every layer of your applications. We cover both the fundamentals of secure coding and the subtleties of identity security, focusing on real-world approaches—from secure authentication and authorization mechanisms to protecting sensitive data and mitigating common threats.

Each chapter in this book covers a specific area of identity security for software development—authentication, authorization, secrets management, machine identity, continuous integration and continuous security (CI/CD), cloud security, and Kubernetes security, among others. Any of these topics could easily provide enough subject

matter for a book on its own, but the goal here isn't to provide technical deep dives. Instead, this book is designed to give those who are new to identity security a strong foundation, helping them understand best practices, recognize risks they might not have considered, and avoid costly mistakes before they happen.

We recognize that not all readers will follow the same path. If you're new to identity security, we recommend that you start with the foundational chapters before exploring more advanced topics. If you're facing a specific challenge—like securing API keys, managing credentials in Kubernetes, or integrating identity into a CI/CD pipeline—you can jump straight to the relevant chapters without needing to read the entire book cover to cover.

This book is also meant to bridge the gap between security and development teams. Security professionals can use it to better understand how developers work, where identity security fits into modern development workflows, and how to provide guidance that speeds up rather than slows down development. By fostering collaboration, we aim to help organizations build secure software without slowing down innovation.

Ultimately, identity security isn't just about preventing breaches—it's about designing applications that are secure by default, resilient to attack, and built with security woven into every stage of development. This book will help you get there.

Who Should Read This Book?

This book is intended for developers, engineers, and dev-adjacent professionals who are involved in building, deploying, and securing software. Whether you're a coder just starting your journey, a seasoned developer, a DevOps engineer managing cloud infrastructure, or a security professional working with development teams, this book will help you understand identity security in the context of modern software development.

Many developers and engineers already interact with identity security—often without realizing it. As automation, AI, and cloud native technologies accelerate development, identity-related security decisions are becoming part of daily workflows. This book will help you recognize those security-related moments, understand best practices, and avoid common mistakes before they lead to security incidents.

At the same time, security professionals can use this book to gain a developer's perspective on identity security—which will help them understand where identity risks emerge in the software development lifecycle, how development teams approach security, and how to collaborate more effectively with them to embed security into development workflows without slowing down innovation.

This book also is for anyone who wants to do the following:

- Develop a strong foundation in identity security—covering authentication, authorization, secrets management, and machine identity.
- Build secure, resilient software systems that integrate security into development workflows without adding unnecessary friction.
- Safeguard user identities and sensitive data in modern applications.
- Mitigate common identity-related vulnerabilities and security threats that affect software development.
- Understand how to secure identities in cloud environments, CI/CD pipelines, and Kubernetes workloads.
- Bridge the gap between security and development teams, fostering better collaboration to build secure software faster.

No matter your role, if you're involved in writing, deploying, or securing software, this book will help you recognize and address identity security risks before they become problems.

What You'll Find Inside

We've structured this book to provide both conceptual knowledge and practical guidance. You'll find coverage of the following topics:

Foundations of secure coding

We start with the basics, laying a strong foundation in secure coding principles, practices, and standards used by developers worldwide.

Identity security

We expand the discussion into identity security, covering topics such as authentication (AuthN), authorization (AuthZ), single sign-on (SSO), and multifactor authentication (MFA).

Data security

We teach you how to protect sensitive data through encryption, access controls, and secure storage practices.

Common vulnerabilities

We explore real-world examples of common security vulnerabilities and teach you how to prevent them.

Security throughout the software development lifecycle

We cover how to integrate security into your development process, from design to deployment, including security automation and CI/CD pipelines.

Cloud security considerations

We teach you about cloud-specific security threats, mitigation strategies, and how different cloud providers support identity security.

Kubernetes security

We also discuss best practices for securing Kubernetes workloads and service meshes.

Finally, remember that you aren't alone. This book teaches you how to collaborate with the relevant members of your organization, and it will serve as your reference for building safer apps faster.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a general note.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/identity-security-for-software-dev>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>.

Watch us on YouTube: <https://youtube.com/oreillymedia>.

Acknowledgments

Writing this book has been a journey, and there have been many contributors along the way. We've been fortunate that partway through the writing process, Jetstack® (the original creator of Kubernetes cert-manager) and its technical team became part of CyberArk. Jetstack's expertise with development and platform engineering teams has helped shape this book and pushed us to make it more useful to developers and platform engineers. While many people made huge contributions, we especially want to thank Steve Judd (for his work on the Kubernetes and CI/CD chapters) and Mattias Gees.

To the O'Reilly reviewers—Vladislav Bilay, Niolet D'mello, and Marcos Mazoti—thank you for your feedback and expertise. We sincerely appreciate the deep technical reviews and insights from our many colleagues: Alyssa Miles, Ashley Davis, Brandon McCaffrey, Chad Miller, Dan Zadik, Darren Khan, David Lang, Evan Litwak, Ivan Wallis, Josh Kirkwood, Maor Franco, Mark Szalkiewicz, Mike Bykat, Przemek Dybowski, Racheli Lotvin, Shlomi Benita, Shlomo Heigh, and Tim Parmenter. And to Roy Sarkar, thank you for sharing your spidey-sense.

Finally, a special thanks to our executive sponsors, Kurt Sand, Kevin Bocek, Charles Chu, Archit Lohokare, and Simon Mouyal. Also, thanks to Chris Smith, for his sponsorship and guidance, and Laura Lasonde, for keeping us all on track.

What You Need to Know About Identity Security

When rideshare company Uber was hacked in 2022, it came close to a total system shutdown. As in many security breaches, hackers exploited a series of human and machine weaknesses to access Uber's internal systems. In one key incident, an Uber employee, frustrated by a flood of multifactor authentication (MFA) messages sent to their phone by the hacker, granted access approval to Uber's network, **.corp.uber.com*.

Once inside, the hacker discovered **Microsoft PowerShell scripts with hardcoded administrative credentials** that gave them access to several sensitive services, including Amazon Web Services (AWS), Google Drive, Slack, SentinelOne, and the HackerOne admin console. Within hours, the hacker announced the breach on the company's own Slack channel and posted screenshots to public sites like HackerOne.

While nothing was stolen, had the hacker sought financial gain, this breach would have had devastating impacts. Today, it serves as a textbook case of the power of social engineering and the consequences of multiple failures of identity security. Attackers using a single exploit and moving through a network by stealing identities is a pattern that has happened many times before and will happen again.

If you were a developer at Uber, what would you change?

Why Identity Security Matters

Organizations have dramatically increased the quantities and types of identities to support cloud computing, DevOps automation, the Internet of Things (IoT), AI, and other advancements in digital services. As a result, more identities and more use cases mean you must secure and manage more credentials, secrets, and accounts than ever before.

Importantly, the number of machine identities will only continue to grow. The CyberArk 2025 Identity Security Threat Landscape Report found that machine identities outnumber humans by a factor of 82 to 1, and this disparity will grow exponentially with the emergence of new AI and AI agent products. The potential volume of unprotected identities cannot be ignored.

Identity security is also a sought-after skill because attackers are getting very good at what they do. According to the *2023 Trends in Securing Digital Identities report* from the Identity Defined Security Alliance (IDSA), 90% of organizations experienced at least one identity-related breach in the past year, and 68% of those surveyed said the attacks directly impacted their business. These impacts include the costs of recovering from the breach and the negative impact that the breach had on the company's reputation.

Zero Trust Identity Security

Traditional barriers, boundaries, roles, responsibilities, and safeguards have melted away so these days, software needs to be written with zero trust in mind. This means that developers can no longer assume that no one will see the code in their integrated development environment (IDE), on GitHub, on their drive, in their test environment, or even in their continuous integration and continuous delivery/deployment (CI/CD).

This also means that developers are a key part of the process of protecting the software supply chain and the enterprise as a whole—so it isn't the responsibility of security teams alone. Identity security is how developers, quality assurance (QA), IT, and other teams within an organization secure all identities within their code, development environments, and software supply chains.

Zero trust identity security requires ongoing verification of identity and device health, and it can use behavior analysis to detect anomalies, especially for privileged identities. Whether packaged as a tool, a framework, or a set of best practices, identity security assumes that any organizational asset can be compromised through a malicious attack path centered on privileged identity. If you've entered a password or used MFA, you've seen identity security in action.

You don't trust anyone, that's your problem.

—Peter Parker (aka Spider-Man)

I trust my barber.

—J. Jonah Jameson

What Peter criticizes in the 2002 *Spider-Man* movie, we say with conviction: identity security must be rooted in skepticism rather than blind trust. *Zero trust* is a security philosophy in which practitioners assume that an application or system is always

vulnerable to attack and do their best to protect against the many ways in which malicious actors operate.

This zero trust challenge gets harder as we see more machine identities in use. The growth in automation and an increasingly complex software supply chain mean entities use more secrets (e.g., passwords, tokens, Secure Shell [SSH] keys, API keys, license keys) to access critical and sensitive data and systems. This means you must figure out where these secrets are used and how to best protect them against unauthorized access.

Table 1-1 provides examples of the two categories of identities.

Table 1-1. Examples of human and machine identities

Human identities	Machine identities
End users	Cloud native and containers applications
IT administrators	DevOps tools and CI/CD pipelines
Developers	Automation scripts
Contractors	AI and AI agents
Vendors	IoT devices
QA testers	Third-party commercial-off-the-shelf including robotic process automation (RPS)
Customer prospects	Secure n-tier/Static homegrown apps and web apps
Grand Moff Tarkin	Imperial code cylinders ^a

^a While fictional, these *Star Wars* props provide a good example of secrets encoded onto a portable device used to access sensitive areas.

If you can identify the security issue in the following code snippet (taken from the **MITRE Common Weakness Enumeration database**), then you have an idea of the issues that an identity security program aims to prevent:¹

```
## Java
int VerifyAdmin(String password) {
    if (!password.equals("Mew!")) {
        return(0)
    }
    //Diagnostic Mode
    return(1);
}
```

Managing human and machine identities is a critical aspect of a security team's responsibilities. Examples of human identity management include the following:

¹ It's not the syntax error of the missing semicolon. (We swear we copied the source site directly!) It's the hardcoded "Mew!" in the password check. If this were exposed, hackers would have the correct password to exploit the system.

- Authentication (AuthN) and authorization (AuthZ)
- User lifecycle management
- Privileged access management (PAM)
- Zero standing privilege

Machine identity management includes the following:

- Secrets management
- Certificate management
- Discovery
- Strong AuthN mechanisms

The Modern Enterprise Makes Identity Security Difficult

Remember when ChatGPT was taken offline? In early 2023, [OpenAI shut down the popular chatbot](#) in response to a cybersecurity breach. Traced to a user AuthZ bug in Redis, the breach was caused by a data race condition between two message queues: if a user canceled a request within a certain time frame, the system delivered their information to the next user who made a similar request. This information could include the user's first and last name, email address, payment address, and credit card type, as well as the last four digits of their credit card number and credit card expiration date.²

This example illustrates how the modern enterprise has changed the nature of identity security. Explosive growth in the software supply chain means there are more opportunities for risk. Automation, DevOps, cloud computing, the IoT, and AI are reshaping our jobs, and the demand for broader collaborative skills has outpaced the need for focused practitioners of one programming language.

Your career in the modern enterprise demands adaptability and versatility, collaborating across departments, and performing tasks that traditionally fell under separate roles. Consider [Glassdoor's top three interview questions for enterprise developers](#):

² As explained by OpenAI, the solution included the addition of redundant checks (to ensure that the data returned by Redis matched the requesting user) and a programmatic examination of logs (to ensure that only the authorized user received their messages).

- Can you walk me through your development process?
- How do you feel about working as a member of a team?
- What would you do to improve our company?

Companies today look for a broader skill set and a better understanding of areas outside pure coding and testing. Also, as enterprises favor smaller and leaner development teams, they're implementing agile processes, automation, and DevOps to achieve more with less. If the words *efficiency* and *productivity* aren't in your development vocabulary, they soon will be.

Our reliance on increasingly complex technologies and a diverse software supply chain requires us to take a cross-functional approach to security that ensures everyone knows how to mitigate threats earlier, at the source. As someone who writes code and selects libraries for integration into production systems, you must understand the impacts of your work on the company's security posture.

Ask yourself questions like, "Where are libraries coming from, and do I trust their level of security testing? Am I setting up access permissions sufficiently, and is code signing configured correctly?"

Machine Identity Challenges

Security organizations understand the critical need to protect all identities—both human and machine.

While the path to protecting human identities is relatively clear, securing machine identities presents far greater challenges that are often summarized by the *three Vs*:

Volume

Organizations have a large number of machine identities that are growing every day.

Variety

Machines and workloads are diverse. They run on premises or in the cloud, and they may be homegrown or provided by a third party. Also, all applications must be identified and secured using different solutions, such as certificates, API keys, or tokens.

Velocity

To support machine identities, organizations need a solution capable of managing the scale and performance demands of modern environments.

There is one more *V*: *visibility*. In many cases, security teams lack visibility into the machine identities within their organization and across different environments.

As you read this book, consider how the tools and techniques discussed meet these organizational needs:

- *Scanning and observability capabilities* to discover and monitor all machine identities across the organization
- *A centralized repository* to manage all machine identities in one place that's easier to manage and maintain
- *Support for different applications across various environments* to ensure compatibility and seamless integration
- *Scalability and high performance* to meet the demands of workloads and applications efficiently

The Real-World Impact of Identity Security

The **Codecov breach** was a supply chain attack in which threat actors exploited infrastructure vulnerabilities to gain potential access to the networks of thousands of Codecov's customers. By compromising Codecov's Bash Uploader script, the attackers were able to trigger the sending of environment variables from the CI pipelines of Codecov customers to a remote server. Among other potential attack vectors, this allowed the hackers to access private Git repositories from unsecured Git credentials stored in a customer's CI environment and to exploit the secrets within.

While the impacts are troubling enough, the timeline of events illustrates a dangerous subtlety behind modern identity security attacks:

1. January to April 2021: attackers successfully manipulated Codecov's Bash Uploader script.
2. April 1, 2021: a Codecov customer noticed that the Bash Uploader had a different hash value than what was published on Codecov's website, indicating something was wrong.
3. April 15, 2021: Codecov investigated and remediated the issue.
4. April 29, 2021: **Codecov publicly disclosed the breach** and urged users to take immediate action.

The systems of Codecov users were potentially exposed for months before the breach was closed, raising concerns about supply chain security and forcing developers to rethink how and where their secrets are stored. In the Codecov scenario, the combination of small security failures between customers and the supplier added up to a big overall impact.

This serves as a reminder that modern enterprises must be extra diligent about who gets access to systems. This includes humans and machines alike, both internally and

externally to the organization. After this attack, developers were advised to secure their systems in the following ways:

1. Eliminate any occurrence of credentials and secrets stored in environment variables in favor of using a secrets manager to ensure that only authenticated users can access clear text data.
2. Perform code signature checks to verify the digital fingerprint of software in the supply chain.
3. Mandate MFA to require two or more AuthN checks before granting access to secure resources.

Common Attack Types

Among the most prevalent security risks are sophisticated attacks that leverage vulnerabilities in both human behavior and technology infrastructure. Five common types of cyber threats are as follows:

Pass the hash

The hacker intercepts a hashed version of a user's password and uses it directly, without deciphering the actual password. This usually happens when the attacker gains access to the password hash through various means, such as intercepting it on the network.

SQL injection

An attacker injects malicious SQL code into the input fields of a script interface or web application. If the application doesn't validate or sanitize user inputs, the injected code can manipulate database queries, potentially allowing unauthorized access or data manipulation. Injection attacks are **high on the OWASP Top Ten list** of the industry's most critical security risks to web applications.

Social engineering

This is a tactic whereby attackers exploit human psychology to manipulate individuals into divulging confidential information or performing actions that could compromise organizational security. This can involve impersonation, building trust, or exploiting human tendencies.

Phishing

This is a type of social-engineering attack in which attackers create fake emails or instant messages by pretending to be a known or authoritative contact. They do this to trick individuals into revealing sensitive information or performing actions that could compromise their organization's security.

Privilege escalation

This is the process of gaining higher-level access and permissions on a system or network than originally granted, such as by exploiting misconfigured networks, using malware, or using stolen credentials. This typically happens after the attacker has gained access to some part of the system, and the attacker uses it to increase the blast radius of the attack.

Understanding Identity Security Lingo

In addition to the concepts directly related to identity security, here are some key terms we use throughout this book:

Blast radius (of a security breach)

The total extent of damage an attacker can cause once they gain unauthorized access.

The common vulnerabilities and exposures (CVE) database

Catalogs publicly disclosed vulnerabilities. It helps security, IT, and development professionals coordinate efforts and prioritize work to address vulnerabilities, thus ensuring everyone has access to the same information. Each CVE vulnerability is referred to by a CVE ID number.

Common weakness enumeration (CWE)

A standardized list of software and hardware security weaknesses maintained by the MITRE Corporation. It helps developers and security professionals identify, classify, and mitigate common vulnerabilities in code.

Distributed denial-of-service (DDoS) attack

Floods a system, server, or network with excessive traffic from multiple sources, overwhelming it and disrupting normal operations. Attackers often use *botnets*—networks of compromised devices—to amplify the attack.

Exploit

Code or a technique that takes advantage of a software vulnerability to gain unauthorized access or execute malicious actions. Exploits are often used in cyberattacks to compromise systems and steal sensitive data.

Honeypot

A decoy system or network that attracts attackers, allowing security teams to observe and analyze their tactics. It helps organizations identify emerging threats and improve their defenses.

The National Institute of Standards and Technology (NIST)

A United States (US) government agency that develops cybersecurity standards, guidelines, and best practices. The **NIST Cybersecurity Framework (CSF)** and **NIST Special Publication 800-53** are widely used for security compliance and risk management.

The National Vulnerability Database (NVD)

A US government repository of vulnerability management data represented using the Security Content Automation Protocol (SCAP). It provides detailed information on vulnerabilities across a wide range of software and hardware, and it supplements the information provided by the CVE database.

The Open Worldwide Application Security Project (OWASP)

A nonprofit organization that produces free resources, tools, and frameworks to help developers secure web applications. The **OWASP Top Ten list** identifies the most critical security risks in software development.

Phishing

A social engineering attack in which attackers impersonate trusted entities, usually via email or text, to trick victims into revealing sensitive information, such as passwords or financial details. More advanced forms include *spear phishing* (targeted attacks) and *whaling* (attacks on high-profile individuals).

Social engineering

Manipulates human psychology to deceive individuals into disclosing confidential information or performing security-compromising actions. Common tactics include phishing, pretexting, baiting, and impersonation.

Identity Security Concepts

Your identity security journey will be most successful if you know how to foster a holistic approach to securing all types of secrets across systems—including your development activities. Hackers are forcing security perimeters deeper inside organizations, so you not only have to secure external APIs, but you must also account for internal and remote users. You must also build tools, microservices, employee devices, third-party cloud services, acquired software, and more.

Let's review the essentials of identity security to help you understand the purpose and relevance of each essential before we go into more detail. Some of these may be familiar to you if you have knowledge of general cybersecurity principles:

Authentication (AuthN)

This is the process of validating a user's identity, such as by usernames and passwords. You may have used tools like signed digital certificates to improve application protections against unauthorized access.

Authorization (AuthZ)

This is the method of granting access to specific resources once a user's identity is verified. For example, your infrastructure lead may have root-level access to all build servers, while junior developers may have their access restricted or disallowed completely.

Code security testing and validation

These are processes for testing, measuring, proving, and improving the resilience of code against cybersecurity attacks. Techniques like penetration testing and code reviews help answer the question, “Is my code vulnerable to attack?”

Data protection

Whether at rest, in use, or in transit, sensitive and classified data must be kept secure at all times to minimize exposure in case of a breach. The core exposure behind the Uber breach of September 2022 was the presence of hardcoded credentials in a misconfigured network share. Harvesting these credentials allowed the attacker to gain high-level access, escalate privileges, and cause chaos inside Uber’s IT environment.

Monitoring and logging

These are techniques that support continuous auditing, measurement, analysis, and adaptations of systems to identity security threats. Monitoring applications offer insights into performance and security-related events, facilitating the identification of anomalies that require further attention. Logging provides a historical record of system actions, which is invaluable for tracing the origin of security breaches, improving incident response, and ensuring compliance.

Software supply chain security

Anything that touches an application or plays a role in application development is considered part of the software supply chain and must be secured. This includes deployment infrastructure, APIs, development tools, and third-party, proprietary, and open source code. The **SolarWinds breach**—in which over 18,000 public-sector and private-sector organizations were impacted by hackers exploiting access to systems with privileged account credentials—illustrates the potential wide-reaching impact of vulnerabilities in any element of the software supply chain.

Transport layer security and secure connections

All AuthN and AuthZ flows should occur over secure encrypted connections, like transport layer security (TLS). Even when identity management is done correctly, failing to encrypt communications can expose credentials to interception by attackers (e.g., through man-in-the-middle attacks).

While reading this book, you should think about whether your team is applying these principles and techniques appropriately now. If it isn’t, perhaps there’s something here that they should consider implementing.

Summary

In this chapter, you learned the basics of identity security to prepare you for the following more detailed chapters. Three overarching security principles will come up repeatedly:

- Identity security matters because identities and use cases will only continue to grow and must be protected from malicious use.
- Zero trust, which means you should implement security principles assuming your application will always be vulnerable to attack.
- Security is a shared responsibility between you, your team, and all stakeholders.

Secure Coding Practices for Identity Security

Before getting into the strategies and methods of implementing identity security, it helps to understand the best practices for securing identity access within applications. Code security is the first line of defense against a malicious attack, and it encompasses more than just writing, compiling, and testing applications. Robust practices should foster strong defenses on every level of the organization, from every line of code to automation scripts to a secure development environment that makes identity security the central focus for human and machine actors alike.

Developers must pick and choose the best practices that make the most sense for their organization and project. An ecommerce application may require compliance with payment security standards like the Payment Card Industry Data Security Standard (PCI DSS), while a health care provider may need to comply with the Health Insurance Portability and Accountability Act (HIPAA) guidelines. Keeping these practices in mind throughout the development process helps you identify and remediate potential issues before they make it into a production application.

The Zero Trust Model

The days of focusing solely on a strong perimeter defense are over. The shift to cloud computing, microservices, open source, AI-generated code, and IoT has broken our reliance on the single wall of security protection offered by firewalls, API gateways, and similar technologies. Now, we must protect access to internal applications and infrastructure just as much as external interfaces. This challenge of creating a holistic enterprise security program led to the development of zero trust models.

Zero trust, at its core, roots a team's security culture in skepticism rather than blind trust. It involves the meticulous validation of human and machine actors, both internal and external, against pre-established security standards. Rather than presume a script, application, or system is invulnerable, zero trust recognizes that it's impossible for developers to protect against the many ways in which a complex system can be exploited.

By advocating that no actor can be trusted until they're verified, a zero trust model strengthens protection against potential breaches at the deepest levels of application code. Like the coach of a sports team, who knows that a single misstep can disrupt an entire play, zero trust guides people, processes, and tools to secure every aspect of the enterprise, including identities, endpoints, applications, data, infrastructure, and networks.



For an example of a zero trust model, the [NIST special publication on zero trust architecture \(ZTA\)](#) defines seven basic tenets for zero trust, replacing the old concept of a monolithic, enterprise-wide security perimeter with a more granular and comprehensive approach.

Best Practices

Given all possible combinations of technologies, application designs, IT environments, external interfaces, and coding decisions, it's impossible to define a concise list of best practices that cover them all. However, we can help you improve code security by identifying the most common techniques that address the most frequent types of attacks that exploit human and machine interfaces with the outside world.

To make referencing this information easier, we've split out the general secure coding best practices from the ones specific to identity security.



The examples presented under each best practice are not exhaustive as it's not feasible to cover all use cases and application environments in this book.

You can make use of other books and secure coding websites, like [OWASP](#), to dig deeper into specific techniques and technologies not listed here.

General Secure Coding Best Practices

Whether it be Java or Python, you should assess source code for security vulnerabilities early in the development cycle to catch potential issues before they become real threats. You have to know where the code is coming from to best validate its security posture.

It's good to keep the following five practices in mind when developing and testing your applications.

Input validation

Input validation is the testing of user or application input against expected values to ensure proper data integrity and guard against malicious data entry. This technique ensures that input is within expected ranges, uses the correct data types, and does not contain illegal characters or special characters that may be used to bypass other security controls.

Without proper input validation, applications become vulnerable to attacks such as structured query language (SQL) injection, cross-site scripting (XSS), and buffer overflows. By applying strict validation rules, you can reduce the likelihood that malformed or malicious input will compromise application integrity or bypass security controls.

A common misconception is that applications should only validate input on the client side, which leads people to believe the myth that any traffic from the client is valid. The reality is that client-side validation alone is insufficient—think of browser vulnerabilities or man-in-the-middle attacks. To mitigate these risks, you should also perform input validation on the server side, ensuring that data remains secure even if the client is compromised. This approach aligns with the zero trust model, which assumes no input should be trusted by default.

The following function validates a username, ensuring that it consists only of alphanumeric characters and is between 4 and 20 characters long:

```
function validateUsername(username) {
  const regex = /^[a-zA-Z0-9]{4,20}$/;
  if (!regex.test(username)) {
    throw new Error("Invalid username. It must be 4-20 characters long and
    contain only letters and numbers.");
  }
  return true;
}

try {
  console.log(validateUsername("ValidUser123")); // Passes validation
  console.log(validateUsername("Invalid!User")); // Throws an error
} catch (error) {
  console.error(error.message);
}
```

This function blocks malicious inputs that contain special characters or overly long strings, thus reducing the risk of injection attacks. While this demonstrates client-side validation, a similar validation process should be implemented on the server side to ensure complete protection.

Encryption

Encryption transforms data into a scrambled format that can only be deciphered by using an appropriate decryption key. Strong encryption methods help prevent unauthorized access, even if attackers gain access to storage systems or intercept network traffic.

Encryption applies to data at rest (stored data) and data in transit (data being transmitted over a network), as follows:

Encryption at rest

This protects stored data by encoding it with cryptographic algorithms, which ensures that even if a storage device is compromised, the encrypted data remains unreadable without the correct decryption key. Common implementations include full-disk encryption, database encryption, and encrypted backups. For example, enterprise databases use AES-256 encryption to secure sensitive fields like passwords, credit card numbers, and personally identifiable information (PII).

Encryption in transit

This secures data as it moves between systems, thus preventing interception or tampering by attackers. *Transport layer security* (TLS) is widely used to encrypt web traffic, thus ensuring confidentiality and integrity. For instance, HTTPS relies on TLS to protect user credentials and sensitive transactions from being stolen or altered during transmission.

Encryption security depends on how well encryption keys are managed. If an attacker gains access to an encryption key, they can decrypt sensitive data and thus render encryption useless. Best practices for protecting encryption keys are as follows:

- Storing keys in *hardware security modules* (HSMs) or dedicated key management systems.
- Restricting access to encryption keys by using strict access control policies.
- Using key rotation to periodically replace encryption keys with new versions to minimize risk. If a key is compromised, rotating to a new key limits potential damage. For example, cloud service providers often implement automatic key rotation to ensure data remains secure over time.

AES-256 is currently one of the strongest encryption standards and is resistant to brute-force attacks. However, the growth of quantum computing poses a potential threat to traditional encryption. Quantum computers could break widely used public-key cryptographic methods like the Rivest-Shamir-Adleman (RSA) encryption algorithm and *elliptic-curve cryptography* (ECC) by solving mathematical problems much faster than classical computers. To counter this, researchers are developing *post-quantum cryptography* (PQC)—which are encryption algorithms designed to

withstand quantum attacks. Some emerging quantum-safe encryption techniques include lattice-based cryptography and hash-based signatures.

Resource management

Securing computing resources used by applications contributes to an effective defense. Resources that need securing include databases, files, and memory because they offer potential access points for attack.

To secure databases, developers should use techniques like strongly typed parameterized queries, input validation, and output encoding to fend off malicious external entities. If these measures fail, the application should log the attempt and not execute database commands.



A *parameterized query* uses placeholders for parameters, and these values are supplied at execution time. These queries help prevent security vulnerability to *SQL injection attacks*, in which malicious actors manipulate the query by adding malicious code. By parameterizing the query, the database knows to treat the input as literal values and not as an executable SQL command.

For file management, requiring authentication (AuthN) before file upload, validation of file types, and restricting file types to business requirements all help mitigate risks.

Applications that directly manipulate memory should enforce input and output control for untrusted data, validate buffer boundaries to protect against overflow, and explicitly close resources rather than rely on garbage collection.

Session management

Securely handling multiple requests to a web-based application or service falls under the domain of session management. In a typical secure interaction between a website and a browser, a session starts when the user authenticates their identity using a password. It is the responsibility of the application to ensure controlled access to resources throughout the session's lifecycle, and similar principles apply to applications and microservices invoking other applications and microservices.

Effective session management guards against unauthorized access, data leakage, and identity breaches. In a banking application, for example, proper session management ensures that a user's access to their account and transactions is secure by blocking unauthorized access through strict AuthN processes and timed session expirations.

Code obfuscation

Code obfuscation is a technique that makes application code harder to understand and reverse engineer by transforming it into a complex, unreadable format without

altering its behavior. Obfuscation prevents attackers from attempting to decompile or disassemble compiled applications to analyze logic, extract sensitive information, or inject malicious code.

Many programming languages, such as Java, compile source code into an intermediate format (e.g., Java bytecode) that can be easily decompiled into a human-readable form. To mitigate exposure risks, you can use obfuscation techniques like renaming variables and methods, inserting dead code, and encrypting critical sections. A more advanced technique is runtime encryption, in which code is decrypted just before execution, making it more difficult to analyze statically.

Consider this simple Java application:

```
public class Secret {
    public static void main(String[] args) {
        System.out.println("API Key: 12345-ABCDE");
    }
}
```

Once compiled, an attacker can use a Java decompiler, such as [JD-GUI](#) or [Procyon](#), to extract the original source code:

```
public class Secret {
    public static void main(String[] args) {
        System.out.println("API Key: 12345-ABCDE");
    }
}
```

As you can see, the decompiled output is identical to the original code.

We recommend using an automated tool rather than doing it yourself when obfuscating code. This will prevent the introduction of unexpected bugs into your application. Consider the following two code samples, with the latter showing manually obfuscated code:

```
// Original code
function calculateTotal(items){
    let total = 0;

    items.forEach(item => {
        const discount = getDiscount(item);
        total += item.price * (1 - discount);
    });

    return total;
}

// Obfuscated code with bug
function _calc(i){
    let d = 0; // Reused variable name 'd' causes shadowing

    i.forEach(x => {
```

```

        const d = _disc(x); // 'd' shadows outer 'd'
        d += x.p * (1 - d); // Bug: Modifies local 'd' instead of outer 'd'
    });

    return d;
}

```

The second sample introduces a potential memory bug that may alter the behavior of your application. If obfuscation is necessary, you should use a stable version of a reliable obfuscation tool and perform all possible tests on the obfuscated code.

Identity Security Best Practices

Identity security is all about ensuring that each person and machine is who they say they are and is authorized to access sensitive information. It supports giving them the proper permissions and providing them access to sensitive information, APIs, and resources in ways that are deliberate and traceable.

Basically, if you can't authenticate it, you shouldn't authorize it. The following practices encompass these goals and set the stage for the rest of this book.

Authentication (AuthN)

AuthN is the process of validating a user's identity so that they can gain access to a system. It answers the question, "Who are you?" by confirming the authenticity of a user's credentials—human and machine—to ensure they are who they claim to be.

Implementing AuthN involves tools and techniques such as the following:

Username and password

This is the most basic and familiar form of AuthN, whereby users input a unique username and a corresponding password to gain system access. Developers must ensure the secure storage of passwords, thus avoiding common pitfalls like plain-text storage that is easily exploitable by hackers.

Biometric AuthN

This uses fingerprints, facial recognition, and other biometric markers to confirm a user's identity.

JSON web tokens

A *JSON web token* (JWT) is a mechanism for AuthN and authorization (AuthZ) that asserts information about the user and uses encryption to ensure data integrity.

Two-factor authentication

Two-factor authentication (2FA) combines two different AuthN methods, like a password and a one-time code sent to the user's mobile device, to strengthen identity verification.

We dive deeper into these techniques in [Chapter 3](#).

Authorization (AuthZ)

AuthZ is the process of granting or denying access to specific resources or functionalities within an application or system. It answers the question, “What can you do?” by determining the actions a human or machine user is permitted to perform once an AuthN mechanism confirms their identity.

Developers can choose to implement AuthZ in three ways:

Role-based access control

Role-based access control (RBAC) assigns roles to users based on their responsibilities, with each role having defined access rights to specific areas of the system.

Attribute-based access control

Attribute-based access control (ABAC) evaluates user attributes like job title, location, and clearance level to determine access privileges.

Policy-based access control

Policy-based access control (PBAC) enforces access based on predefined policies that dictate who can access specific resources under which conditions.

Developers should build AuthZ mechanisms that ensure the most granular level of control (i.e., the principle of least privilege) to ensure that human and machine actors have only the permissions necessary to perform a required task.

Access control

The purpose of *access control* is to restrict entry to sensitive systems and information. This involves confining access to only those users who are who they say they are and possess the necessary AuthZ to keep sensitive information from falling into the wrong hands.

Proper access control measures should achieve these three goals:

Prevent unauthorized entry.

Access control reduces the risk of unauthorized parties gaining entry into sensitive systems, thus minimizing the potential for breaches, cyberattacks, and data leaks.

Preserve data integrity.

Access control does this by limiting access to authorized personnel only, thus ensuring the integrity and accuracy of sensitive data and mitigating the chances of tampering or manipulation by malicious actors.

Maintain availability.

Access control mechanisms are not meant to prevent or delay legitimate users from doing their jobs. Access control measures should keep data accessible and promptly available for authorized personnel.

The following Java code sample uses **Spring Security** to illustrate a simple RBAC access control check. **Spring Security** is a customizable AuthN and access control framework for securing Spring-based applications. This sample requires users to have the "ADMIN" role to access the "/admin/**" URL pattern:

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated()
        .and().formLogin();
}
```

Here, Spring Security's `antMatchers()` method is used to specify a pattern that matches a URL or resource.

Auditing and logging

Developers use various auditing and logging techniques to maintain security accountability and traceability for applications. While *auditing* focuses on capturing specific system events and user actions in support of compliance activities, *logging* centers on capturing information about an application's state, behavior, and interactions with the outside world to support security testing, analysis, and incident response.

The audit process typically follows three stages:

Defining audit trails

Identifying which events and actions warrant auditing, such as login attempts, data updates, and changes to permissions

Capturing relevant data

Live recording of contextual information such as timestamps, user IDs, IP addresses, and actions performed

Storing results

Capturing audit logs in a secure and tamperproof manner to prevent unauthorized changes

Implementing logging functions usually involves three decisions:

Selecting events

This involves choosing which events, errors, and activities warrant logging to satisfy security requirements without bogging down system performance or overloading the security team with too much data.

Protecting data

This involves deciding which details are necessary for accurate logging while avoiding the inclusion of sensitive data like passwords and personal information.

Managing logs

This involves determining how to centralize log data from various application components to facilitate analysis and monitoring.

Here is an example of a log entry that captures a device's IP address and location, identifies that an MFA attempt has failed, and identifies that the account has been locked:

```
2025-01-14T14:32:18Z INFO [AUTH] User login attempt
User: frodo
IP Address: 192.168.1.25
Device: Windows 11, Chrome 120.0
Location: New York, USA
Authentication Method: Password + MFA (OTP)
Status: FAILED (Invalid OTP)
Action Taken: Temporary account lock (3 failed attempts)
```

Secrets management

Secrets management is the practice of ensuring that sensitive information (like passwords, API keys, certificates, SSH keys, and encryption keys) is stored, accessed, and used in ways that safeguard them from unauthorized access, misuse, and exposure.

The biggest challenge with secrets management is one of scale: secrets are everywhere. Application and IT environments vary significantly among organizations, making it difficult to manage secrets effectively for every application, script, automation tool, and other machine identity that relies on privileged credentials to access other tools, applications, and data.

Think of hardcoded credentials in containerized applications (e.g., Red Hat OpenShift, Kubernetes, Azure Kubernetes Service), automation processes (e.g., Ansible Playbooks, Puppet, Chef), software development tools (e.g., compilers, profilers, code analyzers), cloud infrastructure, and IT management software. Multiply that by the speed of automation—accessing protected data and resources much faster than any human—and you'll begin to see the scope of secrets management.



Machine identities are digital and physical entities that interact with systems, applications, and resources. These machine entities include cloud environments and cloud native applications, DevOps tools, CI/CD pipelines, automation scripts, commercial-off-the-shelf and independent software vendor (ISV) applications, legacy applications, and other systems that are responsible for executing specific tasks. Machine identities also encompass IoT devices like sensors, cameras, and smart appliances, which gather and transmit data and perform actions.

Developers must understand the nuances of the different types of machine identities, as they exhibit different properties based on their roles and interactions with systems. For example, a build script within a DevOps pipeline requires different safeguards than a runtime API call to a third-party microservice.

Typical secrets management practices include the following:

- AuthZ of all access requests that use machine credentials
- Enforcement of the principle of least privilege
- Use of RBAC and regular rotation of secrets and credentials
- Tracking all accesses and performing regular comprehensive audits
- Removal of all secrets from code, configuration files, and other unprotected areas

Developers use specialized tools and practices to automate secrets management and apply access policies consistently. These tools typically centralize secrets storage, enable secure access controls, and manage credential rotation. The role of the developer is to know what techniques are available and where to deploy them, whether it's storing credentials in a secrets management system instead of a JSON file or enforcing the principle of least privilege on a containerized build system rather than giving it administrator access.

Securing development environments

Securing development environments involves establishing and maintaining a controlled setting where software is developed, tested, and maintained, to ensure robust application security at every stage of the development lifecycle. This process may be the first line of defense in identity security because it's where developers live and breathe every day.

The following seven techniques that can help you foster a more secure development environment:

Avoid hardcoded secrets.

Never embed sensitive credentials, like passwords or API keys, directly into application code, scripts, or configuration files. Instead, utilize secure secrets management solutions to safeguard these confidential details.

Test in staging environments similar to production systems.

This practice enables developers to simulate authentic usage scenarios, uncover potential vulnerabilities, and validate system behavior under realistic conditions.

Isolate test environments.

This helps you prevent test applications and frameworks from accessing live production systems. Isolating these environments minimizes the risk of accidental data leakage or unintentional system disruption during the development and testing phases.

Enforce segregation of duties.

Implement the principle of segregation of duties, ensuring that different individuals can only access specific development components. This reduces the likelihood of unauthorized actions and improves accountability.

Maintain open source and supply chain vigilance.

Never assume open source and supply chain software components are secure. Developers should thoroughly assess and monitor libraries, packages, products, and open source repositories for potential vulnerabilities, adhering to regular updates and patches. This includes assessing the proliferation of AI-generated code and ensuring that it meets robust security standards. The identity of the developer should also be enforced through the use of signed commits.

Use code signing.

Companies use code signing to ensure their customers can trust the software they deliver and prevent unauthorized changes. Too often, however, organizations fail to properly secure code signing credentials, leaving them vulnerable to cybercriminals and malware. The 2020 SUNBURST malware attack on SolarWinds is a well-known and devastating example of what happens when the code at any point in the software supply chain is left exposed. According to [NIST's "Security Considerations for Code Signing" white paper](#), code signing keys should be stored in an encrypted and secure location, and processes should be created to establish who has access to the keys and under what circumstances.

Test with proper secrets management in place.

Always conduct testing with a robust secrets management system in place to ensure sensitive information remains encrypted, properly managed, and inaccessible to unauthorized parties.

These practices not only minimize potential vulnerabilities before they're introduced into code, but they also fortify the overall tools and IT environment against threats.

Avoiding insecure components

While open source, third-party, and AI-generated components and libraries make your development life easier, they can also become targets for malicious activity by harboring vulnerabilities and potential exploits. Avoiding components known to be vulnerable and scanning them before, during, and after release helps improve the security posture of your applications.

The Equifax data breach arose from an unpatched vulnerability, [CVE-2017-5638](#), in the open source Apache Struts framework. This costly incident underscores the need for developers to know the security status of any component before including it in their application and to stay on top of security patches to maintain the latest versions.

Automated scanning and analysis

Automated scanning and analysis tools reduce the manual effort needed to identify vulnerabilities in code and maintain ongoing code security assessments through development changes, testing, and in-market updates. They use different types of techniques and checks to systematically examine code against known vulnerabilities and vulnerability patterns, with many tools reporting deviations from the best practices identified in this chapter.

There are three major categories of automated scanning and analysis tools:

Static application security testing (SAST)

Analyzes source code, bytecode, or binary files to uncover vulnerabilities without executing the program in a runtime environment.

Dynamic application security testing (DAST)

Assesses applications at runtime, often using an emulation or model of the operating environment, to identify vulnerabilities.

Software composition analysis (SCA)

Scans for vulnerabilities in open source components and third-party libraries that are integrated into an application.

These techniques should be integrated into the development pipeline to help developers pinpoint vulnerabilities earlier in the development lifecycle. This typically

makes it easier and less costly to remediate vulnerabilities than if they were found post-release.

Understanding Security Standards

Industry security standards provide a structured approach to building secure applications that's based on tried-and-tested practices used by the individuals and businesses that make up the standards' communities. These guidelines and examples help developers build security into their development processes. Some industries and customers mandate compliance with security standards before the product can go to market, and an example of this is HIPAA compliance in the health care industry.

This section offers explanations and examples of some key industry standards.

CWE

CWE is sponsored by the MITRE Corporation, and the CWE community comprises representatives from major operating systems vendors, commercial information security tool vendors, academia, government agencies, and research institutions. CWE and the CWE Top 25 provide a catalog of common software security weaknesses and vulnerabilities, including the following:

- Improper input validation that leads to injection attacks
- Insecure direct object references that can result in unauthorized data access
- Incorrect assignment of permissions to critical resources

OWASP

OWASP offers resources and guidance to enhance application security, including the popular OWASP Top 10 used by developers around the world. As a nonprofit and open source initiative to document security threats, OWASP resources are available free of charge to anyone.

Example OWASP guidelines are as follows:

- Ensuring proper input validation to thwart SQL injection and XSS attacks
- Employing secure AuthN and AuthZ mechanisms to prevent unauthorized access
- Logging security and monitoring failures to help detect, escalate, and respond to active breaches

NIST

NIST, which is part of the US Department of Commerce, provides frameworks and guidance on secure software development. Examples include the following:

- The **NIST Cybersecurity Framework** specifies five elements required to protect critical infrastructure—Identify, Protect, Detect, Respond, and Recover—which include activities and target goals to improve cybersecurity outcomes.
- The Secure Software Development Framework (SSDF) describes high-level secure development practices based on established standards and guidance.
- **NIST SP 800-204D**, the Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines, guide the integration of security measures into DevSecOps CI/CD pipelines.
- NIST supports the **President's Executive Order (EO) on Improving the Nation's Cybersecurity** by identifying standards, tools, best practices, and other guidelines to enhance software supply chain security.

PCI DSS

PCI DSS specifies over three hundred security controls for organizations handling payment card data, following these three principles:

- Encrypt payment card data during transmission and storage.
- Store data securely with techniques such as encryption, ongoing monitoring, and testing access to card data.
- Regularly test security systems and processes to identify vulnerabilities.

HIPAA

HIPAA sets guidelines for handling patients' electronic protected health information (ePHI) within the health care sector. This law mandates compliance for both covered entities and their third-party business associates, with guidelines that include the following:

- Implementing strong access controls to limit unauthorized access to sensitive patient data
- Employing encryption to safeguard ePHI
- Implementing audit controls to record and track system activities that involve ePHI

Summary

The power to create secure and trustworthy software begins at the developer's keyboard. From implementing access control and AuthN at every level of the application to ensuring proper secrets management across the organization, these best practices form a robust framework to protect against unauthorized access, data breaches, and other forms of cyberattack.

The understanding and use of industry standards like CWE, OWASP, and NIST further enable developers to educate themselves and foster a culture of security-conscious development to meet the expectations of demanding industries and customers.

By incorporating these secure coding best practices, you not only protect your applications and user data but also contribute to a safer digital environment for the world.

Authentication and Authorization

From here on out, we will build our identity security skills to improve the security posture of our code and applications. In this chapter, we'll start with the fundamentals of authentication and authorization, and in later chapters, we'll integrate these practices into automation and CI/CD pipelines.

Authentication and authorization are often thought of as interchangeable, but they're very different concepts (see [Table 3-1](#)). *Authentication* (AuthN) is the process of verifying the legitimacy of credentials provided by a user and thus ensuring they are who they claim to be. Usernames and passwords are familiar forms of AuthN, but as we'll see here, they're not enough to establish true identity. *Authorization* (AuthZ) is the granting or denying of access rights and permissions to resources after the user's identity is authenticated. For example, your build manager may have root-level or superuser privileges to a build server while your access may be restricted or withheld completely to keep the server secure.

If you can't authenticate a user or machine, you cannot and should not authorize it—this is why identity is so important.

Table 3-1. Differences between AuthN and AuthZ

AuthN	AuthZ
Verifies whether a user (human and machine) is who they claim to be.	Determines what resources a user (human and machine) can and cannot access.
Challenges the user to validate credentials (e.g., username, password, MFA, biometrics).	Verifies a user's access through policies and rules (e.g., RBAC, ABAC).
Performed before AuthZ.	Performed after AuthN completes successfully.
Often implemented by the popular OpenID Connect (OIDC) protocol; other examples are JWTs and Security Assertion Markup Language (SAML).	Often governed by the popular OAuth 2.0 framework; other examples are access control lists (ACLs) and Extensible Access Control Markup Language (XACML).

As we'll see in later chapters, identity and access management (IAM) solutions provide AuthN and AuthZ functions that help you control access to services and infrastructure without developing them yourself.

AuthN

AuthN is a fundamental building block of your application and the components around it because it verifies the legitimacy of user identities. If you're flying to another city, you may be asked to show your passport or driver's license to verify your identity. Similarly, applications, scripts, services, and other tools should ensure that human and machine actors are who they claim to be. This minimizes the risk of data exposure to the wrong entity and allows systems to track access to resources.

You've probably seen this in action. For example, calling a remote script from your command line may prompt you for admin credentials, or logging in to your HR system may require you to submit additional information (known as *factors*) that are sent to your phone.

Let's look at four AuthN methods that are common in application development.

Username and Password

Users use the classic username-password combination to provide their credentials to gain access to services and resources. Imagine yourself writing an API call to access a bank transaction database. You would need a username and password for the bank's identity provider to authenticate your application and authorize access to the appropriate data.

The following code snippet illustrates an AuthN request using a username (`user`) and password (`pass`) through HTTP Basic Authentication:

```
# import requests module
import requests
from requests.auth import HTTPBasicAuth

# Making a get request
response = requests.get('https://api.github.com / user, ',
                        auth = HTTPBasicAuth('user', 'pass'))

# print request object
print(response)
```

This snippet illustrates two things: how a simple AuthN process works and hardcoded credentials, the latter of which is a common security risk. You should never include `user` and `pass` in code explicitly; rather, you should use some form of IAM or secrets management instead.

MFA

Multifactor authentication (MFA) goes a step beyond passwords for human identities, as passwords can be guessed or stolen by bad actors. MFA requires users to present multiple forms of evidence to gain access to an application or system. For example, a support ticketing system may require a one-time, short-lived short message service (SMS) code in addition to a password.

Two-factor authentication (2FA) is a common MFA implementation that requires one additional factor beyond a password, such as a soft token (sent via SMS or mobile app) or a hard token (such as a Universal Serial Bus [USB] key).

It is important to keep the factors separate when including MFA calls in your application, as their underlying implementations may differ. The following code snippet shows that the token validation is performed independently from the username and password combination:

```
# Example of two-factor authentication
def authenticate_with_2fa(username, password, token):
    if authenticate_user(username, password) and verify_2fa_token(token):
        # Successfully authenticated with 2FA
        pass
    else:
        # Authentication failed
        pass
```

Token-Based AuthN

Token-based AuthN uses signed security tokens to verify identity and, in return, provides applications with unique access capabilities to a resource. Unlike passwords, token-based AuthN requires another service to verify identity requests, thus adding another form of verification that is harder to steal and persists over a period of time.

The typical token-based AuthN process follows five steps:

1. Request

The user issues an access request to a server or protected resource, typically with a username-and-password combination.

2. Verify

The resource acting as the identity server verifies the login credentials to determine whether access should be granted or not.

3. Generate token

The identity server generates a secure, signed AuthN token that's valid for a specific period of time and passes it back to the user.

4. Store token

The token sits in the user’s application or browser storage. If the user attempts to access a different part of the resource, the token is decoded and verified again.

5. Expire token

The token remains active until the user logs out or closes the resource or until the identity server revokes it.

Biometric AuthN

Biometric methods validate identity through the physical characteristics of a human user. The most common types of biometric identifiers are fingerprints, facial characteristics, voices, irises, and palm or finger vein patterns. Techniques like face and retina scanning compare the user-provided biometric data to stored records.

Biometric AuthN is often preferred over passwords to improve user experience and avoid *password fatigue*, in which users feel overwhelmed or make mistakes while trying to remember multiple passwords for different applications. Bad actors have a harder time overcoming biometrics as it’s extremely difficult to replicate a user’s retina or fingerprint.

When you’re building biometric AuthN into your app, don’t reinvent the wheel. There are secure, popular standards like *passkeys* and *WebAuthN*, which are developed and maintained by the **FIDO Alliance** standards body. Third-party identity services should generally support all the major standards from the FIDO Alliance, so your web app can support secure, biometric AuthN out of the box.

Common AuthN Techniques

You don’t want to implement AuthN yourself, so knowing the different technologies that are available is helpful.

OpenID Connect

OpenID Connect (OIDC) is an identity layer that sits atop OAuth 2.0, which is the widely used “open ” framework we’ll discuss later in this chapter. The open standard protocol makes verifying a user’s identity easy and provides one login for multiple resources—which is essential for developing apps that access different services.

The most popular use case for OIDC is supporting single sign-on (SSO) capabilities through a federated identity provider that links one identity to multiple systems, like Google or Facebook. Each time a user or your application needs to access resources, it only needs one set of credentials verified through an OIDC service.

For example, if you’re building an application that accesses a protected server on behalf of users, it must verify that users are who they claim to be. A simple approach

would be to build your own database of allowed usernames and passwords, but this is hard to maintain, doesn't scale, and is fraught with security risks.

A more secure and maintainable solution is to use an OIDC-capable identity provider that is shared across your organization. This allows for the AuthN of large volumes of users, and it decouples your application from having to know and manage human and machine identities. It also lets users log in with credentials they use for other applications, thus simplifying their lives.

You're familiar with this approach if you've ever signed in to a service using Google credentials. Google's **OAuth 2.0 API** conforms to the OIDC specification and allows you to access multiple Google services without worrying about implementing SSO yourself.

Figure 3-1 illustrates the OIDC AuthN flow by using the example of a user connecting to a banking application (the "relying party" or client) that uses the CyberArk® Identity Security Platform™ as the OIDC-capable identity provider.

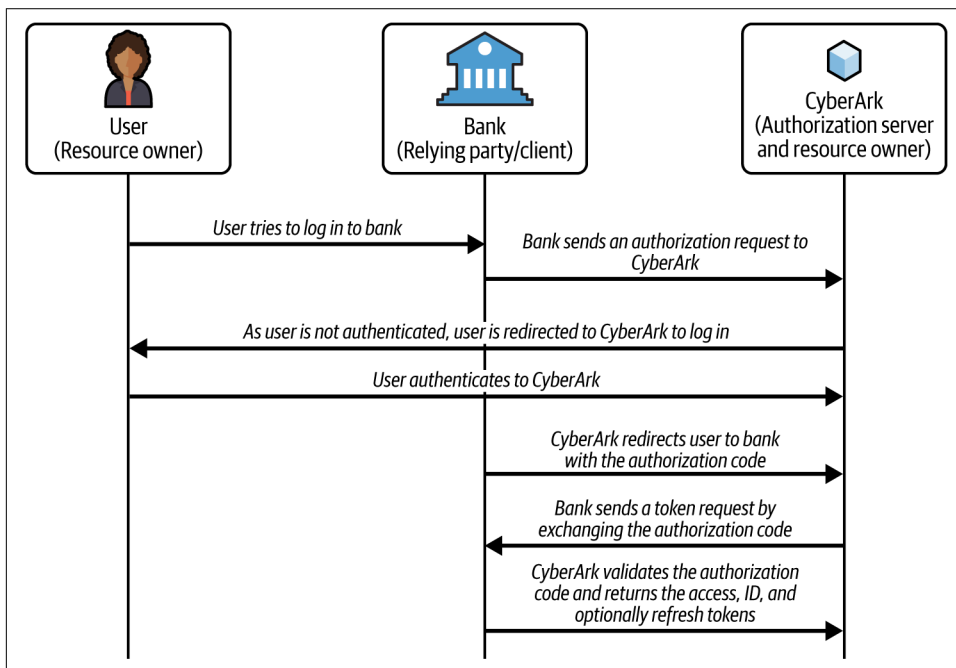


Figure 3-1. Example of OIDC AuthN workflow

In contrast to the first code snippet in this chapter, the following code illustrates this identity provider approach using **OAuthLib for Python**. Note the absence of hardcoded credentials:

```

from oauthlib.oauth2 import BackendApplicationClient
from requests_oauthlib import OAuth2Session

# OAuth provider configuration
client_id = 'your_client_id'
client_secret = 'your_client_secret'
authorization_base_url = 'https://example.com/oauth/authorize'
token_url = 'https://example.com/oauth/token'
redirect_uri = 'https://yourapp.com/callback'

# Initialize OAuth session
client = BackendApplicationClient(client_id=client_id)
oauth = OAuth2Session(client=client)

# Get authorization URL
authorization_url, state = oauth.authorization_url(authorization_base_url)

# Redirect the user to the authorization URL
print('Please go here and authorize:', authorization_url)

# Handle callback URL after user authorization
authorization_response = input('Enter the full callback URL:')

# Fetch access token
token = oauth.fetch_token(token_url, authorization_response=authoriza
tion_response, client_secret=client_secret)

# Use the access token to access protected resources
response = oauth.get('https://example.com/api/resource')
print(response.content)

```

JWT

JWT provides a standardized and self-contained approach to token-based AuthN. Released as the [RFC 7519](#) open standard in 2015, JWT was initially designed to be a simple mechanism for transmitting data for web development, and it later evolved into a solution for facilitating trust between enterprise software resources.

JWT acts as a bearer token that asserts information about the user and uses cryptographic signatures to ensure data integrity. This token can be sent in the header of an HTTP request, through a URL, or through a POST parameter. It contains all the necessary information about an entity to avoid querying a database more than once, and the recipient doesn't need to call an identity server to validate the token.

A JWT is a long string divided into three parts that are base64 URL encoded:

Header

This part contains the token type, which is JWT, and the signing algorithm used, such as hash-based message authentication code (HMAC) SHA256 or RSA.

Payload

This part contains statements about an entity, which are known as *claims*. Claims are either *registered* (predefined terms usable by anyone), *public* (terms defined at will by the organization and usable by anyone), or *private* (custom terms defined by the parties using the JWT and neither registered nor public).

Signature

This part is used to verify that the message wasn't altered during transmission and, when a private key is used, to verify that the JWT sender is who they say they are.

The following code snippet illustrates the creation of a JWT payload that contains user information (hardcoded here for illustration—you'll likely use a secrets manager in real life), sets an expiration time, and encodes the JWT using a secret key. Subsequently, it decodes and verifies the JWT to access the user's claims:

```
import jwt
from datetime import datetime, timedelta

# Define the payload
payload_data = {
    "user_id": 123,
    "username": "enterprise_dev",
    "exp": datetime.utcnow() + timedelta(days=1)
}

# Specify a secret key (ensure it's kept secure)
secret_key = "your_secret_key"

# Generate the JWT
token = jwt.encode(payload_data, secret_key, algorithm="HS256")

# Decode and verify the JWT
decoded_token = jwt.decode(token, secret_key, algorithms=["HS256"])

# Accessing claims from the decoded token
print("User ID:", decoded_token["user_id"])
print("Username:", decoded_token["username"])
```

JWT is a common way to implement *stateless AuthN*, in which much of the data used to authenticate the user is stored on the client side. As no centralized identity provider is necessary, stateless AuthN reduces the overhead on IT resources, making it scalable and interoperable with different SSO models.

The key to maintaining security robustness is ensuring that the JWT has an expiration time set, as in the preceding code snippet, to ensure that it cannot be used if the token is stolen.

Here's a quote from RFC 7519:

The “exp” (expiration time) claim identifies the expiration time on or after which the JWT must not be accepted for processing. The processing of the “exp” claim requires that the current date/time must be before the expiration date/time listed in the “exp” claim.

The flexible nature of JWT requires you to consider the following practices to limit the opportunities for compromise:

Keep payloads light.

Minimize the amount of sensitive information stored in the JWT payload and focus on the essential claims only to reduce token size.

Keep claims simple.

Use simple values in claims and do not put complex data structures inside a claim value. If you need multiple pieces of information, break them out into individual claims.

Set reasonable expiration times.

Configure JWT expiration times under the philosophy that short-lived tokens improve security by minimizing their windows of vulnerability.

Use strong encryption.

Employ strong cryptographic algorithms when signing and encrypting JWTs to minimize the risks of malicious tampering.

Use HTTPS.

Always transmit JWTs over secure channels to add layers of security to prevent interception and eavesdropping.

Lightweight directory access protocol

The *lightweight directory access protocol* (LDAP) is a protocol that gives you a centralized way of storing and organizing information about your organization's users, devices, and other resources. Originally developed as a directory browsing and searching mechanism, it's now a common authenticator that uses a standardized communications protocol to send and receive information.

Unlike JWT, LDAP centralizes user management and AuthN functions and saves you from worrying about them. As a developer building an LDAP-enabled application, you will likely interface with the following LDAP components:

Directory system agent

The directory system agent (DSA) is the server running LDAP on the corporate network.

Directory user agent

The directory user agent (DUA) is the client accessing DSAs, such as your PC or application.

Distinguished name

The distinguished name (DN) is the path through the directory information tree (DIT) for LDAP to navigate through (e.g., cn = Cersei, ou = users, o = Company).

Relative distinguished name

The relative distinguished name (RDN) consists of components within the DN (e.g., cn = Cersei).

Your application would typically connect to an LDAP client that requests AuthN of the DN through the DSA using an API call. LDAP uses the DN and RDN to search through its database to find the user—and if the user’s account is in the directory and has valid credentials, a response is sent back to the application. Once authenticated, your application disconnects from the DSA, allowing users to access services based on their permissions.

LDAP’s disadvantage is that it wasn’t designed with cloud applications in mind, so it requires a client to be installed on endpoint devices. As an older identity management protocol, it’s more suited to environments using Microsoft Active Directory or custom on-premises deployments than newer, more complex systems.

Security Assertion Markup Language

Security Assertion Markup Language (SAML) is an XML-based open standard for exchanging AuthN data between parties. It operates like a session cookie in your browser that gives access to web applications. This means its use cases can be limited outside a browser’s context. Like LDAP, it is centralized and commonly used to enable SSO for organizations.

Similar to JWT, SAML implementations use a claims-based AuthN workflow. For example, these are the possible steps for a test harness script you could write that requires access to a resource using SAML-based AuthN:

1. The script opens a browser and connects to the resource, which uses a centralized identity provider for AuthN.
2. The resource responds with a SAML request.
3. The browser passes a SAML request to the identity provider.
4. The identity provider parses the SAML request.

5. The identity provider authenticates the script's credentials by prompting for a username, password, and likely MFA.
6. The identity provider generates the SAML response and returns it to the browser.
7. The browser sends the SAML response to the resource, which verifies it.
8. The resource grants the user access if the verification succeeds.

Risk Analysis During AuthN

Today, the prevalence of complex systems and increasingly sophisticated attackers means that securing user access requires more than just verifying a username and password. Attackers can steal credentials through phishing, brute-force attacks, and database leaks, making static AuthN methods insufficient for safeguarding sensitive systems.

Developers of modern web apps should consider using *risk-based authentication* (RBA), which is a dynamic approach that evaluates contextual factors and user behavior to determine whether an AuthN request should be approved, be denied, or require additional verification steps.

Effective risk-based AuthN begins with understanding what typical login behavior looks like for a given user. Organizations can analyze historical data to establish baselines for normal login patterns, such as the user's usual geographic location, device, and time of access. If a user typically logs in from a laptop in San Francisco during business hours, an AuthN attempt from another country at 3:00 A.M. would be considered an anomaly and may warrant additional security checks.

Some modern IAM systems, including SSO providers, incorporate *adaptive AuthN*, which adjusts AuthN requirements based on real-time context and risk assessment. These systems incorporate AI and machine learning to analyze AuthN attempts and continuously refine risk assessment models.

These systems use large datasets to detect patterns in login behavior and flag potentially fraudulent attempts with greater accuracy over time. Based on the calculated risk score of a login attempt, organizations can dynamically adjust access policies by doing any of the following:

- Allowing access when the risk score is low
- Requiring additional verification (such as MFA) for medium-risk attempts
- Blocking access altogether if the risk is deemed to be too high

Figure 3-2 illustrates how the User Behavior Analytics engine embedded in CyberArk's Workforce Identity solution incorporates different contextual attributes to determine a risk score that's used to trigger access policies dynamically.

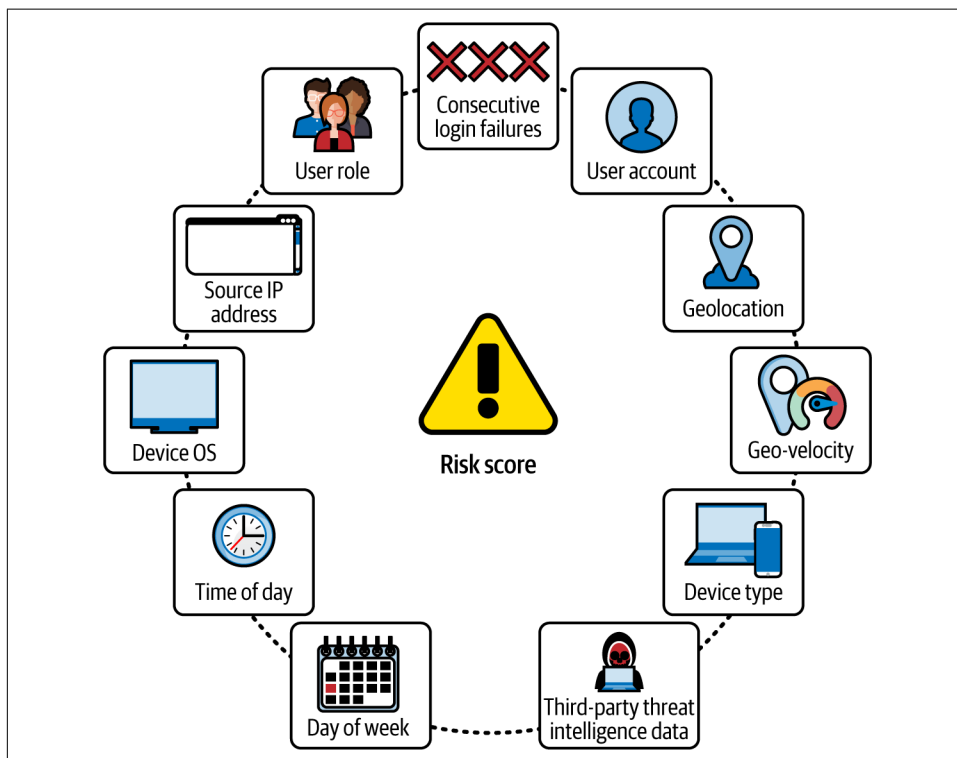


Figure 3-2. Contextual attributes used to determine a risk score

The advantage of these approaches is that they enhance security while minimizing unnecessary friction for legitimate users. Instead of prompting all users for MFA at every login, leading to frustration and fatigue, AI-based and machine learning-based risk analysis ensures that only high-risk attempts face additional scrutiny.

For example, a banking app may allow seamless biometric AuthN (such as Face ID or fingerprint recognition) when a user logs in from their usual phone. However, if the same user attempts to log in from a new device or an unrecognized network, then the system may require additional verification, such as a *one-time passcode* (OTP) sent via SMS or email. Similarly, when unusual financial transactions occur, such as large purchases in a foreign country, the bank might send a text asking, “Did you authorize this transaction?” before processing the payment for any given transaction.

AuthN Best Practices

Implementing strong AuthN practices inside and outside your application improves its security posture. Here are some things to consider when building and testing your code:

- Separate the concepts of AuthN and AuthZ—the thinking behind them is different, and their implementations have distinct requirements.
- Build your AuthN components to be adaptable to the demands of the system and the business needs, such as decoupling the public interface from the specific AuthN technology used, in case it changes.
- Never (ever) hardcode credentials into anything you write or share secrets outside of a protected and authorized mechanism. Developers often forget these common leak sources: scripts, environment variables, build manifests, README files, and even output logs.
- Enforce strong password policies and employ secure password hashing to minimize unauthorized access to protected data.
- Implement MFA with factors like SMS, email, and authenticator apps, to minimize the risk of compromise.
- Use SSO methods to reduce manual password fatigue and associated risks.
- Rotate application credentials, implement token expiration policies, and restrict session times to minimize hackers' window of opportunity in case of compromise.
- Automate and standardize your approaches to AuthN, usually in concert with IT and security teams.
- Empower users and developers to access your applications with fast, responsive, and easy access to everything they need without compromising security.
- Always separate development environments from production, including by using different sets of credentials, identity providers, and AuthN policies.

AuthZ

Authorization (AuthZ) is the process of granting or denying access to specific resources or functions within a system. It's a different type of gatekeeper than AuthN because it ensures that human and machine identities have permission to perform actions or access data once AuthN has passed successfully.

If you have logged in to a remote server successfully and can access certain directories but not others, you've seen AuthZ in action. For example, scripts and services often fail because they cannot run commands on a resource despite being able to ping and connect to it successfully.

AuthZ comes in different forms:

User-based access control (UBAC)

Determines access by the specific user account. Each account is associated with a unique set of permissions, thus offering a granular level of control over resources.

Role-based access control (RBAC)

Assigns permissions based on predefined roles. For example, anyone classified as an admin may have access to modify a test machine, whereas an individual classified as a tech writer may not.

Attribute-based access control (ABAC)

Uses characteristics of the AuthZ request—such as device information, user ID, location, and file type—to determine access. Using attributes allows for more granular control over resources as there are more criteria to choose from than when using the other methods.

Policy-based access control (PBAC)

Uses specific rules or conditions to determine access. It allows for dynamic access decisions based on contextual factors such as department, location, and date, thus enhancing flexibility.

Table 3-2 provides an example of how RBAC controls permissions for a given resource. Even though both roles may be able to access the resource, their allowable actions are governed by the given AuthZ policy.

Table 3-2. Example of RBAC

Role	Create	Read	Update	Delete
Admin	Yes	Yes	Yes	Yes
Tech writer	No	Yes	Yes	No

Once an identity is authenticated, AuthZ controls are applied. Rather than implement this yourself, you should understand the following AuthZ technologies.

OAuth 2.0

OAuth 2.0 is an open standard framework that defines how a client obtains access tokens containing permissions against the resources available to the client. Rather than make a user or application manage AuthZ, OAuth 2.0 delegates decisions to an independent AuthZ server. The standard also specifies refresh tokens that clients use to get new access tokens without requiring them to reauthorize themselves.

The Facebook platform uses OAuth 2.0 to enable access to its app ecosystem through a user’s Facebook account. If you’ve seen the “Log in with Facebook” button on apps and websites, the underlying AuthN mechanism for it is OAuth 2.0.

OAuth specifies four roles:

Resource owner

This role owns the protected resource's data, such as your account on a development network or a Facebook account holder.

Client

This is the application requesting access to the protected resource.

Resource server

This role hosts the protected data the client wants to access.

AuthZ server

This is the OAuth 2.0 implementation that verifies the client's identity and issues access tokens.

OAuth 2.0 is not an AuthN protocol—it does not specify the format of tokens or define how users are validated. OIDC, which we described earlier, is the AuthN and identity layer above OAuth 2.0. For example, the OAuth 2.0 process is separate from the 2FA method Facebook uses to authenticate user access to the platform.



OAuth 1.0a and OAuth 2.0 are two different specifications. They are not interoperable or backward compatible. These days, OAuth 2.0 is by far the most popular standard in use.

While OAuth 2.0 is relatively straightforward to implement, especially with the support of a large tools ecosystem, it can be fragile. If an attacker discovers an exploit on your system, it could be compromised if you don't have the proper protections in place.

Ideally, access tokens expire after a short period of time to minimize the vulnerability window in case they're compromised. If you're integrating OAuth 2.0 capabilities into your application, here are some other best practices to consider:

Manage local credentials securely.

You should securely store the OAuth 2.0 credentials that identify your application's identity (e.g., by using a secrets manager). You should also transmit them securely (e.g., with HTTPS) and not share them publicly. Finally, hardcoding credentials in your application defeats the purpose of identity security.

Authorize incrementally.

Following the *principle of least privilege*, you should only request access to a resource server when the application needs to access it. While it may seem easier to request access when the user opens the app, keeping requests within their scope of use limits security exposure and conforms to the user's security

expectations. For example, if a feature that requires access to a financial transaction database is never used, there's no need for you to authorize access to the database.

Remove unnecessary requests before production.

Your application often needs access to more resources in development than in production. When deploying it, you should ensure that your final implementation limits access requests to only those that are necessary to run.

Handle refresh tokens.

If your application uses refresh tokens, it must also handle their expiration and revocation. For example, if the AuthZ server has revoked a token, you should prompt the user to authorize again rather than do it without their knowledge.

Allow users to revoke support tokens.

A user may decide that they want to de-authorize access to the protected resource, such as when their credentials have been compromised. Allowing the user to invalidate their token supports this goal.

JWT

Once JWT is authenticated through a mechanism like OIDC, it allows clients to access protected resources without going through the AuthN process again. Whenever the client wants access to a resource, it sends the JWT for verification.

However, hackers can often exploit JWT by using malicious links to steal JWT tokens and use them to access protected resources. In addition to the best practices stated earlier, here are things you should keep in mind when writing applications using JWT for AuthZ:

Always validate.

Avoid relying on your application to know the validity of an incoming token and always validate with the AuthZ server. You never know where your code will be deployed or how its threat environment will change, so building in token validation from the start improves your application's security posture.

Don't assume JWTs are secure.

JWT attacks do exist and usually take the form of a malicious entity sending modified tokens to the resource server to compromise the system. For example, if a hacker gains access to or guesses the server's secret key, they can generate a valid signature and use tampered payload values to access protected data.

Access Control Lists

An *access control list* (ACL) is basically a table that defines access permissions for protected resources. It often takes the form of a list of entries with user or role names and associated access privileges. The two basic types of ACLs are as follows:

- *Filesystem ACLs* define access permissions to files and directories, such as those used by an operating system.
- *Networking ACLs* define traffic permissions through a network, such as those used by network switches and routers.

For example, if you're writing code to modify a file on a server managed by an ACL, it would work something like this:

- Your application attempts to modify the file, triggering the ACL to check the application's identity against its list.
- If the application's identity exists, the ACL checks to see whether it has permission to modify the file.
- If the application has permission, the ACL allows the modification to go through; otherwise, it denies the request.

ACLs are simpler to implement and perform faster than more complex frameworks like OAuth 2.0. They are also harder to scale and maintain. As such, they are often used on localized devices (like routers and servers) and entry points into systems, rather than globally for networks and enterprise applications.

As a developer who may adopt or interact with ACLs, here are some things for you to consider:

The order of ACL entries matters.

Much like conditional statement blocks in code, ACLs execute the first rule that applies to the user before moving on to the next. Rules that are entered in the wrong order or poorly written could deny legitimate access to resources. Worse, if a generalized rule is specified before a resource-specific one, it may open that resource to attack.

Implement the principle of least privilege.

You should grant users the minimum level of access they require to perform their tasks, starting with the absolute minimum set of permissions and granting additional powers as necessary.

User entries must be explicit.

As ACLs state permissions explicitly, a user that requires access as a developer and a QA tester must have both declared, rather than inferred from one entry.

User entries could be anywhere.

Permissions and access levels may spread across multiple, standalone ACLs within the organization. Whenever you need to grant, modify, or revoke access through an ACL, you must review every list to ensure the changes are applied correctly. Ideally, your organization will implement ACL management centrally to make maintenance easier.

Extensible Access Control Markup Language

Extensible Access Control Markup Language (XACML) is an XML-based language that is used to specify secure access control policies. It's a standard published by the technical committee of the Organization for the Advancement of Structured Information Standards (OASIS) consortium, and it's designed to work with SAML, which is another OASIS standard.

XACML follows the ABAC model, which gives you more latitude in defining access permissions. You can include the attributes of users, resources, actions, and the context of the action. This approach is popular in the health care and finance sectors, and it's used to do things like enforcing access control policies for different users of an electronic health records (EHR) system or a backend banking system.

XACML architecture comprises these main components:

Policy enforcement point (PEP)

A resource (like a server, database, or filesystem) that contains the protected information. Upon a user's request, the PEP sends a request to the policy decision point with the user's attributes, the requested action, and other relevant information.

Policy decision point (PDP)

Evaluates the PEP request against the XACML policy applied to it and decides whether to grant or deny the request.

Policy information point (PIP)

Supports the PDP in managing and accessing XACML policies.

Policy administration point (PAP)

Manages the PDP, PIP, and all policies.

While XACML is often considered a “developer's language” due to its flexibility, policies can get cumbersome and complex to manage. Common challenges to consider are as follows:

- Because policy sets get complicated—basically, every possible access request is turned into XML—your code must include a robust way of resolving contradictions and omissions in the rules.

- You must carefully consider, propagate, and document any changes to ease maintenance. It's better to make changes first in a staging area that accurately represents production, before pushing and potentially breaking access controls.
- XACML policy decision-making can impact your application's performance. Since the PDP must evaluate every access request, it's important to balance the security of the resource (the PEP) with your application's usability.

Summary

AuthN and AuthZ form the foundation upon which securing applications lies. Here are the key practices to remember:

- Never hardcode secrets and always verify that secrets aren't leaked through development artifacts.
- Enforce the principle of least privilege.
- Rotate secrets and implement expiration policies.
- Always encrypt data at rest and in transit.
- Standardize and automate your approaches to security.
- Empower ease of use while maintaining a high level of security.
- Consider using a third-party identity service that is secured with industry standards and that supports open AuthN standards.

Overview of Identity and Access Management Solutions and Protocols

Managing who can access what resources is the cornerstone of application security. These tasks fall under identity and access management (IAM) solutions that ensure human and machine identities can access the right resources at the right time.

IAM is more than user management; it's a comprehensive framework that safeguards resources by controlling access across multiple platforms, from on-premises systems to cloud services and mobile applications. As IT environments have grown more complex, IAM has evolved to include advanced technologies like SSO, MFA, and federated identity management. As developers, understanding IAM is crucial for building secure applications.

Integrating IAM solutions into your apps means you don't have to worry about managing user access and maintaining compliance—which is especially difficult when the number of identities and the scope of their permissions change and evolve. For example, IAM helps control who can view and manage electronic health records in health care apps, thus ensuring compliance with regulations like HIPAA. In financial services, IAM helps prevent unauthorized transactions and fraud by enforcing strict authentication (AuthN) and authorization (AuthZ) protocols on actors attempting to access the system.

From LDAP directories to federated identity systems powered by OAuth and OpenID Connect, IAM solutions continue to evolve, offering developers a range of tools and standards to secure their applications. Whether you're integrating SSO into a web application, adding MFA to a mobile app, or managing access to cloud resources, IAM will be at the center of your efforts to secure today's application ecosystems.

Core Components of IAM

IAM solutions typically offer features that are structured around five fundamental pillars: identity management, access management, AuthN, AuthZ, and identity governance and administration. Each pillar plays a different role in strengthening the security posture of your applications.

Identity Management

Identity management is the foundation of IAM. It focuses on the creation, maintenance, and deactivation of user identities throughout their lifecycle. To implement these activities, IAM solutions include user registration and profile management features, and they also ensure that identity information is accurate and up-to-date across the organization. Key functions of identity management include the following:

User provisioning

This creates user accounts when employees are onboarded, and *deprovisioning*, which removes access rights when employees leave the organization.

Directory services

These are services such as Microsoft Active Directory and LDAP directories, which store and manage user data at scale (see [Chapter 3](#) for more details).

Self-service password management

This allows users to manage their own passwords.

For example, consider a large health care provider whose staff members need access to medical records. The organization's ehealth app uses an IAM system to ensure that each new employee gets a unique user profile when hired. This profile contains the employee's personal information, job role, and department. Based on this profile, the system automatically assigns the appropriate access rights to clinical applications and databases. For example, an IT administrator would be granted access to database infrastructure and control over ehealth app deployments but not to patient EHRs. On the other hand, a nurse practitioner would be granted access only to EHRs and nothing else. If any employee leaves, their profile will be promptly deactivated and they will no longer have access to systems and sensitive patient information.

The same applies to application provisioning. When an application or script requires an identity to access resources, such as a build server or logging database, the IAM automatically creates user identities and roles in support of these activities. The IAM also manages the maintenance and removal of these identities as status or roles change.

Figure 4-1 illustrates how the CyberArk Identity Security Platform fits into different enterprise systems and applications to support identity management.

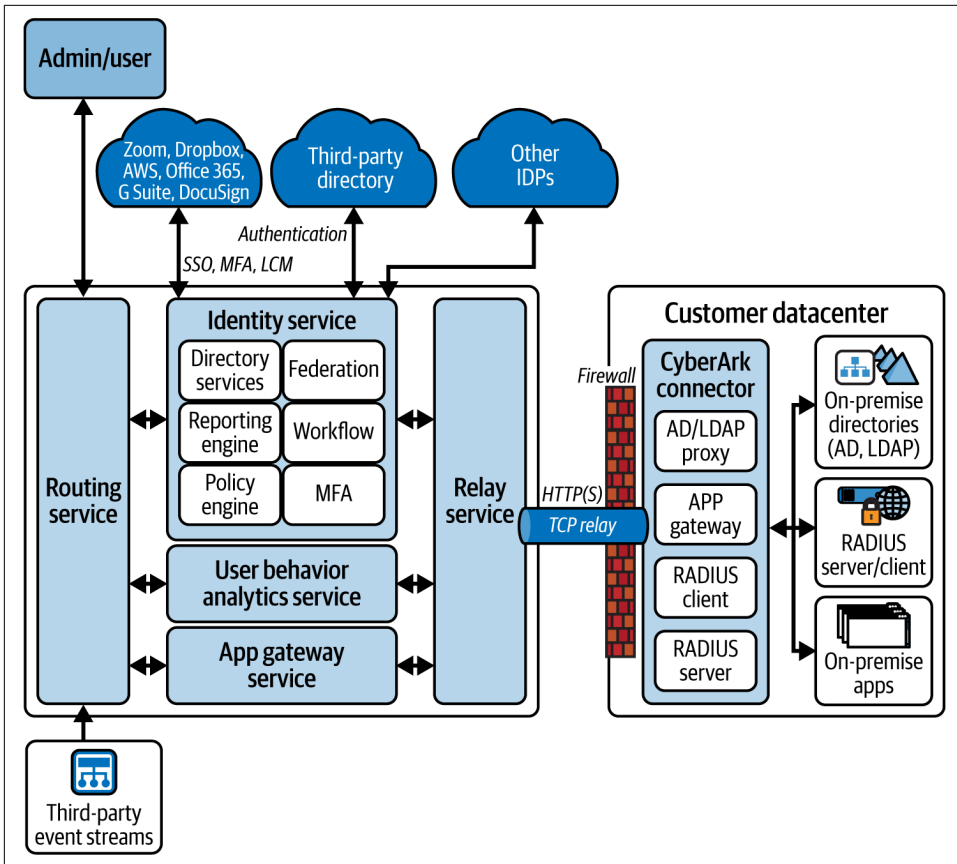


Figure 4-1. Example identity management architecture

Access Management

This pillar focuses on defining and enforcing access policies to determine who can access which systems, applications, and data under what circumstances. Here are some examples of access management:

- MFA, which we covered in [Chapter 3](#), adds an extra layer of security by requiring users to provide two or more forms of verification (e.g., a password and a one-time code sent to their phone) before gaining access to resources.
- SSO allows users to authenticate once and gain access to multiple applications without needing to reenter credentials. SSO enhances the user experience by reducing password fatigue, and it strengthens security by centralizing AuthN.

For example, a financial services company with multiple cloud applications (such as email, customer relationship management [CRM], and accounting software) can use

an IAM solution to implement SSO that simplifies employee access. Then, after initial AuthN, employees can switch between applications without being prompted to log in again. This workflow reduces the risk of password misuse and centralizes password management and auditing.

AuthN

As we discussed in [Chapter 3](#), AuthN is the process of verifying the identity of users and systems before granting access to resources. This is crucial for ensuring that users are who they claim to be and helps reduce the likelihood of unauthorized access.

Common AuthN methods include the following:

- Passwords, although organizations are using stronger methods to supplement them, like MFA
- Biometric AuthN, such as fingerprints and facial recognition
- Security tokens, such as a USB key, an ID badge, and a car key fob

AuthN technologies like SAML and OpenID Connect allow applications to securely authenticate users without directly handling passwords. This makes your job easier, as your app doesn't have to track and synchronize credentials, and it helps users in federated identity environments, where they can authenticate through a trusted third-party identity provider.

For example, a SaaS company that handles sensitive customer data may use an IAM solution to enforce MFA for all user logins through your app. This could require a password and a biometric scan like a fingerprint or a one-time passcode sent to the user's smartphone. This ensures that even if a user's password is stolen, an attacker cannot gain access without the additional AuthN factor.

AuthZ

AuthN determines the permissions and access rights granted to authenticated users. It is commonly managed through RBAC or ABAC policies implemented through an IAM solution.

For example, an IAM may implement an RBAC policy to govern the access rights for an ecommerce platform in this way:

- An administrator has the ability to add or remove users, manage payment processing, and view all customer orders.
- A regular employee only has access to specific order details.
- A customer can only update their profile and manage their orders.

Here, RBAC simplifies permission management by grouping access rights based on roles, making it easier to maintain consistency across the organization.

The following code sample illustrates how Azure identities can be assigned different roles using CyberArk® Secrets Manager™ or Conjur® Open Source.¹ Here, the policy defines a hosts group that contains two host resources representing Azure serverless functions. The `ConjurDemoAccessFunctionSystemAssigned` function is assigned a system-assigned identity, and the `ConjurDemoAccessFunctionUserAssigned` function is assigned a user-assigned identity, `ConjurDemoAccess`:

```
- !policy
  id: azure-apps
  body:
    - !group

    - &hosts
      - !host
        id: ConjurDemoAccessFunctionSystemAssigned
        annotations:
          authn-azure/subscription-id: f89ce...e91dc18
          authn-azure/resource-group: Conjur_Resources
          authn-azure/system-assigned-identity: 7098...0761e13861

      - !host
        id: ConjurDemoAccessFunctionUserAssigned
        annotations:
          authn-azure/subscription-id: f89ced...e91dc18
          authn-azure/resource-group: Conjur_Resources
          authn-azure/user-assigned-identity: ConjurDemoAccess

    - !grant
      role: !group
      members: *hosts

  - !grant
    role: !group conjur/authn-azure/AzureWS1/apps
    member: !group azure-apps
```



In policy files, grant statements determine which roles have access to resources.

¹ Conjur Open Source is the open source version of CyberArk Secrets Manager. All code examples used in this book will work with both.

The first `!grant` tag is part of the policy body, so the resources it creates will also be part of the policy. This tag declares that members of the `&hosts` collection are also members of the `azure-apps` group (`members: *hosts` means “all hosts”). The second `!grant` tag is outside the policy body (note the indentation) and states that members of the `azure-apps` group are also members of the `conjur/authn-azure/AzureWS1/apps` group. This tells Conjur which web service to use to authenticate these hosts.

This hierarchical policy structure allows you to express complex RBAC relationships that link identities to resources without manually identifying each one separately.

Identity Governance and Administration

Identity governance and administration (IGA) defines the policies, processes, and technologies that control and monitor who has access to what and why. Organizations use IGA solutions to automate this framework across all the devices, users, and data in one or more locations. Organizations usually include such solutions as part of a unified identity security platform that includes IAM and PAM services.



Privileged access management (PAM) solutions help protect organizations by monitoring, reporting, and preventing unauthorized privileged access to critical resources. While IAM controls the AuthN of identities to ensure the right user has the right access at the right time, PAM provides more granular control of and visibility into privileged identities and sessions.

Integrating with an IGA solution helps you ensure that access rights to your app are appropriate, regularly reviewed, and compliant with regulations, thus helping the larger organization manage access-related risks more effectively. For example, if HR uses your application to onboard new employees, you can use an IGA solution to automate new user provisioning without you having to deal with it personally.

IGA consists of three core components that work together to provide a comprehensive approach to managing and governing user access:

Access reviews

These ensure that users have the appropriate access rights for their roles and responsibilities. Regular reviews help identify any unnecessary or outdated permissions that could pose security risks. For example, a large retail company with thousands of employees and contractors can perform these reviews and remove users who should no longer have access to their ecommerce app. If users' access rights deviate from established policies, the IGA tool can trigger an alert to notify IT and security teams.

Segregation of duties (SoD)

SoD helps prevent users from gaining excessive control over critical business processes. SoD policies are designed to prohibit users from having access rights beyond what they need for their jobs. IGA tools can help developers include SoD enforcement in their apps, ensuring that access assignments comply with defined policies.

Compliance reporting

This provides visibility into IAM activities, thus allowing organizations to demonstrate that they follow required security and privacy policies. For example, a health care provider handling patient records can generate reports through IGA tools to show that only authorized health care professionals accessed patient data over a period of time.

Let's examine how these IAM pillars apply to real-world enterprise environments.

Why Does IAM Matter?

Understanding and implementing IAM effectively is no longer optional in today's application development. Whether you're working on consumer-facing apps or internal tools, IAM is a key factor in maintaining security, protecting user data, and ensuring a smooth user experience.

A misstep in implementing IAM can have serious consequences. A misconfiguration or omission could expose sensitive user data, leading to financial and reputational damage to your organization. For example, if your organization stores sensitive information in an AWS S3 bucket configured as public rather than private, attackers may be able to access it. This is **what happened to Accenture**: an unsecured bucket was found to store 40,000 plaintext passwords, hashed passwords, email data, and other sensitive information.

Letting identities sprawl throughout your codebase without visibility or upkeep can lead to serious security exposures later. And given our current regulatory landscape—think of data protection laws like the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA)—you must also navigate a minefield of privacy and security requirements.

For example, employees need secure, reliable access to cloud-based and on-premises applications in a hybrid cloud environment. IAM solutions provide a centralized way to manage these domains, thus ensuring that only authorized users can access sensitive information. Whether employees are working from a corporate office or remotely, your applications should be able to translate these needs into access, AuthN, and AuthZ workflows that are easy to use and transparent to the user.

IAM solutions do this for you.

Consumer Applications

If you're developing consumer-facing apps, effective *consumer identity and access management* (CIAM) means providing a good user experience without compromising security. How to design initial user registration, login, and access control are all decisions you'll have to make. CIAM solutions give you implementation options and help ensure that these functions work efficiently and align with auditing and privacy standards.

Whether consumers are registering for the first time or logging in via social media, CIAM can provide secure and company-branded user AuthN using OAuth or similar protocols we described in [Chapter 3](#). You need to understand how your CIAM works and also understand its policies so you can streamline AuthN rather than giving users long registration forms and complicated workflows.

Additionally, ensuring data privacy and compliance with global privacy laws can be challenging. Consumer identity management manages PII by integrating CIAM user privacy management into the AuthN flow of your apps, using standards like OAuth or OpenID Connect. These workflows ensure that users give informed consent when sharing personal information.

Enterprise Applications

Internal enterprise applications have a different set of IAM requirements. While some capabilities of consumer IAM apply, such as privacy and consent management, you may need to add capabilities like SSO and RBAC.

Employees typically need access to various applications, from internal tools to cloud-based services. Managing and maintaining credentials for each system within your app can be time-consuming and prone to error, but implementing SSO through your company's IAM (using standards like SAML 2.0 or OpenID Connect) allows employees to authenticate once and access multiple systems without needing to log in separately to each one.

For example, CyberArk® Identity allows administrators to [add applications using OpenID Connect](#) to manage organization-wide credentials. Once these are added, administrators can configure various settings for credential management, such as token lifetime, authorized parameters, and authorized scopes.

The same principle holds true for machine credentials. Instead of having to keep track of API keys, tokens, and machine identities, an IAM provides one place to manage all of them. Imagine that your app, after months in the field, had to connect to a new DynamoDB database hosted on a new AWS instance provisioned by another team. Without IAM, you would need that team's administrator to manually create new security credentials for your app—and if something changed after release, they would need to get involved again.

On the other hand, with IAM, you would just need to be added as a user (if you're not already provisioned) and assigned to the appropriate role for database access. Once assigned, your role would automatically get access to any new databases or services added to the instance, unless explicitly overridden by the administrator.

How IAM Relates to OWASP

The **OWASP Top Ten** is a well-established set of guidelines created by the Open Worldwide Application Security Project (OWASP) to help developers identify and mitigate the most critical security risks in web applications. These guidelines serve as a roadmap for secure coding practices, and they provide insights into the most common vulnerabilities that attackers exploit.

Many risks highlighted in the OWASP Top Ten involve improper access control, insecure design, and AuthN failures—areas where IAM plays a key role. The following sections explain how strong IAM practices can help mitigate these risks.

A01:2021—Broken access control

Broken access control is the number-one risk in the OWASP Top Ten, and it refers to vulnerabilities that allow malicious actors to access resources they shouldn't be able to access. This could be due to flaws in access control mechanisms that allow them to bypass AuthZ checks and escalate their privileges.

IAM solutions are specifically designed to manage and enforce access control policies, making them a logical choice to solve this problem. Granular access control techniques, such as following the principle of least privilege and implementing RBAC, can help prevent unauthorized access and ensure that users and machine identities operate within the limits of their assigned roles.

A03:2021—Injection

Injection attacks occur when untrusted data is sent to an application through an unprotected or unchecked command or query, allowing attackers to execute malicious code. SQL injection is the best-known example of this vulnerability, but there are other types, such as cross-site scripting (XSS) and code injection.

Combining IAM policies with secure coding practices (e.g., parameterized queries that separate the SQL query from the user input such that the input values are not executable) can better protect your applications from injection attacks; for example:

- Enforcing MFA can reduce the likelihood of attackers gaining access to an application.
- Implementing least privilege for databases and other systems can reduce the impact of an attack by limiting access to sensitive data or critical infrastructure.

A04:2021—Insecure design

Insecure design refers to a lack of attention to security during the architecture and design phases of application development. This is often due to insufficient planning or time allocated to the project, in which security practices are skipped rather than included in the development process.

IAM addresses this by making it easier to develop applications with secure AuthN and access control mechanisms. For example, choosing to use federated identity management in your app early in the design phase means you can allocate development time to integrating an IAM solution later. This may save you time because you can securely manage user identities across multiple domains without worrying about it yourself.

A07:2021—Identification and authentication failures

Identification and AuthN failures occur when attackers exploit vulnerabilities—such as weak passwords, lack of MFA, or improper session management—in an application's AuthN mechanisms. These issues compromise an application's ability to verify the identity of users, thereby leading to unauthorized access and data breaches.

IAM principles are central to mitigating these types of risks. Most solutions allow you to enforce policies for strong passwords, MFA, and federated identity protocols (like OAuth 2.0 or OpenID Connect) to reduce the chances of attackers compromising systems.

A09:2021—Security logging and monitoring failures

Security logging and monitoring failures are scenarios in which applications do not record or monitor security events with sufficient coverage and depth to support attack prevention, detection, and response. These risks become especially problematic when you're trying to identify breaches after they occur and meet security compliance requirements.

IAM solutions typically offer comprehensive logging and monitoring functions, and they can integrate with security information and event management (SIEM) systems. This includes tracking login attempts, access to sensitive resources, and changes to user permissions or roles. IAM tools also support the creation of audit trails that provide clear records of who accessed which resources and when.

Having these features usually means you don't have to worry about logging and tracking all AuthN, AuthZ, and access control activities for your app, but it's a good idea to ensure that no gaps are missed.

Identity Lifecycle Management

Identity lifecycle management (ILM) governs the creation, maintenance, and deactivation of identities throughout their lifecycle. IAM solutions implement ILM to ensure that users have access to resources at the right time while preventing access that's unauthorized because it's no longer needed.

The following sections cover the four aspects of ILM: identity provisioning and deprovisioning, role management, workflow and approval processes, and identity synchronization and reconciliation.

Identity Provisioning and Deprovisioning

IAM solutions automate *identity provisioning*, which is the creation and management of user accounts. For example, when a new employee joins the company, an IAM solution can automatically create accounts in multiple systems (e.g., email, HR platforms, project tools) and assign the appropriate access rights. Alternatively, there could be an automated script that requires access to a variable number of test servers every night to run tests.

Deprovisioning is the process of disabling or deleting human and machine identities and revoking access when it is no longer required. Without timely deprovisioning, dormant accounts can pose a significant security risk as IT teams lose track of them and malicious actors gain more attack opportunities.

Having an IAM solution that automatically deactivates accounts as employees leave and app requirements change ensures that these actors can no longer access sensitive data or proprietary information after such changes.

Role Management

Managing access for hundreds or thousands of users can become overwhelming if done on an individual basis. *Role management* is the practice of defining roles based on job functions or departments and assigning users to those roles, and it simplifies access control and ensures that employees only have the privileges necessary for their work.

IAM solutions typically support three aspects of role management:

Role definitions

Roles are defined based on job functions, departments, or specific tasks within the organization. Each role is associated with the access rights and permissions that are necessary to perform its associated tasks.

RBAC

As we discussed in detail in [Chapter 3](#), RBAC links access rights to roles rather than individuals. For example, employees in a marketing department may have access to the CRM system, while only managers will have access to business intelligence tools.

Role hierarchies and inheritance

Higher-level roles inherit the permissions of lower-level roles. For example, a manufacturing company sets up supervisors to inherit the permissions of operators in addition to access to production reports and machine maintenance logs. This setup allows supervisors to perform their duties while ensuring they don't have unnecessary access to executive-level data.

Workflow and Approval Processes

In complex environments, simply provisioning accounts isn't enough. Organizations need to ensure that access is granted only after proper review and approval. For this reason, IAM solutions provide workflow and approval processes to help enforce governance and compliance through structured steps for access requests.

When a user needs to gain access to a new system or resource, they typically submit a request through the IAM solution. The system then follows a predefined workflow to process the request, and this workflow includes approval chains that ensure the appropriate stakeholders review and approve the request.

For example, imagine you need access to infrastructure code stored in a Git repository. An IAM system will accept your request and send it to your team lead, who will review and approve it. The system then forwards your request to the IT security team for final AuthZ. Once all approvals are received, the IAM system automatically provisions access and generates an audit log for compliance.

Identity Synchronization and Reconciliation

As users move through their identity lifecycle, their data must remain consistent across all systems. *Identity synchronization* reflects all changes to user attributes, such as role or department, across all platforms. Reconciliation resolves any discrepancies between systems to maintain data accuracy.

IAM solutions design their identity synchronization mechanisms to ensure that updates in one system are immediately propagated to connected systems. For example, if a user's department changes, the IAM solution updates their access rights across all systems in real time, preventing access issues and security risks. At the same time, the system reconciles inconsistencies in identity data. If that user was deprovisioned in one system, the request for a department change may need to be flagged and resolved by an IT administrator.

IAM Architecture Models

IAM solutions offer different architecture models—centralized, decentralized, hybrid, and cloud—that define the framework for how their functions operate within an organization. Each model offers distinct advantages, depending on the organization's size, complexity, and security requirements.

Centralized IAM Model

The *centralized IAM model* consolidates all identity and access management functions into a single, unified system. User identities, roles, and access rights are centrally managed, often through a directory service like Microsoft Active Directory (AD) or LDAP.

Consider a multinational corporation that uses centralized IAM across offices in different countries. By integrating all access controls into a single AD instance, the IT team can manage user accounts globally while enforcing consistent security policies. Centralized IAM allows the team to quickly disable access for departing employees and ensure that data is protected.

For developers, using a central IAM means you only need to learn one system and use one set of API calls. This makes integrating IAM capabilities into apps easier.

This model offers several benefits:

Simplified management

A single system controls all identities in the organization, such that administrators can easily manage user accounts, provision new users, and enforce access policies. For example, when onboarding a new employee, the IAM creates their credentials in a central directory, and their permissions across various systems are automatically assigned based on their role. This reduces the complexity of maintaining separate access controls for each system.

Consistent policies and monitoring

Centralized IAM enables uniform policy enforcement and centralized logging, which improves security monitoring and auditing. When an organization needs to conduct an audit for regulatory compliance (e.g., the Sarbanes-Oxley Act or GDPR), a centralized IAM system provides a clear, unified trail of user activity, making it easier to identify and address suspicious behavior.

Improved user experience

SSO is a key feature of centralized IAM. It reduces password fatigue and enhances security by minimizing password reuse.

Decentralized IAM Model

The *decentralized IAM model* distributes identity management across multiple systems or nodes. This architecture uses technologies like blockchain and decentralized identifiers that allow users to control their own identities without relying on a central authority.



Decentralized identifiers (DIDs) are globally unique identifiers that use strings of letters and numbers to act like an identifying address that is verifiable, is persistent, and does not require a centralized registry to work. DIDs are designed to allow the controller of a DID to prove control over it and to operate independently of any centralized registry, identity provider, or certificate authority.

Decentralized IAM is more common in the customer identity and access management (CIAM) space, where AuthN and AuthZ are integrated into consumer-facing applications. This gives customers more control over their identity information than they would have with a centralized IAM. It also makes it more difficult for a malicious actor to gain access, as they would need to hack into each individual's identity information separately to compromise resources.

Decentralized IAM offers several advantages:

Increased privacy and control

Users manage their own identities and the information shared with services. For example, a user interacting with an online retailer could share just enough personal data to complete a transaction without revealing other sensitive details like their full address or date of birth.

No single points of failure

Decentralized IAM is inherently more resilient to downtime and cyberattacks, as there is no central repository of user data to target. This makes it well suited for environments where privacy and data security are critical goals, such as health care systems handling sensitive patient records.

Verifiable credentials

In a decentralized system, users can present verifiable credentials that are cryptographically validated without the need for a central authority. This reduces fraud and provides assurance for both users and service providers. For example, a university could issue a digital diploma that graduates store in their decentralized identity wallets, and employers could verify it without contacting the institution.

Hybrid IAM Model

The *hybrid IAM model* combines elements of both centralized and decentralized architectures. This provides you with the flexibility to use a centralized directory for managing user access while incorporating decentralized elements that enhance privacy and user control. Hybrid IAM is often used in organizations with complex IT environments, such as a parent company with different subsidiaries, and it offers a balance between control and autonomy.

For example, an organization can use hybrid IAM to manage both employee and customer identities. Internally, the company employs a centralized system for managing employee access to corporate resources, enforcing strict security policies through SSO and MFA. Externally, it allows customers to log in using decentralized credentials stored in their own identity wallets. This enables the company to maintain strong internal controls while providing a user-friendly, privacy-enhanced experience for customers.

Hybrid IAM also provides organizations with the following:

Centralized control with decentralized flexibility

Organizations can maintain centralized control for critical access management functions while allowing users more autonomy over their personal data in certain scenarios.

Scalability and adaptability

Due to its flexibility, hybrid IAM can scale to the needs of growing enterprises, by doing things such as integrating both on-premises and cloud-based services. It also allows organizations to meet diverse compliance requirements, such as HIPAA and PCI-DSS, while offering users more control over their personal data.

Enhanced security and user experience

By combining centralized AuthN methods like SSO with decentralized identity solutions, hybrid IAM ensures strong security while giving users flexibility in managing their identities.

Table 4-1 compares different facets of centralized and decentralized architectural models. We did not include hybrid IAM because its capabilities are highly dependent on the implementation chosen.

Table 4-1. Benefits comparison between centralized and decentralized IAM

	Centralized IAM	Decentralized IAM
Data storage	Centralized database and single source-of-truth	Distributed storage across a network
Data ownership	Owned by organizations	Owned by users
Identity model	One identity combining credentials (e.g., username, password, date of birth)	One identity with multiple verifiable credentials (e.g., username, password, date of birth, credit card)

	Centralized IAM	Decentralized IAM
Control	Managed by a central authority, usually IT or security	Managed by users with IT/security oversight
Security	Single point of exposure to malicious actors	Distributed architecture and more resilient to attacks
Access control	Centralized, with limited flexibility	Decentralized, allowing granular permissions
Trust	Centralized authorities	Decentralized authority with users using verifiable credentials
Developer experience	Single set of APIs and integration path	Multiple APIs and workflows
User experience	Less control and centrally managed	More control in hands of users

IAM in Cloud Environments

In [Chapter 6](#), we'll discuss cloud security in depth, but it's worthwhile to cover its overlap with IAM here. If you're connecting apps to an IAM and potentially creating more identities to manage, there are some questions you need to ask:

Should I integrate with my organization's cloud-based IAM?

To maintain consistency with and adherence to company policies and compliance requirements, the answer is *yes*. You do not want to create or expand the problem of *shadow IT*, in which developers build their own cloud services without IT approval.

Does the IAM support the potential number of identities and identity operations needed?

Modern cloud-based IAM solutions can scale dynamically to accommodate fluctuating user loads, such as those provided by AWS, Google Cloud Platform (GCP), and Microsoft Azure. The question is whether the IAM is provisioned to handle your requirements. Traditional IAM systems may struggle to scale effectively, as they are typically architected to meet a certain load in an on-premises environment.

Does the IAM meet my app's compliance requirements?

Application workloads and data storage can span multiple regions and jurisdictions, each with its own data protection regulations. Your IAM should support compliance wherever the app is used.

Does the IAM support a hybrid cloud?

To ensure that your app can access a consistent, comprehensive, and synchronized picture of organizational identities, your IAM should provide a unified view of identities across both on-premises and cloud environments. For example, CyberArk Identity integrates with on-premises directory services (such as Active Directory) while extending identity management capabilities to cloud environments.

Key Standards in IAM

IAM solutions use widely adopted standards for managing identities and access, including SAML, OAuth 2.0, OpenID Connect, and LDAP. We discussed how these standards work in [Chapter 3](#), and here, we'll explain where they fit and how to integrate them into your applications.

SAML: Centralized AuthN for SSO

SAML is an XML-based standard that's designed to exchange AuthN and AuthZ information securely between an identity provider (IdP) and service providers (SPs). IdPs can be part of an IAM solution, but they typically don't manage user access.

SAML is a common protocol used to implement SSO and enable integration with tools like GitHub Enterprise Cloud (for enterprise accounts), Google Workspace, and internal custom applications. Employees authenticate once via an IdP such as CyberArk Identity or Azure Active Directory, and can then access multiple services without re-entering credentials. This approach simplifies user access and centralizes control, allowing administrators to manage authentication and access policies from a single point.

There are two types of SAML requests:

IdP-initiated SAML requests

Users log in directly with their identity provider, and once authenticated, they can navigate to the service provider they wish to access.

SP-initiated SAML requests

Users log in with the service provider, such as their company's web portal, which redirects them to the identity provider for AuthN.

Say you're building a web portal for students to access university services. Using SAML, students can log in once with their credentials and access all services seamlessly. The university's identity provider handles AuthN, while each service provider accepts the SAML assertion to grant access.

The SP-initiated workflow you design may look like this:

1. The student opens their browser and navigates to the university's web application, which uses an identity provider for AuthN.
2. The web application responds with a SAML request.
3. The browser passes the SAML request to the identity provider, which parses it.
4. The identity provider authenticates the student by prompting for a username and password. If the student is already authenticated, this step is skipped.

5. The identity provider generates a SAML response and returns it to the student's browser.
6. The browser sends the generated SAML response to the university's web application, which verifies it.
7. If the verification succeeds, the web application grants the student access.

The IdP-initiated workflows may look like this:

1. Instead of going to their university's web application, the student logs in to their identity provider directly.
2. Once authenticated, the student can see all the services the university has granted them access to.
3. Upon receiving an access request to a specific service, the IdP creates the SAML request and redirects the student to the appropriate SP. As with the SP-initiated workflow, this request contains information about the user and the resources they can access.
4. Once the SP accepts the request, the student is logged in.

OAuth 2.0: Securing Third-Party Access to Resources

OAuth 2.0 allows your application to securely access resources hosted by another service, without you having to track a user's credentials. Because it uses tokens instead of passwords, OAuth 2.0 reduces the risk of exposing sensitive credentials and supports secure integrations with various external services.

Consider a financial application that wants to pull transaction data from a user's bank account. An IAM solution using OAuth 2.0 enables this by letting the user authorize the application to access their banking data without sharing their password. The IAM solution then generates an access token that the app uses to fetch the data, ensuring that the user's credentials remain secure.

OIDC: Adding Identity to OAuth 2.0

OpenID Connect (OIDC) builds on OAuth 2.0 by adding an identity layer for user AuthN using JWTs. It supports IAM solutions for authenticating users and passing identity information securely between systems—particularly for SSO services for web and mobile app clients.

For example, an ecommerce platform can use OIDC to allow users to sign in by using their Google or Facebook accounts. Once a user signs in, the ecommerce site can securely retrieve and store the user's identity details, providing a seamless experience. An IAM solution would manage the AuthN process and the user's permissions within the platform.

The following example illustrates how you can configure a set of JWT claims to authorize a user named Sue Storm:

```
{
  "family_name": "Storm",
  "given_name": "Sue",
  "locale": "en-US",
  "name": "Sue Storm",
  "preferred_username": "sue.storm@fanfour.com",
  "sub": "00u8vme77nxudvxWE0h7",
  "updated_at": 1490198843,
  "zoneinfo": "America/New_York"
}
```

LDAP: Centralized Directory Services

LDAP is used to manage user information and permissions in an IAM solution. It provides a directory that can be queried by applications to authenticate users and control access to resources.

An LDAP-based IAM solution can control access to internal systems like build infrastructure, email, and intranet services. When an employee logs in to their laptop, the IAM solution queries the LDAP directory to verify their credentials and grant appropriate access rights based on their role in the organization.

Here are the key use cases for LDAP in IAM solutions:

AuthN

LDAP acts as the AuthN backend, allowing users to log in with credentials stored in an LDAP directory. This integration enables centralized AuthN across various applications and services, thus reducing the need for multiple user accounts.

User provisioning and deprovisioning

IAM systems can automate the creation, modification, and removal of user accounts in LDAP directories. This ensures that user identities are consistently managed across the organization and that access rights are updated in real time.

Role and group management

Using LDAP, IAM administrators can manage roles and groups centrally, assigning users to specific roles or groups within the directory and automatically enforcing access control policies based on these assignments.

Access control

LDAP supports IAM systems querying the directory for user attributes, roles, and group memberships to make AuthZ decisions. This ensures that access to resources is granted based on up-to-date information.

Thick clients, in which much of the application logic resides on the client device, work best with LDAP because they are less dependent on server functions to operate.

System for Cross-Domain Identity Management: Simplifying User Provisioning

The System for Cross-Domain Identity Management (SCIM) uses a RESTful API to facilitate user management operations, such as creating, updating, and deleting user accounts. It also provides a standardized schema for representing user attributes, making it easier to synchronize identity information across systems.

In an IAM solution, SCIM reduces the administrative burden of user management and ensures that changes to a user's identity—such as role changes or account deactivation—are reflected across all connected systems. While SCIM and AuthN mechanisms like SSO work together, they serve different purposes. SCIM provides an easy way to provision users across multiple domains, whereas SSO performs SCIM AuthN by verifying user credentials.

For example, an organization using multiple cloud services can integrate SCIM into its IAM solution to automate user account management. When a new employee is hired, their account is provisioned across services like Office 365, Slack, and Salesforce automatically. When they leave the company, their access to all these services is immediately revoked, reducing the risk of unauthorized access.

FIDO: Passwordless AuthN for Strong Security

FIDO supports IAM solutions by enabling passwordless AuthN, making it more difficult for attackers to compromise accounts through phishing or credential theft. FIDO fits into IAM solutions by offering strong cryptographic AuthN methods, such as fingerprints or facial recognition, to verify user identities.

For example, an IAM solution supporting the FIDO protocols can provide employees with hardware security keys for AuthN. Then, when accessing sensitive systems, employees use their key and fingerprint to log in, eliminating the need for passwords and reducing the risk of phishing attacks.

Emerging Trends in IAM

Threats will always grow in nature and number, so IAM must evolve to match them. The latest advancements in IAM, based on technologies you've already heard of, will further reduce the need for human intervention in identity management and improve the overall user experience.

AI in IAM

Traditional IAM solutions are often rule based, relying on static policies to grant or revoke access. On the other hand, AI-powered IAM solutions can analyze user behavior in real time, flagging anomalies and automating decisions to improve security across a broader range of scenarios in the following ways:

Behavioral analytics

AI models can be trained on and can analyze historical human and machine identity behavior, such as login times, device types, and frequently accessed resources. Using normal behavior as a baseline, they can then identify actions and patterns that deviate from the norm, triggering manual intervention or automated responses.

Automated decision-making

Machine learning algorithms can assign risk scores to users based on their behavior and then automatically adjust access rights accordingly. For example, if an app frequently accesses sensitive customer data but shows no signs of unusual behavior, its risk score will remain low and access will continue without interruption. However, if the same app begins to access unusually large datasets or different databases, the IAM system can increase the app's risk score and alert the security team, potentially locking out access.

Threat detection and response

If suspicious activities are identified, such as a build server being accessed by multiple users simultaneously, automated IAM actions can be triggered to mitigate the risk. Such actions could include temporarily locking the server, notifying administrators, and requiring additional verification steps.

Blockchain in IAM

Blockchain enables decentralized identities by providing a secure, tamper-proof ledger for storing and verifying identity data. Blockchain's transparency and immutability make it a natural mechanism for improving how IAM solutions secure and maintain identities.

There are various use cases for blockchain in IAM, including the following:

Identity verification

Instead of relying on a central authority to validate user credentials, organizations can use blockchain to confirm identities based on data recorded in a distributed ledger.

Smart contracts

Self-executing contracts with terms of the agreement directly written into code can automate IAM processes like provisioning and deprovisioning, thereby

ensuring that access rights are granted or revoked based on predefined rules. For example, a smart contract could be used to automatically revoke access to patient data after a hospital contractor's project ends. This eliminates the need for manual intervention from the IT team.

Audit and compliance

Blockchain can be used to store a transparent and immutable record of IAM transactions per user, per role, or for the entire organization. This is invaluable for audit and compliance purposes because it lets organizations demonstrate that access rights were granted, modified, or revoked according to established policies.

Summary

If you've never had to configure and manage multiple services in an enterprise environment, it may be hard for you to see the benefits of using an IAM solution. However, for many applications, IAM saves time and reduces the risks of a security breach. Instead of juggling API keys and potentially losing track of which users can access what systems, IAM solutions control all access and user information you can leverage in one convenient place.

If you want to further explore IAM solutions and best practices, we recommend that you use the following resources and references:

- [“What Is Identity Provisioning”?](#)
- [“Security Best Practices in IAM”](#)
- [*Buyer's Guide: What to Look for in an Identity Security Platform*](#)

Secrets Management

Up to this point, we have discussed the fundamental concepts and practices that apply to identity security as a whole. It's important for you to understand these principles because they lay the foundation for any practical identity security exercise you'll perform on the job. They also give us the necessary background as we move into the rest of this book: practical methods of implementing identity security.

Secrets management is the core activity behind securing machine identities. While IT environments and use cases vary among groups and organizations, every app, script, automation tool, and other machine identity relies on some form of privileged credential to access other apps, scripts, tools, and data. These secrets are essential for authenticating and authorizing users, services, and applications to protect data, communications, and business reputations.

The most common types of secrets are as follows:

- Privileged account credentials
- SSH keys
- Passwords
- API keys
- Certificates
- Encryption keys

Nearly every enterprise application and infrastructure component interacts with secrets, with the potential exception of small, air-gapped embedded devices. Such an interaction could be a web application needing credentials to connect to a backend database, a timekeeping service using an API key to access a Google Maps API to validate employee mileage reimbursement, or an EHR microservice using a private key to authorize the transmission of prescription orders to a pharmacy. These secrets must be securely stored, accessed, and managed throughout the software lifecycle to prevent malicious actors from gaining access to them.

As we discussed in “[The Real-World Impact of Identity Security](#)” on page 6, ineffective secrets management can have disastrous consequences. The challenge developers have is threefold:

Secrets are everywhere.

They are integral to containerized applications like Red Hat OpenShift and Kubernetes; automation processes such as Ansible Playbooks, Puppet, and Chef; and business-critical applications, including both internally developed and commercial-off-the-shelf solutions. They are also crucial to security software, IT management software, CI/CD toolchains, and more. They can appear in new web applications and lie buried in legacy mainframe code. If left unmanaged, this *secrets sprawl* makes it harder to keep resources secure as time goes on. This underscores the importance of effective secrets management to minimize exposure risks.



Secrets sprawl is the uncontrolled and opaque distribution of sensitive information throughout the organization, like passwords, encryption keys, and other credentials. It results from many developers, applications, and systems creating and using their own secrets without proper secrets management practices in place.

It's easy to forget or bypass good secrets management.

The scenarios under which a secret can be exposed, either intentionally or unintentionally, are impossible to count. For example, a developer writing a script may choose to hardcode a secret, making it easier for a hacker to exploit if the file is committed to a version control system like Git. Without *rotation*—meaning a policy to update credentials periodically—secrets can remain static for extended periods, thus increasing the risk of compromise. Also, an IT administrator may grant excessive privileges to a resource or manually share a password with a developer to resolve a bug. There are unintended actions too, such as committing typos in configuration files or forgetting to let IT know when a secret is no longer needed.

Automated processes can pose significant exposure risks.

Rapid development processes open up the potential for many types of secrets exposure. Automation scripts, cron jobs, background services, CI/CD tools, and more can access protected data, scale at unparalleled rates, and execute business processes instantaneously. For example, a Puppet server with admin access spinning up and spinning down virtual test machines at scale could pose a denial-of-service risk if a hacker gains access to its credentials. As Codecov and other well-publicized security breaches demonstrate, automated processes are

susceptible to sophisticated cyberattacks, which can occur suddenly and spread rapidly. We'll discuss security automation in depth in [Chapter 8](#).

Going deeper into the challenges and pitfalls of secrets management and understanding the practices and tools you can use to overcome them will help you improve how you deal with machine identities to protect your apps and infrastructure.

Why Does Secrets Management Matter?

Secrets management isn't a single action or tool, but rather, it's the implementation of different techniques to ensure secrets are controlled in ways that minimize exposure to nonauthenticated users and machine actors. When implemented well, it can prevent secrets sprawl and support other cybersecurity activities such as compliance and continuous monitoring and improvement.

There are different ways in which secrets are stored, including the following:

Plain text/hardcoded

This is a human-readable format, and it's the most insecure method because anyone who can read the data can gain access to the sensitive resource. The [Uber breach](#) is a well-known and costly example of an attacker using hardcoded credentials to gain access to a system.

Hashed

Hash algorithms provide a one-way transformation of a secret to a hash value that cannot be converted back into the original input. Hashes are commonly used when storing user passwords since it's easy to check whether the input matches the hash, and a stolen hash doesn't expose the password. As hashed passwords can be attacked with GPUs or *rainbow tables* (precomputed tables of reversed password hashes used to crack passwords in a database), many companies choose very strong hashing algorithms using 512 bits or more and use "salting" (and sometimes "peppering"), a technique to increase the strength of the hashing process.

Encrypted

In *encrypted* secrets storage, the plain-text secret is transformed into a ciphertext version using a complex mathematical formula and a cryptographic key. This is a two-way function, meaning the ciphertext can be returned to plain text by using the key to decrypt it. An attacker who accesses an encrypted secret cannot decrypt it without having the key.

Secrets follow a lifecycle that your applications and management tools must align with to ensure secure operations:

Creation

Secrets must be securely generated and encrypted, and they must also be assigned the absolute minimum privilege level necessary to support their required use cases.

Rotation

Secrets must be rotated frequently, according to a defined schedule, to minimize the risk of stolen credentials being exploited by malicious actors and to reduce the likelihood of users reusing them.

Revocation

To avoid potential compromise, you must securely revoke access to credentials when they're no longer needed or after they've been compromised.

Expiration

Secrets must expire after a defined period of time to reduce the risk of exposure over time.

For example, assume you're writing a NoSQL script to access a MongoDB authentication (AuthN) database named *shire* with a user that has read and write permissions. This user, *frodo*, has a password that was created by a human and stored in the database. At a minimum, this password is known by three entities:

- The MongoDB database, as it's the entity granting and rejecting access
- Your script, as it needs to submit the username-password combination to do its job
- The human who created the username and password in the first place

Depending on how the credentials are stored, a fourth entity may be involved: you. They may have been passed to you over email or Slack or stored in human-readable format in the database. (MongoDB doesn't enforce encrypted credentials.) They could also be stored in a config file resident in a Git repository, meaning there could be exposure to more entities who clone the file and claim they're *frodo* to gain access to the database. You can see how easy it is for secrets sprawl to occur.

If you decide to hardcode *frodo* and his password into your script, as illustrated in the following MongoDB Shell code sample, anyone who accesses it now and in the future will also have database access:

```
mongosh --port 27017 --authenticationDatabase \
  "shire" -u "frodo" -p "lembas"
```

Secrets management aims to prevent this spread of information: rather than store human-readable credentials in exposed resources, your script would access a secrets management tool to retrieve encrypted credentials and send them to the database over encrypted transmission methods. You may even have access to a shared identity provider, like OIDC or LDAP (which we discussed in [Chapter 3](#)), to offload more work and risk from your script.

Secrets management tools also help you enforce best practices, such as regularly rotating *frodo's* password and logging all database accesses for security analytics and compliance purposes.

Principles of Secrets Management

As teams and dependencies grow, the secrets lifecycle must be secured across applications, services, infrastructure files, and other software artifacts in a purposeful and consistent manner. It's easy for developers and end users to forget or bypass secrets management principles, and poor lifecycle management can leave outdated or unused secrets lingering in the system—where they'll become easy targets for attackers.

That's why it's important to understand and integrate the following principles into your development process to ensure effective secrets management at every lifecycle stage.

Principle #1: Encryption

The key to effective secrets management is ensuring that sensitive information is encrypted for added security. Secrets should be encrypted at rest and in transit, using strong and up-to-date encryption algorithms and keys, such as TLS for data transmitted between clients and servers and AES-256 for database field-level encryption.

Essentially, you should never store or send secrets in plain text.

Your encryption keys are considered secrets and must be protected and managed. We therefore recommend that you adopt a *key management service* (KMS)—such as the [AWS Key Management Service](#) or [Google Cloud Key Management](#)—to automatically create, store, and manage keys for different application components. Many secrets managers include these features so that you can work with secrets and keys within one known environment.

Effective key management also means avoiding building and maintaining your own encryption features. KMS and secrets management tools are designed to offload and scale encryption services to support any developer's need to create, store, and manage keys safely outside their applications. Some solutions offer encryption as a service (EaaS) to give large or distributed organizations a subscription-based encryption model to protect their data. For example, you can use an EaaS provider to allow users

to create, rotate, and revoke keys across a set of web applications that are used for EHR management between pharmacies, customers, and a central distribution center.

Principle #2: Access Control

Access to secrets should only be granted to the entities that need them, and only for the purposes that they are intended for. Access control ensures that secrets are not shared or disclosed to unauthorized or unnecessary parties.

As we discussed in [Chapter 3](#), it is important to follow the principle of least privilege, in which users are granted privileges only if they are necessary to perform their role. When access to secrets is no longer needed, it should be revoked. If privilege escalation is necessary, such as for an urgent bug fix or unplanned maintenance, it should be carefully considered and the privilege should be revoked after a limited time.

The most common method of access control for secrets, RBAC, should be coupled with MFA. While RBAC is effective at authorizing access to secrets based on an established policy, it's insufficient protection if the secrets provider provides a secret to an imposter. Therefore, strong multifactor AuthN, using attributes only available to trusted resources (such as certificates), provides an additional layer of secrets protection.

Principle #3: Monitoring and Auditing

Your organization needs full visibility into how, why, and when secrets are being used. Tracking who has access to what secrets, from creation to revocation, helps improve secrets management processes, feeds into tools like IAM and threat analytics platforms, and informs remediation activities when there's a breach.

Your secrets management tool needs comprehensive observability features to support various cybersecurity activities and technologies. It should create detailed logs every time a human or machine identity is authenticated to retrieve a secret and when the secret is used to access a resource—including when AuthN and access fail. Logfiles, real-time alerts, and reports enable visibility into and accountability of the secrets lifecycle to help IT and security teams do their jobs.

The following data should be captured by your secrets management tool or, in the absence of one, your application code:

- The type of secret (JWT, encryption/cryptographic keys, etc.)
- The secret's creation, usage, rotation, and deletion date and time
- Who (which user or role) created, used, rotated, and deleted the secret
- What application or system created, used, rotated, and deleted the secret

Monitoring and auditing also help you take action. Whether it's provided by a centralized secrets manager or built into your own code, logging of secrets accesses and usage helps you understand the security state of your application (e.g., who's logged in, who has historically logged in) and triggers activities in the secrets lifecycle. For example, if an audit log shows that an encryption key for database access was last updated a year ago, you can decide to revoke it and create a new one. Audit data, from your own logs or a secrets manager, can also be used to prove compliance with cybersecurity regulations.

Principle #4: Compliance

Any secrets management activity should feed into and comply with industry regulations, standards, and laws around information security and data privacy. Depending on your organization, these include local, national, and global mandates. Some examples are as follows:

- Health care software operating in the USA must comply with **HIPAA** to protect patient health information.
- Financial institutions and retailers fall under **PCI-DSS** regulations that protect credit card data.
- Systems must comply with region-specific privacy laws, such as the **General Data Protection Regulation (GDPR)** in Europe, the **Personal Information Protection and Electronic Documents Act (PIPEDA)** in Canada, and the **Federal Information Security Modernization Act (FISMA)** and the **California Consumer Privacy Act (CCPA)** in the United States.
- Companies that operate in certain industries, such as defense or manufacturing, are subject to sector-specific regulations such as **ISO/IEC 27001:2022 Information security, cybersecurity and privacy protection**, **ISO/SAE 21434:2021 Road vehicles—Cybersecurity engineering**, and various **NIST standards**.

These standards typically define processes and procedures that support a secure development lifecycle (SDL), including secrets management. This SDL often requires the generation of evidence to prove that secure practices are being followed. This involves conducting software security reviews and design documentation during development and collecting audit logs and security metrics during production.

For example, ISO 27001:2022, Annex A Control 8.28 specifies guidelines for preventing security risks and vulnerabilities arising from poor software coding practices for both internally developed and third-party software. These guidelines include the following:

- Prohibiting the use of insecure software coding methods, such as unapproved code samples and hardcoded passwords

- Tailoring secure coding principles for each programming language and technique
- Implementing and configuring software development tools (including IDEs) to ensure the security of all code created
- Considering the use of coding techniques, such as following the least privilege principle, before implementing software in a live application environment

Your responsibility here is to understand your organization's compliance requirements and ensure that your applications are designed, implemented, and maintained in accordance with the applicable standards. A good secrets manager can help here, as it's designed to enforce compliance with many standards and collect the necessary information without you having to worry about it.

Principle #5: Testing

As with all processes and technologies, you and your organization should test the implementation of secrets management to ensure that it meets requirements and identify any gaps that need addressing. Testing helps validate the functionality and quality of secrets management—such as the correctness, completeness, and security of secrets—by using methods to simulate common and unexpected scenarios.

This also includes running scenarios where the secrets management service becomes unavailable, such as planned downtime or an internet outage. If the service goes down, how will your applications that require its stored credentials work? Running these scenarios often leads to the development of stronger backup procedures and tools, including an emergency “break-glass” process to restore services if all else fails.

Principle #6: Automation

Reducing the human elements in the secrets management mix better serves complex enterprise environments with different teams that operate under different conditions. Trying to do everything manually, such as by requiring IT personnel to triage and action API key requests or track secrets in a spreadsheet, increases the risk of error, inconsistencies, and frustration. It's also impossible to support development, security, and operations (DevSecOps); security analytics; and compliance activities if the people running them are forced to wait for other people to collect and send the data.

Automating secrets management keeps the processes of creating, validating, and integrating secrets robust and scalable. It also helps integrate secrets more efficiently into DevOps and CI/CD pipelines. Examples of this are as follows:

- Automating secrets management eliminates the human tendency to create “easy” passwords that are more susceptible to compromise.

- Automatically rotating secrets on a regular schedule minimizes the impact of compromised credentials.
- Automating secrets management allows the use of **dynamic secrets**, which are ephemeral and short-lived secrets that automatically expire once a predetermined time-to-live (TTL) is reached.
- Automating secrets management helps you enforce RBAC and the principle of least privilege whenever an access request is created and granted.
- Automating mundane tasks, such as secrets creation and revocation, frees up your time to work on code and reduces the possibility of human error.
- Automation helps you adhere to and report on compliance with many standards, such as those requiring secrets rotation and evidence of user login tracking.

Principle #7: Centralization

Proper secrets management allows you to safely and securely create, store, and manage secrets in a central repository or system. This prevents the inconsistencies and overhead incurred when individuals and teams manage secrets on their own, and it also prevents secrets sprawl.

There are plenty of opportunities for junior employees or those who are too busy to consider security to leave plain-text passwords in code or API keys in config files. Worse, if IT and security teams are not involved in the selection process, this behavior may lead to the creation of a mix of open source and commercial tools and thus may lead to increased risks.

An additional risk is the spread of *security islands* within an organization. This occurs when a security tool or platform is deployed to manage secrets but does not facilitate interoperability with other tools, policies, management, and audit data. These islands make it much harder to manage secrets across the organization, often leading to a shadow IT that does its own thing independently from the actual IT department.

While storing secrets in one place may seem counterintuitive, it's far easier to harden, protect, and avoid security islands using one centralized repository or a series of redundant repositories.

Everyone needs boundaries around what they can and cannot access, and using a centralized tool is the best way to manage access control and reduce risks consistently and repeatedly across the organization.

A side effect of centralization is the risk of creating a *secret zero* situation, in which all your secrets fall under a tool or platform that a single master key can access. This often happens at home when all your cloud services passwords (like streaming and banking services) are stored in a password manager locked under one easy-to-

remember password. You can mitigate this risk of having this single attack point by using multiple AuthN mechanisms to authenticate access to your master key.

Secrets Management in Code

Your applications and scripts can access secrets in any number of ways, including the following:

Environment variables

Environment variables are key-value pairs that are set in the operating system or the application's runtime. While they're easy to use and integrate, environment variables are equally easy to leave unsecured and unreliable, as they can be leaked, overwritten, or lost. It's difficult to apply different permissions to these variables, as they are only on the system or per-application level.

Configuration files

These are files that contain your application's settings and parameters. They're flexible and portable, but they're also insecure and inconsistent. They can be exposed, modified, or out of sync, and they're usually written to disk, which makes them vulnerable to tampering if disk-level access is obtained.

Vaults

Secrets can be stored and accessed through dedicated systems or services that provide secure storage and access such as [HashiCorp Vault](#), [CyberArk Secrets Manager Self-Hosted](#) (formerly known as Conjur Secrets Manager Enterprise), and Conjur Open Source. Vaults are secure and scalable but can also be complex and costly, thus requiring additional infrastructure and maintenance.

Cloud services

Secrets can be stored and accessed through services from cloud providers or platforms that offer secrets management, such as [CyberArk Secrets Manager SaaS](#) (formerly Conjur® Cloud Secrets Manager), [CyberArk® Secrets Hub™](#), and [AWS Secrets Manager](#). Cloud services are convenient and reliable but can also depend on external providers, thus incurring additional fees.

Regardless of the method you use to access secrets, you should take additional steps in your code to minimize exposure risks. Relying on the operating system or cloud provider alone to provide protection isn't sufficient, especially if your app will be ported to other development or production environments outside your control.

In addition to the secure coding practices we described in [Chapter 2](#), here are techniques you can use to protect secrets in your code:

- Do not hardcode secrets as static text *anywhere!* (We're belaboring this point because it is the most common method of secrets exposure.)
- Externalize secrets as much as possible by using a secrets manager. Or, if absolutely necessary, you can use [local environment variables](#) to enforce best practices like credential rotation without requiring code updates and potential rebuilds. If you're developing a mobile app, consider storing and accessing all secrets on the server side, which is easier to secure and manage than multiple client instances out in the wild.
- Protect secrets in memory by using techniques such as encrypting memory space (if your OS or hardware supports it) and zeroing memory once a secret has been used.
- Avoid using immutable data structures, like Java `String`, to store secrets. You cannot force these structures to be garbage collected, and there will be an indeterminate period when stored secrets may be accessible by an outside source.
- To minimize exposure risks at runtime, use code obfuscation to make it harder for attackers to reverse engineer and extract secrets like JWT and API keys. Obfuscation techniques typically include renaming variables, modifying control flow structures, and inserting dummy code.
- Ensure all secrets accesses are logged, either from your application's logger or an external secrets management tool.

You should also ensure that sensitive files are not stored in source code repositories. For example, you can use the `.gitignore` file in Git to prevent files with environment variables and hashed database keys from being stored in the repository.

Secrets Management Tools and Frameworks

At this point, you may be wondering how to implement secrets management effectively. Security experts and organizations have built many different types and technologies to manage secrets, from open source to enterprise-class products. Your job isn't to reinvent the wheel; rather, your goal is to support your company's security technologies if they exist or recommend tools that best fit into your workflow.

To illustrate how a secrets management solution works, we'll explain how you can interact with a popular option, [CyberArk Secrets Manager, SaaS and self-hosted](#).

CyberArk Secrets Manager centralizes access to secrets across cloud and DevOps environments, integrating with your application to provide many of the practices

described in this chapter. As with many secrets managers, your primary interactions with CyberArk Secrets Manager after installation and setup will be as follows:

- Setting up a secrets policy
- Creating a secrets store
- Using a secret

Here, we'll assume your application or script is accessing secrets within a Kubernetes application using *External Secrets Operator (ESO)*. ESO is used to synchronize secrets from external APIs into Kubernetes, and it supports different secrets management systems, such as AWS Secrets Manager, HashiCorp Vault, Google Secret Manager, CyberArk Secrets Manager, and Conjur Open Source.

To install and set up ESO for yourself, read the ESO “Getting started” guide.

Setting Up a Secrets Policy

As a common secrets management best practice, CyberArk Secrets Manager and Conjur Open Source uses an RBAC authorization (AuthZ) model (the example in this section uses Conjur Open Source but it also applies to CyberArk Secrets Manager). Each transaction between your app and Conjur Open Source includes three components:

Role

This is the person or technological asset that performs the action.

Privileges

These are the action or actions (read, write, and/or execute) the role is allowed to perform.

Resources

These are the secret(s) under protection.

Conjur Open Source uses policy statements to manage your organization's roles, privileges, and resources. For example, a group of developers called `secrets-users` could have `read` and `execute` access to a database `db`. These policy statements are stored as YAML files.

For this scenario, here is the Conjur Open Source policy file, *policy.yml*:

```
- !policy
  id: secrets
  body:
    - &variables
      - !variable test_secret
      - !variable db/url
      - !variable db/username
```

```

- !variable db/password
- !variable db/platform

# declare two users and a group
- !user frodo
- !user samwise
- !group secrets-users

# add members to the group
- !grant
  role: !group secrets-users
  members:
  - !user frodo
  - !user samwise

# give the group privileges on resources
- !permit
  resources: *variables
  role: !group secrets-users
  privileges: [read, execute]

```

To load this file into Conjur Open Source from the command line, you run this command (assuming the policy file is saved as *policy.yml*):

```
conjur policy load -b root -f policy.yml
```

Creating a Secrets Store by Using ESO

ESO uses a configuration object called a *SecretStore* to specify how the external secrets API is accessed to separate AuthN from the actual secrets data. Before you can create and use secrets, you must create this SecretStore, as in the following example of a *conjur-secret-store.yml* file. Here, we're using JWT for AuthN:

```

# Create a secret store that uses a Kubernetes Service Account
# JWT to authenticate to Conjur
---
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: conjur-jwt
  namespace: demo-apps
spec:
  provider:
    conjur:
      url: https://my_conjur_url
      caProvider:
        type: "Secret"
        name: "conjur-details"
        namespace: "demo-apps"
        key: "certificate"
      auth:
        jwt:

```

```
account: conjur
serviceID: eso-env
serviceAccountRef:
  name: test-app-sa
  audiences:
    - https://my_conjur_url
```

Setting a Secret

Here's how you set a secret `test_secret` from the Conjur Open Source CLI:

```
conjur variable set -i "secrets/test_secret" -v "MyS3cretContent!"
```

Retrieving a Secret

And here's how you retrieve the secrets `test_secret`, `url`, `username`, and `password` from the SecretStore into your Kubernetes app:

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: external-secret
  namespace: demo-apps
spec:
  refreshInterval: 10s
  secretStoreRef:
    name: conjur-jwt
    kind: SecretStore
  target:
    name: db-credentials
    creationPolicy: Owner
  data:
    - secretKey: test_secret
      remoteRef:
        key: secrets/test_secret
    - secretKey: url
      remoteRef:
        key: secrets/db/url
    - secretKey: username
      remoteRef:
        key: secrets/db/username
    - secretKey: password
      remoteRef:
        key: secrets/db/password
```

Summary

In this chapter, you have seen a variety of practices to consider when you're strengthening the security of secrets used by your applications and organizations:

- Deploying practices such as encryption and access control across your secrets lifecycle
- Adopting secure coding practices to minimize exposure risks
- Ensuring that secrets management supports auditing and compliance activities

The best approach for you depends on the nature and scale of the secrets in use and whether you can integrate with an automated, centralized secrets management tool.

For further learning and exploration, we recommend that you consult the following resources and references:

- [OWASP Secrets Management Cheat Sheet Series](#)
- [NIST Special Publication 800-57 on key management](#)
- [ISO/IEC 27001:2022 standard on information security management](#)
- [CyberArk Secrets Manager \(SaaS and self-hosted\)](#)
- [AWS Secrets Manager documentation](#)
- [Azure Key Vault documentation](#)
- [CyberArk Secrets Hub](#)
- [Google Cloud Secret Manager documentation](#)

Cloud Security and Cloud Native Considerations

Attackers don't break in. They log in.

—Unknown

While some IT pundits would have you believe that cloud computing makes any app fly, we're a little more down to earth here. Cloud service providers (CSPs) and on-premises cloud deployments (sometimes referred to as *private clouds*) can offer significant benefits in scale and time to market. They can also present major security risks if you're not careful.

Before cloud environments existed, attackers had to overcome protections around on-premises infrastructure. Applications, servers, and networks were secured by IT departments working with users within a defined perimeter. Now, however, we have cloud deployments whose very nature is to break down barriers, spread workloads, and distribute responsibility. While this is great for cost savings, rapid delivery, and scalability, the cloud's dynamic environment and shared responsibility model open up potential attack vectors and call traditional security practices into question.

Microsoft's 2023 State of Cloud Permissions Risks Report found that cloud environments have over 40,000 permissions that can be granted to identities. Of those, over 50% are “capable of causing catastrophic damage if used improperly.” For example, a single compromised user with misconfigured permissions can potentially access surrounding infrastructure and sensitive data resources.

In this chapter, we'll cover the challenges of and best practices for cloud security related to developer roles. We'll cover misconfigurations, the four Cs of cloud native security, and a comparison between popular cloud-based IAM services.

Background on Cloud Computing and Security

Before we start discussing cloud security, you need to understand the different models that classify cloud computing. They are as follows:

- Infrastructure as a service (IaaS) provides virtualized resources over the internet, such as virtual compute and storage.
- Platform as a service (PaaS) provides tools and services for application development, testing, and deployment, so development teams don't have to worry about the infrastructure.
- Software as a service (SaaS) provides applications that are accessible through a browser or API.

These models can be deployed in different ways:

- Public clouds are operated by a third-party CSP that rents access to users, usually on a pay-as-you-go basis or a consumption-based model. Popular providers are AWS, Microsoft Azure, GCP, and Alibaba.
- Private clouds are hosted and maintained within a single organization, with infrastructure onsite or offsite.
- Hybrid clouds combine public and private cloud models to share data and workloads. Popular providers include Microsoft Intelligent Cloud (Azure), Oracle Cloud, and HPE GreenLake.
- Multicloud architectures use multiple CSPs to balance features and workloads across the providers that best suit them. This tends to optimize performance and cost, and it avoids provider lock-in. However, it can be extremely challenging to maintain security across all deployments.

Regardless of the model and provider you use, you share responsibility for ensuring that security best practices are applied. It's like an office building where you have a security guard and gate that requires badge access (the CSP) and the floor you work on requires you to lock rooms and filing cabinets.

Your responsibility is determined by the AWS service that you use. You are also responsible for other factors, including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

—“Security in AWS Organizations”, AWS

In other words, if your app passes insecure keys from an internal system to an Amazon S3 Bucket, then Amazon won't take responsibility for any security risks.

The practices covered in this book also apply to cloud security, including trusting no actor (human or machine), applying the principle of least privilege, and always

managing your secrets. With the dynamic nature of cloud deployments and the lack of static resource boundaries, you can never guarantee who has access to the data and resources you've deployed.



Security responsibilities are shared between you and your chosen cloud provider. In what is known as the *shared responsibility model*, the CSP protects certain aspects of its infrastructure while the customer secures its cloud-hosted data and applications. For example, in a SaaS model, the CSP secures the infrastructure that runs all its services, including its hardware, software, networking, and physical facilities. You and your organization are responsible for securing the guest operating systems, your applications, and your data, and for ensuring cloud services are configured correctly. Different models shift responsibility between the CSP and the consumer.

Most cloud deployments are fed by automated development pipelines with resources, data, and access points that could be compromised. Because it includes development and production, the cloud presents the following unique risks for you and your infrastructure and IT teams:

Reduced control

Using public cloud services and infrastructure means configuring their security policies, practices, and tools to meet your requirements and supplementing them with your services, if necessary. The actual separation depends on the CSP and the services you use. For example, data deletion cannot be guaranteed in a multitenant environment if you cannot verify that the CSP securely deletes data across all virtual and physical storage.

Reduced visibility

Services can be provisioned and decommissioned easily through a CSP's dashboard and tools, making it harder for you to know what's happening. If development teams use their cloud access to provision services without IT's consent, it increases the chances of attackers bypassing security controls and introducing new risks.

API compromise

CSPs are managed through an extensive and complex set of APIs. These APIs operate on the entire stack, from OS to the application, making it almost impossible to secure all the touchpoints with potential access to your resources. If attackers discover a weak link, your applications will be put at risk.

Tenant compromise

In a multitenant environment, your application's protection is only as strong as the policies and tools of the weakest tenant. If proper security practices and separation are not ensured, an attacker that compromises one tenant may be

able to gain access to yours, via a tactic known as *tenant hopping*. If the CSP outsources parts of its infrastructure and operations, then there may be further security risks to your organization.

Watch Out for Security Misconfigurations

Despite all the protections and certifications offered by your cloud provider, the human factor is still the biggest threat to cloud security. The *2024 Cloud Security Study by Thales* reports misconfiguration or human error as the largest root cause of cloud data breaches. One example, the *breach of Toyota* cloud environments that took about a decade to discover, shows that misconfiguration issues can be difficult to detect, even if they're relatively easy to fix.

The following example illustrates how improperly configured permissions—which constitute a security risk on the *OWASP API Security Top Ten list*—can cause a potential security exploit. This code creates an Amazon S3 bucket using *Terraform* (a popular infrastructure-as-code tool for automating infrastructure tasks) and sets up its access policies:

```
resource "aws_s3_bucket" "s3_bucket" {
  bucket = "my-bucket"
}

resource "aws_s3_bucket_public_access_block" "s3_bucket" {
  bucket = aws_s3_bucket.s3_bucket.id

  block_public_acls       = false
  block_public_policy     = false
  ignore_public_acls     = false
  restrict_public_buckets = false
}

resource "aws_s3_bucket_policy" "s3_bucket_policy" {
  bucket = aws_s3_bucket.s3_bucket.id
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Principal = "*"
        Action = "s3:GetObject"
        Resource = "${aws_s3_bucket.oidc_bucket.arn}/*"
      }
    ]
  })
}
```

Here, the **Amazon S3 Block Public Access feature**, which is defined through `aws_s3_bucket_public_access_block`, sets all access settings to false. This tells Amazon S3 to remove all access blocks as follows:

`block_public_acls`

This specifies whether public ACLs for the bucket are blocked (e.g., whether attempts to set permissions will fail if the specified ACL allows public access).

`block_public_policy`

This specifies whether public bucket policies are blocked (e.g., whether attempts to apply public policies are rejected).

`ignore_public_acls`

This specifies whether public ACLs for the bucket are ignored.

`restrict_public_buckets`

This specifies whether public bucket policies are restricted (e.g., whether only the bucket owner and AWS Services can access the bucket if it has a public policy).

Additionally, the policy is set to public through `Effect="Allow"`.

This example illustrates that while the cloud services you interact with may be functionally correct, they may not adhere to security best practices. Also, while it used to be simpler to understand and set access policies with AWS S3, its current complexity makes it harder to prevent accidental leakage. On the other hand, if the principle of least privilege had been applied, you would have less to worry about.

The next example shows Terraform code with a hardcoded username and API key. This may not pose an issue when used in a development environment under your control, but it's a security risk if pushed to a publicly accessible resource:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.37.0"
    }
  }
  required_version = ">= 1.2.0"
}

provider "aws" {
  region = "us-east-1"
}

resource "aws_iam_user_ssh_key" "user1_ssh" {
  username = "bruceb"
  encoding = "SSH"
```

```
    public_key = "dGhpc19pc19ub3Rfc2VjdXJl"  
}
```

This type of issue may have led to the Toyota security breach we discussed earlier, in which **sensitive information** belonging to over two million customers was leaked. We can learn something from Toyota’s response to the breach:

We believe that this incident also was caused by insufficient dissemination and enforcement of data handling rules; since our last announcement, we have implemented a system to monitor cloud configurations...We will also work to prevent a recurrence by thoroughly educating our employees once again.

The lesson is that cloud security is best served by combining education and tools to reduce risks.

Cloud Native Versus Lift and Shift

Lift-and-shift applications typically have a monolithic architecture, and they are built for on-premises deployment and are migrated to a cloud later. They don’t follow cloud design, development, or deployment principles—such as dynamic scaling and continuous integration—but they can be an easy option for rapid deployment without much rework.

The Cloud Native Computing Foundation (CNCF) defines *cloud native systems* as “loosely coupled systems that interoperate in a manner that is secure, resilient, manageable, sustainable, and observable.” Furthermore, it states that these architectures typically consist of “some combination of containers, service meshes, multitenancy, microservices, immutable infrastructure, serverless, and declarative APIs.” Development processes rely on automation to achieve the velocity necessary to keep cloud applications current and running.

Cloud native development builds on cloud practices and technologies from the start. Cloud native applications tend to be smaller, service-based components that work independently and consume fewer system resources. Developers use microservice architectures, containers, and orchestration to deploy and distribute workloads, enabling the speed and scalability necessary to support dynamic demands.

The challenge of cloud native security is ensuring the protection of all resources, from the developer’s desktop to the cloud container, without sacrificing the cloud’s benefits. Whereas traditional applications are relatively unchanging in design and implementation, cloud native architectures are highly fluid, and their implementations are dynamic. It’s common for systems to add microservice instances to meet demand during the day—allocating work across multiple cloud resources—and then scale them back during nonpeak times. To meet changing needs, large enterprises deploy several times per day. For example, Airbnb **deploys over 125,000 times per year**, using multicluster Kubernetes.

CSPs have native security services—such as IAM, data protection, and compliance mechanisms—to make your job easier. However, they don't eliminate your need to secure your development pipeline and application deployments. There are three elements to consider when developing your cloud native security strategy:

Development environment

The tools you use to write, build, test, and deploy your code are often connected to cloud resources—which are sometimes hidden from you. For example, your IDE may connect to a cloud-based ticketing system or use cloud resources to compile code in the background.

Management platform and services

The main access point to a CSP's cloud services is its CLI or management console. These tools configure, control, and monitor various aspects of your cloud deployment, and they require some form of access control to ensure only authorized users can use it.

Infrastructure workloads

These are the resources used to perform cloud computing, including memory, databases, containers, virtual machines, storage, and cloud native apps.

When you're working with cloud native apps, it's crucial for you to implement a strategy that covers all these bases.

The Four Cs of Cloud Native Security

If there's one certainty in cybersecurity, it's that no system is inherently “hack-proof.” However, what we can do is break down cloud native systems into smaller chunks with layers of controls and protections that minimize the blast radius of a breach.

The four Cs of cloud native security divide responsibilities into manageable layers that follow most cloud application architectures: cloud security, cluster security, container security, and code security. These categories form a nested security structure that protects resources from the lowest level of the cloud stack to the highest and fosters the sharing of responsibilities across teams.

Cloud security

Cloud security involves the practices, policies, and tools used to secure cloud infrastructure. The CSP usually manages this level to protect cloud resources from threats such as unauthorized access, denial-of-service attacks, and vulnerabilities that may compromise resources. CSP-provided security services include IAM, network security, encryption, and security monitoring, but you are always responsible for ensuring that they are configured correctly.

Your responsibility here is to understand what protections are available and how to configure them to serve your applications best.

Cluster security

A *cluster* is a group of machines operating together to run containerized applications. Securing a cluster requires knowing its components, such as Kubernetes pods, applications, and their configurations. Working with your infrastructure team, you should know how the cluster is configured and operates to build the right security practices into your code.

For example, if you're deploying to a cluster where access controls are unknown, it might be a good idea to employ authentication (AuthN) mechanisms within your app to protect sensitive data.

Kubernetes **provides some tips** on how to secure a cluster, including the following:

- Control access to the APIs that are used to manage the cluster through AuthN, authorization (AuthZ), and encryption.
- Set container privileges to limit who can access them.
- Restrict network access to only those users who are authorized to use the cluster's components.
- Limit resource usage (CPU and memory) to prevent malicious users from running unreasonably high or low workloads.
- Enable audit logging to support remediation analysis in the event of a breach.

Container security

Containerization packages an application, libraries, frameworks, and dependencies into a single unit that can be deployed and scaled easily. There are four security questions to ask about your container deployments:

How secure is the container image?

A container image holds the code, runtime, system libraries, configuration files, and settings your application needs to run. These components should be free from vulnerabilities and have the latest security patches applied.

Can the container be trusted?

You should avoid using and interacting with images from untrusted sources. Signing images built by your team and deploying only images signed by known sources helps you to maintain a system of trust across your organization.

Does the container follow the principle of least privilege?

Every container should only have the minimum privileges required to perform its function.

Is the container's build pipeline secure?

To minimize the risk of an attacker compromising containers, you must extend security controls to the build server, code repositories, and developer workstations.

Code

To protect application code, you should use the secure coding practices explained in [Chapter 2](#). These include input validation, encryption, resource and session management, and identity security practices.

You can also employ automated security verification tools to provide more comprehensive checks without taking time away from other development tasks. These include the following:

- SAST is a fast and effective way to detect code vulnerabilities through *open testing*, in which the tool can access the code. By embedding a SAST tool into your development pipeline, you can check for known vulnerabilities before code is committed and packaged into containers. OWASP has selection criteria and [a list of popular open source and SAST commercial tools](#).
- DAST checks for vulnerabilities while the application is running. As a form of closed testing, in which the tool cannot access source code, DAST can examine applications in the context of a container or cluster, allowing the tool to perform input range checks and simulate attacks. OWASP has [a list of DAST tools on its site](#).
- SCA examines a codebase to determine the origin, security status, and licensing compliance of units under testing. SCA tools compare source code, binaries, manifest files, container images, and more to various databases to identify potential issues. For example, an SCA tool connected to the [NVD](#) can identify code snippets that match patterns in that repository before deploying to the cloud.

How Different CSPs Structure Resources

Different CSPs implement the four Cs differently, as illustrated in [Figure 6-1](#). Here, we'll compare how AWS, GCP, and Microsoft Azure structure cloud resources.

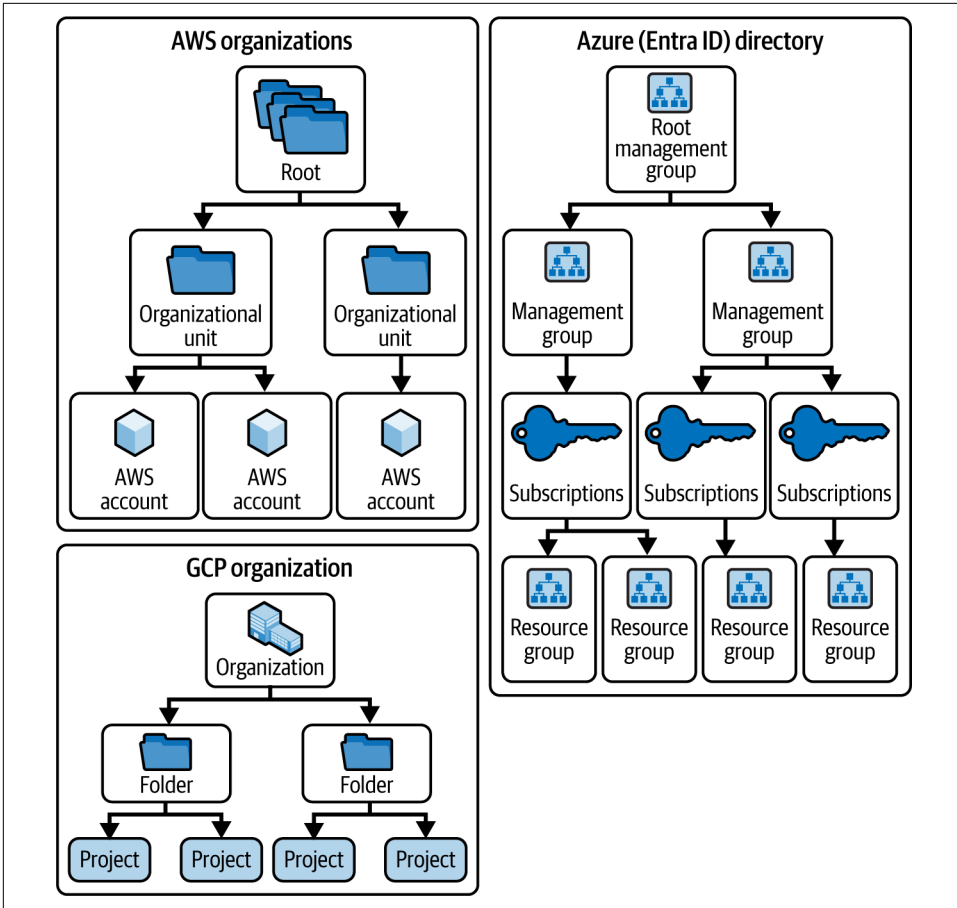


Figure 6-1. Resource structures for different CSPs

AWS

The top-level entity on an AWS instance is an **AWS organization**, which is broken down into a management account, member accounts, organizational units (OUs), and policies. The parent container of an AWS organization is called *Root*. Security policies are JSON-based and can be set at any level.

While **AWS states** that using “multiple AWS accounts is a best practice for scaling your environment, as it provides a natural billing boundary for costs, isolates resources for security, [and] gives flexibility [for] individuals and teams,” you must balance this against the complexity and effort of managing identities and resources across accounts.

One misconfigured IAM policy on one account could give bad actors access to your entire AWS instance. Also, running everything in a single account opens a potential single point of security compromise.

In general, we advise you to use a multiaccount environment to make it easier to maintain the principle of least privilege and contain the impact of a breach. As per AWS:

We recommend that you organize workloads in separate accounts and group accounts based on function, compliance requirements, or a common set of controls rather than mirroring your organization's reporting structure. In AWS, accounts are a hard boundary. For example, account-level separation is strongly recommended for isolating production workloads from development and test workloads.

Leveraging an identity provider and single sign-on offsets much of the complexity of managing multiple identities. For example, AWS IAM Identity Center supports having one federated identity across multiple AWS accounts.

Google Cloud Platform

The **root node on GCP** is an *organization* that contains folders and projects. A *project* contains all the resources necessary to create, configure, and use the Google Cloud services required by your application. A *folder* is used to aggregate multiple projects. IAM manages security policies, which can be set at any level.

The key to securing GCP is the super admin account, which has a broad set of administrative capabilities that includes IAM under cloud identity. Google also has [a list of super admin account best practices](#).

As with AWS, you can create as many projects in GCP as you have apps (within Google's quota, but you can increase the limit). The trade-off comes with managing all the identities across them.

Microsoft Azure

Beneath the top-level Microsoft Entra ID directory (formerly known as Azure Active Directory [Azure AD]), which holds users' identities and service principles, the *Root management group* contains Azure's **cloud management hierarchy**, which contains subscriptions to map users with resources and resource groups to logically contain cloud resources for your app. Security policies follow RBAC and can be set at any level.

As with AWS and GCP, we recommend that you structure your Azure resources to make following the principle of least privilege easy. You can do this by configuring subscriptions to follow your organizational structure to align permissions by team or by development phase to separate permissions between coding, testing, and production.

Guiding Principles for Securing Cloud Applications

You should consider these seven principles when building your application and configuring it for use on the cloud:

Zero standing privileges

The complex and dynamic nature of cloud development makes it challenging to manage and secure standing privileges (to access privileges that are “always on”) for every resource and every actor. Rather, the principle of **zero standing privileges (ZSP)** advocates for the removal of all persistent privileges until a user needs access. Implementing ZSP and the time, entitlements, and approvals (TEA) approach ensures that only the required permissions are granted to users who access cloud resources for a specific period of time. At all other times, access is denied. Even with ZSP, you should follow the principle of least privilege we described in [Chapter 2](#).

Centralized identity management

A centralized system for AuthN and AuthZ facilitates better control, monitoring, and maintenance of access across cloud environments.

AuthN and AuthZ

Strong AuthN mechanisms, such as MFA, and proper AuthZ controls ensure that only the right users can access cloud resources.

Credential management

Avoiding the use of hardcoded credentials, regularly rotating credentials, using federated identities to minimize the number of credentials, and using IAM reduce the risk of cyberattacks that steal and exploit credentials.

Secure DevOps practices

Integrating security measures, such as SAST and IAM, into the DevOps pipeline ensures that identity and access controls are implemented and monitored throughout the development lifecycle.

Automated compliance checks

Automated tools and processes can be used to assess and ensure compliance with security best practices, IAM policies, and industry guidelines. For example, regular reviews of inactive identities, identities with unused privilege permissions, and identities with standing privilege permissions are essential to preventing the exploitation of stale credentials.

Audit trails and monitoring

Logging and monitoring all identity-related events supports the timely detection of and response to security incidents and helps with many compliance activities.

Time, Entitlements, and Approvals

The TEA approach minimizes the need to manage standing credentials without impacting development velocity. The three components of the approach are defined as follows:

Time

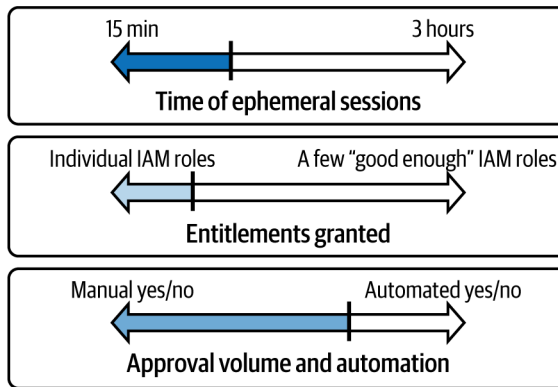
How long is access granted

Entitlements

The level of access granted

Approvals

The checks undertaken to grant access



There are no standing privileges in the TEA approach. Rather, entitlements are added for a specific window of time after the required approvals are met. This approach ensures that only the right users have a limited period of time to access the approved resources.

The time you take to understand and adopt these practices will be well worth it—your cloud applications will have an improved security posture.

Comparing IAM Services

IAM grants human and machine actors the access required to do their jobs within organizational security policies. As we explained in [Chapter 4](#), IAM solutions automate and manage the following key identity security processes:

- Identity provisioning and deprovisioning
- SSO services
- MFA
- Role management (RBAC)
- Workflow and approval processes
- Identity synchronization and reconciliation

When you're building and configuring your apps, you need to know how your chosen CSP implements IAM. This is because different CSPs do it in different ways.

Let's look at the three most popular CSPs—AWS, GCP, and Microsoft Azure—in terms of how permissions are assigned and how access is managed.

Permissions

All three CSPs use some form of a permissions document to assign permissions to human and machine actors.

AWS

AWS uses a JSON permissions document called an *IAM policy* that specifies permissions and resources. Since both sets of information are managed in one document, there is no separation between the resources an identity has access to and the actions the identity can perform given its permissions.

The following policy example allows a user to perform all Amazon DynamoDB actions (`dynamodb:*`) on the Superheroes table in the 123456789123 account within the `us-east-1` region:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "dynamodb:*",
    "Resource": "arn:aws:dynamodb:us-east-1:123456789123:table/Superheroes"
  }
}
```

GCP

GCP decouples permissions from resources by storing them in different locations. Permissions are defined in documents called *roles* and are linked to the resources they grant access to in IAM policy documents. An IAM policy only applies to the scope on which it resides, such as a project, folder, or organization. Placing this file on the appropriate resource is typically the responsibility of your IT or security department.

To grant a permission in an IAM policy file, you create a *binding* to link a role to a scope. The following example shows that `frodo@example.com` is granted the `organizationAdmin` role in the first role binding. This role contains permissions for organizations, folders, and limited project operations. The second role binding grants `frodo@example.com` and `sam@example.com` the ability to create projects via the `projectCreator` role:

```
{
  "bindings": [
    {
      "members": [
        "user:frodo@example.com"
      ],
      "role": "roles/resourceManager.organizationAdmin"
    },
    {
      "members": [
        "user:sam@example.com",
        "user:frodo@example.com"
      ],
      "role": "roles/resourceManager.projectCreator"
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

Together, these role bindings grant fine-grained access to Frodo and Sam, and Frodo is granted more access than Sam. Note that the roles themselves don't contain information about where their permissions apply. This allows them to have different scopes and permissions, which helps apply the principle of least privilege.

Microsoft Azure

Azure also decouples permissions from resources by using a JSON file that's usually accessed through the Microsoft Azure Portal. As with GCP, permission assignments are set by a *scope* that's defined by a resource or a container of resources.

The following policy example assigns the Frodo Baggins custom role access to all read actions of `Microsoft.Compute`, `Microsoft.Storage`, and `Microsoft.Network` resource providers (`subscriptionId1` and `subscriptionId2` only):

```

{
  "Name": "Frodo Baggins",
  "IsCustom": true,
  "Description": "Can monitor and restart virtual machines.",
  "Actions": [
    "Microsoft.Storage/*/read",
    "Microsoft.Network/*/read",
    "Microsoft.Compute/*/read"
  ],
  "NotActions": [
  ],
  "AssignableScopes": [
    "/subscriptions/{subscriptionId1}",
    "/subscriptions/{subscriptionId2}"
  ]
}

```

Access

Knowing how each CSP manages human and machine accesses helps you fill any security gaps.

AWS

Though it's not recommended, AWS administrators use an *IAM user* and *IAM user groups* to represent the identities and IAM policies that specify access capabilities to manage access resources. This framework can get cumbersome when you're giving IAM users access to multiple accounts, as each user maps to a specific identity and would require giving them cross-account access—potentially violating the principle of least privilege.

A best practice for AWS is to manage users through an external IdP. With an IdP, users are represented through a proxy identity called an *IAM role*, thus allowing the IdP to delegate access to the user. This includes machine identities, such as an AWS Lambda service that's allowed to access resources.

AWS provides the [AWS IAM Identity Center](#) to help you centralize identity management. Here, you can create and manage user identities directly—or you can connect an existing identity source, including Microsoft Active Directory, Microsoft Entra ID, and CyberArk Identity via SAML federation.

GCP

With GCP, users and groups are managed through [Google Cloud Identity](#). It supports entities such as domains, service accounts, authenticated users, and all users. Like AWS's IAM role, GCP has a proxy identity called a *service account*. This identity is used to attach permissions to GCP cloud services.

The “all users” and “authenticated users” groups are not tied to a specific instance of Google Cloud Identity. Rather, they are scoped to all Google accounts and users on the internet, and they are essentially public.

Microsoft Azure

In addition to RBAC, Azure has **Microsoft Entra ID** to help you manage users, groups, and permissions. Unlike AWS and GCP, Entra ID has two options for managing resource access for machine identities:

- The first option is to create a *managed identity* to grant access to Azure resources without needing to manage credentials. A managed identity can be system assigned, in which it sets a one-to-one relationship between a role and resources, or user assigned, in which the identity is created separately from resources and granted access as needed. Having a user-assigned managed identity is useful when access to multiple resources is required to perform the same job, such as an analytics function sending work to multiple cloud compute instances.
- The second option is to use a *service principal* identity for an application or service to access resources within a specific Azure tenant or other Microsoft services.

In general, you use managed identities to authenticate and access Azure resources from within Azure itself. If you interact with services outside Azure, using a service principal is the appropriate choice.

Switching Between CSPs

It’s clear that no two CSPs are alike and that migrating between them could take time and effort. However, service issues, licensing costs, features, and other considerations may drive your organization to move from one cloud environment to another.

As a developer, here are the most common security practices you should consider when switching between CSPs in single or multicloud environments. We’ve also included ways to overcome them to discuss with your IT team:

IAM roles and policies

With each CSP using different elements, hierarchies, and capabilities (e.g., AWS uses JSON, Azure uses RBAC), you should study the target CSP’s IAM model and map your current structures to it. This will help you avoid permissions issues that could lead to overly broad or unnecessarily restrictive access.

Principle of least privilege

With different granularities and structures for permissions, you should review and minimize permissions when moving to a new CSP. Also consider using the tools and features provided by the new CSP to enforce least privilege.

MFA configuration

Assuming MFA configurations are the same among CSPs can result in you having insufficiently protected accounts. You should ensure that MFA is set up according to the target CSP's recommendations and test configurations to ensure they're set up correctly without your application's features.

Management of access keys and secrets

Each CSP has its own methods of managing keys and secrets, so be sure to apply the principles outlined in [Chapter 5](#) (e.g., rotation, encryption, monitoring, auditing) to the target CSP.

Service accounts

CSPs handle service accounts and managed identities differently. To avoid granting inappropriate permissions, you need to understand the target CSP's service account management practices and apply the principle of least privilege.

Data encryption

Assuming all CSPs have the same level of data encryption, default or otherwise, can lead to sensitive data exposure. You need to understand how the target CSP can encrypt data at rest and in transit, and you need to leverage these capabilities appropriately.

Policy monitoring and enforcement

Each CSP has different tools to monitor and audit activities. Failing to set these up properly could lead to inadequate detection and response to security breaches. We recommend that you use the target CSP's tools to implement monitoring and audit logging as closely as possible to what you had before.

Identity federation

Improper or inadequate identity federation can lead to AuthN failures or unauthorized access. You should therefore adapt your current identity federation policies to the new CSP to ensure the correct AuthN and AuthZ of users.

Conditional access policies

Overlooking or misapplying conditional access features can result in inadequate access controls and the exposure of sensitive resources. Leveraging the conditional access features of the target CSP and performing regular reviews and updates keeps these risks to a minimum.

It's important to plan your switchover carefully and follow the preceding recommendations. Not doing so can lead to frustrations and extra work during your migration process.

Summary

In this chapter, you've learned what makes cloud native application development special, and you've seen how different CSPs implement and manage identities and resources. Three cloud security principles came into play repeatedly:

- Visibility into security protections and services is limited.
- No two CSPs are alike.
- You are responsible for applying security best practices to your chosen CSP.

For further exploration of cloud security best practices, we recommend the following resources and references:

- [“What is Cloud Native?” \(AWS\)](#)
- [“Security Best Practices for Cloud-Native Development” \(CyberArk\)](#)
- [“Cloud Native Security Whitepaper” \(TAG Security\)](#)
- [AWS IAM Identity Center documentation](#)
- [GCP IAM documentation](#)
- [Microsoft Entra ID documentation](#)

Securing Kubernetes

Before containerization and Kubernetes came along, you had to spend weeks provisioning servers, carefully documenting IP addresses, and praying that your application would work in production like it did on your local machine. We called it the “*works on my machine*” syndrome, and it caused real issues when code behaved in mysterious and unexpected ways due to environmental differences.

Containers are self-contained application packages that include everything you need to run your application: code, runtime environment, tool and library dependencies, and configuration settings. While the concept of containers has been around for a long time (in 1979, Unix introduced `chroot` as a form of process isolation), Docker revolutionized the usage of containers in 2013 by making them accessible to everyday developers. Thereafter, applications and their dependencies could be packaged together and deployed easily to run consistently in different environments.

As containerization took off, so did the challenges associated with it. Teams ran hundreds, even thousands of containers across multiple servers and clouds—and manually managing these containers became impossible.

Enter Kubernetes, which was originally developed by Google based on its experience with Borg and Omega. It was released as open source in 2014, and it solved several critical problems:

Scaling

Automatically scaling applications based on demand using defined parameters

Self-healing

Automatically replacing failed containers without manual intervention

Load balancing

Distributing traffic across multiple instances to optimize performance

Rolling updates

Updating applications to the latest versions without downtime

The following code sample illustrates one of the approaches that made Kubernetes revolutionary. This config file sets up three instances, called *replicas*, of *Kubernetes Pods*, which are the smallest units of deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3 # Run 3 instances of your application
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app:1.0
```

Replicas are the automatic fail-safe and load-balancing components of Pods. As the system detects failures or changes in loads, it can add or delete replicas to maintain workload integrity.

The layers of abstraction and scale that make Kubernetes so powerful, such as Pods and replicas, also introduce security challenges. If the system isn't designed with security in mind, you may see the following:

- Containers escaping their boundaries to access other containers and even attack machines from other networks (e.g., via DDoS attacks)
- Pods accessing sensitive data they shouldn't access
- Exposure of critical services to the internet
- Configuration mistakes that affect entire *clusters* (groups of machines that run containerized apps), often without your knowledge

For example, a single misconfigured role binding between a user and a resource could give a developer or development machine access to production secrets.

Whether you're just starting out with Kubernetes or running production clusters, understanding security is crucial. Once you understand the fundamental principles and patterns of security, securing your Kubernetes workloads will become a natural part of your development workflow.

How Kubernetes Works

Before we dive into the security specifics, let's run through a refresher on how Kubernetes works. (We assume that you're somewhat familiar with it.)

Kubernetes is basically a highly sophisticated orchestra conductor that coordinates numerous workloads across servers, based on how you configure it. Kubernetes operates in clusters, which are groups of machines running containerized apps.

Figure 7-1 illustrates the key components of a cluster: nodes, control plane, pods, and services.

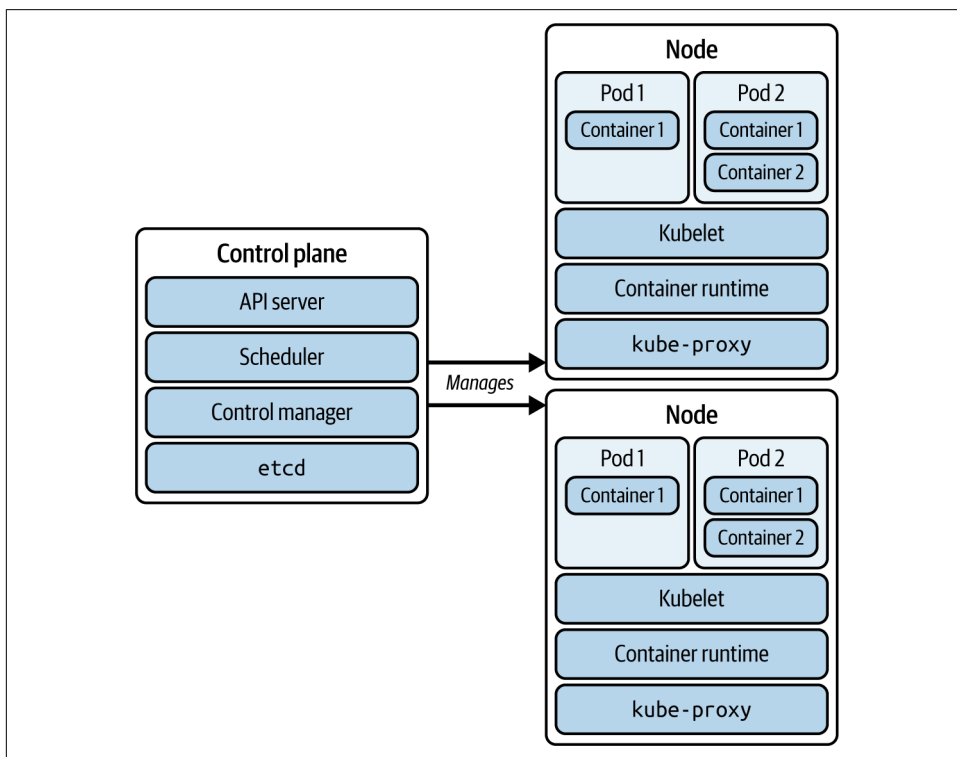


Figure 7-1. Components of a Kubernetes cluster

Nodes are where your applications actually run. Each node contains the following:

- A *kubelet* manages the node's containers and reports to the control plane.
- *Container runtime* is the software that's actually running your containers.
- The *kube-proxy* handles network routing to service container-to-container communications.

A control plane is a collection of processes that control Kubernetes nodes. A control plane includes the following:

- The *API server* (*kube-apiserver*) is the Kubernetes HTTP API through which all commands and queries flow.
- The *scheduler* (*kube-scheduler*) decides where to run new workloads.
- The *controller manager* (*kube-controller-manager*) ensures that the cluster's state matches what's specified and requested through the API server.
- The *etcd* is the cluster's key-value store for all configuration and state information.

Pods are the smallest execution units you can deploy in Kubernetes. A Pod can contain one or more containers that work together.

Services define a logical set of Pods and a policy to access them, often through a network. They provide a stable way to access your Pods as they come and go.

Kubernetes Security Challenges

The evolution of containerized applications has introduced complex security challenges that you must understand and address. Security oversights can cascade into major incidents, and the impact goes beyond just technical issues.

Imagine you're an infrastructure developer at a SaaS company, and one morning, you discover that your cloud bill has suddenly skyrocketed. After investigating the issue, you find that crypto miners have compromised your containers through an exposed API. Their hack costs your organization money.

This isn't a hypothetical scenario—Trend Micro **reported that a malicious actor targeted remote Docker API servers** for crypto mining. In the attack, the actor initiated a gRPC protocol upgrade to HTTP/2 over clear text (h2c), thereby evading many security tools that monitor traditional HTTP traffic. Bypassing these security layers enabled the actor to deploy the SRBMiner crypto miner on the Docker host and mine XRP cryptocurrency.

The financial impacts of such attacks can be severe. According to Infosys, the **estimated average cost of a container security incident is \$250,000**. It also found that 39% of security incidents resulted in lost revenue and 34% led to customer attrition.

Beyond the financial and reputational damage from these attacks, there's the cost of paying down the technical debt incurred due to poor security practices. A prevalent example of this is the management of sensitive data that needs to be made available to cluster workloads (e.g., API key and database credentials). Although Kubernetes makes it straightforward via its Secrets resource type, it does not provide any built-in mechanisms for the management and governance of this type of data. This makes the introduction of security risks easy if secrets are managed poorly.

According to Red Hat's *The State of Kubernetes Security in 2024* report, 42% of respondents cite security as a top concern with container and Kubernetes strategies. Red Hat notes "the difficulties that can surface as security incidents, vulnerabilities, and misconfigurations at different stages of the application lifecycle."

Secrets management in Kubernetes is often misunderstood, so let's look at how to approach securing secrets effectively.

Secrets Management in Kubernetes

As developers, we must adopt a security-first mindset when working with Kubernetes. This means implementing the practices we've discussed earlier in this book, such as zero trust and the principle of least privilege, and wrapping them all into some form of security automation.

The foundation lies in understanding Kubernetes secrets and how to manage them. For example, if a web application container running in a Kubernetes cluster needs to have access to an API key to communicate with an external API, then the necessary credentials need to be made available to the container. This applies to other types of secrets, too, such as the following:

- API tokens
- Database credentials
- Secure Sockets Layer/TLS certificates
- SSH keys
- OAuth tokens
- GitHub access tokens

Kubernetes comes with a basic secrets management functionality known as **Kubernetes Secrets**. In this context, a *Secret* is an object that contains a small amount of sensitive data, such as a password or an API token, that a resource needs to access another resource.

When defining a Secret, you can specify its type by using one of the built-in types or your own. The built-in types support the following:

- User-defined data
- Service account tokens
- Serialized `~/.dockercfg` files
- Serialized `~/.docker/config.json` files
- Basic Auth credentials
- SSH credentials
- TLS certificates
- Tokens used by a Node when bootstrapping

The benefit of using built-in types is that they make it easier to programmatically handle secret data and impose constraints on stored data. For example, the TLS type requires `tls.crt` and `tls.key` data items. Refer to the Kubernetes documentation for more details on these [Secret types](#).

The following code sample shows what a basic Kubernetes Secret looks like:

```

apiVersion: v1
kind: Secret
metadata:
  name: app-credentials
  type: Opaque
data:
  db_password: cGFzc3dvcnQxMjM= # base64 encoded 'password123'
  api_key: c2VjcmV0a2V5NDU2 # base64 encoded 'secretkey456'

```

This example illustrates one of the limitations of Kubernetes Secrets: data is stored as base-64 encoded strings in configuration files. While it may be obscured from casual browsing, this encoding type is not secure, and visibility is limited to the cluster where the secret is kept. You must also maintain and update these values manually every time they change.

Kubernetes does support [encrypting sensitive data at rest](#) but the secrets will need to be manually rotated and managed.

As we discussed in [Chapter 5](#), using a secrets manager like CyberArk Secrets Manager or Conjur Open Source reduces these risks. [Figure 7-2](#) illustrates how applications running inside containers in one Kubernetes namespace can securely access secrets stored in a CyberArk-managed namespace. This approach minimizes exposure risks and simplifies the scalability and maintenance of your secrets.

If your organization uses third-party secrets managers, such as the [Secrets Store CSI Driver for Kubernetes](#) or [ESO](#), CyberArk Secrets Manager and Conjur Open Source offer providers that allow you to push and retrieve their secrets. [Figure 7-3](#) illustrates how an application running inside a Kubernetes namespace would consume the secrets from secret files in a mounted Secrets Store CSI Driver volume.

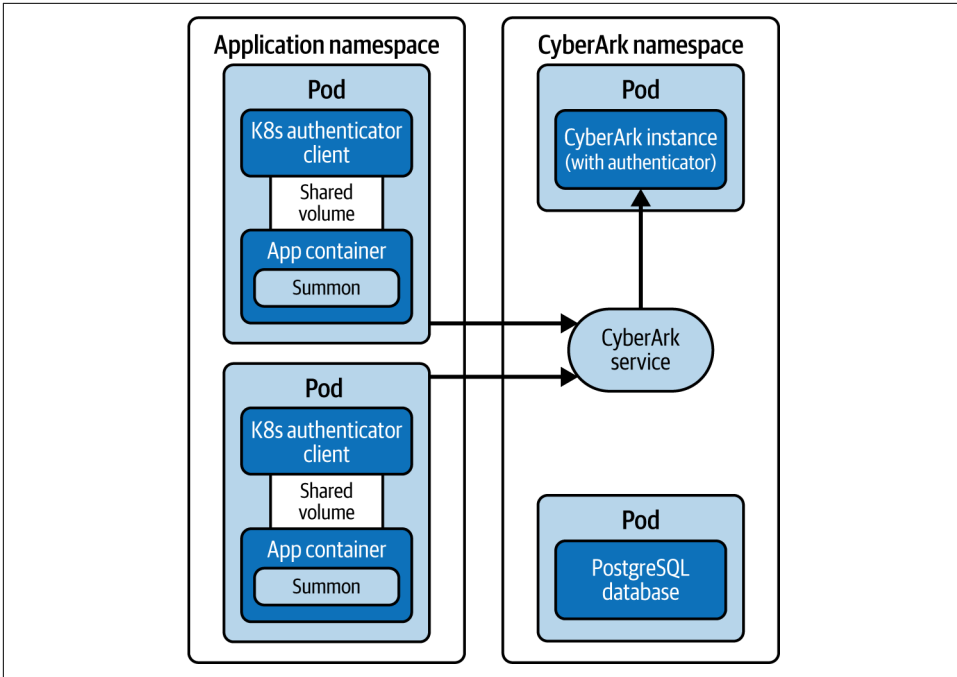


Figure 7-2. Example architecture of CyberArk Secrets Manager and Kubernetes

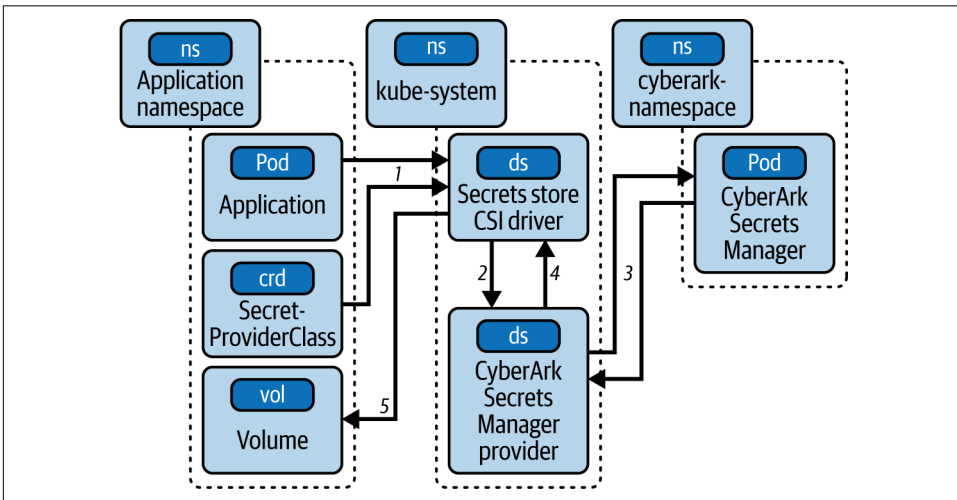


Figure 7-3. Secrets workflow among a Kubernetes application, CyberArk Secrets Manager (similar to the flow of Conjur Open Source), and Secrets Store CSI Driver



A Kubernetes CSI driver implements the **container storage interface (CSI)** that exposes persistent storage systems to a containerized workload. Using CSI, third-party storage providers (like the Secrets Store CSI Driver for Kubernetes) can write and deploy plug-ins for Kubernetes applications to reliably store, manage, and access data.

The details of the workflow shown in **Figure 7-3** are as follows:

1. The application Pod is created, requesting a mounted CSI volume from the Secrets Store CSI Driver via a reference to a `SecretProviderClass`.
2. The Secrets Store CSI Driver passes configuration from the referenced `SecretProviderClass` to the CyberArk Secrets Manager provider.
3. The CyberArk Secrets Manager provider authenticates to CyberArk Secrets Manager using a JWT authenticator.
4. The CyberArk Secrets Manager provider reads all CyberArk Secrets Manager secrets listed in the `conjur.org/secrets` Pod annotation and passes the content, along with their target filepaths, back to the Secrets Store CSI Driver.
5. The Secrets Store CSI Driver creates a `tmpfs` volume, writes the CyberArk Secrets Manager secrets to the provided filepaths in the volume, and mounts the volume to the application Pod. Note that `tmpfs` is only available in-memory. It is never written to disk.

Best Practices for Kubernetes Security

To understand where and how to improve Kubernetes deployment security, let's look at the challenges and best practices for the following key Kubernetes elements:

- Service accounts
- Kubernetes RBAC
- Pods and containers
- Networking

Use Service Accounts Diligently

Service accounts function as identity providers for Pods and workloads. Every namespace comes with a default service account, and a few security considerations:

- There's a lack of explicit identity assignment.
- Service account tokens are stored indefinitely, without proper secrets management principles applied.
- There's the potential for privilege escalation.

It's like password sharing between applications. For these reasons, and to support workloads requiring a higher level of resource access beyond the defaults, you should create a custom service account for Pods that require access to the Kubernetes API. As we explain in “[Examples of RoleBinding and ClusterRoleBinding](#)” on page 115, this means specifying a role (which specifies the granted permissions) and binding that role to the service account. This allows you to easily implement the principle of least privilege and better protect your workloads.

We recommend these other best practices for service account security:

- Disable the automatic mounting of service account tokens.
- Use unique service accounts for each application.
- Avoid binding additional RBAC roles to the default service account.
- Always use a custom service account if additional privileges are required.
- Implement clear naming conventions and documentation for service accounts.
- Rotate and audit service account credentials regularly.
- Implement time-bound access where possible.

Leverage the Kubernetes RBAC Framework

The Kubernetes RBAC API is the primary and most crucial security capability for administrative-level control of clusters. It provides a systematic and consistent way to manage access to computing and networking resources based on human and machine roles. It also facilitates the separation of duties by using role assignments to control what different teams and individuals can do, based on their responsibilities (e.g., allowing developers to only deploy into development namespaces).

The four main components of Kubernetes RBAC are as follows:

Roles and ClusterRoles

These define the actions subjects can perform within a cluster or namespace. A *Role* sets permissions within a namespace, and a *ClusterRole* is a non-namespaced resource. These permissions are additive, and there are no “deny” rules.

RoleBindings and ClusterRoleBindings

These grant subjects (service accounts, groups, and users) the permissions specified in a role. *RoleBindings* grant permissions within a specific namespace, and *ClusterRoleBindings* grant cluster-wide access.

A *ClusterRole* sets permissions for these scenarios:

- Permissions for namespaced resources (such as Pods and Secrets) in a specific namespace

- Permissions for namespaced resources across all namespaces (e.g., allowing this command: `kubectl get pods -all-namespaces`)
- Permissions on cluster-scoped resources

You can configure custom Roles or use default user-facing Roles, including but not limited to the following:

`cluster-admin`

This is a “superuser” Role that can perform any action on any resource in a cluster. If applied to a ClusterRoleBinding, it grants full control over every resource in the cluster (and in all namespaces). If applied to a RoleBinding, it grants full control over every resource in the respective namespace.

`admin`

This Role permits unlimited read/write access to resources within a namespace. It can also create Roles and RoleBindings within a particular namespace.

`edit`

This Role grants read/write access within a given Kubernetes namespace. It cannot view or modify Roles or RoleBindings.

`view`

This Role allows read-only access within a given namespace. It does not allow the viewing or modifying of Roles or RoleBindings.

Examples of Roles and ClusterRoles

Each rule applied to a Role or ClusterRole specifies a list of resources and the permitted actions (verbs) against them:

`get`

Read the specified resource manifest.

`list`

List and read the manifests of a collection of resources.

`watch`

Be informed when a resource is updated.

`create`

Create a resource.

`update`

Replace an existing resource.

`patch`

Partially update an existing resource.

`delete`

Remove a resource.

`deletecollection`

Remove a collection of resources.

The following example illustrates granting `get`, `watch`, and `list` actions on the `Pods`, `ConfigMaps`, and `Deployments` resources to the Role, which is `reader-role`. This could be a workload watching for new resources, such as `Pods` and `Deployments`, being created in the same namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: reader-role
rules:
- apiGroups: [""]
  resources: ["pods", "configmaps"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "watch", "list"]
```

You would use this configuration to grant a workload with this role read access to `Pods`, `ConfigMaps` (API objects used to store nonconfidential data), and `Deployments` in the same namespace. Note that these are read-only permissions that do not grant modification (`create`, `update`, or `patch`) or deletion (`delete` or `deletecollection`) privileges.

The following code shows the granting of `Pod` read permissions to the `ClusterRole`, which is `reader-clusterrole`. This could be a workload watching for new resources being created in any namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: reader-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Examples of RoleBinding and ClusterRoleBinding

The following code sample illustrates how a `RoleBinding` grants the `reader-role` Role to the `my-service-account` service account in the default namespace. Note that in a single `RoleBinding`, you can grant multiple subjects the same Role:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: reader-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: reader-role
  apiGroup: rbac.authorization.k8s.io

```

The following sample illustrates a ClusterRoleBinding granting reader-clusterrole permissions to the my-service-account service account in the default namespace. This permits that service account to read Pod details in any namespace:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: reader-binding
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: reader-clusterrole
  apiGroup: rbac.authorization.k8s.io

```

Securing Your Pods and Containers

A Kubernetes security context defines privilege and access control settings for Pods and containers. These settings include the following:

- Discretionary access control (DAC), in which permission to access a resource is based on a user ID (UID) and a group ID (GID)
- Running as privileged or unprivileged
- Allowing or disallowing a process to gain more privileges than its parent process (aka *privilege escalation*)
- For Linux, granting a process a limited set of privileges but not all the privileges of the root user

If you're running on Linux, you can configure these settings to use Linux kernel features to harden your containers. These features are provided by the following settings:

Secure computing mode (seccomp)

This setting filters which system calls a process can make.

AppArmor

This setting restricts the privileges of individual programs.

Security-Enhanced Linux (SELinux)

This setting provides security policy enforcement.

AppArmor and SELinux settings are only applicable if the host Node supports them. In general, Debian and Ubuntu-based distros support AppArmor, and Red Hat Enterprise Linux (RHEL)-based distros support SELinux. The major cloud providers also provide their own Linux distros as the recommended default for their managed Kubernetes services.

If you're running on Microsoft Windows Nodes or Pods, these security context mechanisms and privileged containers are not supported. However, you can use HostProcess containers instead. Pod and container security contexts provide [Windows-specific security options](#). The complete list of security context settings is found in the [Kubernetes documentation](#).



A *privileged container* is a container that can access all host system resources, including kernel features and devices. Unlike nonprivileged *containers*, which are isolated and restricted in their capabilities, privileged containers can perform tasks requiring higher-level access.

A security context is defined at a Pod or container level, as follows.

Pod security context

Security settings for a Pod are defined in the Pod specification, and they apply to all containers in the Pod unless they are overridden at the container level. In the following code, the Pod must run as a nonroot user, which Kubernetes validates at runtime:

```
apiVersion: v1
kind: Pod
metadata:
  name: partial-pod-spec
spec:
  securityContext:
    runAsNonRoot: true
  ...
```

Container security context

The security settings for a container are defined in its manifest and apply only to it. These settings override any settings made at the Pod level where there is an overlap. In the following sample, the container cannot run in privileged mode (which is equivalent to root) and cannot gain more privileges than its parent process has:

```
apiVersion: v1
kind: Pod
metadata:
  name: no-privilege-pod
spec:
  containers:
  - name: no-privileges
    securityContext:
      privileged: false
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
```

Pod Security Standards

Kubernetes *Pod Security Standards* (PSS) are policies for maintaining Pod and container security and integrity. The standards include security context settings, and they define three different profiles with varying restrictions to ensure containerized workloads are secure against known privilege escalations. As per the [Kubernetes documentation](#), these profiles are as follows:

Privileged

This is an unrestricted policy with the broadest possible permissions level. This profile allows for known privilege escalations.

Baseline

This is a minimally restrictive policy that prevents known privilege escalations.

Restricted

This is a heavily restrictive policy that follows current Pod-hardening best practices but may limit compatibility.

Adhering to PSS helps ensure that containerized applications meet industry-standard security requirements within Kubernetes environments.

Auditing and enforcing these policies is the responsibility of the Pod Security admission controller (PSA), which checks a Pod's isolation level against the PSS. The PSA uses labels to specify the enforcement mode for each policy level, which can be one of the following:

Enforce

Policy violation causes the Pod to be rejected and not deployed.

Audit

Policy violation allows the Pod to be deployed and triggers an audit annotation in the audit log event record.

Warn

Policy violation allows the Pod to be deployed and triggers a user-facing warning.



An *admission controller* intercepts and processes requests to the Kubernetes API before object deployment. It operates like middleware that validates, mutates, and rejects requests to the API. Admission controllers can be *validating* (to accept or deny requests based on custom rules), *mutating* (to modify attributes before deployment), or both.

You configure the PSA controller using two labels:

Level

Privileged, baseline, or restricted

Mode

Enforce, audit, or warn

You can configure the mode for any or all policy levels within a namespace and set different modes for different levels. You can also set the enforcement mode for a specific Kubernetes minor version. The following code demonstrates the various PSA labels in action:

```
apiVersion: v1
kind: Namespace
metadata:
  name: myns
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: v1.28

    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: v1.29
```

Here's what the labels mean:

- Any Pod that tries to deploy but doesn't specify the security policies defined in the Kubernetes v1.28 baseline policy will be rejected and not deployed.

- Any Pod that tries to deploy but doesn't specify the security policies defined in the Kubernetes v1.29 restricted policy will be deployed, and a warning message will be displayed to the user.

While using PSS is a good way to start securing your Pods, it's often not good enough for real-world enterprise environments. The PSA policies are too broad and don't offer sufficient customization, so we recommend looking at two popular alternatives: [Kyverno](#) and [OPA Gatekeeper](#), which offer greater flexibility and extendability than PSA. Both products are CNCF projects and run as separate, dynamic admission controllers to control what goes on in your cluster.

Sandboxed Pods

Sandboxed Pods provide an isolation boundary between the container application and other resources on the same host, such as a shared kernel or networking components. They complement other security measures, like PSS, in protecting workloads from security risks.

As [their documentation states](#), the Kubernetes API doesn't offer a way of sandboxing Pods:

There is not currently an API standard that controls whether a Pod is considered sandboxed or not. Sandbox Pods may be identified by the use of a sandboxed runtime (such as gVisor or Kata Containers), but there is no standard definition of what a sandboxed runtime is.

As this statement alludes to, there are Kubernetes distributions that provide their own sandboxing mechanisms. They are as follows:

- Google Kubernetes Engine (GKE) has [GKE Sandbox](#), which uses gVisor.
- Azure Kubernetes Service (AKS) uses [Kata](#).
- Amazon Elastic Kubernetes Service (EKS) supports gVisor, but [you have to configure it yourself](#).

Use a minimal base image

Which base image to select is perhaps your most crucial security decision. Selecting a minimal base image that contains only what your application needs reduces its potential attack surface and makes security auditing more manageable.

There are various strategies for building small images, each of which has pros and cons.

Slim images. Popular Linux distributions, like Debian, Ubuntu, and RHEL, provide slim versions that are built specifically for containerized environments and have documentation, locale files, and unimportant packages removed. These tend to be

30% to 40% smaller than a full image—around 75 MB, compared to approximately 130 MB for a full image.

These images are suitable for cases in which you need a full Linux distribution that's similar to and compatible with existing development and hosting environments, such as building and testing new application software releases.

Alpine Linux is a very small distribution comprising a stripped-down Linux kernel, BusyBox (for essential utilities), a service manager, and a lightweight C standard library. Alpine images come in at around 5 MB in size.

Slim images containing Debian or Ubuntu are great for builder- and development-type images, as they contain much of the necessary tooling and packages, and it's easy to install other dependencies. As Alpine ships with very little, you may have to install a lot more to get a working builder image, thus increasing its size relative to that of the full distro slim images.

Furthermore, Alpine uses `musl` as its standard C library, rather than `glibc`, which is used by other slim distros. Although both conform to the C standard and provide the required standard functions, they differ in ways that could result in code that compiles and builds under one distribution and not another.

Alpine is often a more attractive base for runtime images, in which you just need to contain the compiled binaries or packages. These images don't need to include any build-related tooling, and from a security perspective, they shouldn't.

Distroless images. The ultimate distroless image is a [Docker scratch image](#), which is basically empty. If you have a static compiled binary that requires no external dependencies, using a distroless image could be a viable option.

In practice, your application usually requires additional dependencies, such as the following:

- Root certificates so TLS connections work
- Runtimes for languages such as Python and Node
- Common directories used by libraries such as `/tmp`

Distroless images were conceived to meet these requirements. These images contain the bare minimum needed to host and run an application written in a particular language.

Google produced the first distroless images for Java, Python3, Go, Node, and Rust, and it continues to support them. Further details are found in Google's [GitHub repo](#). [Chainguard](#) also produces distroless images, and it offers a larger selection than Google, though most require a paid subscription.

A common complaint with distroless images is that they don't contain any troubleshooting tools, like a shell, thus making debugging harder. A solution is to use [Kubernetes ephemeral containers](#) for troubleshooting.

Another challenge with distroless images comes when trying to extend them with additional dependencies. As these images don't contain a package manager or many of the standard libraries, anything extra will need to be self-contained, statically compiled, and copied over.

All that said, distroless images are an effective choice for most application runtime use cases, and the Docker multistage build process makes using them for this purpose easy.

Multistage Docker builds. A multistage image build uses multiple FROM statements in your Dockerfile. Each FROM statement can use a different image that represents a different stage of the build. Selected artifacts can be copied from one stage to another so that the final stage can use a distroless base image and copy the runtime artifacts built or compiled by previous stages.

The following code sample shows a multistage Dockerfile for a Go application:

```
# Building the application
FROM golang:1.23 as build

WORKDIR /go/src/my-app
COPY . .

RUN go mod download
RUN CGO_ENABLED=0 go build -o /go/bin/my-app

# Copy it into a base image.
FROM gcr.io/distroless/static-debian12
COPY --from=build /go/bin/my-app /
CMD ["/my-app"]
```

Other considerations

Here are other best practices for securing your container images:

Use a specific and known version tag.

With container image security, the latest is not always the greatest. Always using the latest tag introduces unpredictability and potential security risks. Instead, you should always specify the exact versions of your base images and verify their integrity through digital signatures and SHA digests.

Automate security checks.

As we'll discuss in [Chapter 9](#), your CI/CD pipeline should automatically enforce container security practices. For example, it should reject any container that uses

the latest tag instead of a specific version, runs as the root user, contains known high-severity vulnerabilities, and uses unauthorized base images.

Run vulnerability scans.

Container vulnerability scanning is particularly challenging, due to the layered nature of containers. Each layer can introduce its own vulnerabilities, and these can compound in unexpected ways. Automated scans reduce these risks and should check the base image, dependencies, and configuration.

Two examples of open source automated tools for improving container security are CyberArk's **Kubesploit** and **KubiScan**. *Kubesploit* executes complex attack vector coverage on a cluster to help you understand its resilience. *KubiScan* helps diagnose permissions breaches that might jeopardize clusters by scanning them for risky permissions in Kubernetes' RBAC AuthZ model.

Ensure Network Security

Kubernetes networking can be particularly challenging because traffic is unrestricted by default and networks can be configured in various ways with no guarantees of security success. You might use a service mesh, which decouples communications between applications and infrastructure. (We discuss the security implications of this in “[Using Service Meshes to Secure Clusters](#)” on page 132.) Alternatively, you may directly connect components together. Some of your cluster communications may be limited to internal networks, while others may go outside your perimeter. These permutations make manual implementation of security practices almost impossible.



Kubernetes doesn't have a default network plug-in built in; rather, it supports the Container Network Interface (CNI) and leaves it up to each distribution to determine what its support will be. The key consideration here is that not all network plug-ins support network policies.

Two of the most popular network plug-ins are **Calico** and **Cilium**, which both support network policies.

Kubernetes provides network policy management to help with network security. A *NetworkPolicy* resource specifies the rules for traffic flow within clusters, between Pods, and to the outside world. They operate at the IP address and port levels (OSI layers 3 and 4), and they allow you to define the allowed ingress (incoming) and egress (outgoing) connections for a group of Pods.

There are three ways to identify connection endpoints:

Specific Pods

Pods matching a label are allowed.

Specific namespaces

All Pods in the namespace are allowed.

IP address blocks

Endpoints with an IP address in the block are allowed.

Examples of NetworkPolicies

The following code samples illustrate ingress and egress network policies that ensure:

Ingress

Only the echo2 Pod in the echo2 namespace can connect to the echo1 Pod in the echo1 namespace and use echo1's service hostname.

Egress

Only the echo1 Pod in the echo1 namespace can connect to the echo2 Pod in the echo2 namespace and use echo2's service hostname.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: echo1-ingress-policy
  namespace: echo1
spec:
  podSelector:
    matchLabels:
      app: echo1
  policyTypes:
  - Ingress
  ingress:
    # allow ingress only from the echo2 Pod in echo2 namespace
    - from:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: echo2
      - podSelector:
          matchLabels:
            app: echo2
    ports:
      - protocol: TCP
        port: 5678
## YAML
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: echo1-egress-policy
  namespace: echo1
spec:
  podSelector:
    matchLabels:
      app: echo1
```

```

policyTypes:
- Egress
egress:
  # allow egress only to the echo2 Pod in echo2 namespace
  - to:
    - namespaceSelector:
      matchLabels:
        kubernetes.io/metadata.name: echo2
    podSelector:
      matchLabels:
        app: echo2
    ports:
      - port: 5678

```

For NetworkPolicies, the Pod and namespace selectors only match on labels and not on Pod or namespace names. Fortunately, Kubernetes automatically applies an immutable label (`kubernetes.io/metadata.name`) containing the namespace's name when it is created. Also, the specified port(s) should be the Pod container port(s) and not those defined in the Pod's service.

Another type of egress policy is required to allow your Pods to perform DNS lookups (such as resolving the service hostnames in the preceding examples). The following code enables the echo1 Pod to perform DNS lookups when connecting to service `echo2.echo2.svc.cluster.local`:

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-allpods-to-dns
  namespace: echo1
spec:
  policyTypes:
  - Egress
  podSelector: {}
  egress:
  - to:
    # - namespaceSelector:
    #   matchLabels:
    #     kubernetes.io/metadata.name: kube-system
    # - podSelector:
    #   matchLabels:
    #     k8s-app: kube-dns
  ports:
  - port: 53
    protocol: UDP
  - port: 53
    protocol: TCP

```

To make this policy more secure, you can uncomment the `namespaceSelector` and `podSelector` fields, assuming your cluster Pods use the Kubernetes DNS service (`kube-dns`) for DNS resolution.

Network security best practices

To improve your network security, we recommend reviewing and applying the following practices:

Start by defining your ingress policies.

It's usually simpler to focus on one policy type at a time, and it's typically easier to determine incoming network connections than outgoing ones (Domain Name System lookups being a case in point).

Use descriptive naming.

Policy names that describe what the policy does make them easier to search for and understand later.

Avoid default ports.

As Kubernetes uses default ports for most of its services, using custom ports makes it harder for attackers to locate and access resources.

Implement separation of concerns.

Combining policies that serve different purposes makes isolating access harder and troubleshooting more difficult. On the other hand, separating policies into units with specific purposes and minimal overlap makes management easier and reduces the impact when changes are made.

Restrict who can change labels.

As policies depend on selectors to specify which Pods and namespaces they apply to, limiting who can change these labels minimizes the risk of accidental or malicious changes that may render the policies useless.

Test policies to ensure they're working.

It's tough to know whether network policies are working without trying them out first. For example, you can test allowed and disallowed connections by creating a new Pod with labels that match your NetworkPolicy selectors and using cURL to determine responses to test data.

Review policies regularly.

As Kubernetes architectures and workloads evolve, performing regular policy reviews helps to ensure that network security is appropriate for the current environment.

Consider adopting a default deny-all policy.

To ensure that all Pods in a namespace are secure by default, adopting a “deny-all” policy sets a minimum baseline to avoid exposing resources that don't have a policy explicitly defined.

The following code sample illustrates a policy that denies all traffic to all Pods in the test namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  Namespace: test
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

Identity Management on Kubernetes Using SPIFFE

The **Secure Production Identity Framework for Everyone (SPIFFE)** is an important standard for establishing a secure, consistent, and scalable identity for workloads. This open standard offers a streamlined approach for identifying and authenticating applications, workloads, and service identities. This allows SPIFFE to be used for workload-to-workload authentication (AuthN) and authorization (AuthZ).

The framework relies on each service or workload being associated with a unique SPIFFE identity (SPIFFE ID) that's verified by using cryptographic materials. Implementing SPIFFE offers these benefits for how machine identities are managed and verified:

Remove reliance on secrets managers

By using SPIFFE, you eliminate the need to retrieve secrets like API keys, usernames, and passwords from the secrets manager. Rather, you use your SPIFFE ID to authenticate and authorize external services like workloads and databases.

Cross-cloud AuthN

You can use a SPIFFE ID to access cloud services from another cloud provider. An example is using your SPIFFE ID to access an AWS S3 bucket directly from Azure without requiring AWS access tokens.

CI/CD security

You can provide a unique SPIFFE ID for each step in your CI/CD pipeline that can be used to sign artifacts such as binaries, software bills of materials (SBOMs), and other generated provenance. Signing artifacts allows for cryptographic verification down the line and provides a reliable auditing mechanism.

SPIFFE is a way to give each workload a unique identity that's encoded by cryptographic key materials such as X.509 certificates or JWT tokens. A SPIFFE ID defines the unique name of the workload.

Here's an example of what a SPIFFE ID looks like:

```
spiffe://dev.example.com/pricingervice/api
```

Here, `spiffe://` defines that what follows is a SPIFFE ID, and `dev.example.com` defines the trust domain under which the identity is issued. Most organizations have a few trust domains for different functions, such as `dev`, `prod`, and `staging`. By default, workloads in different trust domains can't communicate with each other, but you can change that by configuring federation between trust domains. The remainder of the definition, `pricingservice/api`, is how the user uniquely identifies the application. It can be prescriptive, a randomly generated hash, or a random term.

A workload retrieves its identity through a **workload API** that is exposed through a Unix socket. Supporting the X.509 and JWT cryptographic standards supports a broad range of use cases, including service meshes, direct application integrations, databases, cloud providers, and many more. Integrating SPIFFE with your favorite programming language is easy, thanks to its wide support of SDKs.

The following code sample illustrates a configuration of a Golang web server with SPIFFE support:

```
ctx, cancel := context.WithCancel(ctx)
defer cancel()

// Set up a '/' resource handler
http.HandleFunc("/", b.rootHandler)

// Create a `workloadapi.X509Source`, it will connect to Workload API using
provided socket.
// If socket path is not defined using `workloadapi.SourceOption`, value
from environment variable `SPIFFE_ENDPOINT_SOCKET` is used.
source, err := workloadapi.NewX509Source(ctx)
if err != nil {
    return fmt.Errorf("unable to create X509Source: %w", err)
}
defer source.Close()

// Allowed SPIFFE ID
clientID := spiffeid.RequireFromString(b.spiffeAuthz)

// Create a `tls.Config` to allow mTLS connections, and verify that presen
ted certificate has SPIFFE ID `spiffe://example.org/client`
tlsConfig := tlsconfig.MTLSServerConfig(source, source, tlsconfig.Authori
zeID(clientID))
server := &http.Server{
    Addr:           b.serverAddress,
    TLSConfig:      tlsConfig,
    ReadHeaderTimeout: time.Second * 10,
}

// Serve the SPIFFE mTLS server.
if err := server.ListenAndServeTLS("", ""); err != nil {
    return fmt.Errorf("failed to serve: %w", err)
}
```

```
}  
return nil
```

The logical place to start adopting SPIFFE is to begin experimenting with the product-ready implementation of SPIFFE, which is called **SPIRE**. SPIRE implements the full SPIFFE specification and is used by organizations such as Bloomberg and Uber.

If you are interested in learning more about SPIFFE, read *Solving the Bottom Turtle*, which is published by the SPIFFE community.

Using cert-manager for TLS Identity Security Automation

A reliable pattern in identity security is focusing on automation to obtain and rotate secrets. This is commonly seen with TLS certificates, especially with the use of the Automatic Certificate Management Environment (ACME) protocol with free services like **Let's Encrypt**.

In Kubernetes, the de facto standard for solving certificates for TLS is *cert-manager*. As a CNCF-graduated project, cert-manager sits on the same rung of the CNCF ladder as Kubernetes itself. It adds Kubernetes CustomResourceDefinitions (CRDs) to clusters to allow users to specify which certificates are needed and how they should be issued. It also watches these resources in the background to ensure they match their specification and renew them if needed.

Issuers of cert-manager are pluggable and independently configurable, allowing easy use of various sources of privately managed certificates, like ACME, which are **listed on the cert-manager website**.

The following code sample illustrates how cluster-internal private certificate issuers are natively supported by cert-manager:

```
apiVersion: cert-manager.io/v1  
kind: ClusterIssuer  
metadata:  
  name: selfsigned-issuer  
spec:  
  selfSigned: {}  
---  
apiVersion: cert-manager.io/v1  
kind: Certificate  
metadata:  
  name: my-private-ca  
  namespace: cert-manager  
spec:  
  isCA: true  
  commonName: my-private-ca  
  secretName: private-ca-secret  
  privateKey:
```

```

    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: ClusterIssuer
    group: cert-manager.io
  ---
  apiVersion: cert-manager.io/v1
  kind: ClusterIssuer
  metadata:
    name: my-issuer
  spec:
    ca:
      secretName: private-ca-secret

```

These resources create a self-signed *root certificate*, which can be used to sign other certificates, such as a certificate for a domain:

```

  apiVersion: cert-manager.io/v1
  kind: Certificate
  metadata:
    name: my-cert
    namespace: default
  spec:
    secretName: cert-secret
    issuerRef:
      name: my-issuer
      kind: ClusterIssuer
    dnsNames:
      - example.com
      - www.example.com

```

The certificate’s progress can be tracked with `kubectl`, just like any other Kubernetes resource.

When the “Ready” condition on the certificate’s status field is “True,” the cert can be used and will be available in the specified Kubernetes secret.

Also, cert-manager supports issuing certificates by using methods that don’t require the Certificate resource:

- In response to *certificate requests* (CSRs)
- As service mesh certificate requests, such as Istio
- As virtual volumes attached privately to Pods (CSI driver volumes)

CSI driver volumes are especially useful since they enable the issuance of separate certificates per Pod, without the need to store the certificate in a Kubernetes Secret. While this does lead to increased reliance on the certificate issuer and a larger overall number of certificates issued, it reduces the attack surface by limiting where certificates can be read from.

The cert-manager *csi-driver* project is installed separately, and once it's configured, you can use it by changing your Pod resources to add a new volume, as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-csi-app
  namespace: default
  labels:
    app: my-csi-app
spec:
  containers:
  - name: my-container
    image: busybox
    volumeMounts:
    - mountPath: "/tls"
  name: tls
  command: [ "sleep", "1000000" ]
  volumes:
  - name: tls
    csi:
      driver: csi.cert-manager.io
      readOnly: true
      volumeAttributes:
        csi.cert-manager.io/issuer-name: my-issuer
        csi.cert-manager.io/issuer-kind: ClusterIssuer
        csi.cert-manager.io/dns-names: ${POD_NAMESPACE}.example.com
```

In the preceding example, certificates are mounted to the */tls* directory and are only available inside the Pod once issued. CSI driver certificates are then deleted when the Pod is deleted, further reducing the risk of private keys being stolen.

Beyond *csi-driver* and *cert-manager* itself, the wider *cert-manager* project includes other projects that assist with various tasks relating to TLS in Kubernetes. These include the following:

trust-manager

The easiest way to manage trust bundles in Kubernetes and OpenShift clusters. It orchestrates bundles of trusted X.509 certificates which are primarily used for validating certificates during a TLS handshake but can be used in other situations too.

approver-policy

This is a policy engine for *cert-manager* certificates, and it provides fine-grained control of issuance.

csi-driver-spiffe

This is a *csi-driver* variant that focuses on issuing X.509 SPIFFE Verifiable Identity Documents (SVIDs), and it provides trivial workload identity provisioning in Kubernetes.

istio-csr

This is an Istio compatibility layer that enables cert-manager to issue certificates for the Istio service mesh.

The [cert-manager website](#) has extensive documentation for cert-manager and all of its subprojects, including API reference documentation that details exactly which fields are configurable in the CRDs provided, along with installation and usage guides.

Using Service Meshes to Secure Clusters

While Kubernetes excels at container orchestration, it lacks sophisticated tools to secure service-to-service communication, implement fine-grained access controls, and maintain comprehensive audit trails of service interactions. This is where service meshes come in. They provide a dedicated layer that handles service-to-service communication, but implementing them securely requires an understanding of their capabilities and limitations.

Service meshes implement security through a dual-plane architecture (see [Figure 7-4](#)). The control plane orchestrates operations and makes decisions about security policies, identity management, and traffic routing. The data plane, which is typically implemented through proxy sidecars, acts as the enforcement arm of this security framework. Each proxy operates as a dedicated security guard for its associated service because it inspects and controls all traffic between Pods.

This architecture ensures consistent security policy enforcement throughout your application network, regardless of the underlying application code.

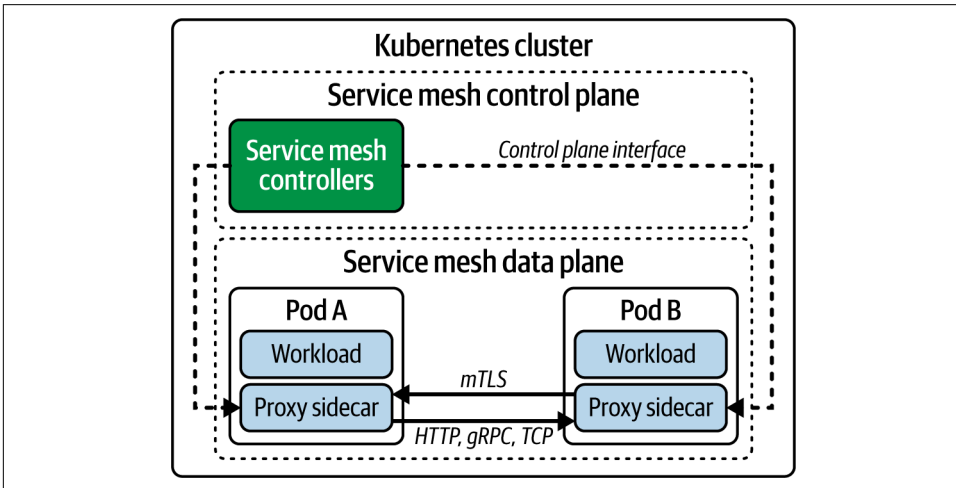


Figure 7-4. Example service mesh architecture

Several service mesh solutions are available, including [Istio](#), [Linkerd](#), [Kuma](#), and [Cilium](#). Which one you should choose depends on your requirements for ease of implementation, performance, security features, and community support.

The next sections detail the security challenges and best practices to consider when working with service meshes.

Identity Management in Service Meshes

Identity management underpins all other security features in a service mesh. For example, it's common for a Pod to send an X.509 certificate request to the service mesh control plane when it starts. The control plane then validates this request and signs it to generate the certificate, which it sends back to the Pod. The Pod's proxy sidecar then uses this certificate for mutual TLS (mTLS) AuthN and access control via AuthZ policies. The Pod then communicates with the control plane periodically to renew its certificate. There are also other configurations where another Pod performs the certificate request on behalf of the workload.

Using this certificate-based identity system solves several security challenges. First, it ensures that services can trust communications from other services, thus preventing impersonation attacks. Second, it enables fine-grained access control based on service identity rather than network location. Finally, it provides a foundation for comprehensive audit logging, allowing security teams to track exactly which services are communicating with one another.

AuthZ in Service Meshes

Service mesh AuthZ works on multiple factors: the identity of the calling service, the specific operation being requested, the path being accessed, and even custom metadata about the services involved.

As each Pod has its own confirmed identity, the service mesh can control access using AuthZ policies. Each sidecar proxy runs an AuthZ engine that evaluates Pod access requests in real time. If the evaluation results in an ALLOW response, the proxy will forward the request to the Pod's workload for processing; otherwise, a DENY response will be returned to the request sender.

These AuthZ policies are specified using custom resources, and the control plane is responsible for updating the proxies with the latest policies.

The following code samples illustrate [Istio Authorization Policies](#). Other service meshes will look different, but the principals will remain the same.

Example 7-1 illustrates how to prevent network access to workloads in the echo namespace.

Example 7-1. Definition of a policy denying network access to workloads in a namespace

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-nothing
  namespace: echo
spec:
  {}
```

Example 7-2 allows the echo2 Pod running in the echo2 namespace to access the echo1 Pod using HTTP GET (assuming the echo2 Pod is using the echo2 service account).

Example 7-2. Definition of a policy defining access rights between Pods

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: "echo1-access"
  namespace: echo
spec:
  selector:
    matchLabels:
      app: echo1
  action: ALLOW
  rules:
  - from:
    - source:
      principals: ["cluster.local/ns/echo2/sa/echo2"]
    to:
    - operation:
      methods: ["GET"]
```

If the from field were removed, any workload could access echo1 using an HTTP GET request.

Example 7-3 allows any workload in the echo2 namespace to execute an HTTP POST request to the echo1 Pod, but only on path /add.

Example 7-3. Definition of a policy defining HTTP POST permissions within a namespace

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: "echo1-create"
  namespace: echo
spec:
```

```
selector:
  matchLabels:
    app: echo1
action: ALLOW
rules:
- from:
  - source:
    namespaces: ["echo2"]
  to:
  - operation:
    methods: ["POST"]
    path: ["/add"]
```

Trading Off Performance and Security

One of the more significant challenges in implementing service mesh security is managing the performance impact. The sidecar proxy model introduces additional resource overhead and potential latency, and network requests involve multiple proxy hops. As the number of workloads increases in a cluster, so does the impact of this challenge.

This overhead shouldn't be viewed as purely negative, as the security benefits often outweigh the performance costs. The maintainers behind Istio and Cilium have implemented features that reduce or even remove the need to rely on sidecar proxies in the data plane. Istio's solution, **Ambient mode**, replaces them with a node-based proxy for handling Layer 4 network traffic and namespace-based proxies for handling Layer 7 traffic. Cilium has also created an Extended Berkeley Packet Filter (**eBPF**)–based **service mesh** without sidecars that offers the same features as a traditional model but is significantly more efficient.

Service Mesh Best Practices

We recommend the following best practices for securing and using your service meshes.

Prioritize security

A security-first approach builds resilience against service mesh attacks from the ground up. You should consider these recommendations when building and maintaining your service meshes:

Lock down network traffic.

Start implementation by using AuthZ policies that follow the default-deny pattern. Then, consider which services need to communicate with one another to add more policies that provide these permissions.

Use strict mTLS.

Implementing strict mTLS provides AuthN and encryption between communicating services and helps mitigate man-in-the-middle attacks.

Deploy an egress gateway.

Rather than allow each mesh service to access external services directly, you should configure them to access these services indirectly via a dedicated egress gateway. This allows greater control of and governance over access to services outside the mesh.

Manage mesh traffic

Service meshes offer various traffic management capabilities to help you optimize application performance and minimize the impact of failures. Examples of these capabilities that you should use include the following:

Take advantage of network resilience mechanisms.

A service mesh provides features—such as circuit breaking, retries, and timeouts—to ensure application resiliency and reliability by preventing connection requests to stressed and failing workloads. Using these mechanisms, requests are sent to other workloads in the same pool, giving the failing ones a chance to recover.

Use traffic routing rules for traffic shifting during deployment of new releases.

These rules allow for weight-based routing and routing based on HTTP header values, facilitating canary and blue-green-style deployments.



A *canary deployment* progressively rolls out an application release by splitting traffic between a previously deployed version and the new one. This could be rolling out the new release to a few servers, Nodes, or users and expanding deployment only after tests have passed.

On the other hand, *blue-green deployment* splits application rollout between two runtime environments: one for the current application (e.g., blue) and one for the new version (e.g., green). Once the team ensures the new application meets all requirements in the green environment, traffic switches to it, and the blue environment goes idle.

Implement effective observability

The nature of service meshes, in which all traffic flows through proxies, means that you have additional observability options without needing to integrate them into application codebases. These options include the following:

- Service behavior metrics, such as traffic volume, error rates, and response times
- Integration with tracing backends, in which the mesh proxies send trace information directly to backend services like [Jaeger](#) and [Zipkin](#)
- Access logs for service traffic to provide extensive details of service-to-service communication

Summary

Security in Kubernetes is a journey, not a destination. You start with the fundamentals in this chapter, practice in a safe environment, and gradually build up to more sophisticated security configurations as your architecture changes. Every security control you implement on every abstraction layer is another barrier between your applications and potential threats.

To further explore Kubernetes security best practices, we recommend that you consult the following resources and references:

- [“Kubernetes Security: Best Practices for Kubernetes Secrets Management”](#)
- [Kubernetes Security Checklist](#)
- [Pod Security Standards](#)
- [Amazon EKS Best Practices for Security](#)
- [Google GKE guide to harden your cluster’s security](#)
- [Microsoft Best practices for cluster security and upgrades in Azure Kubernetes Service \(AKS\)](#)

Security Automation

While reading this book, you may have wondered how any one developer or team could implement all the practices and tools we've discussed without doing it full-time. Each effort requires planning, implementation, and testing and can overwhelm already busy schedules. Even security teams get bogged down by the growing number of cyber threats, more complex tools, and shrinking budgets—and that's where security automation comes in to make everyone's lives easier.

Security automation is the use of automated tools, scripts, and processes to perform security tasks that would otherwise be manual, time-consuming, and prone to human error. For developers, it's a powerful way to integrate security into the development process without significantly slowing down their workflows—and for security teams, it helps simplify and enforce protections throughout an organization. Security automation includes orchestrating security technologies and workflows to detect, investigate, and respond to security events without human intervention. This includes threat detection, incident response, compliance monitoring, vulnerability management, access control, and security policy enforcement across an organization's infrastructure.

According to IBM's *Cost of a Data Breach Report 2024*, applying AI and automation to security prevention reduces an organization's cost of a data breach by an average of \$2.22 million relative to organizations that don't deploy those technologies. Automation streamlines regular security operations across development infrastructure, IT systems, clouds, and applications. It can also act as a template for security setup and configuration at the beginning of projects and as a dynamic set of tests for continuous and proactive threat detection. The goal of any automation strategy is to remove human error and improve the efficiency, consistency, and scalability of security processes so teams can focus on what they do best.

For example, rather than relying on your time and effort to identify security vulnerabilities in your code, you can use static application security testing (SAST), dynamic application security testing (DAST), and interactive application security testing (IAST) tools to automate the process. We discuss these in more detail in [Chapter 9](#). Also, instead of manually coding permissions for all the human and machine identities that access your application, you can use policy as code (PaC) mechanisms to define and manage security rules, criteria, and conditions through a high-level programming language. We'll discuss these tools later in this chapter.

Why Does Security Automation Matter?

Security automation provides significant value to a business, in the following ways:

- It reduces the risk of data breaches and cyberattacks by reducing human error, automatically enforcing security practices (processes, tools, and reporting) across the organization, and identifying and potentially remediating vulnerabilities before they're exploited.
- It improves the efficiency and effectiveness of security operations by reducing manual and repetitive tasks and enhancing visibility and control over security status.
- It increases agility by enabling the development and deployment of secure and reliable applications and by facilitating the adoption of new technologies and platforms.
- It helps your business keep up with changes in industry standards and customer requirements over time. For example, governance over personal information, health data, financial data, and other protected data types requires constant enforcement and monitoring of compliance status.

Efficiency is the difference between creating secure coding tests yourself and running an automated tool that does it for you, which frees you up to focus on writing code rather than repetitive tasks. It's also the difference between a breach taking months to remediate rather than days (depending on the nature and scope of the breach). Even then, you cannot fully remediate a breach because once your data is exposed or stolen, you can't get it back.

A recent [Mend.io report on open source security](#) found that “only 13 percent of vulnerabilities were getting fixed, and on average, it takes 271 days to fix a vulnerability.” While breaches cannot be undone, automation can significantly reduce the time needed to identify compromised systems, stop unauthorized access, and implement necessary security fixes.

The [2023 CircleCI breach](#) offers a compelling recent example of this problem. Malicious actors compromised an engineer's laptop with malware, stole their 2FA-backed

SSO session cookie, and impersonated the engineer in a remote location. As CircleCI's chief technology officer (CTO) explained, "Because the targeted employee had privileges to generate production access tokens as part of the employee's regular duties, the unauthorized third party was able to access and exfiltrate data from a subset of databases and stores, including customer environment variables, tokens, and keys."

CircleCI's remediation strategy included steps that are ideally automated:

- Rotate potentially exposed production hosts to ensure clean machines.
- Revoke project API and personal API tokens.
- Rotate GitHub OAuth tokens.
- Rotate all Bitbucket tokens on behalf of customers (in partnership with Atlassian).

It's not clear whether or not CircleCI had automated these actions before the breach, but either way, implementing them as proactive and continuous measures helps reduce the chances of a malicious actor gaining entry and limits the exposure radius if they do.

Types of Security Automation

From developer tools to endpoint protection, security automation comes in three flavors:

Full code

This consists of tools and applications that require developed code to automate security operations. This could come as a PowerShell script that runs security tests against every new code check-in or a configuration file for a policy as a code platform such as CyberArk Secrets Manager or Conjur Open Source. We provide an example of this later in the chapter.

No code

No-code security automation needs little to no programming to implement. These tools usually have visual interfaces and prebuilt templates to run security tasks with minimal human effort. Essentially, they replace manual scripting with drag-and-drop functionality. The trade-off is that no-code tools may not provide the flexibility and functionality required by an organization.

An example of no-code security automation is an incident response workflow defined through a visual interface and executed automatically when a specified trigger is detected. Another example is **CyberArk® Identity Flows™**, which orchestrates identity management workflows without writing code.

Low code

These tools provide a visual interface for automating security tasks, and they have an API for users and machines to configure and customize behavior. Unlike no-code tools, low-code solutions allow you to use as much code as possible to customize automation and integrate with other systems.

AI and machine learning are bringing new methods of security automation to the identity management industry. These solutions can ingest and analyze massive amounts of data to predict potential threats and identify them in real time. They can also train on organization-specific data to suggest optimal security measures to implement and make recommendations as attacks occur. AI and machine learning also enhance frameworks like NIST and MITRE ATT&CK by ingesting large amounts of data, eliminating reliance on static rules, and significantly increasing the speed of predictions—thus enabling faster reactions.

Remember, AI can help humans make smarter, faster decisions by processing large amounts of data quickly and presenting a selection of options with the most relevant data. However, a human still needs to check the AI's work and make the final decision.

We'll discuss more about AI's role in security automation later in this chapter.

What Security Processes Can Be Automated?

Security automation manages mundane work and the most critical aspects of your organization. It can offload the repetitive tasks of a single developer and orchestrate the activities of multiple teams with different security needs.

Here are six security processes that benefit from automation:

Secure development workflows

You'll often find yourself stuck in the crunch between feature development and release schedules, meaning you'll need tools that make security testing and compliance easier. As you code, test, integrate, and submit your code, security automation tools can run through checks against secure coding standards, vulnerability databases, and dynamic security scenarios. For example, a process that automatically blocks code check-ins when it detects a pattern that matches the NVD database can help you fix the vulnerability faster than if you had to wait for a manual test.

Identity security

IAM and similar tools can automate the creation of user accounts, assigning access permissions, and revoking access when needed. Through automation, you don't need to worry about securing and rotating credentials for your app, as your IAM solution does it for you. Similarly, automated tools can elevate your

permissions to access a test server when needed and lower them when work is completed.

Threat investigation

Software can perform the forensic work necessary to identify the causes of breaches and exposures to reduce the time you need to develop a **remediation plan**. For example, if your company is going through a ransomware attack, an automated security tool can identify spikes in file accesses and network traffic that indicate where and how the attack is spreading.

Incident response

Once you've identified a response plan, automation can run through its actions faster than you or any other human. This is crucial for environments where multiple applications, tools, and devices are spread throughout the organization—remediating them all can be a complex and time-consuming process. For example, an intrusion detection and prevention system (IDPS) can identify a potential threat through unusual patterns of network activity, and it can automatically block the offending IP addresses while an administrator looks into it.

Endpoint protection

Endpoint-based software can automatically detect, investigate, and remediate suspicious activities on endpoint devices. This includes monitoring network traffic, database requests, filesystem accesses, and other data for anything that's considered an endpoint: laptops, smartphones, servers, virtual machines, and IoT devices (cars, cameras, smart home products, etc.). Because cybercriminals find endpoints to be easy targets—they're usually dependent on users implementing security measures—they're ideal use cases for security automation.

Reporting and compliance

Security software can manage routine logging and reporting activity in support of threat detection, investigation, and remediation, and it can also provide evidence in support of compliance activities. Through PaC systems, teams can automate rules, conditions, and instructions that govern compliance activities. For example, they can automate a build script that prevents unauthorized users from accessing a resource or a custom GPT that incorporates policy into ChatGPT's learning model.

Security Automation at Work

A good security automation solution answers the needs of the entire organization in a holistic manner. It standardizes security workflows and integrates with the systems that are necessary to ensure that data and resources are protected. For example, security automation could respond to malware infection by quarantining suspect files across teams or running RBAC checks against file share access requests.

It all boils down to two goals: making security processes more efficient and making them less prone to human error. Implementing automation isn't easy, as you have to integrate with multiple systems and processes to get things right, but there are many tools to help.

Security Automation Tools

Unless you're responsible for developing security automation tools, most of your interactions with them will occur around the edges, making it crucial for you to understand their impact on decision-making, threat response, and your organization's overall security posture.

For architects and engineers, understanding the role and implications of automation is equally valuable, as it directly influences how they design resilient systems that can seamlessly integrate and leverage these tools for enhanced security. It's important for them to understand how these tools work as one day, they may be asked to push user activity data to a centralized system or provide an API to control specific functions as quickly as possible and with little to no resources.

The following sections introduce the most common security automation tools and identify those areas where you could become involved.



As a developer, your interactions with the following security automation tools may be limited—so we've highlighted your potential responsibilities where appropriate. More important is building proper logging and monitoring mechanisms into your apps. As **called out by the OWASP Top Ten**, security logging and monitoring is one of the most common security failures. Implementing these features—such as logging auditable events like logins, access requests, and high-value transactions—helps automation tools perform data enrichment and analysis of suspicious activities.

Digital forensics

Security and computer security incident response teams (CSIRTs) automate digital forensics tasks, such as collecting and analyzing breach data, to accelerate the reconstruction of cybersecurity incidents. They often take data from other security platforms to find out how a breach impacted filesystems, networks, and applications. They then use this evidence to pinpoint the root cause of an attack and identify the entities that are responsible.

If your software is involved in a breach, a security team may ask for logs to support forensic activities. More proactively, it may require your app to supply data to a centralized security platform to support forensic analysis later.

EDR and XDR

Endpoint detection and response (EDR) manages endpoint security, while *extended detection and response* (XDR) takes a more holistic view of security across various organizational systems. Most EDR solutions analyze endpoint activities to detect and report on suspicious patterns. They track detailed information—such as process activities, file changes, network traffic, and system events—to help security teams monitor and respond to threats. XDR platforms use a broader range of inputs—such as email servers, network and cloud security tools, and EDR solutions—to provide a unified view across the organization.

Unless you're an EDR/XDR developer, you won't interact with these platforms because they're meant to connect with the security APIs and tools outside your scope. However, you may be asked to supply information or support control from a platform, so you may need to learn its API and data requirements.

Firewalls

Firewall automation takes care of the configuration and management of firewall policies, rules, and runtime actions. IT and security teams can automate rules updates based on emerging threats and how logs are analyzed and reported for more effective threat management. More commonly, firewalls are part of a larger automation strategy using security orchestration, automation, and response (SOAR) and SIEM platforms. These platforms take firewall logs as inputs to threat identification algorithms, and they send commands back to adjust rules if suspicious activities are detected. As a developer, you may be required to set up and manage firewalls as part of your cloud development environments.

Intrusion detection

An *intrusion detection system* (IDS) monitors network devices and traffic for activities that match a known malicious pattern, violate corporate security policies, and may appear suspicious. IT and security teams use IDSs to identify and respond to potential threats faster, and you may need to supply them with runtime application data. IDSs also support compliance efforts; for example, the **PCI DSS standard** explicitly calls for their use:

Use network intrusion detection and/or intrusion prevention techniques to detect and/or prevent intrusions into the network. Monitor all traffic at the perimeter of the cardholder data environment as well as at critical points inside of the cardholder data environment and alert personnel to suspected compromises. Intrusion Detection and Prevention (IDP) Systems, baselines, and signatures must be kept up to date.

Privileged access management

Privileged access management (PAM) platforms control how privileged human and machine identities interact with systems and monitor their activities. PAM platforms are key to automating the principle of least privilege, in which users only receive the minimum levels of access required to do their jobs. They also support the automatic rotation of secrets, analyze potential threats, and generate reports for compliance purposes.

Developers often use PAM platforms to manage how users work with their apps. **Figure 8-1** illustrates an example in which **CyberArk® Privileged Access Manager™** integrates with a Unix environment to manage permissions. Here, it automates user authentication (AuthN) by following these steps:

1. User `dave@unix01.comp.com` attempts to log in to the target system using their active directory (AD) credentials.
2. CyberArk checks the AD credentials and user group through a preconfigured connection with the organization's AD.
3. If the user has sufficient permissions to access the target system by using their credentials, an SSH session is established.
4. PAM-aware applications can authenticate with the same AD credentials, providing seamless interaction and workflows.

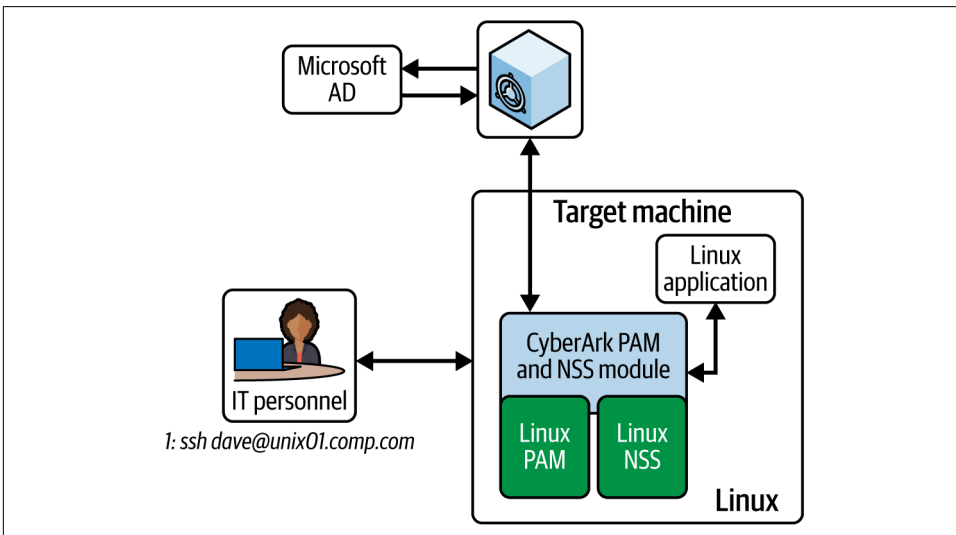


Figure 8-1. How a PAM solution integrates with a Unix environment

SIEM

SIEM tools analyze logs from various sources—such as firewalls, servers, and your apps—to aggregate data and identify suspicious activity. Typical SIEM use cases are log management to collect and centralize data, event correlation to detect patterns, and incident monitoring to track and audit activities related to security incidents. Some solutions also offer incident response to coordinate actions across different systems.

If you're asked to supply data to a SIEM tool, it usually means your app needs to create a paper trail of security-related actions that assist IT and security teams. For example, PCI DSS requirement 10.1 states that organizations must “Implement audit trails to link all access to system components to each individual user,” and 10.7 says to “Retain audit trail history for at least one year; at least three months of history must be immediately available for analysis.” These requirements would be impossible to meet if every app and system needed to manually implement them.

At a minimum, logging for audit purposes includes tracking user logins and access to important resources. This may also mean logging password resets, username changes, data downloads, search strings, and other artifacts that help build a picture of what's going on with your app.

SOAR

SOAR platforms let you integrate separate security tools in one place to coordinate information, analysis, and actions. They have a central console to manage operations and interact with different systems. *Orchestration* refers to how these platforms connect and coordinate software and hardware tools toward a common goal, *automation* refers to how they automate low-level and time-consuming tasks across all the individual systems, and *response* is how they serve as a central authority to manage incident response actions across connected components. Together, these features support faster security workflows and more consistent actions across the organization.

SOAR platforms are well outside your area of responsibility, but your app may be required to send data or accept control as part of a larger security orchestration strategy.

Static code analysis

Static analysis of application code is nearly ubiquitous in enterprise environments, as it provides fast and comprehensive testing against security threats. With automated vulnerability scanning, you can continuously assess code for potential weaknesses by using pattern matching with vulnerability databases, known language exploits, and secure coding standards.

Static code analysis tools work on any level of your application, from scanning lines of code as you write to build-level tests. OWASP provides a [basic overview of how static code analysis works](#).

Threat intelligence

Threat intelligence tools collect information about potential and existing threats to a system. This information is usually divided into *strategic intelligence*, which covers high-level trends; *tactical intelligence*, which focuses on specific threat behaviors; and *operational intelligence*, which focuses on known cyberattacks. Security teams use these tools to better understand their threat environment and make more informed decisions about implementing security processes.

Security Automation Best Practices

Security automation helps you stay one step ahead of cyberattackers, who have nothing but time and patience. To help you implement it right, we've compiled this short list of best practices for developers and security teams.

Know where to start

You need to identify the security processes that would most benefit from automation. These are repetitive, time-consuming, and error-prone tasks, such as vulnerability scanning, identity access management, threat detection and analysis, and compliance reporting. Prioritizing these processes establishes the implementation roadmap.

Enforce zero trust

Zero trust ensures that no human or machine identity is granted access to resources by default, regardless of their position or location. Automatically enforcing zero trust makes it easier to verify users at every access point across all your applications, systems, and networks.

Choose a solution

Selecting and deploying an implementation, whether in house or outsourced, requires knowing your use cases, system requirements, staff capabilities, and budget. The necessary steps are beyond the scope of this book, but as starting points, we recommend that you read [the CyberArk guides to selecting an IAM solution and managing endpoint privileges](#).

Use scripts and playbooks

You need to ensure consistency and scalability by using clearly defined and documented automation rules. Scripts and playbooks collect all processes in one place and should include contingencies associated with every step. For example, Red Hat

Ansible uses playbooks to orchestrate the steps of various security processes, and this is illustrated by the following code sample from the [Ansible documentation](#) that targets web and database servers:

```
- name: Update web servers
  hosts: webservers
  remote_user: root

tasks:
- name: Ensure apache is at the latest version
  ansible.builtin.yum:
    name: httpd
    state: latest

- name: Write the apache config file
  ansible.builtin.template:
    src: /srv/httpd.j2
    dest: /etc/httpd.conf

- name: Update db servers
  hosts: databases
  remote_user: root

tasks:
- name: Ensure postgresql is at the latest version
  ansible.builtin.yum:
    name: postgresql
    state: latest

- name: Ensure that postgresql is started
  ansible.builtin.service:
    name: postgresql
    state: started
```

Monitor continuously

Automated security processes must be monitored to ensure that they function as intended and to identify optimization opportunities. This is critical for keeping processes and tools current with the latest security vulnerabilities and threats, as they constantly change.

Infrastructure as Code

Infrastructure as code (IaC) can be used to automate the secure configuration and deployment of cloud resources. By using editable text files [that describe how cloud infrastructure is provisioned and managed](#), IaC tools automatically create, configure, and update resources to known states. Rather than relying on a confusing mix of command-line operations, scripts, and GUIs, IaC tools simplify and standardize provisioning in one place.

The three major cloud providers offer these IaC tools: [AWS CloudFormation](#), [Azure Resource Manager](#), and [Google Cloud Deployment Manager](#).¹ All providers come with some level of security template that you can customize and extend for your use cases. For example, Azure Resource Manager has the [Azure security baseline](#) that applies the [Microsoft cloud security benchmark](#) to configure cloud resources.

This benchmark includes the following recommendations,² many of which follow the principles described in earlier chapters of this book:

- DS-2: Ensure software supply chain security.
- DS-3: Secure DevOps infrastructure.
- DS-4: Integrate static application security testing into the DevOps pipeline.
- IM-1: Use a centralized identity and AuthN system.
- IM-3: Manage application identities securely and automatically.
- IM-5: Use single sign-on (SSO) for application access.
- IM-6: Use strong AuthN controls.
- IM-7: Restrict resource access based on conditions.
- LT-1: Enable threat detection capabilities.
- LT-2: Enable threat detection for identity and access management.
- PA-1: Separate and limit highly privileged/administrative users.
- PA-2: Avoid standing access for user accounts and permissions.

You can find detailed descriptions of these recommendations under the [Controls section of the benchmark](#).

From a developer's perspective, here are some security best practices when working with an IaC tool:

Manage secrets.

Always store and maintain secrets used by IaC scripts in a centralized secrets management solution. This solution should include AuthN, authorization (AuthZ), and lifecycle management.

Enforce the principle of least privilege.

IaC scripts should ensure that the permissions granted to the resources they create are limited to what is required for them to perform their work.

¹ Note that the third of these hasn't been updated since 2020.

² Here, DS stands for DevOps security, IM stands for identity management, LT stands for logging and threat detection, and PA stands for privileged access.

Perform security tests on IaC scripts.

As IaC scripts are just like application code, they should conform to secure testing principles. Therefore, you should use tools like static code analysis to identify potential vulnerabilities in IaC code, including known threats and potential misconfigurations.

Log activities.

IaC scripts should log all activities as they provision infrastructure. This data provides valuable information to security tools like IDS and SIEM, and it helps you analyze the root causes of security incidents.

Use version control.

Track all changes to IaC code to maintain history and support reverting back to an older version if something isn't working. It's important to align check-ins with the infrastructure features they support so that a consistent baseline is maintained across the system. You wouldn't want an SSO configuration against a version of the identity provider that doesn't support it.

Policy as Code

As its name implies, policy as code (PaC) uses high-level programming languages to abstract away how policies are defined and managed for compliance and governance purposes. It's more effective than manual methods, as teams can rely on one consistent set of instructions that's easy to deploy, update, test, and track under version control.

A *policy* could be a rule that defines which conditions must be met for user AuthZ or for an app to access a server. It could also be a remediation tool, such as defining a set of steps executed automatically in response to a security incident. Policies are coded using a language such as Python or YAML, making them easy to share and update. There are also open source languages specifically designed for PaC use, such as **Rego** and **Cedar**. Rego is the language of the Open Policy Agent (OPA) and is hosted by the Cloud Native Computing Foundation. Cedar is a policy language for AuthZ policies.

PaC is useful when you need to do the following:

- Abstract away policy enforcement such that policies are easier to create, maintain, and track across various resources.
- Promote the reuse of policies across your environment, where teams can use them without knowledge of the specific platforms where those policies will be deployed.
- Make deployment faster and easier through code that can run through existing test workflows and DevOps delivery pipelines.

While PaC may seem similar to IaC, there are notable differences. IaC supports the provisioning of infrastructure only, while PaC manages operations, security, and compliance across any function within an organization.

Here's an example of a PaC definition using [CyberArk Secrets Manager](#), a cloud-agnostic solution for automating secrets management:

```
- !policy
  id: policy-1
  body:
    - !user
      id: barbara.gordon
      owner: !group /ops-admin
    - !host
      id: web-oracle
```

This syntax is human readable and easy to maintain. In this definition, the `policy-1` object owns `web-oracle`, and the `ops-admin` group owns the `barbara.gordon` user. If your security department decides to delete `policy-1`, `web-oracle` is deleted, but `barbara.gordon` remains because it is owned by `ops-admin`. This example illustrates how policy object ownership can have more sophisticated relationships than are possible with simple scripts or manual procedures.

The best practices for PaC implementations and IaC are identical. In addition, we recommend these practices that we've adapted from [the CyberArk Secrets Manager documentation](#):

Separate development from production.

Policies usually differ between development and production, with developers requiring more open access to resources than in a live environment. To avoid granting access beyond what's necessary, you should ensure that your policy files match the intended use case and are deployed to the appropriate resources.

Organize policies in separate files.

Keeping policies physically separate promotes the segregation of duties and allows for easier composition of existing declarations without propagating changes across multiple files later.

Use a branching strategy.

For PaC systems that use a hierarchical structure, creating a branching strategy ensures that rules are maintainable and enforces the principles of least privilege. For example, CyberArk Secrets Manager separates policies into “root,” “system,” and “application” to define their structure. *Root* is the top-level organizational element, *system* manages services used by the policy manager, and *application* contains the managed resources. Enterprise-wide policies would be appropriate under root, while team-specific ones fall under application.

Secure Automation Tools

Developing automation tools securely is fundamental to protecting the organization, and an in-house PAM solution is of no use if hackers can gain access to it. If you're writing the automation code, you should keep secure coding practices in mind, such as securely managing tokens, using secure storage mechanisms for sensitive information, and validating all inputs and outputs to prevent attacks.

Security Automation with AI

No chapter on automation is complete without a discussion of the influence of AI and machine learning. Every cybersecurity function, from identity management to threat intelligence, is an ideal use case for these technologies. They benefit from AI's ability to understand, learn, and adapt to large amounts of data, with teams gaining the ability to predict where and how threats will occur.

Use cases for AI and machine learning in security automation include the following:

Automating routine tasks

AI-based automation can manage repetitive security tasks based on an organization's past data and behavior, often without explicit instructions. This could include scanning for ransomware, managing software patches, detecting vulnerabilities, and managing identity access. As AI operates more on training data than manual programming, it can recommend automating tasks where you least expect them. For example, you could run an AI-based vulnerability scanner through your codebase and see what it reports—the results could be eye-opening. It's also important to remember to check your AI tool's licensing model, as your organization may not allow you to use proprietary data to train third-party AI models.

Anomaly detection

Machine learning can take historical and real-time data from APIs and device logs to identify anomalies that may indicate a potential attack or one that has already happened. If machine learning is tied to SIEM and SOAR platforms, it can also trigger immediate responses to block suspicious activity before it becomes a significant issue.

Predictive analytics

AI can analyze patterns in various data types—such as network traffic, identity access policies, and API calls—to forecast security incidents before they occur. This helps teams to be more proactive in their cybersecurity strategies. AI can also suggest optimal security actions to implement and tailor recommendations based on an organization's past data and behavior.

Simplifying user interactions

An ML technique known as *natural language processing* (NLP) enables security platforms to process and respond to human language. It's used to understand the syntax and context of text- and voice-based inputs to perform operations on data. This could be used to analyze text-based messages for signs of a social engineering attack or respond to a user asking questions about security-related behavior across sensitive resources.

Many security vendors now use AI and machine learning technologies in their products. An example is **CyberArk® CORA™ AI**, which extracts insights from identity data points across an organization and allows users to ask questions and get recommendations on how to manage access policies. For example, it can detect and notify users of noncompliant accesses to secrets stored in AWS Secrets Manager, and it can understand natural language instructions to query data. It also surfaces security recommendations automatically to help users make better decisions based on assessed risk, community data, industry verticals, market segmentation, and security best practices.

While AI can enhance security automation, it may also introduce several risks. Bias and false positives can lead to incorrect threat classifications, causing unnecessary disruptions or missed attacks. AI's dependence on input data can also lead to adversarial attacks, in which attackers manipulate training data to deceive AI models into making incorrect security decisions. The potential lack of transparency in AI-driven security decisions can also make it difficult for security teams to understand why an action was taken, leading to trust and compliance concerns.

Example: Security Automation with Ansible and Conjur Open Source

Let's apply these broader principles to a specific example using *Red Hat Ansible*, which is a popular automation platform used by development and IT teams. It covers a broad range of scenarios, including software builds, configuration management, application deployment, and continuous integration and delivery. Instructions for running configuration, deployment, and orchestration functions are kept in *playbooks*. They can be relatively simple, such as an ordered list of build tasks, or complex enough to orchestrate actions among multiple users, hosts, and monitoring tools.

Consider a situation where you're automating security procedures using a combination of Ansible and the **Conjur Open Source secrets management provider**.³ In this

³ Conjur Open Source is the open source version of CyberArk Secrets Manager and this example will work with both.

example, we'll store a secret in Conjur Open Source by using PaC, and we'll use Ansible to configure a set of host machines.

Assuming all the components are installed, the first step is to use the Conjur Open Source CLI to create a new policy and store our secret value. The CLI comes as a Docker container:

```
$ docker-compose conjur authn login admin
```

Conjur Open Source will ask for an admin password, which is the API key that it generated when it was first set up.

Next, you will create a PaC description file in YAML called *Database_Password.yml*. This policy creates a user who can set the password, a host that can read the password, and the password itself:

```
- !policy
  id: Database
  body:
    # Define users, hosts, and secret variable
    - !user DatabaseAdmin
    - !host Ansible
    - !variable DatabasePassword
    - !permit
      # Give read and write permissions to the administrator.
      role: !user DatabaseAdmin
      privileges: [read, update, execute]
      resource: !variable DatabasePassword
    - !permit
      # Give read permissions to the Ansible host account to fetch the password.
      role: !host Ansible
      privileges: [read, execute]
      resource: !variable DatabasePassword
```

This policy file is loaded into Conjur Open Source as follows:

```
$ docker cp Database_Password.yml conjur_client:Database_Password.yml
$ docker-compose conjur policy load root Database_Password.yml
```

The password for our database can then be stored in a secure store, such as AWS Secrets Manager, Azure Key Vault, or CyberArk Privileged Access Manager. A benefit of using Conjur Open Source is that if the password needs to be changed, it only requires an update in the platform rather than everywhere it is used:

```
$ docker-compose conjur variable values add Database/DatabasePassword
S3cr3tP4ssw0rd
```



We're using a simple password here as an example; in reality, you should follow your corporate password policy.

Before using Ansible, we need two more policies for the automation to work. The first, *Layer.yml*, adds a layer representing an application and its running environment:

```
- !layer
  id: Applications
```

We'll also define a Host Factory, *HostFactory.yml*, for the layer. This allows us to create unforgeable tokens to assign to new hosts within a layer, which is useful as it saves us from having to define an identity for multiple machines in the policy:

```
- !host-factory
  id: demoFactory
  annotations:
    description: Factory to create identities for new application servers
  layers: [ !layer Applications ]
```

After loading these policy files into Conjur Open Source, we'll create a token that lives for 365 days. This token allows machines within the Applications layer defined in *Layer.yml* to identify themselves and gain access to the resources that are permissible through its role, such as the database password defined earlier.

We then set the token as an environment variable on the Ansible Node, using the Conjur Open Source CLI:

```
$ docker-compose conjur hostfactory tokens
create --duration-days 365 demoFactory
```

To use Ansible to automatically configure the host machines, we need to create an environment variable on the Ansible node and set it to equal the token that we generated from the host factory. The following token string is an example:

```
$ export FACTORY_TOKEN=bqwktntxa3sjn1jjwr192p1r0c31s8bs07dzd70g24vp21sn02a4m
```

The next step is to use an Ansible playbook to create an identity on the host machine. To do this, we'll need to download the Conjur Open Source Ansible Role, allow Ansible to create an identity on a host machine, and install **Summon**, which is used to retrieve information from Conjur Open Source. We download it using **Ansible Galaxy**, which is a free site for community-developed Ansible roles:

```
$ ansible-galaxy install cyberark.conjur-host-identity
```

Before building the playbook, we assume two prerequisites:

- The Ansible hosts file (*/etc/ansible/hosts*) defines the collection of host machines.
- The public certificate from the Conjur Open Source server was downloaded upon installation. We'll assume this file is called *conjur_demo.pem*.

Using the Ansible Role we downloaded, we can build a basic Ansible playbook to create identities and install Summon on each host, which is configured to access our

Conjur Open Source instance. We'll name this file `Server_Conjur_Install.yml`, using `clock-tower` as the instance's hostname:

```
- hosts: localhost
  become: yes
  roles:
    - role: cyberark.conjur-host-identity
      conjur_appliance_url: 'clock-tower'
      conjur_account: 'demo'
      conjur_host_factory_token: "{{lookup('env', 'FACTORY_TOKEN')}}"
      conjur_host_name: "{{inventory_hostname}}"
      conjur_validate_certs: false
      conjur_ssl_certificate: "~/conjur_demo.pem"
```

This playbook is run from the command line like this:

```
$ ansible-playbook Server_Conjur_Install.yml
```

Ansible will loop through all the hosts in the Ansible hosts file, create a Conjur Open Source identity for each one, and install and configure Summon.

To demonstrate how you would retrieve a secret whenever one of the hosts is started, we can use a standard Linux `systemd` file. In this example, we use Summon to retrieve the database password stored previously and store it in `/usr/local/bin/application`:

```
[Unit]
Description=Application
After=network-online.target

[Service]
User=Database/Ansible
ExecStart=/usr/local/bin/summon --yaml 'DATABASE_PASSWORD: !var root/Data-
base/DatabasePassword' /usr/local/bin/application
```

The [Conjur Open Source](#) and [CyberArk Secrets Manager documentation for Ansible](#) have more examples and tips for automating PaC.

Summary

Automation is critical for improving the security posture of your apps, development environments, and software supply chains—no team should do without it. It makes routine tasks easier, enforces consistency, reduces human error, and takes much of the security burden off your plate.

To further explore security automation best practices, we recommend that you consult the following resources and references:

- [“Better Security with Threat Response Automation”](#)
- [“Infrastructure as Code Security Cheatsheet”](#)
- [“Automated Policy as Code with Red Hat Ansible Automation Platform”](#)

- “Analyzing 3 Offensive AI Attack Scenarios”

CI/CD Pipeline Security and Software Supply Chains

Any organization that develops software, for either itself or others to use, has processes for creating and delivering code from concept to deployment. Known as *software supply chains*, these processes comprise four stages:

Development

Writing code and tests, reviewing code, testing features, and committing code into source code repositories

Build

Building and compiling the code into packages that can be imported (as dependencies) or deployed (as applications)

Test

Verifying that the code meets requirements and validating different aspects of the code, such as quality, performance, compliance, and security

Deployment

Releasing packages into production environments or launching SaaS tenants internally to the business or externally to the customer

Development organizations use continuous integration (CI) and continuous deployment (CD) tools to automate and accelerate these stages so as to deploy releases faster and offload effort. CI/CD is often illustrated as a DevOps pipeline consisting of discrete steps that build toward a release (see [Figure 9-1](#)). This pipeline usually begins when code is pushed to a repository and ends with the software solution built, tested, and deployed to a production environment.

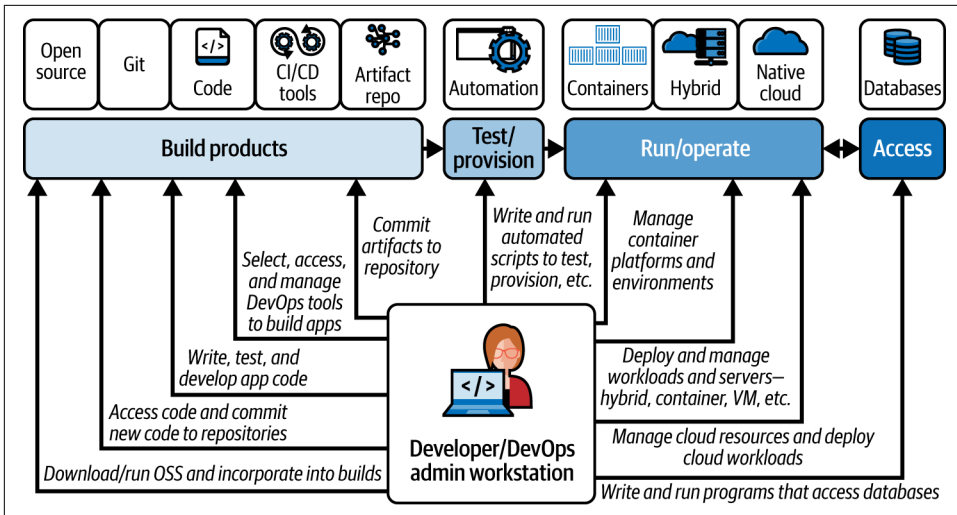


Figure 9-1. Example CI/CD pipeline from development to deployment

CI focuses on build and testing automation, while CD focuses on promoting and testing built code to a staging or production environment. Taking it a step further into operations and IaC practices, many teams use CI/CD pipelines to manage configuration files for building out the infrastructure that their applications run on. This means that CI/CD tools can also handle change management, testing, and deployment for artifacts like Docker files and Kubernetes manifests.

The exact steps in a CI/CD pipeline vary from team to team, and larger organizations may have multiple pipelines, but the security question is always present. Like production environments, software supply chains driven by CI/CD are vulnerable to attacks—arguably more so given that in many organizations, they are not viewed as being as important as production, and security is not as tight. However, as has been shown over the past few years, attacking software supply chains can be just as disruptive and costly as attacking production environments. The [Codecov](#) and [SolarWinds](#) breaches we discussed in earlier chapters are just two examples of the potential impact of CI/CD compromise.

While most discussion of CI/CD focuses on application code—avoiding bugs and crashes—the shift to managing IaC configuration with CI/CD tools highlights another potential issue: managing the secrets needed to run your applications and systems. These secrets can end up exposed and may put your systems at risk.

The people, processes, and technology involved in CI/CD are all potential avenues of attack. Application code, automation servers, deployment tools, IaC configuration files, and the Nodes responsible for running CI/CD pipelines are just a few areas that

malicious entities can exploit. Since privileged identities often execute CI/CD actions, successful attacks often have high-risk outcomes.

In this chapter, we discuss the core principles of secure software delivery in a CI/CD pipeline, the possible attack vectors in software supply chains, and how such attack vectors can be mitigated.

Why Does Securing Your CI/CD Pipeline Matter?

Attackers have many methods at their disposal to exploit CI/CD environments, but certain risks are more common than others. The following sections explain the basic challenges involved with securing these pipelines.

Securing Secrets in a CI/CD Pipeline

Automated processes are critical components of DevOps infrastructure. CI/CD tools such as Jenkins, GitHub Actions, CircleCI, and GitLab CI/CD are commonly used to facilitate faster software deployment and product delivery.

These tools are the biggest consumers of secrets, and they have access to many sensitive resources, such as other apps and services and information stored in configuration files and databases. As the number of secrets grows, it becomes harder to store, transmit, synchronize, and audit secrets securely. Additionally, secrets aren't just for authentication (AuthN) between tools—they're often used in the build and deployment process for deployed resources to access what they need. This is particularly important in hybrid cloud and microservices deployments, with automated scaling capabilities of tools like Kubernetes.

When secrets are compromised, it means that someone could make unwanted changes or leak information. Recent examples of high-impact software supply chain attacks include Log4j, Uber, and CircleCI. In the CircleCI breach, the code integration tool (the pipeline) was compromised, leading to the exposure of all the secrets and credentials that had access to other pieces of the software supply chain.

A Jenkins Example

Jenkins' configurability and use of plug-ins make it challenging to securely determine who can use Jenkins at any given time, what it is allowed to do, and where it can deploy artifacts. Jenkins communicates with many applications and systems across DevOps environments, meaning secrets can reside in configuration details within Jenkinsfiles and Jenkins' central credentials store for each pipeline. Jenkins jobs and plug-ins can access these credentials at runtime, and the server obscures their usage in the output of each job.

This makes it challenging to secure access to critical secrets such as passwords, source control, and artifact deployment. For example, the following security risks may arise if developers leave Jenkins consoles in an insecure state within development environments:

- The Jenkins scripting console can run arbitrary Groovy code and can be found under Manage Jenkins > Script Console or via `/script` from the root of the Jenkins install path.
- If developers store passwords within Jenkins, they may not be visible from within the web console, but they can be extracted from the system.
- Any Jenkins users with Job/Configure permissions can run any executable on the Jenkins agents and can ask for any set of credentials defined in their scope (global or otherwise) to be injected into agents using the credentials binding plug-in.
- The credentials binding plug-in enables application secrets to be passed from job configurator users to the build agent. For example, with the credentials ID, users can add a `withCredentials` statement to their Groovy pipeline script.

An example pipeline using `withCredentials` might be as follows:

```
node {
  withCredentials( [usernamePassword( credentialsId: 'googlecompute', username
  Variable: 'USERNAME',passwordVariable: 'PASSWORD')]) {
    // build project
    sh 'mvn clean install'
  }
}
```

Here, the pipeline job instructs Jenkins to grab the Google Cloud credentials by using the `googlecompute` credentials ID, injecting the Google Cloud `USERNAME` and `PASSWORD`, and running a Maven build.

This exposes all credentials in the global scope to any pipeline and any job configurator user. An attacker can shift from low privilege to highly privileged with access to Google Cloud secret keys, passwords, and Git credentials.

These risks are not exclusive to Jenkins. Any system that allows user-defined build processes and uses credentials in build processes faces the same challenges in preventing credentials exposure.

What OWASP Says About CI/CD Security

Learning about [OWASP's Top Ten CI/CD Security Risks](#) helps you understand the challenges of securing your software supply chain. This list identifies the Top Ten CI/CD risks in three general categories: access and credential risks, integration and dependency risks, and configuration risks.

Access and credential risks include the following:

Inadequate identity and access management

Modern CI/CD environments involve numerous human and machine identities interacting among multiple systems, and even a single compromised identity can lead to significant security breaches. According to OWASP, these identities can be compromised in many ways, from granting too-broad permissions to lapsing out of date. To mitigate these issues, OWASP recommends implementing the following:

- Continuous identity mapping and access review
- Removal of unnecessary permissions
- Prohibition of shared accounts and local credentials
- Implementation of just-in-time access protocols

Insufficient credential hygiene

Credentials are high-value targets for attackers because they provide access to critical resources and deployment capabilities. If they are shared to a common repository or unknowingly printed to a console's output, they can be read by anyone with access to those channels. For credential hygiene, OWASP recommends implementing the following:

- Regular credential rotation
- Use of temporary, scoped credentials
- Use of secure secret management systems
- Use of automated secret detection in code and artifacts

Insufficient PBAC

CI/CD execution nodes require access to resources that may be exploitable by malicious attackers if not secured properly. OWASP recommends implementing the following:

- Pipeline isolation between sensitivity levels
- Granular secret access control per pipeline stage
- Clean environment provisioning for each run
- Regular security patches and updates

Integration and dependency risks include the following:

Dependency chain security

Your CI/CD pipeline likely depends on other code to function, and attackers can exploit those dependencies to import malicious packages into the development process. OWASP identifies four types of dependency chain attacks:

dependency confusion (giving malicious packages the same names as legitimate packages), *dependency hijacking* (replacing legitimate packages with malicious ones), *typosquatting* (naming malicious packages variations of common misspellings of popular legitimate packages), and *brandjacking* (camouflaging malicious packages as trusted brands). OWASP recommends the following:

- Prohibiting package fetching from the internet and untrusted sources
- Verifying that private packages are within the organization's scope
- Preventing installation scripts from accessing secrets or sensitive resources

Poisoned pipeline execution

The CI/CD pipeline may be *poisoned* by malicious code if an attacker gains access to source control systems. This often happens when unreviewed code is submitted to the pipeline. OWASP recommends doing the following:

- Restricting pipelines that run unreviewed code to isolated Nodes and reviewing all CI configuration files before the pipeline runs
- Limiting the pipelines that can be run on public repositories and protecting sensitive pipelines with branch protection rules in the source control management (SCM)
- Following the principle of least privilege when setting SCM repository permissions policies

Ungoverned use of third-party services

Third-party services often drive CI/CD pipelines, and a compromised entity can expose access to the organization. OWASP provides an example of an attacker exploiting a third-party service with write permissions to push code to a repository, which triggers a build and infects the build system. To mitigate this, OWASP recommends the following:

- Imposing governance controls on third-party services
- Vetting third parties before granting access to resources
- Following the principle of least privilege when granting permissions to third parties

Improper artifact integrity validation

If artifacts in the CI/CD pipeline (such as code used in the build process) aren't validated, then they can push malicious code into the pipeline. OWASP recommends doing the following:

- Validating the integrity of all resources from development to production
- Using secure code-signing technology and artifact verification software

- Watching out for *configuration drift* in CI/CD assets where unmanaged versions go untracked and untested

Configuration risks include the following:

Insufficient flow control mechanisms

The impact of a security breach can be exacerbated by an attacker gaining access to the CI/CD pipeline to push malicious code wherever they want. OWASP recommends limiting this impact by ensuring that no human or machine identity can “ship sensitive code and artifacts through the pipeline without external verification or validation” and doing the following:

- Setting protection rules for branches containing sensitive or production-related code
- Limiting the use of auto-merge rules
- Preventing pipelines from being triggered without approval or review, where possible

Insecure system configuration

Insufficient system configurations can introduce risks into CI/CD pipelines, such as excessive access permissions, inadequate secrets hygiene, and unpatched software versions. OWASP recommends doing the following:

- Maintaining a complete inventory of systems and versions and checking them for known vulnerabilities
- Following the principle of least privilege and conducting periodic configuration reviews

Insufficient logging and visibility

The lack of logging capabilities and visibility into CI/CD pipeline operations makes it difficult to proactively detect security issues and remediate breaches once they’ve happened. For these risks, OWASP recommends the following:

- Setting up appropriate human and programmatic log sources at each pipeline stage
- Aggregating and correlating logs from different systems for intelligence gathering
- Employing alerts to flag anomalies and potential malicious activity

Foundations of CI/CD Security

The principles of securing software supply chains are no different from those for securing other aspects of software engineering, and they are equally important:

Principle of least privilege

As explained in [Chapter 2](#), this means granting users and workload accounts only the permissions required to perform their tasks.

Secure credentials

This means ensuring that credentials, API keys, private certificate keys, and other credentials are protected and rotated at regular intervals.

Isolate build environments

This means ensuring that each CI pipeline run is inaccessible by any other run.

Automation

The CI/CD process should be fully automated, with approval gates if necessary. Any changes to the process should be subject to review and approval.

Monitoring

This means continuous monitoring for unusual activities and logging all actions and events to facilitate auditing and incident review.

[Figure 9-2](#), from the [Supply-chain Levels for Software Artifacts \(SLSA\) framework](#), shows the various threat vectors in a typical software supply chain. Throughout the rest of this chapter, we will show how applying the preceding principles can help mitigate these threat vectors.



The SLSA framework helps ensure the integrity of software artifacts through a checklist of standards and controls for evaluating trust. Developed by Google, it covers software development areas such as signing, artifact provenance, build and release processes, vulnerability scanning, and attestation.

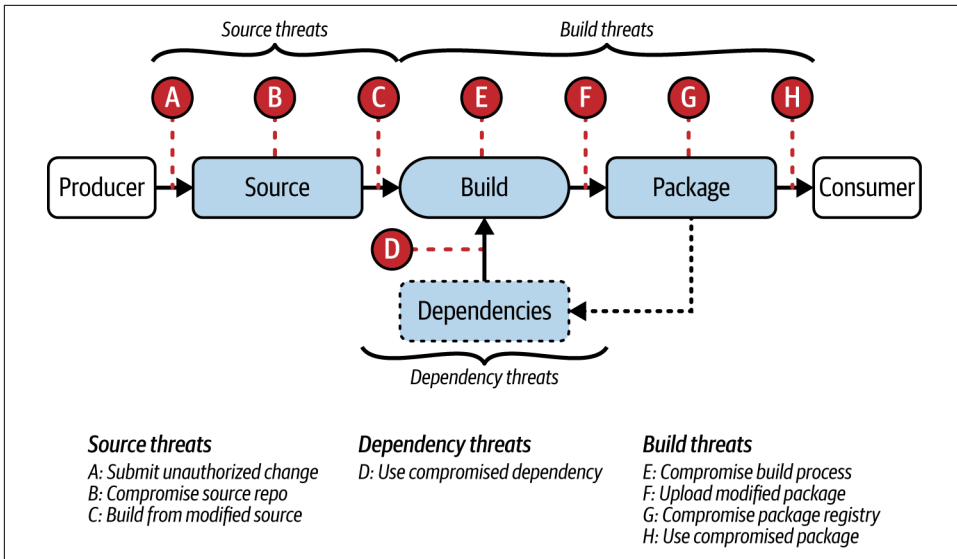


Figure 9-2. Possible threat vectors in a typical software supply chain

Source Code Threats

Source code threats cover the potential for a malicious actor to introduce unauthorized code changes into a codebase, even if that actor is a legitimate maintainer. An example was a source code threat to the XZ Utils data compression library, where a **trusted maintainer introduced a backdoor** designed to enable attackers to take over systems. As XZ is a core Linux utility used everywhere (including in the popular SSH program OpenSSH), this represented a serious problem. Fortunately, a curious engineer discovered this backdoor before its blast radius became significant.

Other source code threat examples are as follows:

- In 2022, the maintainer of the popular Node-ipc Node package manager (npm) **added malicious and destructive code** in protest of the war in Ukraine.
- The same year, the ctx Python library was **updated with malicious code** after an attacker took over the domain that owned the source code.
- The Great Suspender, a popular Chrome extension for suspending browser tabs, was **blocked by Google** after a newer maintainer added malware to the codebase.

Notably, these examples involve open source projects, which are particularly vulnerable to source code poisoning attacks. Often, the maintainers don't know each other well, and unpaid, time-constrained volunteers run most projects. This leads to less-than-ideal security hygiene, but the same issues apply to a commercial organization's private source code repositories.

The following sections describe security best practices that will help you mitigate source code threats.

Access control

The first step in preventing source code threats is to always apply the principle of least privilege when assigning RBAC permissions to users, especially regarding write and approval type access.

For critical, long-lived branches, such as `main`, you should also use branch protection rules to limit the actions that can be performed on a branch. For example, if you use rules to prevent pushing commits directly to `main` and requiring a pull (or merge) request before merging, it will limit the potential security risks to the branch.

Other mechanisms to consider are as follows:

- Before merging occurs, you can ensure that the source branch is up-to-date with the target branch and that at least one human has approved the change. This approver should be someone other than the creator of the pull request, because with version control systems like GitHub, you can configure yourself to approve your own PRs and merge them to `main`.
- You can use automated checks, such as static analysis, on requests that must pass before merging occurs. Typically, these checks run actions on pipelines in the CI system that build and test the code.

We also recommend that you use an IaC tool, like [Terraform](#) or [OpenTofu](#), to provision your Git repositories. (Both GitHub and GitLab provide Terraform providers.) These tools help you ensure that any new repository automatically conforms to your access control and protection rules.

Security-focused testing

As part of your CI pipelines, you should include the following steps that focus on security testing:

- You should use SAST to analyze your code for potential security vulnerabilities, such as SQL injection, buffer overflows, and insecure direct object references.
- You should also use code linting as an additional check against potential security issues. Although linters are primarily aimed at improving code quality and consistency, they often include security rules.
- You should use fuzzing to feed invalid, unexpected, or random data into your application and monitor how it reacts. This helps identify unusual behaviors that may lead to security vulnerabilities or unexpected outputs.

Dependency management

Dependency management is essential for reducing the security risks inherent in the production of software, and there are several best practices to consider:

Minimize dependencies.

Each *direct dependency* you specify will depend on others—sometimes, up to hundreds of other libraries and packages. These are *indirect* or *transitive dependencies*, and you have little control over their appearance in your application binary or package. Any dependency carries some risk of introducing security vulnerabilities into your application, so the fewer you have, the smaller the risk.

Use dependency tooling in your application projects to identify vulnerabilities.

Package managers like npm, Maven, and Golang modules, along with automated dependency management tools like **Dependabot**, can inform you about vulnerabilities. Dependabot also raises PRs to update those dependencies when new versions are released.

Update dependencies with the latest security patches, including transitive dependencies.

You should use the dependency lock file of your language and dependency tool to ensure consistency across builds. All dependency tools support the ability to specify ranges of versions for each direct dependency and pin a specific version.

Perform due diligence on external dependencies that you introduce into your projects to identify known security vulnerabilities.

You can check public security vulnerability databases such as **NVD** and **CVE**, and ideally, you should use automated tools to do this. This applies to open source and non–open source packages.

For open source, we recommend the **Scorecard** tool that's maintained by **the Open Source Security Foundation (OpenSSF)**. It's designed to help open source maintainers improve the security hygiene of their projects and assist consumers of open source to evaluate the security risks of the dependencies they want to use. Scorecard performs a set of checks against a GitHub project (GitLab is not currently supported) to evaluate its security posture, and it awards a score of 0 to 10 for each one. OpenSSF currently scans over one million of the most critical open source projects and publishes results in a public BigQuery dataset.

Scorecard checks include branch protection, CI tests, dependency management, code analysis, and review. You can run Scorecard against any public GitHub repository and any repository you own through its CLI and as part of a CI pipeline.

Commit integrity

When making a commit in Git, you can specify any values for the commit author's name and email address—with nothing to prevent you from impersonating someone

else. While this works as intended since Git doesn't take responsibility for authenticating committers, you have no guarantee that the commit author is correct or that the code change is actually what the author wrote.

Fortunately, you can verify or cryptographically sign a commit. Doing so allows you to prove that you were the author of a commit, and it ensures that the commit's contents cannot be tampered with. To sign your commits, you need to create a GNU Privacy Guard (GPG) key pair, add the name and email address of your Git identity, and configure Git to sign your commits using this key.

For GitHub or GitLab to show your commits as “verified,” you must ensure that the service has verified the email address you used in your key and that you have uploaded the public part of your GPG key to your account.

As an extra layer of protection, you can add a branch protection policy rule to your repository that allows only signed commits to specified branches.

Hardening the Build Process

Automated build pipelines are crucial parts of any software supply chain, and it's essential to protect them from attack. As pipelines tend to operate with high privileges, an attacker gaining access may be able to do the following:

- Add malicious code to the package being built.
- Access sensitive data that the pipeline uses, such as credentials or private keys.
- Access other environments.

The following sections describe methods of protecting build pipelines against malicious access.

Access control

If an attacker can access and alter the build process, then they can introduce modified source code that ends up in a legitimately produced package. In 2020, this mechanism was likely used to **infect SolarWinds' Orion product with a backdoor**. As we noted earlier in this chapter, software build systems should be treated like production environments, especially where access control and system changes are concerned. Thus, audit logging should be enabled, and admin permissions should only be granted when necessary.

Build protection

Running pipelines should be isolated from each other to prevent one from affecting or changing another. Each build pipeline execution should be performed in a *clean* manner, with environments from previous pipeline runs not being reused. This helps

prevent an attacker from using an earlier run to replace build tools with malicious versions, which can affect subsequent builds.

Managing Artifacts

Whether you are consuming artifacts or producing them for others to use, you will need to consider the following:

- How much you can trust public registries and what mitigations are needed to de-risk their use
- Securing your registries
- Providing incontestable proof that packages and images you are responsible for have been produced legitimately and have not been tampered with

Public registries

Public package registries like Docker Hub, npm, and the Python Package Index (PyPI) allow anyone to create an account and upload packages. Since they're popular and they host packages used by millions of applications, malicious actors target them with attacks that include typosquatting, account hijacking, and dependency confusion.

Typically, these registries are managed by small numbers of volunteers, so resolving security issues and improving overall security hygiene can be a lengthy process. It's impractical to avoid these public registries altogether, but you can implement the steps in the following sections to help mitigate the security risks.

Access public registries via private proxy. In your development and production environments, you should restrict direct access to public registries and use a private artifact repository manager instead. This manager acts as a central hub and proxy for all registries used by your developers, CI pipelines, and Kubernetes clusters.

The most popular commercial products are [Artifactory from JFrog](#) and [Nexus from Sonatype](#); the major cloud providers also provide managed artifact registries. These products allow public registries to be proxied through their service, meaning that developers, pipelines, and clusters can request images and packages directly from the private repository manager.

If the manager doesn't host the requested artifact, then it will automatically download and cache it from the appropriate public registry. This approach makes it easier to control from which registry packages can be taken, and it reduces the likelihood that you'll inadvertently use a risky registry.

Perform event-driven package assessments. A further benefit of using a private repository manager is that when a new or updated package is requested and pulled from its

source, various automated processes can be configured to run. These could include a vulnerability scan, a software composition scan, or a license scan.

The benefit is having these scans run at the time of the request, rather than later, when the package is already part of a CI pipeline.

Registry security

Securing your package registry is crucial because it serves as a central repository for critical components. A compromise can lead to the following severe consequences:

- Legitimate artifacts could be replaced with compromised ones.
- Sensitive information contained within private artifacts could be exposed.
- Services or environments could be disrupted.

We recommend that you use these strategies to secure your registry instances:

- Avoid exposing your registries to the public internet. If you produce artifacts for external consumption, use a separate registry service for these.
- Follow the principle of least privilege and use RBAC to facilitate granting users and service accounts only the necessary permissions to perform their tasks. Ideally, anonymous access to private registries should be prohibited.
- Enable audit logging to track activities and events.
- Schedule regular vulnerability scans (at least once per week) and ensure that there is a process for dealing with CVEs (especially for those with a severity of Critical or High).
- Digitally sign artifacts to enable verification of their integrity and authenticity.

Integrating Security Testing into CI/CD Pipelines

Security testing is a crucial component of a robust CI pipeline. It helps identify vulnerabilities early in the development process and helps reduce the risk of security incidents.

The following sections describe different types of security testing that you should consider for any CI/CD process.

Static application security testing

Static application security testing (SAST) tools examine source code to detect vulnerabilities and suspect code patterns, such as cross-site scripting, SQL injection, and a lack of input validation. The SAST process typically consists of the following steps:

- Parse code to understand its structure in a way that can be used in the various analysis steps.
- Analyze the control flows through and within functions, such as conditional and loop statements.
- Analyze the data flow through the unit under test.
- Perform pattern matching and rules-based analysis to identify code patterns that are known to indicate a vulnerability.
- Perform taint analysis to look at how data is validated and sanitized before it's used.

This process produces a report that lists the vulnerabilities found, their severity, and any recommended mitigations.

There are many open source and commercial SAST tools available, and you can plug the developer-focused ones into popular IDEs to give you immediate feedback and integrate them into CI pipelines.

However, SAST does have a few limitations to be aware of:

- *False positives* (findings that are not true vulnerabilities) can become overwhelming. To mitigate this, you should ensure that your tool has features for triaging SAST alerts. Sifting through too many false-positive alerts will result in alert fatigue or ignored reports.
- Certain code patterns can't be interpreted by SAST tools. For example, *reflection* (self-modifying code) cannot be analyzed unless the code is running. The [OWASP Testability Patterns for Web Applications initiative](#) was created to address this.

Dynamic application security testing

Dynamic application security testing (DAST) is a testing technique that simulates attacks against a running application to identify vulnerabilities. Also known as *closed testing*, it tests software units from the outside in. Since it operates by running application code, it can find vulnerabilities that SAST tools would miss.

DAST tools require specialized expertise to configure correctly, and they are typically slow to run. Therefore, to avoid bottlenecks, teams often include DAST in a CI pipeline that's separate from the main application build pipeline.

The most popular open source DAST tools are [Zed Attack Proxy \(ZAP\)](#) and [Nikto](#), and there are many commercial options as well. [A comprehensive list of tools](#), both open source and commercial is maintained by OWASP.

Fuzzing

Another dynamic security testing technique, which we described earlier in this chapter, is fuzzing. A popular open source fuzzing tool is **OSS-Fuzz**, which was developed by Google as part of the Chrome browser and OS project.

Interactive application security testing

Interactive application security testing (IAST) uses *sensor modules* that are embedded in application code to analyze the application while it's run by an activity interacting with it, such as an automated test. If these sensors detect any vulnerabilities, they trigger an alert for further inspection.

Examples of vulnerabilities that IAST looks for include using cleartext API keys, a lack of sanitizing user inputs, and a lack of Secure Sockets Layer encryption. Similar to DAST, IAST requires the application to run. However, it can pinpoint the vulnerable line(s) of code more accurately due to the sensors embedded in the code.

A downside of IAST is language support, as the sensor modules need to be embedded in your code and work with your application's programming language.

Common vulnerabilities and exposures scanning

Common vulnerabilities and exposures (CVE) is the de facto standard for recording details of known security vulnerabilities. Each vulnerability is given a unique identifier (a CVE ID), a description, a severity rating, and known mitigation details.

CVE scanning tools compare a list of all dependencies in your application (based on its dependency file, such as *requirements.txt* for Python) to public CVE databases. Any matches indicate known vulnerabilities for the specific dependency and version. These tools can also scan OS packages for most Linux distributions, and they can analyze Docker images by looking for package manifests and dependency lock files.

Typically, these scanning tools can be integrated into IDEs and CI pipelines. This makes results easy to access and reports quick to generate. Commercial tools shine here, and several noteworthy open source tools provide both scanning and reporting functionality:

- **Dependency-Track** from OWASP is a software composition analysis platform that uses SBOMs to identify vulnerabilities in software components. It doesn't generate SBOMs, but it accepts existing ones via its API. Ingesting an SBOM triggers various vulnerability scans, the results of which are visualized in Dependency-Track's web-based dashboard.
- **OpenClarity** is a platform for identifying security vulnerabilities using SBOMs. Unlike Dependency-Track, it generates an SBOM from Docker and VM assets

before executing various security scans. The results of these scans are visualized via a web-based dashboard.

Monitoring and Incident Response

Your CI/CD pipelines should be monitored for security anomalies in the same way that applications and their hosting environments are monitored. Software supply chains are an attractive target for malicious actors, and one of the lines of defense is to implement an anomaly detection mechanism. If your organization already uses a SIEM solution as part of its security posture, you can configure your CI environment to push logs and metrics into it and work with your security team to develop anomaly detection rules and alerts.

The types of unusual CI/CD activity to look for include the following:

- Excessive pipeline executions and failures
- Pipeline execution times that are substantially different from the average
- Excessive CPU, memory, and networking usage

If you don't have access to a SIEM solution, you can use an application monitoring system instead. For example, Prometheus aggregates data in real time and **can be configured to detect anomalies**.

If detected anomalies result from a cyberattack, it's important for you to have a security incident response plan. You can adapt NIST's *Incident Response Recommendations and Considerations for Cybersecurity Risk Management* for software supply chain incidents.

There are four phases in the NIST framework:

Preparation

Establish a plan that identifies the people, resources, and tools that should be involved in handling a response.

Detection and analysis

Identify possible attack vectors, ensure appropriate monitoring for anomalies, and devise processes for validating whether detected anomalies are false positives or indicators of a genuine incident.

Containment, eradication, and recovery

Develop a containment strategy to limit the blast radius, eradicate malicious components, disable compromised accounts, replace API and certificate keys, restore operations to normal, and remediate vulnerabilities to prevent similar incidents.

Post-incident activities

Hold lessons-learned sessions to improve security measures and incident handling process.

An important aspect of any incident resource plan is that it must be prepared before an incident occurs. Creating one after an incident has been detected is unlikely to produce a successful outcome.

Provenance, Attestations, and Digital Signatures

Some of the most significant challenges of using software artifacts created by someone else are deciding whether to trust them and gauging the level of risk you will be incurring. To help you with this decision, you should consider the following:

- The provenance of the artifact, meaning how it was created and built and other components it contains
- Your ability to verify the authenticity and integrity of the provenance and thus the artifact

Although many people associate trust and risk problems with open source, such issues also affect closed source and an organization's internal source.

The following sections offer these explanations:

- What *provenance* means in the context of software components
- How provenance is created, consumed, and verified
- Useful specifications, standards, and patterns to consider adopting

SLSA is a useful framework and set of guidelines. Aimed at software component producers and consumers, the current version of SLSA (v1.0) focuses on making software more secure. It defines the following four levels in ascending order of security hardening, and it also defines guarantees that must be met by either the software producer or the build platform used to transform the source code into a package:

Build L0: there are no guarantees.

This level represents a lack of SLSA. It is intended for development or test builds of software that are built and run on the same machine, such as unit tests.

Build L1: provenance exists, but it can't be verified.

At Build L1, organizations must maintain a build environment that is consistent and reproducible. This means the build environment must be documented, and changes to the environment must be tracked. Additionally, the build process must be reproducible using the same inputs and build environment.

To achieve Build L1 compliance, organizations must also track critical components, including dependencies and their versions. This ensures that any changes or vulnerabilities in these components can be quickly identified and addressed.

Build L2: provenance can be verified, but risks remain that it could be forged.

After achieving L1, organizations must record and keep current the details of their build environment. This means documenting the build environment and any tooling or scripts used in the build process.

Build L2 compliance relies on proving a repeatable build process and verifying the integrity of critical components. This includes regularly scanning the components for vulnerabilities and ensuring that any vulnerabilities are quickly addressed.

Build L3: provenance can be verified, and forging it would be beyond the capabilities of almost all adversaries.

After meeting L1 and L2, organizations must use automated tooling to enforce consistent builds and verify that only authorized individuals can modify the build environment. This means that the build process is automated and repeatable, with clear checks and balances to ensure that only trusted individuals can change the process.

The benefit of SLSA is that it specifies security hardening requirements and a mechanism for demonstrating and verifying that a software component's build process has achieved a particular level. SLSA includes a **provenance spec**, which is part of a broader **attestation framework** from the **in-toto project**. The purpose of this spec is to describe how an artifact was produced so consumers can verify that the expected process was followed and others can rebuild the artifact if desired. The *in-toto Attestation Framework* defines lightweight *statements*, each of which communicates information about some aspect of the software supply chain used to produce an artifact.

These statements are known as *attestations*, and each one has a particular schema or *predicate* (referred to as a *predicate type*). SLSA provenance is one such predicate type.

Provenance generation

Unfortunately, provenance generation is not a straightforward process, especially for levels 2 and 3, and not all build platforms have support. Known options for the different SLSA levels are as follows:

Level 1

- Azure DevOps proof of concept
- Jenkins SLSA provenance attestation plug-in
- Jenkins SLSA provenance generator proof of concept
- Jenkins in-toto provenance plug-in

- Tekton Chains
- Tejolote

Level 2

- Factory for Repeatable Secure Creation of Artifacts (FRSCA)
- GitLab

Level 3

- GitHub Actions
- Google Cloud Build

Provenance storage

The storage convention is to keep the provenance attestation with its artifact. Various package repositories, like *Maven for .jar files* and *deps.dev for npm packages*, support this. You can also use OCI image registries to store artifacts other than images (such as plain-text files containing provenance). An open source tool called *Cosign* is an easy way to sign and push artifacts into OCI registries.

SBOMs

SBOMs are documents that provide an inventory of packages and components that are included in a piece of software. They include key information such as names, version numbers, checksums, and license information. Support for SBOMs is growing across the IT industry, aided by support from government organizations in many countries.

SBOMs are useful when you want to answer questions such as, “Are any of my software components affected by this critical vulnerability in library X?” or “What critical vulnerabilities are contained within OCI image Y?”

You can create SBOMs during the build process by examining the source and build metadata to retrieve a list of all dependencies used. You can also generate them after the fact by scanning the compiled artifacts and OCI images. Both approaches have challenges, such as scanners missing dependencies (especially with artifacts that don’t include all dependency metadata) and trusting that third-party SBOMs were created correctly. (SLSA provenance can help provide a sufficient level of confidence here.)

The two main SBOM formats are *System Package Data Exchange (SPDX)* by the Linux Foundation and *CycloneDX* by OWASP. SBOM tools typically support one or both of these formats. Converting from one format to another is possible, but the resulting format may lose some information, as each format has properties not supported by the other.

Popular open source tools for generating SBOMs are as follows:

- [Syft](#) generates both SPDX and CycloneDX format SBOMs.
- [The SBOM tool](#) generates SPDX format SBOMs.
- [The CycloneDX generator](#) generates CycloneDX format SBOMs.

Many organizations that consume third-party packages and OCI images have developed automated processes to generate an SBOM for a package or image as soon as it is pulled in. Typically, these are stored in centralized, private registries that make it straightforward to invoke a security-focused pipeline as soon as a new artifact is imported into it. These pipelines can run security scans and generate SBOMs and other attestations against these artifacts.

Digital signatures

Generating, storing, and sharing SBOM and SLSA provenance attestations is a big step toward securing your software supply chains. Equally important is the ability to verify these attestations as genuine and untampered with.

Achieving this requires the SBOM and provenance attestations to be digitally signed, ideally through an automated process. An open source project that's focused on providing tools for signing and verifying artifacts and attestations is [sigstore](#). This project provides several tools to facilitate the process, including [Cosign](#) for signing and verifying attestations, artifacts, and images. Cosign can be used independently in a CI pipeline or combined with other sigstore tools. [Fulcio](#) acts as a certificate authority that generates short-lived certificates that assert the producer's identity. These can be used by Cosign to sign the artifacts. [Rekor](#) provides a transparency log that stores the artifact hash, signature, and certificate for later public audit.

Summary

Securing CI/CD pipelines and software supply chains requires a comprehensive approach that addresses identity management, access control, dependency security, and system configuration. Implementing the frameworks and practices outlined in this chapter can significantly reduce your risk of supply chain attacks while maintaining development velocity.

To further explore CI/CD security best practices, we recommend that you consult the following resources and references:

- [Software Supply Chain and DevOps Security Practices from NIST](#)
- [OWASP Software Assurance Maturity Model](#)
- [Defending Against Software Supply Chain Attacks](#)
- [Information and Communications Technology Supply Chain Security](#)

- *CIS Software Supply Chain Security Guide*

A

- ABAC (attribute-based access control), 20, 41, 45
- Accenture, IAM failure at, 53
- access control, 20
 - (see also identity and access management)
 - broken, 55
 - CI/CD pipeline, 163, 168, 170
 - code security best practices, 20
 - IAM service comparison, 100-101
 - IGA reviews, 52
 - as LDAP use case, 65
 - policies for, ix, 16
 - secrets management, 74
 - session management, 17
- access control lists (ACLs), 44-45
- ACME (Automatic Certificate Management Environment) protocol, 129
- Active Directory (AD), 37, 95
- adaptive authentication, 38
- admin Role, Kubernetes, 114
- admission controller, 119
- AES-256 encryption standard, 16, 73
- AI (artificial intelligence)
 - access policies for AI-assisted deployments, ix
 - in IAM, 67
 - impact on security, ix
 - security automation with, 139, 142, 153-154
- AKS (Azure Kubernetes Service), 120
- Alpine Linux, 121
- Amazon
 - DynamoDB, 98
 - Elastic Kubernetes Service (EKS), 120
 - S3 Block Public Access feature, 89
 - S3 Bucket with Terraform, 88
- Amazon Web Services (see AWS)
- Ambient mode, service mesh, 135
- anomaly detection, with machine learning, 153
- Ansible Galaxy, 156
- Ansible, automation example, 154-157
- Apache Struts, 25
- API server, Kubernetes, 108
- AppArmor, Linux, 117
- application provisioning, 48
- approver-policy, 131
- architecture models, IAM, 59-62
- artifact integrity validation, CI/CD pipeline, 164
- artifact management, CI/CD pipeline, 171-172
- Artifactory (JFrog), 171
- attestations, CI/CD pipeline, 176-179
- attribute-based access control (ABAC), 20, 41, 45
- auditing and compliance
 - automating, 143
 - with blockchain, 68
 - cloud security audit trails, 96
 - code security best practices, 21-22
 - IGA reporting, 53
 - secrets management, 75-76
 - SPIFFE standard, 127-129
- authentication (AuthN), 9, 29-40
 - best practices, 19, 40
 - biometric, 19, 32
 - cloud security, 96
 - cross-cloud, 127
 - and failures in identification, 56

- and IAM, 50
- JWT, 34-36
- LDAP, 36, 65
- MFA, 31, 49, 74, 102
- OIDC, 32-33, 64
- passwordless, 66
- risk analysis during, 38-39
- SAML, 37
- stateless, 35
- token-based, 31, 34-36
- username and password, 30
- workload-to-workload, 127-128
- authorization (AuthZ), 9, 29-30, 40-46
 - ACLs, 44-45
 - cloud security, 96
 - code security best practices, 19, 20
 - and IAM, 50-52
 - JWT, 43
 - OAuth 2.0, 41-42
 - service meshes, 133-134
 - workload-to-workload, 127-128
 - XACML, 45-46
- AuthZ server role, OAuth 2.0, 42
- Automatic Certificate Management Environment (ACME) protocol, 129
- automation tools and processes, ix, 142-143
 - (see also security automation)
 - AI-based decision-making, 67
 - CI/CD pipelines, 166
 - cloud native verification tools, 93
 - code security scanning and analysis, 25
 - compliance checks for cloud security, 96
 - container image security checks, 122
 - managing machine identities across pipelines, ix
 - secrets management, 70, 76
- AWS (Amazon Web Services), 88-89
 - access, 100
 - cloud resource structures, 94
 - CloudFormation, 150
 - IAM Identity Center, 100
 - Key Management Service, 73
 - permissions, 98
 - Secrets Manager, 78
- Azure, 95
 - access control, 101
 - Active Directory, 95
 - AKS, 120
 - Portal, 99

Resource Manager, 150

B

- base image, deciding on, 120-122
- behavioral analytics, AI in identity security, 67
- BigQuery dataset, 169
- binding, 99, 106, 113
- biometric authentication, 19, 32
- blast radius of security breach, 8
- blockchain, in IAM, 67-68
- blue-green deployment, 136
- botnets, 8
- branching strategy, for PaC, 152
- brandjacking, 164
- broken access control, 55
- build pipeline, securing container, 93

C

- Calico, 123
- California Consumer Privacy Act (CCPA), US, 75
- canary deployment, 136
- centralization principle, secrets management, 77
- centralized IAM model, 59
- centralized identity management, cloud security, 96
- cert-manager for Kubernetes TLS identity, 129-132
- certificate request (CSR), 130
- certificate-based identity system, service mesh, 133
- certificates
 - digital signatures, 179
 - root, 130
 - service meshes, 133
 - TLS automated identity security, 129-132
- Chainguard, 121
- ChatGPT, cybersecurity breach of 2023, 4
- CI/CD pipeline security, 159-179
 - artifact management, 171-172
 - digital signatures, 179
 - hardening the build process, 170
 - monitoring and incident response, 175-176
 - OWASP Top Ten security risks, 162-165
 - provenance attestations, 176-179
 - secrets management, 161-162
 - source code threats, 167-170
 - SPIFFE security for, 127

- testing, 168, 172-175
- CIAM (customer identity and access management), 54, 60
- Cilium, 123, 133, 135
- CircleCI breach, 140, 161
- client role, OAuth 2.0, 42
- client-side validation, 15
- closed testing, 173
- Cloud Native Computing Foundation (CNCF), 90
- cloud native environments, 90-95
- cloud security, 85-103
 - audit trails and monitoring, 96
 - automated compliance checks, 96
 - centralized identity management, 96
 - cross-cloud authentication, 127
 - CSP comparisons, 93-95, 98-102
 - DevOps security practices, 96
 - IAM in cloud environments, 62, 98-102
 - misconfiguration problem, 85, 88-90, 106
 - risks in cloud deployments, 87
 - zero standing privileges, 96
- cloud service providers (CSPs), 98-102
 - access control, 100-101
 - IaC tools for security automation, 150
 - IAM resource structures, 93-95
 - permissions, 98-99
 - secrets management, 78
 - switching between, 101-102
- cluster security, 92, 106, 132-136
- cluster-admin Role, 114
- ClusterRoleBindings, Kubernetes RBAC, 113, 115
- ClusterRoles, Kubernetes RBAC, 113-115
- clusters, Kubernetes, 92, 107-108
- CNCF (Cloud Native Computing Foundation), 90
- CNI (Container Network Interface), 123
- code linting, 168
- code obfuscation, 17-19, 79
- code security, 13-28
 - access control, 20
 - auditing and logging, 21-22
 - AuthN and AuthZ considerations, 19, 20, 96
 - automation, 25, 141
 - avoiding insecure components, 25
 - cloud native environments, 93
 - code obfuscation, 17-19
 - credential management, 96
 - development environments, securing, 23-25
 - encryption, 16-17
 - input validation, 15
 - resource management, 17
 - secrets management, 22-23
 - security standards, 26-27
 - session management, 17
 - testing and validation, 10
 - zero trust model, 13-14
- code signing, 24
- code to access secrets, 78-79
- Codecov breach, 6
- commit integrity, CI/CD pipeline, 169
- common vulnerabilities and exposures (CVE) database, 8, 169, 174
- common weakness enumeration (CWE), 8, 26
- compliance (see auditing and compliance)
- computer security incident response teams (CSIRTs), 144
- conditional access policies, CSPs, 102
- ConfigMaps, 115
- configuration files, secrets management, 78
- configuration risks, CI/CD pipeline, 165
- Conjur Cloud Open Source, 110
- Conjur Open Source, 80, 154-157
- consumer applications, IAM, 54
- container images, securing, 92, 120-122
- Container Network Interface (CNI), 123
- container runtime, 108
- container security (containerization), 92 (see also Kubernetes)
- container storage interface (CSI), 112
- containers, 105, 116-123
- continuous integration/continuous delivery (see CI/CD pipeline security)
- control plane, 108, 132
- controller manager, Kubernetes, 108
- Cosign, 179
- CRDs (CustomResourceDefinitions), 129
- credentials
 - CI/CD pipeline, 163, 166
 - code security, 96
 - embedding in generated scripts, ix
 - password and username as basic, 30
 - revocation of access, 72
- cross-cloud AuthN, 127
- CSI (container storage interface), 112
- CSI driver volumes, 130
- csi-driver-spiffe, 131

- CSIRTs (computer security incident response teams), 144
- CSPs (see cloud service providers)
- CSR (certificate request), 130
- customer identity and access management (CIAM), 54, 60
- CustomResourceDefinitions (CRDs), 129
- CVE (common vulnerabilities and exposures) database, 8, 169, 174
- CVE scanning tools, 174
- CWE (common weakness enumeration), 8, 26
- cyber attack threat, 2
- cyber-attack types, 7-9
- CyberArk, 146
 - behavior analytics, 39
 - container security automated tools, 123
 - Cora AI, 154
 - identity solutions, 54
 - Kubernetes architecture, 110
 - OPM, 146
 - Secrets Management, 79-82, 110
 - Secrets Manager SaaS, 78, 152
- CycloneDX, 178

D

- DAC (discretionary access control), 116
- DAST (dynamic application security testing), 25, 93, 173
- data at rest, encryption for, 16
- data in transit, encryption for, 16
- data plane, 132
- data protection, 10
- DDoS (distributed denial-of-service attack), 8
- decentralized IAM model, 60
- decentralized identifiers (DIDs), 60
- default ports, avoiding, 126
- deny-all policy, as network default, 126
- Dependabot, 169
- dependency confusion, 164
- dependency hijacking, 164
- dependency management, CI/CD pipeline, 169
- Dependency-Track, 174
- deprovisioning, identity, 48, 57, 65
- developers
 - extended skillset for enterprise, 4
 - protection of supply chain role, 2
- development and production, separating with PaC, 152
- development environments, securing, 23-25

- DevOps
 - CI/CD pipeline, 159
 - cloud security, 96
- DevSecOps, 27
- digital forensics, 143, 144
- digital signatures, CI/CD pipeline, 179
- direct dependencies, 169
- directory information tree (DIT), 37
- directory services, 48
 - (see also lightweight directory access protocol)
- directory system agent (DSA), 37
- directory user agent (DUA), 37
- discretionary access control (DAC), 116
- distinguished name (DN), 37
- distributed denial-of-service attack (DDoS), 8
- distroless images, 121
- DIT (directory information tree), 37
- Docker
 - and crypto mining attack, 108
 - development of, 105
 - distroless image, 121
- DSA (directory system agent), 37
- DUA (directory user agent), 37
- dynamic application security testing (DAST), 25, 93, 173
- dynamic secrets, 77
- DynamoDB, 98

E

- EaaS (encryption as a service), 73
- eBPF (Extended Berkeley Packet Filter), 135
- ECC (elliptic-curve cryptography), 16
- edit Role, 114
- EDR (endpoint detection response), 145
- egress NetworkPolicies, 124-125
- EHR (electronic health records), 45, 69, 74
- EKS (Amazon Elastic Kubernetes Service), 120
- electronic health records (EHR), 45, 69, 74
- electronic protected health information (ePHI), 27
- elliptic-curve cryptography (ECC), 16
- embedding credentials in generated scripts, ix
- encryption
 - code security best practices, 16-17
 - of connections, 10
 - Kubernetes support for, 110
 - secrets management, 73
 - secrets storage, 71

- switching between CSPs, 102
- encryption as a service (EaaS), 73
- encryption keys, 16, 73
- endpoint detection response (EDR), 145
- endpoint protection, automating, 143
- enterprise applications, IAM, 54
- environment variables, secrets management, 78
- ephemeral containers, Kubernetes, 122
- ePHI (electronic protected health information), 27
- Equifax data breach, 25
- ESO (External Secrets Operator), 80, 81, 110
- etcd key-value store, 108
- event-driven package assessments, 171
- expiration of secrets, 72
- exploit, 8
- Extended Berkeley Packet Filter (eBPF), 135
- extended detection response (XDR), 145
- Extensive Access Control Markup Language (XACML), 45-46
- external dependencies, 169
- External Secrets Operator (ESO), 80, 81, 110

F

- factors, in AuthN, 30
- failures in identification and AuthN, 56
- Federal Information Security Modernization Act (FISMA), US, 75
- FIDO Alliance, 32
- FIDO, passwordless AuthN, 66
- file management, security practices for, 17
- filesystem ACLs, 44
- firewall automation, 145
- FISMA (Federal Information Security Modernization Act), US, 75
- flow control mechanisms, CI/CD pipeline, 165
- Fulcio, 179
- full code security automation, 141
- fuzzing, dynamic testing technique, 168, 174

G

- garbage collection, 17
- GCP (Google Cloud Platform), 95, 99, 100
- General Data Protection Regulation (GDPR), Europe, 75
- Git, commit integrity in, 169
- GKE (Google Kubernetes Engine), 120
- GKE Sandbox, 120
- GNU Privacy Guard (GPG), 170

- Google Cloud
 - Deployment Manager, 150
 - Key Management, 73
- Google Cloud Platform (GCP), 95, 99, 100
- Google Kubernetes Engine (GKE), 120
- Google OAuth 2.0 API, 33
- GPG (GNU Privacy Guard), 170
- group management, as LDAP use case, 65
- gVisor, 120

H

- hard token versus soft token, MFA, 31
- hardcoded secrets, avoiding, 24
- hardware security modules (HSMs), 16
- hash algorithms to store secrets, 71
- HashiCorp Vault, 78
- health care software, identity security, 75
- Health Insurance Portability and Accountability Act (HIPAA), 27
- honeypot, 8
- HSMs (hardware security modules), 16
- HTTP API, Kubernetes, 108
- human identity management, 3
- hybrid clouds, 86
- hybrid IAM model, 61

I

- IaaS (Infrastructure as a Service), 86
- IaC (infrastructure as code), 149-151, 152
- IAST (interactive application security testing), 174
- identity and access management (IAM), 30, 47-68
 - access management, 49, 100-101
 - AI in, 67
 - architecture models, 59-62
 - AuthN methods and technologies, 50
 - AuthZ methods and technologies, 50-52
 - AWS IAM Identity Center, 100
 - blockchain in, 67-68
 - CI/CD environment risks, 163
 - cloud environments, 62, 98-102
 - CSP resource structures, 93-95
 - identity management, 48
 - IGA, 52-53
 - ILM, 57-58
 - importance of, 53-56
 - security automation, 142
 - standards in, 63-66

Identity Defined Security Alliance (IDSA), 2
identity federation, switching between CSPs,
102
identity governance and administration (IGA),
52-53
identity lifecycle management (ILM), 57-58
identity management, 48
identity provider (IdP), 63-64
identity security, 1-11
 automation of, 139-157
 coding best practices, 13-28
 common attack types, 7-9
 machine identities, 5-7
 modern enterprise's impact on, 4-5
 zero trust identity security, 2-4
identity synchronization and reconciliation, 58
identity verification, with blockchain, 67
IdP (identity provider), 63-64
IDS (intrusion detection system), 145
IDSA (Identity Defined Security Alliance), 2
IGA (identity governance and administration),
52-53
ILM (identity lifecycle management), 57-58
immutable data structures, avoiding for secrets
storage, 79
in-toto Attestation Framework, 177
incident response plan, 175
indirect dependencies, 169
industry standards for secrets management, 75
Infrastructure as a Service (IaaS), 86
infrastructure as code (IaC), 149-151, 152
ingress NetworkPolicies, 124
ingress policies, starting with, 126
injection attacks, 7, 17, 55
input validation, code security, 15
insecure components, avoiding in code, 25
insecure design, 56
integration and dependency risks, CI/CD pipe-
line, 163-165
interactive application security testing (IAST),
174
intrusion detection system (IDS), 145
isolation of build environments, 166
isolation of test environments, 24
Istio, 133-134
istio-csr compatibility layer, 132

J

Jenkins, 161

JSON web token (JWT), 19, 34-36, 43, 64

K

Kata, 120
key management services (KMSs), 73
key management systems, 16
key rotation, 16
kube-apiserver, 108
kube-controller-manager, 108
kube-proxy, 108
kube-scheduler, 108
kubelet, 108
Kubernetes, 105-137
 best practices, 112-126
 cert-manager for TLS identity, 129-132
 cluster component, 92, 107-108
 ClusterRole, 113-116
 ClusterRoleBinding, 113-116
 CSI Driver, 110-112
 ephemeral containers, 122
 network security, 108, 123-126
 NetworkPolicies, 123-125
 Pod Security Admission controller, 118-120
 Pod Security Standards, 118-120
 Pods and containers, 106, 108, 116-123
 RBAC API, 113-116
 Role, 113-116
 RoleBinding, 113-116
 sandboxed Pods, 120
 secrets management, 109
 security challenges, 108
 service accounts, 112
 service meshes for cluster security, 132-136
 SPIFFE for identity management, 127-129
 workings of, 107-108
Kubernetes Secrets, 109
Kubesploit, 123
KubiScan, 123
Kuma, 133
Kyverno, 120

L

label changing restrictions, network policies,
126
LDAP (lightweight directory access protocol),
36, 65
least privilege
 ACLs, 44
 CI/CD pipelines, 166

- container security, 92
- IaC tools, 150
- OAuth 2.0, 42
- secrets management, 74
- switching between CSPs, 101
- lifecycle of secrets, 72
- lift-and-shift versus cloud native applications, 90-95
- lightweight directory access protocol (LDAP), 36, 65
- Linux, 116, 120
- load balancing, containers, 105
- local environment variables, secrets management, 79
- logging and monitoring, 10
 - CI/CD pipelines, 165, 166
 - cloud security, 96
 - code security best practices, 21-22
 - IaC scripts, 151
 - reporting activity automation, 143
 - secrets management, 74, 79
 - security automation, 149
 - security failures in, 56
- low-code security automation, 142

M

- machine identities, 2, 4, 23
 - challenges of securing, 5-7
 - increase in, 2
 - managing across pipelines, ix
 - secrets management, 69
 - and zero trust challenge, 3
- machine learning (see AI)
- memory
 - protecting secrets in, 79
 - secure code practices for, 17
- MFA (multifactor authentication), 31, 49, 74, 102
- Microsoft, 95
 - (see also Azure)
 - Active Directory, 37
 - Entra ID, 95
 - Windows Nodes or Pods, 117
- minimal base image, deciding on, 120-122
- misconfiguration problem, cloud security, 85, 88-90, 106
- modern enterprise's impact on identity security, 4-5
- monitoring

- auditing secrets management, 74
- incident response in CI/CD pipeline, 175-176
 - and logging (see logging and monitoring)
- multicloud architectures, 86
- multifactor authentication (MFA), 31, 49, 74, 102
- multistage Docker builds, 122
- mutual TLS (mTLS), 136

N

- naming conventions, network policies, 126
- National Institute of Standards and Technology (NIST), 8, 27, 175
- National Vulnerability Database (NVD), 9, 169
- natural language processing (NLP), 154
- network security
 - ACL connections, 44
 - CNI, 123
 - Kubernetes, 108, 123-126
 - NetworkPolicy resource, 123-125
 - Nexus (Sonatype), 171
 - Nikto, 173
 - NIST (National Institute of Standards and Technology), 8, 27, 175
 - NIST Cybersecurity Framework, 27
 - NIST SP 800-204D, 27
 - NIST SSDF, 27
 - NLP (natural language processing), 154
 - no-code security automation, 141
 - Node package manager (npm), 167
 - nodes, cluster, 108
 - NVD (National Vulnerability Database), 9, 169

O

- OASIS (Organization for the Advancement of Structured Information Standards), 45
- OAuth 2.0, 41-42, 64
- OAuth 2.0 API (Google), 33
- observability
 - secrets management, 74
 - service meshes, 136
- OIDC (OpenID Connect), 32-33, 64
- on-premises cloud deployments, 85
- one-time passcode (OTP), 39
- OPA Gatekeeper, 120
- Open Policy Agent (OPA), 151
- Open Source Security Foundation (OpenSSF), 169

open source software, protecting, 24
open testing, 93
Open Worldwide Application Security Project (OWASP), 9, 26, 55-56
OpenClarity, 174
OpenID Connect (OIDC), 32-33, 64
OpenTofu, 168
orchestration, container, 107
Organization for the Advancement of Structured Information Standards (OASIS), 45
OSS-Fuzz, 174
OTP (one-time passcode), 39
OWASP (Open Worldwide Application Security Project), 9, 26, 55-56
OWASP Top Ten, 55-56, 162-165

P

PaaS (Platform as a Service), 86
PaC (policy as code), 151-152
PAM (privileged access management), 52, 146
PAP (policy administration point), 45
parameterized query, 17
pass the hash attack type, 7
passkeys, 32
password fatigue, 32
passwordless AuthN, 66
passwords
 self-service management, 48
 username and password (AuthN), 19, 30
Payment Card Industry Data Security Standard (PCI DSS), 27, 75, 145
PBAC (policy-based access control), 20, 41, 163
PDP (policy decision point), 45
PEP (policy enforcement point), 45
performance and security trade-off, 135
permissions
 cloud services, 98-99
 least privilege principle, 150
 RoleBindings and ClusterRoleBindings, 113
Personal Information Protection and Electronic Documents Act (PIPEDA), Canada, 75
phishing attack type, 7, 9
PIP (policy information point), 45
plain text/hardcoded secrets, 71
Platform as a Service (PaaS), 86
playbooks, security automation development, 148
Pod Security admission controller (PSA), 118
Pod Security Standards (PSS), 118-120
Pods, Kubernetes, 106, 108, 116-123
poisoned pipeline execution, CI/CD pipeline, 164
policies
 access control, ix, 16
 approver-policy, 131
 CSP monitoring and enforcement, 102
 Kubernetes network management, 123-126
 NetworkPolicy resource, 123-125
 OPA, 151
 PaC, 151-152
 PSS, 118-120
 secrets management, 80-81
 switching between CSPs, 101
policy administration point (PAP), 45
policy as code (PaC), 151-152
policy decision point (PDP), 45
policy enforcement point (PEP), 45
policy information point (PIP), 45
policy-based access control (PBAC), 20, 41, 163
post-quantum cryptography (PQC), 16
predictive analytics, automating with AI, 153
President's EO on Improving the Nation's Cybersecurity, 27
principle of least privilege (see least privilege)
privacy regulations, compliance with, 75
private artifact repository manager, 171
private clouds, 85, 86
private proxy, accessing public registries with, 171
privilege escalation attack, 8, 74, 116
privileged access management (PAM), 52, 146
privileged container, 117
privileged or unprivileged settings, Kubernetes, 116
provenance attestations, CI/CD pipeline, 176-179
provisioning, identity, 57, 58, 65
proxy sidecars, service mesh, 132, 135
PSA (Pod Security admission controller), 118
PSS (Pod Security Standards), 118-120
public clouds, 86
public package registries, CI/CD pipeline, 171

Q

quantum computing, encryption vulnerability, 16

R

- rainbow tables, 71
- RBA (risk-based authentication), 38
- RBAC (see role-based access control)
- RBAC API, Kubernetes, 113-116
- Red Hat Ansible automation example, 154-157
- refresh tokens, 43
- registry security, CI/CD pipeline, 171-172
- Rekor, 179
- relative distinguished name (RDN), 37
- replicas (Kubernetes Pods), 106
- reporting and compliance, automating, 143
- resource management, code security, 17
- resource owner role, OAuth 2.0, 42
- resource server role, OAuth 2.0, 42
- revocation of access to credentials, 72
- RFC 7519 open standard, 34, 36
- risk analysis during AuthN, 38-39
- risk-based authentication (RBA), 38
- Rivest-Shamir-Adleman (RSA) encryption
 - algorithm, 16
- role management, 57, 65, 101
- role-based access control (RBAC), 20, 41
 - IAM access management, 50-52
 - IAM role management, 58
 - secrets management, 74
- RoleBindings, Kubernetes RBAC, 113, 115
- Roles, Kubernetes RBAC, 113-115
- rolling updates, with containers, 106
- root certificate, 130
- rotation of secrets, 70, 72
- RSA (Rivest-Shamir-Adleman) encryption
 - algorithm, 16

S

- S3 Bucket with Terraform, 88
- SaaS (Software as a Service), 86
- SAML (Security Assertion Markup Language), 37, 63-64
- sandboxed Pods, 120
- SAST (static application security testing), 25, 93, 168, 172
- SBOM (software bill of materials), CI/CD pipelines, 174, 178
- SCA (software composition analysis), 25, 93
- scaling of applications, with containers, 105
- SCAP (Security Content Automation Protocol), 9
- scheduler, Kubernetes, 108

- SCIM (System for Cross-Domain Identity Management), 66
- SCM (source control management), 164
- Scorecard tool, 169
- scripts, security automation development, 148
- SDL (secure development lifecycle), 75
- secret zero situation, 77
- SecretProviderClass, 112
- secrets management, 25, 69-83
 - access control principle, 74
 - automation principle, 76
 - centralization principle, 77
 - CI/CD pipeline security, 161-162
 - code security best practices, 22-23
 - code to access secrets, 78-79
 - compliance principle, 75-76
 - encryption principle, 73
 - ESO for secret store creation, 81
 - hard-coded secrets, avoiding, 24
 - IaC tools for security automation, 150
 - Kubernetes, 109
 - lifecycle of secrets, 72
 - monitoring and auditing principle, 74
 - retrieving a secret, 82
 - secrets policy setup, 80-81
 - setting a test secret, 82
 - SPIFFE to eliminate secrets managers, 127
 - storage methods, 71
 - switching between CSPs issue, 102
 - testing principle, 76
 - types of secrets, 69
- secrets policy setup, 80-81
- secrets sprawl, 70, 72
- Secrets Store CSI Driver for Kubernetes, 110-112
- SecretStore, 81
- secure computing mode (seccomp), Linux, 117
- secure development lifecycle (SDL), 75
- secure development workflows, automating, 142
- Secure Production Identity Framework for Everyone (SPIFFE), 127-129
- Secure Software Development Framework (SSDF), 27
- Security Assertion Markup Language (SAML), 37, 63-64
- security attack types, 7-9
- security automation, 139-157
 - with AI, 139, 142, 153-154

- automation example, 154-157
- best practices in, 148-149
- business value of, 140-141
- developing tools securely, 153
- infrastructure as code in, 149-151
- PaC, 151-152
- tools for, 144
 - (see also automation tools and processes)
- types of, 141
- Security Content Automation Protocol (SCAP), 9
- security information and event management (SIEM), 56, 145, 147, 175
- security islands, 77
- security orchestration, automation, and response (SOAR), 145, 147
- security teams and developers, coordination of, ix
- Security-Enhanced Linux (SELinux), 117
- segregation of duties (SoD), 24, 53
- self-healing with containers, 105
- self-service password management, 48
- self-signed root certificate, 130
- SELinux (Security-Enhanced Linux), 117
- sensor modules, IAST, 174
- separation of concerns, network policies, 126
- server-side validation, 15
- service accounts
 - CSP management of, 102
 - Kubernetes, 112
- service meshes for cluster security, 132-136
- service providers (SPs), 63-64
- session management, code security, 17
- setting a test secret, 82
- shared responsibility model, CSP, 87
- sidecar proxy model, service mesh, 132, 135
- SIEM (security information and event management), 56, 145, 147, 175
- signed security tokens, 31
- sigstore, 179
- single sign-on (SSO), 32, 49, 64, 66
- slim images, 120
- SLSA (Supply-chain Levels for Software Artifacts), 166, 176
- smart contracts, with blockchain, 67
- SOAR (security orchestration, automation, and response), 145, 147
- social engineering attack type, 7, 9
- SoD (segregation of duties), 24, 53
- soft token versus hard token, MFA, 31
- Software as a Service (SaaS), 86
- software bill of materials (SBOM), CI/CD pipelines, 174, 178
- software composition analysis (SCA), 25, 93
- software supply chains, 4, 10, 24, 159
 - (see also CI/CD pipeline security)
- SolarWinds breach, 10, 170
- source code repositories, avoiding sensitive file storage in, 79
- source code threats, CI/CD pipeline security, 167-170
- source control management (SCM), 164
- SPDX (System Package Data Exchange), 178
- SPIFFE (Secure Production Identity Framework for Everyone), 127-129
- SPIFFE Verifiable Identity Documents (SVIDs), 131
- SPIRE, 129
- SPs (service providers), 63-64
- SQL injection attack type, 7, 17
- SSDF (Secure Software Development Framework), 27
- SSO (single sign-on), 32, 49, 64, 66
- staging environments, testing in, 24
- standards
 - AES-256 encryption, 16, 73
 - code security best practices, 26-27
 - compliance with (see auditing and compliance)
 - in IAM, 63-66
 - NIST, 8, 27, 175
 - OASIS, 45
 - PCI DSS, 27, 75, 145
 - PSS, 118-120
 - RFC 7519, 34, 36
 - secrets management, 75-76
 - SPIFFE, 127-129
- stateless AuthN, 35
- static application security testing (SAST), 25, 93, 168, 172
- static code analysis tools, 147
- static text, avoiding hardcoding of secrets as, 79
- storage methods, secrets, 71
- strict mTLS, 136
- supply chain attacks, 6
- supply chain software, 4, 10, 24, 159
 - (see also CI/CD pipeline security)

Supply-chain Levels for Software Artifacts (SLSA), 166, 176
SVIDs (SPIFFE Verifiable Identity Documents), 131
Syft, 178
system configuration risk, CI/CD pipeline, 165
System for Cross-Domain Identity Management (SCIM), 66
System Package Data Exchange (SPDX), 178

T

TEA (time, entitlements, and approvals) approach, 97
tenant hopping, 88
Terraform, 89, 168
testing
 CI/CD pipeline, 168, 172-175
 code security, 10
 DAST, 25, 93, 173
 IaC scripts, 151
 IAST, 174
 isolation of test environments, 24
 network policies, 126
 SAST, 25, 93, 168, 172
 secrets management, 76, 82
 in staging environments, 24
thick clients, and LDAP, 66
third-party services risk, CI/CD pipelines, 164
threat detection and response, AI in identity security, 67
threat intelligence tools, 148
threat investigation, automating, 143
time, entitlements, and approvals (TEA) approach, 97
TLS (transport layer security), 10, 16, 73
token-based AuthN, 31, 34-36
Toyota cloud environment breach, 88, 90
traffic management, service mesh, 136
transitive dependencies, 169

transport layer security (TLS), 10, 16, 73
Trend Micro API servers breach, 108
trust manager, TLS, 131
two-factor authentication (2FA), 19, 31
typosquatting, 164

U

Uber breach, 1
user interactions, simplifying with AI, 154
user provisioning, 48, 65
user-based access control (UBAC), 41
username and password (AuthN), 19, 30

V

vaults, secrets management, 78
version control, IaC code, 151
version tags, container images, 122
view Role, 114
vulnerability scanning, containers, 123, 147

W

WebAuthN, 32
workflow and approval processes, 58
workload-to-workload AuthN and AuthZ, 127-128

X

XACML (Extensive Access Control Markup Language), 45-46
XDR (extended detection response), 145

Z

Zed Attack Proxy (ZAP), 173
zero standing privileges (ZSP), 96
zero trust architecture (ZTA), 14
zero trust model, 2-4, 13-14, 148

About the Authors

John Walsh is a distinguished authority in software security, with over 15 years of experience in various roles within enterprise security companies. His extensive professional history as a software developer, product manager, product marketing manager, and open source community manager has equipped him with a comprehensive understanding of how to secure the entire software supply chain. This unique perspective enables him to communicate the intricacies of enterprise software security with clarity and depth while catering to both technical and nontechnical audiences.

Uzi Ailon is a seasoned software engineering professional with over 25 years of experience, including 15 years focused on customer engagement and sales. With his deep expertise in the entire software development lifecycle, security, and DevOps, Uzi understands the unique challenges enterprises face in balancing innovation with security.

As a leader in CyberArk's machine identities solutions, Uzi is committed to helping organizations strengthen their security posture while fostering seamless collaboration between DevOps and security teams. By leveraging automation to reduce security risks and enhance operational efficiency, Uzi enables businesses to protect their most critical assets without compromising agility. With a strong grasp of the business objectives of Fortune 500 companies, Uzi bridges the gap between technical excellence and strategic goals, ensuring that security solutions align with broader enterprise needs. His customer-centric approach and extensive industry knowledge make him a trusted advisor in driving secure, scalable, and efficient digital transformations.

Over the years, Uzi has worked with hundreds of customers, helping to design their machine identity security solutions and define best practices that enhance security, compliance, and operational effectiveness.

Colophon

The animal on the cover of *Identity Security for Software Development* is an African crested or maned rat (*Lophiomys imhausi*). It resembles a porcupine or skunk with its black and white coloring and bushy mane, which stands on end when the rat feels threatened. These visual warnings are not empty threats; the maned rat's fur stores plant toxins that can poison its predators. It is the only rodent known to use poison as a defense mechanism.

The maned rat is a nocturnal creature of African forest, savanna, and shrubland. It feeds primarily on plants and can grow up to 21 inches long (from head to tail).

Maned rats have a stable population in East Africa and an ICUN conservation status of Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Cassell's Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.