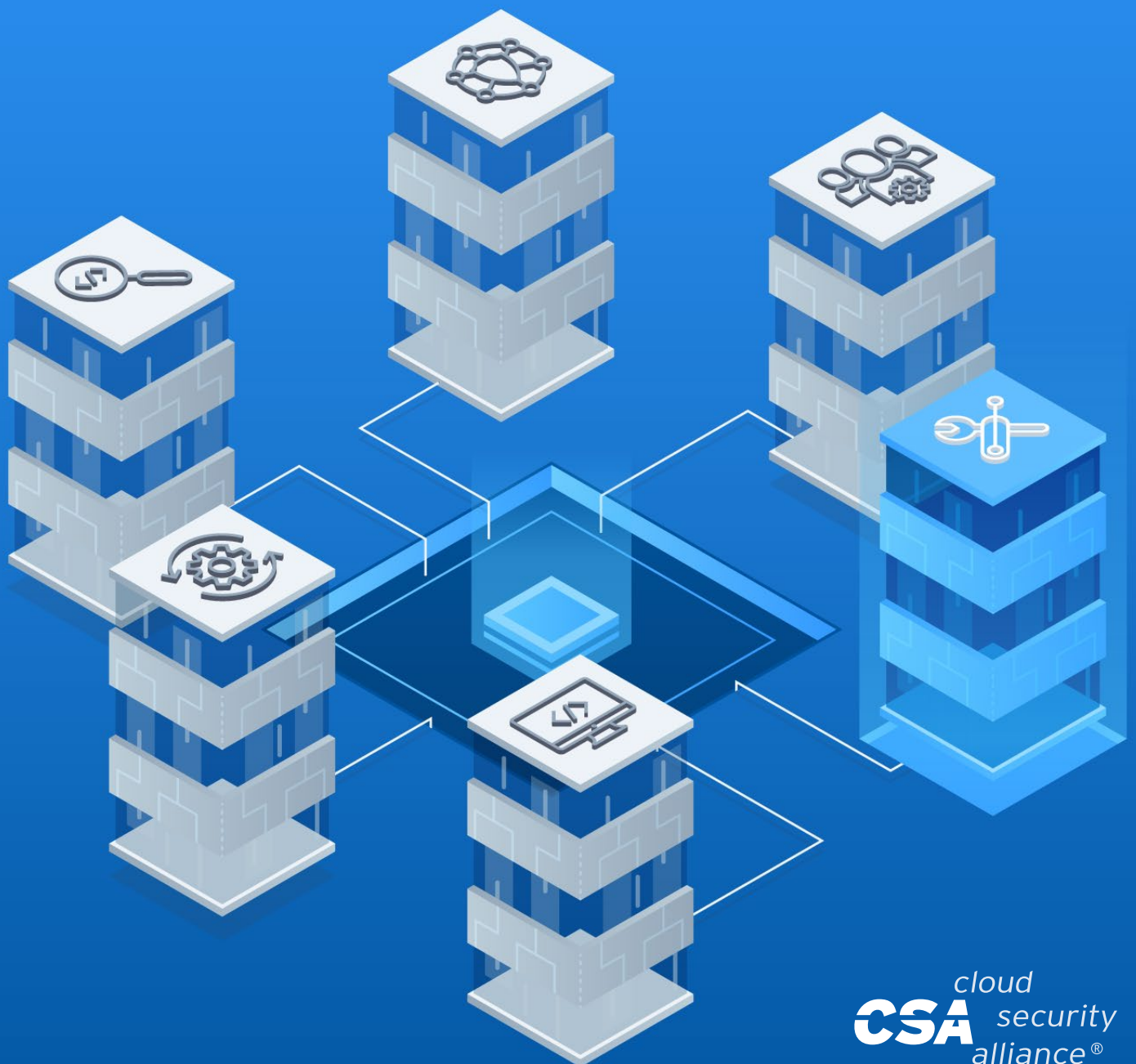


The Six Pillars of DevSecOps: **Pragmatic Implementation**



The permanent and official location for DevSecOps Working Group is
<https://cloudsecurityalliance.org/research/working-groups/devsecops/>

© 2022 Cloud Security Alliance – All Rights Reserved. You may download, store, display on your computer, view, print, and link to the Cloud Security Alliance at <https://cloudsecurityalliance.org> subject to the following: (a) the draft may be used solely for your personal, informational, non-commercial use; (b) the draft may not be modified or altered in any way; (c) the draft may not be redistributed; and (d) the trademark, copyright or other notices may not be removed. You may quote portions of the draft as permitted by the Fair Use provisions of the United States Copyright Act, provided that you attribute the portions to the Cloud Security Alliance.

Acknowledgments

Lead Author

Roupe Sahans

Contributors

Charles Bideau
Aristide Bouix
Mauricio Cano
Eric Gauthier
Daniel Gora
Michael Holden
Brynna Nery
Abdul Rahman Sattar
Michael Roza
Sreeni Sharma
Damian Zawodnik

Reviewers

Chris Latham
Fabiola Moyón
Douglas Needham

Co-Chairs

Kapil Bareja
Chris Kirschke
Sam Sehgal

CSA Analyst

Josh Buker

CSA Global Staff

Claire Lehnert
Stephen Lumpe

Table of Contents

Acknowledgments	3
Table of Contents.....	4
Foreword.....	6
Introduction	7
Goals	7
Audience.....	8
Overview	9
Using DevOps to Introduce Security	9
Security Transformation on Software Lifecycles.....	11
People are Key Enablers in Successful DevSecOps Transformation	12
Culture Will Accelerate Velocity and Improve Performance	16
Technologies and Processes Underpin the Pragmatic Implementation of DevSecOps	20
DevSecOps Stages.....	20
Design and Architecture	20
Coding	21
Integration and Testing	21
Delivery and Deployment	21
Runtime Defense and Monitoring	21
Technology and Process: Detailed Activities	27
Design and Architecture	27
Threat Modeling	27
Security User Stories	29
Value Stream Security Mapping	29
Architecture Principles and Artifacts.....	31
Risk Management (Shifted-Left)	34
Coding	36
Developer Training.....	36
Security Hooks	38
Code Linting	39
Software Composition Analysis	40
Static Application Security Testing.....	41
Container Hardening	44
Infrastructure as Code Analysis	47

Security Unit Testing.....	49
Peer Reviews	51
Integration and Test	52
Dynamic Application Security Testing	52
Interactive Application Security Testing	53
API Testing.....	55
Fuzz Testing.....	56
Penetration Testing.....	57
Container Testing	63
Integrity Checking	64
Delivery and Deployment	65
GitOps	65
Guardrails	66
Environment Separation	70
Secrets & Key Management	74
Securing the CI/CD Pipeline	76
System Hardening	79
Runtime	81
Chaos Engineering	81
Cloud Security Posture Management	82
Monitoring and Observability	85
Attack Surface Management	89
Postmortems.....	91
Purple Teaming	92
Vulnerability Management (Post-identification)	93
Incident Response	96
References	99
Acronyms.....	102

Foreword

The Cloud Security Alliance (CSA) and SAFECode are deeply committed to improving software security outcomes. The paper Six Pillars of DevSecOps, published in August 2019, provides a high-level set of methods and successfully implemented solutions its authors use to build software at speed and with minimal security-related bugs. Those six pillars are:

Pillar 1: Collective Responsibility (Published 02/20/2020)

Pillar 2: Collaboration and Integration (Estimated 04/15/2023)

Pillar 3: Pragmatic Implementation (Published 12/14/2022)

Pillar 4: Bridging Compliance & Development (Published 02/08/2022)

Pillar 5: Automation (Published 07/06/2020)

Pillar 6: Measure, Monitor, Report and Action (Estimated 04/15/2023)

The successful solutions that underpin each of these pillars are the subjects of a much more detailed set of joint publications by the Cloud Security Alliance and SAFECode. This paper is the fourth of those follow-on publications.

Introduction

Organizations have a wide array of tools and solutions to choose from when implementing security into their Software Development Lifecycle (SDLC). Organizations often procure tools and solutions that are either hard to deploy, challenging to operationalize and scale, or do not provide actionable insights that can help mitigate the actual security risks. Since every SDLC is different in terms of structure, processes, tooling, and overall maturity, there is no one-size-fits-all binary blueprint to implement DevSecOps.

Using a framework-agnostic DevSecOps model-focused on application development and platform security to ensure safety, privacy, and trust in the digital society-organizations will be able to approach security in DevOps pragmatically. This model will fulfill the unmet need of connecting all the stakeholders (development, operations, and security) to embed security into the software lifecycle.¹

The DevSecOps implementation guidance in this paper is organized into a menu of practical responsibilities and activities to enable digital security leaders to make pragmatic decisions when embarking on DevSecOps.

DevSecOps adoption is typically organically established either by software development and platform engineering teams, or centrally by leadership. Regardless of the drivers and stakeholders, organizations should view their DevSecOps implementation as an iterative continuous improvement effort rather than a one-time waterfall project.

The scope of this paper identifies and expands on the 4 key elements of a successful DevSecOps initiative-culture, people, process, and technology. This paper touches on privacy, but it doesn't offer a complete view on this domain (i.e. privacy by design).

Goals

The Cloud Security Alliance DevSecOps Working Group (WG) issued high-level guidance in "Information Security Management through Reflexive Security: Six Pillars in the Integration of Security, Development and Operations."² The six pillars are considered critical focus areas for implementing DevSecOps, with one of the recommended pillars on pragmatic implementation of a security program. *Pragmatic* is defined as making calculated decisions sensibly and realistically, based on practical rather than theoretical considerations.³

For digital and security leaders reading this paper, pragmatism in DevSecOps will be driven by implementing approaches that yield the highest return of investment. The guidance provided will help organizations implement DevSecOps with success that embeds security across the SDLC and existing workflows of DevOps.

1 <https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/>

2 <https://cloudsecurityalliance.org/artifacts/information-security-management-through-reflexive-security/>

3 <https://www.lexico.com/en/definition/pragmatic>

Audience

The target audience of this document includes those involved in the management and operational functions of risk, information security, and information technology. This audience consists of the CISO, CIO, CTO, and those leading digital transformation initiatives.

Additionally, this guidance can also be used by application, platform, and security engineers and architects as a reference to improve an organization's DevSecOps program.

Overview

The pragmatic implementation guidance within this paper focuses on security covering internally developed and third-party packaged software regardless of where it runs, whether on-premise or in the cloud.

There are typically two distinct viewpoints on how an organization addresses a DevSecOps program:

1. From the perspective of DevOps, where an organization builds and integrates security into the SDLC.
2. From the security team's perspective, where security controls are typically implemented, considering process and technology.

Our guidance in this paper fuses these two common viewpoints, providing complete coverage of security activities. Organizations should approach DevSecOps as an iterative and continuous improvement effort rather than a one-time project. The intent is for readers to think about where within their organization they want to focus their improvement efforts and then reference this paper and its guidelines for activities to prioritize during each software lifecycle stage.

Using DevOps to Introduce Security

From the DevOps viewpoint, a team can address security earlier in the design and coding stage, otherwise known as "shifting left." This aims to integrate security as early as possible with non-security stakeholders in an organization. Security processes are established as default, so that doing something insecurely requires extra effort and includes exception workflows. Security from the DevOps viewpoint is represented in **Figure 1**. Organizations seeking to introduce security through the DevOps viewpoint typically include the following steps:

1. Review the development and IT organization's current software lifecycle. This is how new technology is evaluated, built, tested, deployed, and operated. There are usually ways in which the SDLC handles change (i.e., primarily feature requests) that security teams can leverage.
2. Review DevOps and IT organizations' existing tooling, especially tools that automate deployment and development tasks.

Understanding the development workflow and existing toolset enables decision-making for security controls across each stage of the software lifecycle. This paper explores in detail how both points above can be achieved, coupled with automation approaches detailed in *Pillar 5 Automation*.⁴

⁴ <https://cloudsecurityalliance.org/artifacts/devsecops-automation/>

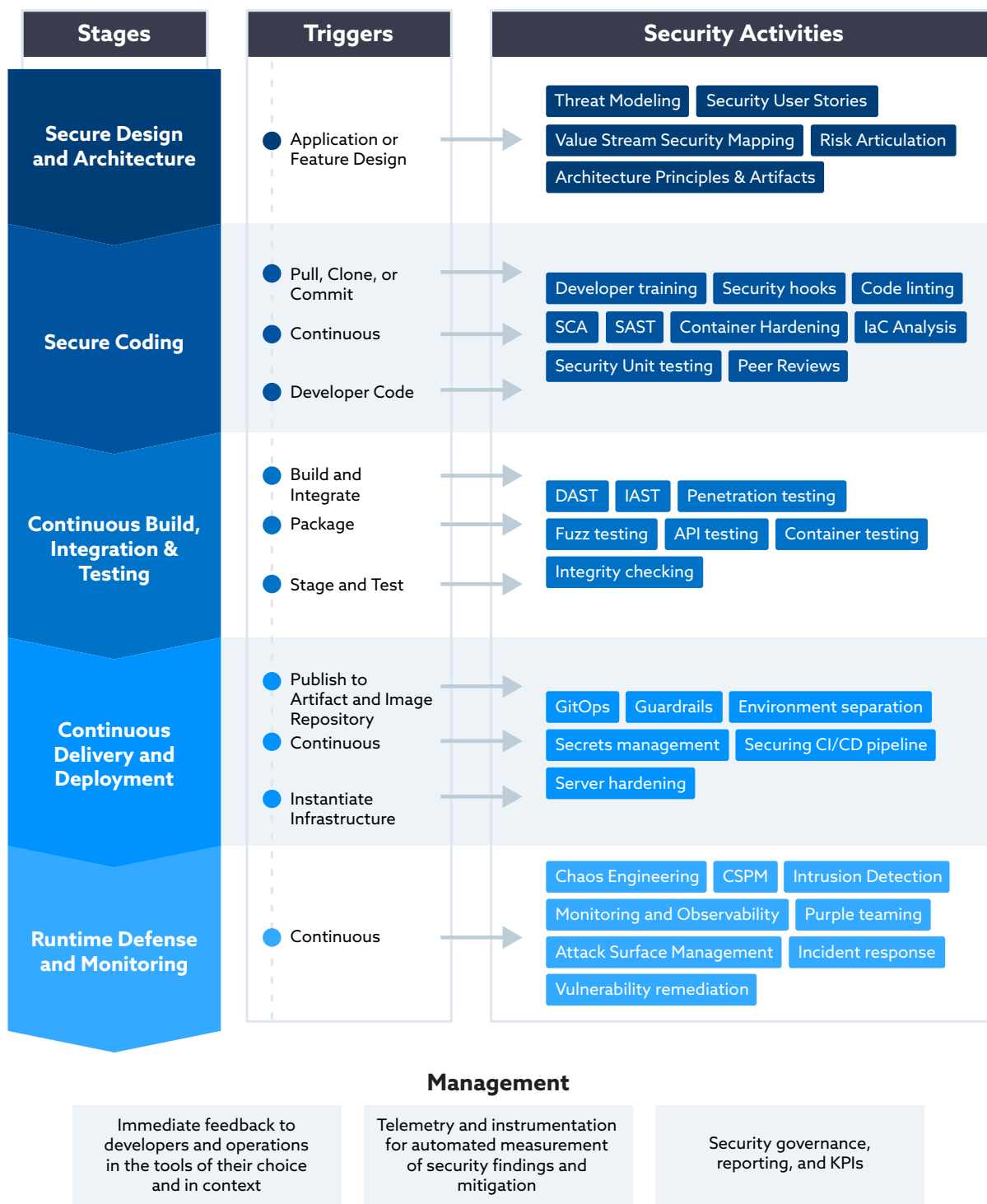


Figure 1: Secure Development Lifecycle

Whilst there are many different ways to define a Software Development Lifecycle (SDLC) and its stages, the five-staged SDLC referenced in **Figure 1**: Design, Coding, Integration & Test, Deployment, and Monitor offers a universal view for software development.

Security Transformation on Software Lifecycles

From the Security viewpoint, organizations seeking security improvements look to robust security programs that rely upon effective integration of security best practices into the organization's software development paradigms. This is typically led by culture and processes, operation of security infrastructure and technology, and workforce awareness. Security at every software lifecycle stage will require some combination of people, culture, process, and technology:

- **People:** The individuals and their roles within your organization and the delivery of secure and successful applications, platforms, and infrastructure. There is a focus on leadership, technical stakeholders, implementers, and decision-makers.
- **Culture:** How an organization approaches and manages work and risk. Often the softer and difficult-to-measure area, culture is perception of software development work and security, which plays a direct role in performance and success.
- **Process:** The common workflows, manual actions, and well-documented activities. People execute processes with the support of automation or some combination of the two.
- **Technology:** Technology focuses on securely creating and operating infrastructure and applications using tooling and automation to help protect assets and detect vulnerabilities, events and incidents.

The combination between these will vary depending upon the size and sophistication of an organization's security program. Smaller organizations typically rely primarily on people filling multiple roles and managing application security, while larger organizations may leverage technology and automation. An organization's culture may be influenced by compliance requirements that lead to significant oversight and audit, while others embed checks and balances into standard workflows.

People are Key Enablers in Successful DevSecOps Transformation



The most important facet of DevSecOps are the people. Without the understanding of utilizing skills and knowledge of your existing workforce and building successful cross-functional teams, your DevSecOps efforts will become expensive and inefficient. Cross-functional teams and roles are typically an organization's biggest enabler in DevSecOps. Cross-functional teams are mixed groups of people with complementing capabilities, mitigating the risk of operating in silos and avoiding the overhead of heavy, multi-staged approvals. With people being a key dependency, we've identified in **Table 1** the different role profiles to consider in DevOps and the pragmatic application of security. We map each role to where each would typically fit within the DevSecOps stage. An example of how it is achievable can be seen in **Figure 2**, with DevOps and Security roles articulated in each DevSecOps stage aligned with the security activities recommended in this paper.

Stages	People		Security Activities
	Software Development using DevOps	Security Embedded	
Secure Design and Architecture	<ul style="list-style-type: none"> Solution Architect 	<ul style="list-style-type: none"> Security Architect Security Navigator Security Champion 	<ul style="list-style-type: none"> Threat Modeling Security User Stories Value Stream Security Mapping Risk Articulation Architecture Principles & Artifacts
Secure Coding	<ul style="list-style-type: none"> Cloud Developer Operations Engineer Software Developer 	<ul style="list-style-type: none"> Application Security Engineer Security Champion Security Operations Engineer 	<ul style="list-style-type: none"> Developer training Security hooks Code linting SCA SAST Container Hardening laC Analysis Security Unit testing Peer Reviews
Continuous Build, Integration & Testing	<ul style="list-style-type: none"> Performance Tester Integration Engineer Cloud Developer Operations Engineer 	<ul style="list-style-type: none"> Security Tester Security Champion 	<ul style="list-style-type: none"> DAST IAST Penetration testing Fuzz testing API testing Container testing Integrity checking
Continuous Delivery and Deployment	<ul style="list-style-type: none"> Solution Architect Integration Engineer Cloud Developer Operations Engineer 	<ul style="list-style-type: none"> Solution Architect Security Champion Security Operations Engineer 	<ul style="list-style-type: none"> GitOps Guardrails Environment separation Secrets management Securing CI/CD pipeline Server hardening
Runtime Defense and Monitoring	<ul style="list-style-type: none"> Solution Architect Cloud Developer Operations Engineer 	<ul style="list-style-type: none"> Solution Architect Security Navigator Security Operations Engineer 	<ul style="list-style-type: none"> Chaos Engineering CSPM Intrusion Detection Monitoring and Observability Purple teaming Attack Surface Management Incident response Vulnerability remediation

Figure 2: DevSecOps Structure Example

Role Definitions

Security Architect	Consults with the business on how to design applications and consume infrastructure services (including cloud) to serve business goals in a secure manner. Develops methods, frameworks, and patterns and helps developers operate secure development and best practices. Identifies future-state security approaches to evolve existing practices, respond to industry, regulatory and technology changes, and identify DevSecOps strategic and operational fit (e.g., what controls to shift left/right). Works with developers to senior business stakeholders to advise and assess security methods, approaches, and use of automated tools across deployment pipelines and repositories.
Security Navigator	Leads strategies to communicate key initiatives and successes, and assesses the impacts of security tooling and method adoption across the organization, individual teams, and lines of business. Aligns security initiatives with business value and strategic goals. Connects high-level digital strategy for security and business goals with low-level developer objectives. Assists with management and remediation of security risks, vulnerabilities, and threats to applications.
Security Champion	Helps product teams and developers adopt security initiatives consistent with the organization and is typically already developing or managing software or infrastructure. Continuously reviews the security posture within his/her/their product team against threats, vulnerabilities, and risks. It can be shared as a hybrid role.
Application Security Engineer	Identifies potential flaws in application code and mitigates vulnerabilities by fixing security issues. Applies secure coding and testing standards and documents secure code guidelines. Integrates security testing tools like static and dynamic application testing tools into source code repositories and continuous integration and delivery pipelines. Applies security controls (like encryption, identity, and authentication) on the application code to reduce the attack surface and minimize the risk of exploitation.
Security Tester	Addresses security concerns in the software testing phase, including risk acceptance, security acceptance criteria, and methods of security testing. Performs regular manual penetration tests to expose application and infrastructure weaknesses and confirms correct implementation of security enforcing features. Makes remediation recommendations based on testing outputs.
Security Operations Engineer	Delivers recommendations for the security posture of cloud, infrastructure, and platform services. Applies system hardening practices on infrastructure and platforms and ensures strong security mechanisms. Conducts vulnerability scanning and remediation to reduce the attack surface on infrastructure.

Cloud Developer Operations Engineer	Provisions and operates cloud infrastructure, including servers, storage, and networks. Automates the provisioning of cloud infrastructure and applications using Continuous Integration and Deployment methods.
Solution Architect	Develops solution architecture and relevant system components to meet user requirements. Designs systems and interfaces between systems and determines the impact of the target environment, including but not limited to security posture.
Software Developer	Engages with architects in identifying, designing, and writing code as software features. They will help test, maintain and update the product to ensure that all security, performance and functional problems are resolved. Security roles and outcomes enable the developer to code securely according to software engineering methodologies, tools, and practices and secure code guidelines.
Performance Tester	Responsible for writing and executing test plans for load and stress testing infrastructure, platforms, and applications.
Integration Engineer	Helps to ensure security activities are applied to integrated software packages and integrated application features. Introduces holistic applications security controls and processes that impact the entire application – utilizes tools for integrity checks and scanning applications in runtime.

Table 1: DevSecOps Roles

Modern cloud-native architectures are more distributed - smaller, security-capable, distributed teams can better tackle the end-to-end security design as one value stream. Although there is no silver bullet for a DevSecOps team topology and structure that will suit every organization, some characteristics influence efficiency. An organization with strong technical leadership and technical management will understand the difference and balance development, operations, and security with the appreciation that DevSecOps activities are integrated into each phase of DevOps. When designing your team structure, it's important to recognize and avoid some of the common mistakes (anti-patterns), such as:

- **Organizational silos:** Stakeholders belong to their teams and don't have an understanding of each other's work, shared responsibility between teams are non-existent. Security operates in silo and are involved just before deployment - as a single security gate - fostering an unproductive and inefficient software development practice. This can start with leadership deciding that they want DevSecOps and start building a team. This approach is cost inefficient and will only begin to serve a return of investment after significant investment and is common where Dev and Ops teams are separate and operate in a silo.
- **Devs doing DevSecOps:** Although typically done in good spirit and can yield initial benefits, it's difficult to scale consistently and lacks completeness in security activities. A common pitfall developers may encounter is assuming Security Teams only perform compliance and governance activities. It can be easy to underestimate the complexity and knowledge

required to securely architect a system, as the skills required do not necessarily overlap with feature development.

- **Rebranding the Security team:** Starts with simply hiring DevSecOps staff into your Security team to improve practices and reduce costs, yet they fail to see the core drivers of DevSecOps to the business. With the industry hype with DevSecOps now evident, teams may want to adopt DevSecOps without understanding how security can increase operational efficiency and in some cases increases release cadence and reduce security effort.

Understanding the different stages of DevSecOps (Design, Coding, Integration & Test, Deployment, and Monitor) against your existing DevOps and software development paradigms will help you avoid team topology anti-patterns.

Culture Will Accelerate Velocity and Improve Performance



Changing the mindset of product development underpins culture and requires processes to ensure the right cultural characteristics are easily adopted. As we break down the characteristics of DevSecOps success, the shared responsibility model must be established between each discipline. Development, security, and operations roles, should work harmoniously to deliver fast, secure, and stable software objectives.

The business proverb *"Culture eats strategy for breakfast"*⁵ has relevance when introducing DevSecOps to software development paradigms and digital transformation. The most innovative and ubiquitous software products can often develop new features at a speed that isn't solely reliant on large investments and well-defined strategies. Research from the DevOps Research and Assessment (DORA) team⁶ shows that an organizational culture that values learning contributes to software delivery performance with:

- Increased deployment frequency
- Reduced lead time for changes, time to restore service, and change failure rate
- Strong team culture

An organizational culture that is high-trust and emphasizes information flow is predictive of software delivery performance and organizational performance in technology. DORA research shows that changing the way people work changes culture⁶. Teams can identify helpful practices to create a generative culture that fosters information flow and trust by examining the six aspects of Westrum's model⁷ of organizational culture, focusing on those behaviors seen in the generative culture:

- High cooperation
- Messengers are trained
- Risks are shared
- Bridging is encouraged
- Failure leads to inquiry
- Novelty is implemented

Culture references the principles and foundational capabilities which support DevSecOps activities. These key practices drive a culture of continuous improvement and early involvement of security as embedded members of the cross-functional teams. Culture can be an arbitrary concept that makes it difficult for teams to quantify and measure; we've recognized the cultural characteristics

5 <https://www.forbes.com/sites/forbescoachescouncil/2018/11/20/why-does-culture-eat-strategy-for-breakfast/>

6 <https://cloud.google.com/architecture/devops/devops-culture-learning-culture>

7 <https://cloud.google.com/architecture/devops/devops-culture-westrum-organizational-culture>

of successful DevSecOps teams that can apply security at speed to meet business demand. **Figure 3** outlines the cultural characteristics and heatmaps these with the level of applicability each has in applying the respective characteristic to the CSA DevSecOps software lifecycle in **Figure 1**.

Culture Characteristic	Design	Coding	Integration & Test	Delivery & Deployment	Runtime
Accelerate Change	●	●	●	●	●
Create Guidelines	●	●	●	●	●
Maintain communication/ transparency	●	●	●	●	●
Security failure readiness	●	●	●	●	●
Security baselines	●	●	●	●	●
Continuous security learning	●	●	●	●	●
Collective security responsibility	●	●	●	●	●
Innovation	●	●	●	●	●
Measure cost of work	●	●	●	●	●
Continuous improvement	●	●	●	●	●
Start small/think small	●	●	●	●	●
Incentivization	●	●	●	●	●

● High Applicability
● Medium Applicability
● Low Applicability

Figure 3: DevSecOps Culture Applicability Matrix

As we think about the measurable cultural features within DevSecOps, we also need to dissect where each is typically applied across each stage. There are only three cultural characteristics applicable in every DevSecOps stage. Those are security baselines and continuous improvement referenced in **Table 2** and **Figure 3**; which emphasize the importance and prioritization of:

- Security Baselines - Understanding the minimum viable secure product (security baseline) at each stage
- Continuous Improvement - Always seeking improvement at each stage whether those are operational or strategic
- Incentivization - Building rewards to objectives and goals to incentivise high performance.

The relevance of mapping characteristics to DevSecOps stages helps identify the applicability of security controls and whether they can be counterproductive to software development. For example, the 'Innovation' characteristic only holds high applicability to the Coding stage. Thus overly restrictive security controls in the Coding stage can stifle innovative work. However, this should be balanced with the security-relevant cultural characteristics in Coding: security baselines, continuous security learning, and collective security responsibility.

Table 2 identifies the characteristics of an optimized and productive culture to achieve a high return on investment from DevSecOps. Each characteristic is mapped by applicability to the stage of DevSecOps; Design, Coding, Integration & Test, Deployment, and Monitor.

Definitions	
Accelerate change	Continuous change and improvement are encouraged and accelerated. Change activities are communicated and understood, whilst security also looks to compliment the acceleration of change.
Create guidelines	Use of security design processes and technologies are well documented for product stakeholders to understand security design work and remove reliance on individual employee knowledge. Guidelines are continuously reviewed to improve efficiency.
Maintain communication/ transparency	Communicate key initiatives and successes. Align security initiatives with business value and strategic goals. Connect high-level digital strategy for security and business goals with low-level developer objectives. Constructive code reviews are the norm. Security is valued and included in these reviews
Security failure readiness	Security measures are assumed to fail or be compromised. Failure is expected, and response plans are widely in-place, discussed, exercised and understood. Where failures do occur these are reviewed to determine further improvements.
Security baselines	A baseline for security appetite and a minimum viable, secure product is well understood. Unacceptable outcomes are well-understood. Secure and reusable components are encouraged and packaged as templates.
Continuous security learning	Information sharing about security is continuous and universal across the organization. Good security practice is shared.
Collective Security Responsibility	Everyone shares a collective responsibility for developing secure products and features. Security is not someone else's problem.
Innovation	Work is well balanced between day-to-day operations and future improvements to products. Time is allocated for innovative work external to project goals. Environments are designed to enable innovation.
Measure cost of work	Shifting security left doesn't overburden developers to impact speed and innovation adversely. Security work and effort is measured.

Continuous improvement	The ongoing improvement of products, services, or processes through incremental or breakthrough improvements.
Start/ think small	Start as small as possible – while still achieving the other characteristics. The difference between short-term wins/sprints and longer-term builds and activities is understood. Minimum viable, secure products are well understood where security is progressively developed with product features.
Incentivization	Is the systematic building of rewards to motivate the participants to achieve objectives. Security is incentivised where the reduction such that the vulnerabilities is rewarded.

Table 2: DevSecOps cultural characteristics

Technologies and Processes Underpin the Pragmatic Implementation of DevSecOps



The activities that yield the highest return of investment in DevSecOps are supported by the effective use of tooling/technology and process. As we break down the DevOps paradigm, we can identify where security activities (technology and process) fit. The five stages of the SDLC referenced in **Figure 1** (Design, Coding, Build, Deployment, and Monitoring) each have security risks and activities that are expanded upon within the matrices **Figures 4 - 8**.

The security activities within this section do not reflect the security resources included in a solution design. Much like how a web application firewall (WAF), antivirus (AV) or an intrusion detection system (IDS) will better secure an application/infrastructure, it's exclusive to the design of a product, which is why we recognize this as a security control and not a security activity supported by tooling or processes.

DevSecOps Stages

Design and Architecture

The Secure Design and Architecture stage precedes any code being written, but can be the most critical step in the SDLC when implementing security. Insecure implementation details at this stage can amplify future security risks in the long run. Conversely, identifying design weaknesses and product risks at this stage can save many hours of work and entirely prevent vulnerabilities from being introduced into the codebase before it's even written. Risks that can be addressed in the design stage include:

- Failing to identify and secure data flows
- Vulnerabilities inherent to a choice in framework and programming language
- Common risks addressed by security control frameworks
- Poor design decisions identified from threat modeling
- Vulnerabilities tied to an incomplete or flawed business logic

Coding

Secure coding is where applications are developed based upon the requirements detailed in the Design stage. Here, developers choose frameworks and libraries, write code, and create unit tests. The application is expected to undergo significant change as design choices are made and code is rewritten. This stage focuses on securely implementing the design and business logic and using secure coding practices before code commits to a source code repository (SCR). Risks at this stage that can introduce security vulnerabilities include:

- Use of insecure protocols, frameworks, or libraries
- Vulnerable source code and dependencies
- Poorly coded infrastructure with vulnerability in hosting environments
- Lack of transparency of changes on critical code

Integration and Testing

Integration and Testing focuses on assembling the components of an application into releasable artifacts and testing the new releases to ensure that it meets the design requirements. This should include explicit testing for security requirements not only of the running application but also for the assembly and release process. This stage should focus on testing the security of the running application as it will exist in production. Risks at this stage include:

- Lack of integrity in the build process
- Runtime environment or source code security misconfiguration
- Exploits in the application or infrastructure during runtime
- Improper testing or lack of testing security requirements

Delivery and Deployment

Delivery and Deployment focuses on taking a pre-build release artifact and releasing it into production. This requires ensuring that security checks have been completed and deployed with the correct new artifact and environment configuration. This stage should focus on controlling production changes, logging change activities, and the overall integrity of a production release. Risks at this stage include:

- Release of incorrect or corrupted artifacts
- Introducing insecure environment configurations into production
- Hijacking the release process to introduce unauthorized changes

Runtime Defense and Monitoring

Runtime Defense and Monitoring focuses on managing the security of applications after they are deployed into production. Managing includes monitoring for indicators of compromise or other malicious and abusive behavior as well as continual security testing to detect either security regressions or newly discovered vulnerabilities. Risks at this stage include:

- Business logic attacks
- Denial of service or other availability attacks
- Newly discovered exploits
- Compliance monitoring

Design and Architecture

Overview: The design section references the technologies and processes that can be applied during the design of a product. We recognize design as continuous, so new product features and changes route through design activities. Without including security in the design phase, security measures are introduced with a higher operating impact and cost at deployment or runtime. As a result, measures are difficult to scale, resulting in security bottlenecks that slow development speed and impact release timelines.











Capability			
Activity	Description	Technology	Process
Threat Modeling	A structured process to identify security requirements, pinpoint security threats and potential vulnerabilities at an abstract level, quantify threats and prioritize remediation design methods.		
Security User Stories	A statement of needed security functionality that ensures one of many different security properties of software is satisfied. Raising security requirements in the same fashion as features are raised as stories. Security stories can be attributed to developer tasks and are prioritized in a sprint backlog.		
Value Stream Security Mapping	Mapping the activities in a process map from requirements gathering to release to identify process inefficiencies and opportunities to automate.		
Architecture Principles and Artifacts	The underlying general principles and goals for the use and development of all organization technology resources (incl. IT, IoT, and OT). They reflect a level of consensus among the various elements of the enterprise and form the basis for making future decisions and changes to technology, process, and people.		
Risk Management (Shifted-Left)	The process for product teams to identify and effectively describe product risk (incl. application, component, platform, and dependent resources) and implement improvements to alleviate or prevent possible negative events.		
Benefits	Drawbacks	Output	
<ul style="list-style-type: none"> • Increased velocity of review • Easier to scale consistently • Reduce operating impact and reliance on people • Plan and measure security work/backlog • Understand risks and threats early 	<ul style="list-style-type: none"> • Requires security architecture expertise 	<ul style="list-style-type: none"> • Architecture design reviews that are threat-led • Security requirements and prioritization of backlog • Measurement of security work by effort/time • Documented principles, patterns and methods • Recorded design risks and unacceptable risk scenarios 	

Figure 4: Design and Architect Matrix



















Coding			
Overview: The capabilities that can be applied as applications are being developed. Coding security controls rely on automation as tools can better and more consistently identify weaknesses and vulnerabilities in code compared to manual human reviews. Without including security in the coding phase, there is a risk that vulnerabilities in source code will be unidentified and deployed into production environments. Security vulnerabilities and weaknesses will be more expensive to fix during later stages of the SDLC			
Capability			
Activity	Description	Technology	Process
Developer Testing	With the requirement for high-quality, secure code, role-based security training helps developers identify bugs while writing code in their respective languages without common vulnerabilities.		
Container Hardening	Addresses the base image and container orchestration weaknesses by updating packages and looking for insecure default configuration and known vulnerabilities. It enables new golden base images for safe use within pipelines.		
Security Unit Testing	Takes small testable parts of source code to determine the effectiveness of security control.		
Security Hooks	Scripts triggered as code is committed to a git repository (git commit actions) either as a pre-commit or post-commit activity. They automatically identify issues in code like sensitive information in the source code.		
Code Linting	A form of analysis of the integrated development environment to flag programming errors, bugs, and stylistic and construct errors.		
Software Composition Analysis	Helps identify and remediate vulnerabilities in open-source and third-party components of a software product. SCA tools scan application code bases and artifact repositories to identify security vulnerabilities.		
Static Application Security Testing	Analyzes source code, byte code, and binaries to find security vulnerabilities in static code. It is also known as static code analysis or white box testing.		
Infrastructure as Code (IaC) Analysis	Enables engineers to version control, deploy, and improve cloud infrastructure while performing security verification checks. This presents an opportunity to proactively improve the build of cloud infrastructure and fix security issues early.		
Peer Review	A team member's source code review by one or more people before being implemented into the codebase.		
Benefits	Drawbacks	Output	
<ul style="list-style-type: none"> Identify security issues early and often Increased feedback on code quality Reduction in time to remediate vulnerabilities Safety checks to prevent risky code behaviors 	<ul style="list-style-type: none"> Code scanning covers only a subset of vulnerabilities and does not include runtime issues or design flaws issues May introduce false positives. 	<ul style="list-style-type: none"> Reporting on code via SAST and SCA tools Integrated capability in a git repository Security conscious developers Security activities embedded in development workflows 	

Figure 5: Coding Matrix











Integration and Testing			
Overview: The tools and processes to security test the functionality of an application/product. Without the tools and methods, security vulnerabilities and weaknesses will be the source of exploitation and result in data breaches and availability issues.			
Capability			
Activity	Description	Technology	Process
Dynamic Application Security Test	A form of black-box security testing that analyzes a running application to identify security vulnerabilities by simulating external attacks against the application's external interfaces.		
Interactive Application Security Testing	A method that combines the techniques of SAST and DAST to scan applications in runtime while also scanning individual lines of code.		
API Testing	A method of testing on application programming interfaces (APIs) as common messaging channels to ensure features cannot be exploited by weaknesses or vulnerabilities.		
Fuzz Testing	An automated and dynamic software testing technique that involves providing invalid, unexpected, or random data as inputs into an application or process to assess how it responds and reveals security defects.		
Penetration Testing	A method of manual security testing that targets weaknesses found in networks, systems, or application components to determine whether a vulnerability can be exploited to compromise the resources within the environment, the application as a whole, or its data.		
Container Testing	The testing of container base images and container orchestration methods against exploitable attacks.		
Integrity Checking	An automated process to cryptographically sign code artifacts and container images and verify their integrity before the Continuous Deployment stage.		
Benefits	Drawbacks	Output	
<ul style="list-style-type: none"> Catch security issues during runtime Identify and fix security issues during integration Automated process with a faster feedback loop to the developers 	<ul style="list-style-type: none"> High effort to establish the process, write integration packages with security testing tools 	<ul style="list-style-type: none"> Demonstrable assurance of security posture Reportable content for realistic exploits 	

Figure 6: Integration and Testing Matrix













Delivery and Deployment			
Overview: Pre-deployment safety checks to ensure that an application/product is deployed onto secure infrastructure. Without including security in deployment, there is a risk that vulnerabilities and poor security practices weaken an application/product, exploiting it for attacks in production environments.			
Capability			
Activity	Description	Technology	Process
GitOps	Using git repositories to manage infrastructure and application code deployments. In GitOps, the git repository is the source of truth for the deployment state rather than the server configuration files.		
Guardrails	A high-level rule that provides ongoing governance for building cloud environments. Guardrails are implemented preventive, or detective controls that help govern resources and monitor compliance across resources		
Environment Separation	The logical separation and management of development, testing, pre-production, and production environments.		
Secrets & Key Management	The tools and methods for managing digital authentication credentials (secrets) and cryptographic key material. This includes passwords, encryption keys, APIs, and tokens for use in applications, services, privileged accounts, and other sensitive credentials for authentication and cryptographic usage.		
Securing the CI/ CD Pipeline	Verifying the integrity of the CI/CD pipeline through logging and monitoring modifications and referencing the pipeline state in version control.		
System Hardening	A process that involves securing a server's data, ports, components, functions, and permissions.		
Benefits	Drawbacks	Output	
<ul style="list-style-type: none"> Fast and safe deployments Consistent configuration for hosting services Baseline establish for security controls Self-sufficient once established 	<ul style="list-style-type: none"> Requires re-architecture of hosting services for infrastructure and code 	<ul style="list-style-type: none"> Securely configured pipeline to deploy code Securely configured hosting environments 	

Figure 7: Delivery and Deployment Matrix

















Runtime Defense and Monitoring			
Overview: The capabilities and practices that can be applied after an application/product has been released into production. Runtime security enables continuous improvement by better identifying inefficiencies, vulnerabilities, and weaknesses and enabling incident response.			
Capability			
Activity	Description	Technology	Process
Chaos Engineering	Testing a distributed computing system to ensure it can withstand unexpected disruptions. They use failure mode and effects analysis or other tactics to get insight into potential points of failure in their organization's systems.		
Cloud Security Posture Management	The identification of misconfiguration issues and compliance risks in the cloud. An important purpose is the continuous monitoring of cloud infrastructure for security policy enforcement gaps.		
Performance Management	The process of collecting, analyzing, tagging log data to identify and respond to incidents and events, on applications and infrastructure, whilst also identifying metrics, the root cause, trends and performance.		
Attack Surface Management	External Attack Surface Management (EASM) identifies and manages the risks presented by internet-facing assets and systems. It refers to the tooling to discover external-facing assets.		
Postmortems	Used to identify the causes of a project failure (or significant business-impairing downtime) and how to prevent them in the future. They are intended to inform process improvements that mitigate future risks.		
Purple Teaming	Cybersecurity simulation is a way to test technologies and processes in response to simulated cyberattacks. This typically requires a replicated environment to effectively "war-game" against potential attacks in realistic scenarios.		
Vulnerability Remediation	The process of evaluating, treating, and reporting security weaknesses and misconfigurations found in systems and the software that runs on them.		
Incident Response	The process to prepare, analyze, and respond to incidents.		
Benefits	Drawbacks	Output	
<ul style="list-style-type: none"> Focus on incident response improvements Continuous measurement of security posture in production Opportunity to generate compliance metrics Data feedback loop into Design stage 	<ul style="list-style-type: none"> High effort to establish process 	<ul style="list-style-type: none"> Continuous reporting on infrastructure hosting environment Optimized preparedness for incident response Meaningful data and metrics to improve security and performance management 	

Figure 8: Runtime Defense and Monitoring Matrix

Technology and Process: Detailed Activities



Design and Architecture

The design section references the technologies and tools that can be applied during the design of a product. We recognize that design is continuous, so new product features and changes will route through design activities. Without including security in the design phase, security measures will be introduced with a higher operating impact and cost at deployment or runtime. As a result, measures will be difficult to scale, resulting in security bottlenecks that slow development speed and impact release timelines.

Threat Modeling

Threat modeling consists of techniques for identifying and outlining key threats, attack vectors, and preventive measures for a planned or existing system or application. The purpose of threat modeling is:

- Identifying, analyzing, and rating security threats
- Producing prioritized mitigations
- Assisting and informing attack surface analysis and risk reduction

The core threat modeling activities, as referenced in the [CSA Cloud Threat Modeling](#) paper, are:

1. Identify threat modeling security objectives for the threat modeling exercise, focusing on critical aspects such as confidentiality, integrity, availability, and privacy, e.g.,
 - a. Protect the company's databases containing customer or regulated information from external attackers;
 - b. Ensure high availability for the e-commerce web application, and
 - c. Select a cloud application model with the least attack surface or customer security responsibility.
2. Set the scope of the assessment concerning the systems and/or cloud infrastructure under consideration by providing an overview of the system or cloud application. This typically covers areas such as the various organizational assets, including the technology stack used, existing security controls, deployment scenario, type of users, and any specific security or regulatory requirement which needs to be addressed in the threat modeling.
3. System/application decomposition covers breaking down the system into subsystems and examining the interaction among the various components. The key activities done in this phase are:
 - a. Understand trust boundaries (external and internal facing, privileged, unauthenticated, etc.).

- b. Identify input and egress to the system (input and output) and data format.
 - c. Map the data flow in the system.
4. Identify and rate the potential threats, type of attacks, and how a malicious user can misuse the given system or its functionalities. The [OWASP Top Ten](#) survey results can be used to ensure the most critical security risks (per the survey) are considered in threat modeling activities. Some common threats are unauthorized access, denial of service, information disclosure, and so on. [STRIDE](#) can be used to classify a threat, while the severity of the threat can be assessed using a framework, such as [DREAD](#). The [MITRE ATT&CK®](#) framework can be used to help understand preparedness and the effectiveness of detection and mitigation actions.
5. Identify weaknesses and gaps in the system design and components to aid the security decisions and define the scope and nature of security testing.
6. Design and prioritize mitigations and controls applicable to the predetermined threats and reflect on how those controls would reduce the threat or risk level.
7. Communicate the identified threats, their potential impact and severity, and the applicable and proposed controls. Make the modeling data and insights available and call to the action of threat mitigation by design or effect.

Threat modeling provides organizations the benefit of performing design reviews early and often, as early as the solution design stage, which can result in productive design decisions (including technology procurement and decisions for including product features). The ability to thematically review a solution and identify mechanisms to control an attack technique helps security refine solutions and equally allows product team stakeholders the time to make changes during coding.

Threat modeling can be achieved with tooling that applies a digital record of threats on an architectural diagram and, in some cases, automates the identification of threat methods. Threat modeling is, however, most commonly performed as a manual review through analyzing a proposed design within the scope of the critical systems; based on knowledge of the scoped systems, they must model the data flow and trust boundaries between components in a data flow.

Whilst privacy by design elements are out of scope in this paper, the threat modeling activity is an effective method of baking privacy design reviews into a product. Depending on the type of product and geographical location and jurisdiction of an organization, regulatory requirements (e.g. GDPR) will mean organizations need to demonstrate compliance (e.g. data processing agreement, data privacy impact assessment) for the processing, handling and storage of personal and medical data.

Organizations that automate threat models typically select licensed vendors. Examples of those are [IRIUS Risk](#) and [ThreatModeler](#). [OWASP Threat Dragon](#) and [Cairis](#) are relevant examples for those seeking open source solutions.

Security User Stories

Security user stories are a statement of security functionality that ensures one of many different security properties of software is satisfied. Security requirements can be identified using tools or effective processes to define new features or additions to existing features to solve a specific security issue or remediate vulnerabilities.

Establishing security user stories for software developers/engineers is an effective method of baking security into software development. Existing security compliance requirements, organizational policies, industry guidelines, and standards can be referenced, which will require collaboration between project stakeholders to materialize. Approaches like INVEST⁸ can help support the creation of security user stories and tasks where each should be:

- **Independent:** should be self-contained
- **Negotiable:** should leave space for discussion
- **Valuable:** must deliver value to the stakeholders
- **Estimable:** should be able to estimate the size
- **Small:** should be able to plan, task, and prioritize it
- **Testable:** The development should be able to be tested

The bridge built between security and development should not be one-way. Security policies should be mapped to tightly scoped technical stories that developers and operations can complete. Software developers, DevOps engineers, and architects should provide feedback on the completion status so that security compliance teams that write information security policies have visibility of the security posture of the current software lifecycle stage/sprint. The creation of security patterns and hardening guides will support security compliance activities.

While security user stories can better demonstrate to business stakeholders that code has been designed and configured with organizational-approved security controls, they're also effective for measuring activities by effort and time, allowing security work to be prioritized in a backlog.

Security stories are typically manually produced against the organizational policy. However, where policies are irrelevant to the product's development, the [OWASP Application Security Verification Standard](#) can be useful for sourcing application security requirements, and the [CSA CCM](#) can effectively establish cloud requirements. However, opportunities for automating the security story process (e.g., SD Elements) may not address organizational policy alignment and prioritization of work.

Value Stream Security Mapping

A value stream security map (VSSM) will depict how work moves through product development from idea to customer. Having visibility of work represents how teams understand the flow of work from the business to customers and whether they have visibility into this flow, including the status of products and features.

⁸ <https://www.agilealliance.org/glossary/invest/>

Understanding how work moves through the application development lifecycle is important before considering security controls. VSSM is a useful tool that includes:

- **Stakeholders:** The accountable and responsible teams and individuals
- **Activity:** The generic activity performed during the phase
- **Lead time:** The time from when a process accepts a piece of work to the point the process hands that work off to the next downstream process
- **Process time:** The time it takes to complete a single item of work if the person performing it has all the necessary information and resources to complete it and can work uninterrupted
- **Percent complete and accurate (%C/A):** The proportion of times a process receives something from an upstream process that it can use without requiring rework

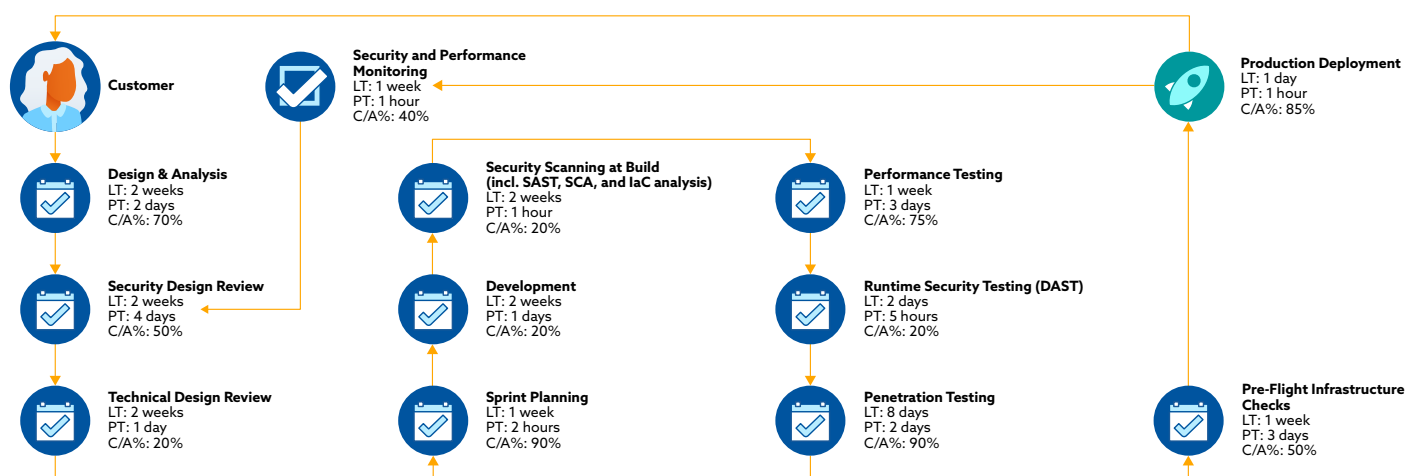


Figure 9: VSSM Example

A VSSM can be created⁹ in the following manner:

1. Identify the problem: Define the problem from the customers' point of view. Document the problem.
2. Present the problem: Ensure all team members understand the problem. Document the understanding and agreement.
3. Identify the team: Select the team with the right knowledge and experience.
4. Empower the team: Provide the team with the authority and resources to solve the problem.
5. Perform walk through: Walk through the complex process to ensure understanding of the steps and environment and establish a mutually shared purpose.
6. Initial Mapping: prepare a rudimentary mapping as a starting point.
7. Guardrail the process: Establish guardrails (measures/metrics) to ensure and maintain proper scoping.
8. Collect data - Gather/analyze the data - Data includes assets, resources, cycle times, lead times, and up times.
9. Map the current process: The previous steps in this map represent the current state and serve as a baseline.

⁹ <https://www.leanix.net/en/wiki/vsm/value-stream-mapping#How-to-create-a-value-stream-map>

10. Map the future process: Next, develop clear goals for what needs to be accomplished and map those into the current process.
11. Implement the future map: The future map guides the improvement process by giving team members targeted objectives to work towards, requiring a phased approach to implement improvements.

Value Stream Maps provide a wide array of industry-accepted benefits. However, the integration of security work provides additional benefits,^{10 11}:

- End-to-end security and visibility influences product development
- Better connects DevSecOps to business and time objectives, providing the information for better decision making
- Better collaboration for faster feedback between different stakeholders when deciding to include or remove security tools and processes
- Identify security bottlenecks and overly restrictive time-consuming tasks that can be better optimized at different value stream stages

Although the VSSM activity is typically performed manually and locally (e.g., [Atlassian](#)), per project, there are commercial vendors (e.g., [LeanIX](#), [Plandek](#), [HCL tech](#)) that provide a technology platform to create and manage Value Stream Maps.

Architecture Principles and Artifacts

Architecture principles and artifacts are the underlying general principles, goals and practices for the development and operation of all organization technology resources. Architectural principles reflect a level of consensus and form the basis for making future decisions and changes to technology, process, and people. DevSecOps architectural principles help plan for the creation of working patterns and guidelines used to build secure and approved solutions, as well as the understanding of effective team topologies. When followed, methods and patterns can re-engineer how people build products to help avoid vulnerabilities from being introduced and detect and prevent errors that could compromise the security of a product.

DevSecOps principles vary per organization and should align to business objectives while complementing the targeted value derived from software development. DevSecOps, an evolution of DevOps, should appreciate the value of speed and automation. It eliminates security gates and time-consuming activities by embodying values that focus on speed, innovation, and collaboration. Some common principles to build an effective DevSecOps foundation are:

- **Minimum viable, secure product:** A baseline to implement security controls and processes - the minimum threshold each product an organization develops should achieve. The [Minimum Viable Secure Product](#) provides an example checklist of how this can be achieved. However, a minimum viable secure product should go beyond a checklist and inform organizations to build effective baselines to refine further and build a product's security features.

¹⁰ <https://www.leanix.net/en/wiki/vsm/value-stream-management#what-are-the-benefits-of-value-stream-management>

¹¹ <https://www.virtusa.com/perspectives/article/value-stream-mapping-in-devsecops>

- **Security as a product function:** An approach to introduce security capabilities as product features to improve security. Product teams and solutions can address risks and threats instead of adding security solutions at the end of development pre-deployment. The [Security User Stories](#) activity provides an example method to help achieve this. However, it has a dependency on key stakeholder buy-in.
- **Security is collective responsibility:** DevOps culture is all about a shared understanding between developers and operations and sharing responsibility for the software they build. That means increasing transparency, communication, and collaboration across development, IT/operations, and “the business.”¹² Security responsibilities should be considered beyond the security team. All stakeholders should assume accepted security responsibility for building and managing services. See [CSA Pillar 1: Collective Responsibility](#) for information on implementation.

The three principles above should be treated as foundational. Principles beyond these 3 need to be relevant to the organization’s goals, team topologies, budget, IT landscape, and architecture. To understand an example of a full suite of DevSecOps principles, the United States Department of Defense has published a DevSecOps reference architecture guide¹³ that outlines the following:

- Remove bottlenecks (including human ones) and manual actions.
- Automate as many of the development and deployment activities as possible.
- Adopt common tools from planning and requirements through deployment and operations.
- Leverage agile software principles and favor small, incremental, frequent updates over larger, sporadic releases.
- Apply the cross-functional skill sets of Development, Cybersecurity, and Operations throughout the SDLC, embracing a continuous monitoring approach in parallel instead of waiting to apply each skill set sequentially.
- Security risks of the underlying infrastructure must be measured and quantified to understand the total risks and impacts on software applications.
- Deploy immutable infrastructure, such as containers. The concept of immutable infrastructure is an IT strategy in which deployed components are replaced in their entirety rather than being updated in place. Deploying immutable infrastructure requires standardization and emulation of common infrastructure components to achieve consistent and predictable results.

Beyond the foundational DevSecOps principles, we should recognize that DevSecOps principles should encapsulate what the industry evangelizes in DevOps and Cloud. AWS recognizes 5 key principles¹⁴ that form part of its well-architected framework for the cloud:

- **Perform operations as code:** In the cloud, the same engineering discipline for application code can be applied to infrastructure. Entire workloads (applications, infrastructure) can be deployed as code and updated with code. Operations procedures can be implemented as code and automated by triggering in response to events which limits human error and enables consistent responses to events by performing operations as code.

¹² [Building a DevOps culture | Atlassian](#)

¹³ [DoD Enterprise DevSecOps Reference Design](#)

¹⁴ [Design principles - AWS Well-Architected Framework](#)

- **Make frequent, small, reversible changes:** Design workloads to allow components to be updated regularly. Make changes in small increments that can be reversed if they fail (without affecting customers when possible).
- **Refine operations procedures frequently:** Improve operations procedures during use. As workloads change, evolve procedures appropriately. Regular game days to review and validate that procedures are effective and that teams are familiar with them.
- **Anticipate failure:** Perform “pre-mortem” exercises to identify potential failure sources so they can be removed or mitigated. Test failure scenarios and validate understanding of impact. Test response procedures to ensure that they are effective and that teams are familiar with their execution. Set up regular game days to test workloads and team responses to simulated events.
- **Learn from all operational failures:** Drive improvement through lessons learned from all operational events and failures. Share what is learned across teams and through the entire organization.

Google Cloud¹⁵ also recommends a set of principles for cloud-native architectures:

- **Design for automation:** Automation has always been a best practice for software systems. Still, the cloud makes it easier than ever to automate the infrastructure and components that sit above it. Although the upfront investment is often higher, favoring an automated solution will almost always pay off in the medium term in terms of effort and the resilience and performance. Automated processes can repair, scale, and deploy systems faster than people.
- **Be smart with state:** Storing of ‘state,’ be that user data (e.g., the items in the users shopping cart, or their employee number) or system state (e.g., how many instances of a job are running, what version of code is running in production), is the hardest aspect of architecting a distributed, cloud-native architecture. Therefore architect systems to be intentional about when and how storage and design components are stateless.
- **Favor managed services:** Cloud is more than just infrastructure. Most cloud providers offer a rich set of managed services, providing all sorts of functionality that relieve the headache of managing the backend software or infrastructure. However, many organizations are cautious about taking advantage of these services because they are concerned about being ‘locked in’ to a given provider. This is a valid concern, but managed services can often save the organization hugely in time and operational overhead. Whether or not to adopt managed services comes down to portability vs. operational overhead in money and skills.
- **Practice defense in depth:** Traditional architectures place a lot of faith in perimeter security, crudely a hardened network perimeter with ‘trusted things’ inside and ‘untrusted things’ outside. Unfortunately, this approach has always been vulnerable to insider attacks and external threats such as spear phishing. Moreover, the increasing pressure to provide flexible and mobile working has further undermined the network perimeter.
- **Always be architecting:** One of the core characteristics of a cloud-native system is that it’s always evolving, and that’s equally true of the architecture. Always seek to refine, simplify and improve the architecture of the system, as the needs of the organization change, the IT landscape, and the capabilities of the cloud provider.

15 [5 principles for cloud-native architecture—what it is and how to master it | Google Cloud Blog](#)

Although principles can feel like high-level abstract statements, there is value in identifying and mapping technology and security targets with business goals through principles. Define principles early within DevSecOps activities to yield the following benefits:

- In a complex technology landscape where DevSecOps can be difficult to grasp, establishing principles helps demystify the work that needs to be achieved to measure success
- They form a connecting target to business goals and can be reviewed by senior business stakeholders
- They are used as a guide to direct future decisions and changes to technology, people, and processes
- They can link to the Agile ceremonies and practices
- They can link to the day-to-day operational work teams spend their time on

Further examples of DevSecOps and cloud security principles beyond those referenced by the U.S. Department of Defense (DoD), Amazon Web Services (AWS), and Google Cloud can be found by the [NCSC's cloud security principles](#) and [Grafana's cloud-native security manifesto](#).

Risk Management (Shifted-Left)

The process for product teams to identify and effectively describe product risk (including application, component, platform, and dependent resources) and implement improvements to alleviate or prevent possible negative events. Although risk identification and assessment should occur early during the design phase of the development process, there are cases in which new risks may arise over time.

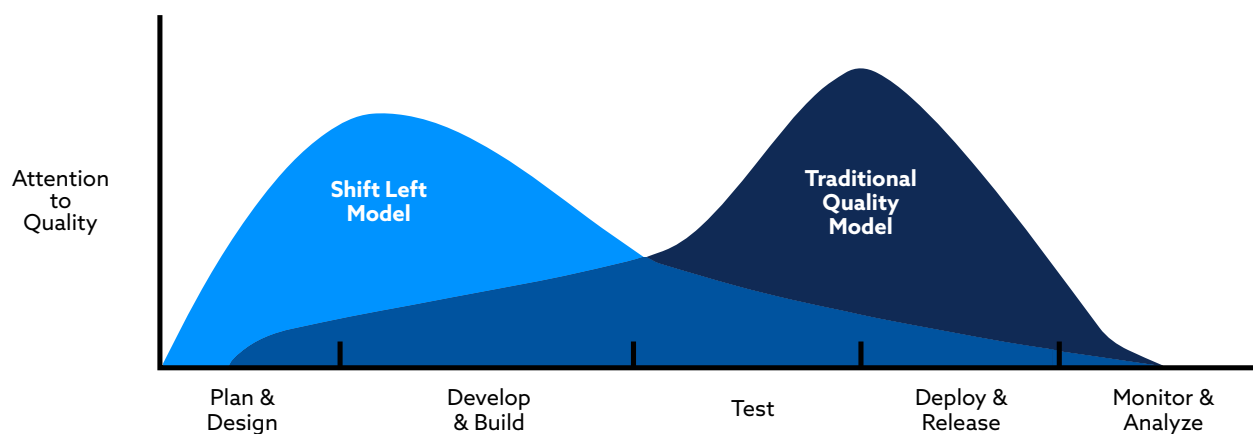


Figure 10: Risk Management Shifted Left

The ability to form risk statements bears importance in describing security issues and prioritizing work. In DevOps and modern software development paradigms, risk articulation will be operationally effective, reference threats (identified in threat modeling), and the types of vulnerabilities exploitable, see **Figure 11**. Pragmatic decisions are made by prioritizing work and the ability to form risk statements that include the following areas:

- **Asset:** The type of asset being protected, which can range from customer data to intellectual property (IP), and whether the customer is being affected or not.
- **Threat:** The type of threat is considered an internal threat, such as malicious insiders, or an external threat, such as Advanced Persistent Threats (APTs).
- **Vulnerability:** Assess which vulnerability could be exploited for the threat to execute an attack (e.g., remote code execution, code injections, reverse engineering) should be considered.
- **Compensating controls** (i.e., predisposing conditions): Assess and enumerate existing mitigations, for example, firewalls segmenting the network, input filtering capabilities of a web framework or product interface, binary obfuscation.
- **Describe the risk:** The impact is obtained by considering the type of asset being protected and the threat (e.g., IP and data leaks), while the likelihood considers the likelihood of the impact being exploited (e.g., by a vulnerability) and potential mitigations (e.g., Server Side Request Forgery (SSRF) in an internal, firewalled network).

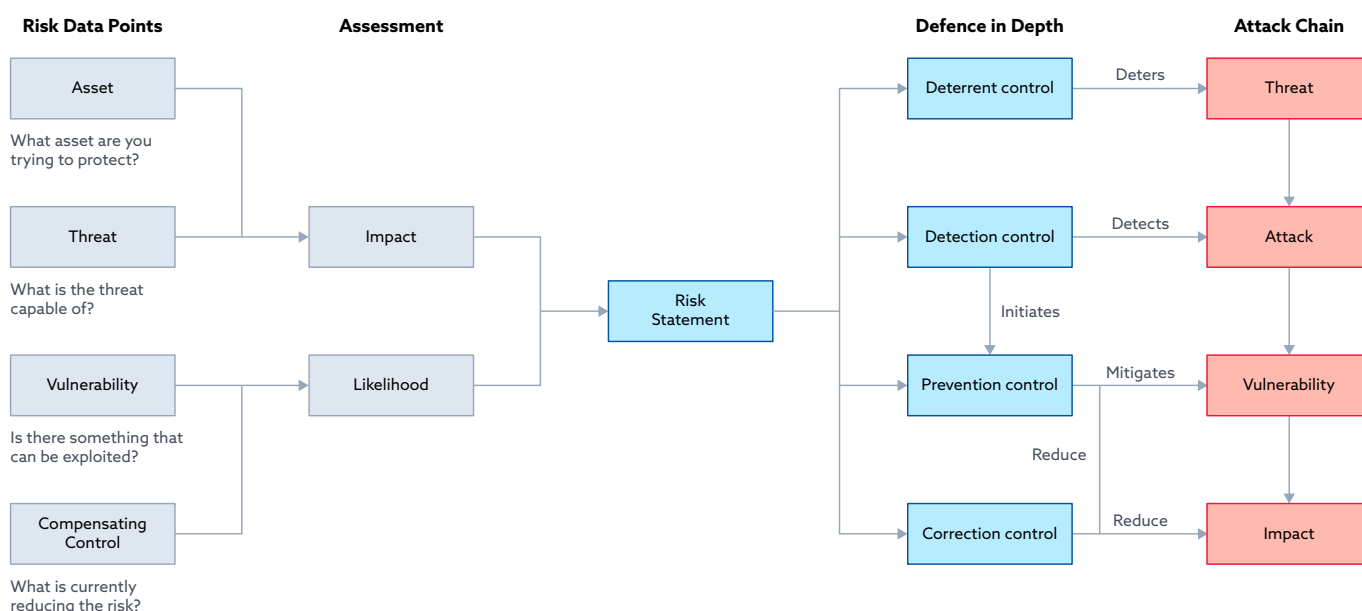


Figure 11: Risk Articulation and Control

Shifting risks left means tackling unknowns to reduce or eliminate risks associated with product design. Performing risk analysis at design rather than later ensures it helps manage time, costs and resources and better predict management obstacles. Shifting risk left also allows product team stakeholders like Solution Architects and Software Engineers to more easily identify and raise risks rather than going through monolithic and time-consuming risk management processes with which they are not otherwise familiar.

An organization's risk appetite and the approach to risk management is influential in decision making when utilizing the people, process and technology capabilities. The absence of risk management could jeopardize an organization's running budget and bring chaos to the overall morale of the company and its stakeholders. Organizations should be in a much better mature state in security operations to adopt DevSecOps, as it requires long term investments and fine tuning.

While example risk frameworks are influenced by organization, industry sector, and geographic location, the material is like the [CSA's Perspective on Cloud Risk Management](#), which helps identify and manage key risk themes.



The capabilities that can be applied as applications are being developed. Coding security controls that rely on automation as tools can better and more consistently identify weaknesses and vulnerabilities in code compared to manual human reviews. Without including security in the coding phase, there is a risk that vulnerabilities in source code will be unidentified and deployed into production environments. Security vulnerabilities and weaknesses will be more expensive to fix later in the SDLC.

Developer Training

Developer training provides development teams with the responsibility, adequate training, tools, and resources to validate that software design and implementation are secure. With the requirement to secure code as part of the secure systems lifecycle, adequate secure coding training helps developers identify bugs while writing secure code in their respective languages. This minimizes the risk of introducing vulnerabilities and design flaws.

Research studies^{16 17 18} have shown that developers are commonly self-taught, which results in over 50% of software developers lacking secure coding awareness and skills. Even if developers have had some education in software engineering, this curriculum often does not include secure coding practices. This lack of training leads to insecure coding practices and common application vulnerabilities in the SDLC.

A prerequisite for effective developer training is to assess the maturity of the secure SDLC and corresponding secure coding practices. For developer training to be successfully achieved and implemented, it focuses on the roles and responsibilities associated with each development team member. For any case of developer training, a successful implementation is to provide an incentive to actually do the training by offering a reward for the completion through gamification of the training experience. Further, such training includes guidance and examples training that matches the technology and platforms that the developer is using. Training can come in many forms. The major ones are discussed in the following.

16 Patel, S.: 2019 Global Developer Report: DevSecOps finds security roadblocks divide teams (July 2020), <https://about.gitlab.com/blog/2019/07/15/global-developer-report/>, [Online; posted on July 15, 2019]

17 Assal, H., Chiasson, S.: 'Think secure from the beginning' A Survey with Software Developers. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. pp. 1-13. CHI '19, Association for Computing Machinery, New York, NY, USA (2019)

18 Gasiba, Tiago Espinha, et al. "Awareness of Secure Coding Guidelines in the Industry-A first data analysis." *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020.

- **Instructor-led training:** Instructor-led training is the practice of facilitating education for a group of developers conducted either online or in person. The advantage of this training is to have real-time access to the instructor for feedback and discussion. This training is particularly effective when using engaging demos, particularly how a particular vulnerability can be exploited. This benefits developers in understanding what the attackers are and provides an understanding of why input validation and sanitization are needed and the potential impact if the code does not behave as expected. SANS offers instructor-led developer training for software security via on-demand through online community resource offering via on-demand webcasts. The [OWASP Secure Knowledge Framework \(SKF\)](#) also has practical secure coding examples across several that are used for instructor-led training.
- **Online Training:** A form of online training where developers participate when they have time. This method is preferred for more advanced material allowing the person taking the course to review the material if required repeatedly. The [OWASP Juice Shop](#) is an insecure web application that can be used for security training and CTF awareness demos. The OWASP Juice Shop also encompasses secure coding challenges, particularly aimed at secure coding developer training. Also, the Open Security Foundation (OpenSSF) provides [free courses for software developers](#) on secure coding provided through the Linux Training & Certification platform.¹⁹ A commercial solution for secure coding training is [Secure Code Warrior](#), which provides an e-learning platform including training, tournaments, and assessment for skills-based pathways.
- **Security Awareness Training:** Security awareness training aims to provide end-users with information on current threats, security best practices, and policy expectations. However, solely relying on annual security awareness training has proven ineffective, particularly in establishing secure coding practices²⁰. Instead, effective security awareness training targets the role of the developer. Organizations like [SANS Web Application Security Awareness Training](#) provide specialized training with role-based and progressive paths.
- **Mentoring:** Mentoring is an effective form of developer training as it typically encourages a developer in the team to take ownership of security practices. A form of mentoring is formalized through a security champion program, where the security team works closely with nominated security champions to educate fellow developers on security vulnerabilities and secure coding practices. The [OWASP Security Culture project](#) and the [OWASP Security Champions Playbook](#) provide a step-by-step process to set up a security champions program that focuses on the security team providing mentorship to selected security champions in development teams.

Utilizing a combination of the developer training mentioned above provides several benefits. Research²¹ has shown that the level of security training a developer has is positively related to the security awareness of the entire software development team. Effective developer training also enables developers to produce secure code faster and prevent common security vulnerabilities from being introduced in the design and implementation phases of the secure systems development lifecycle.

¹⁹ <https://safecode.org/training/>

²⁰ Patel, S.: 2019 Global Developer Report: DevSecOps finds security roadblocks divide teams (July 2020), <https://about.gitlab.com/blog/2019/07/15/global-developer-report/>, [Online; posted on July 15, 2019]

²¹ Weir, C., Rashid, A., & Noble, J. (2017). Developer Essentials: Top Five Interventions to Support Secure Software Development (<https://core.ac.uk/download/pdf/301368474.pdf>)

Security Hooks

Git hook scripts are useful for identifying simple issues in code before a code review. Developers often use these hooks to prevent the introduction of light bugs in the code base, such as a missing semicolon, secrets in plain text, or trailing whitespace. Security hook scripts can also block a commit when security credentials are detected, using a secret scanning tool, or even after a match to common Software Application Security Testing rules.

Git security hooks can be set at any stage of a Git workflow. However, they are typically observed at pre and post-commit stages, see **Figure 12**. Other stages of a Git workflow where hooks can be applied are:

- pre-commit
- pre-push
- pre-merge-commit
- prepare-message-commit
- commit-message
- post-commit
- post-checkout
- post-merge
- post-rewrite

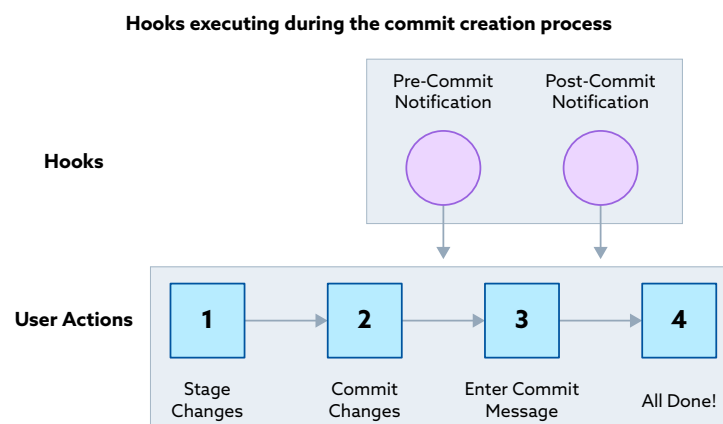


Figure 12: Git Hooks

It has to be noted that git security hooks don't substitute to other Continuous Integration security testing methods for two reasons:

1. Developer time is precious, and they will likely commit or push several times a day, while security hooks prevent the introduction of obvious security issues, such as leaked credentials in the code base, they shouldn't obstruct development workflows by taking more than a few seconds to a grand maximum of one minute.
2. For a non-container standardized development environment, there is no guarantee the hooks will be properly configured as in such a scenario; the set-up of this control would be based on trust and developer's opt-in.

While security hooks are effective for catching developer mistakes, the tooling landscape has not yet matured for these to be complete code scans. Some example security hooks can be found on [Git-scm](#) and [Pre-commit](#).

Code Linting

Code linting is defined as automated checking of source code for “bad code smells” to flag programming errors, bugs, stylistic and construct errors²². Linting tools are a basic form of static code analysis. Linter tools are commonly provided as configurable plugins for identifying and reporting patterns of bad code smells like misuse or deprecations. The boundaries between linting and advanced static analysis tools are blurred. Linting tools are varied and are specific to a particular programming language and are particularly used by development teams in an Integrated Development Environment (IDE) as a plugin or during code commit hooks.

A pragmatic approach to code linting is carefully selecting one linting tool for a coding language²³. When choosing a code linting tool, it is important to find the configuration and select only one linter to reduce verbosity. Also, it is important to ensure that the linter sufficiently supports the programming language for which that linting tool is used. Linters can also stand alone and build integration tools that allow custom extensions and API support. Areas where code linting is a particularly effective method, are to enforce secure coding standards and secrets detection and detect depreciated code constructs. Another area where linting has proven useful is detecting cyclomatic complexity, which indicates the complexity and hence also the readability and maintainability of a program.

Various tools are available for linting, the majority of which are focused on a particular programming language, from which a few are discussed:

- **Java:** CheckStyle²⁴ is a linter for Java that can be used both for linting and static code analysis. It is a tool focused on following coding standards for Java Developers. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard, including secure coding. Another
- **JavaScript:** For JavaScript the most widely used linters are JSLint²⁵ and ESLint²⁶, where ESLint is focused on identifying bad code smells, where JSLint is used as a static analysis tool to comply with coding rules.
- **Infrastructure as Code:** TFlint is an open-source pluggable Linter for Terraform. It warns developers about deprecated syntax and unused declarations and enforces best practices and naming conventions in Infrastructure as Code²⁷. It is possible to extend TFlint with other plugins and can also be run during a pre-commit hook. For Kubernetes, the KubeLint²⁸ analyzes Kubernetes YAML files and Helm charts and checks them against various best

22 OWASP DevSecOps Guidelines <https://owasp.org/www-project-devsecops-guideline/>

23 Tómasdóttir, Kristín Fjóra, Mauricio Aniche, and Arie Van Deursen. “The adoption of javascript linters in practice: A case study on eslint.” IEEE Transactions on Software Engineering 46.8 (2018): 863-891

24 <https://github.com/checkstyle/checkstyle>

25 <https://www.jshint.com/>

26 <https://eslint.org/>

27 <https://github.com/terraform-linters/tflint>

28 <https://docs.kubelinter.io/#/>

- practices, focusing on production readiness and common security misconfigurations.
- **Python:** Not strictly being a linter, Bandit²⁹ is an advanced static code analysis tool designed to detect common security issues in Python code.
- **Cross-language:** An example of cross-language static code analysis is PMD³⁰, which detects coming programming flaws, including unused variables, unnecessary object creation, and empty catch blocks. Besides Java, it supports other languages like Scala, Apex, Swift, and PLSQL.

Code linting overall has several benefits and security efficiencies. A major advantage of using code linting is giving immediate feedback to the developer when writing the code. Linting can detect errors in a code and which may lead to security defects. In particular, linting can notify a developer using vulnerable and deprecated classes and methods and reduce the risk of using vulnerable software components. Linting is important to reduce errors and improve the overall quality, helping in accelerating development and reducing costs by finding errors already when the developer writes source code. Linting can scan code for security defects and best practices like formatting or styling issues, making code easier to maintain and read. Linting increases the quality of the code and hence reduces the likelihood of introducing security defects by the developer. Readable code also allows code to be more secure due to an easier understanding of the code for developers. When configured correctly, linting allows enforcing of secure coding standards, particularly for error and exception handling.

However, it is important to point out that not every language has “quality” standard linter tools as each framework usually has one or several linters. Also, the linting tool landscape varies, and linting tools, versions, or configurations can lead to different results. Code linting also causes false positives and information overload due to its verbosity.

Software Composition Analysis

A key method to gain development velocity in building modern applications is to leverage open-source or third-party components. However, using such components brings risks such as security vulnerabilities, licensing obligations, and potential code quality issues. Software Composition Analysis (SCA), a subset of Component Analysis, is a process and a set of tools to identify and remediate risks associated with using open-source and third-party software product or application components.

SCA tools provide teams visibility across applications and the ability to create and enforce policies to mitigate risks from using these components. SCA tools help developers detect and remediate issues within their development environments and CI/CD pipelines. They are typically deployed as a security scan code after code has been committed to a Source Code Repository (SCR), as the SCA doesn't need to scan code in runtime. Typically, SCA tools provide the following benefits:

- Continually scan a codebase (e.g., source code, binaries, package managers, manifests, container images, serverless functions) and create/maintain an inventory of open-source and third-party components used. This inventory can be the starting point for a software bill

²⁹ <https://pypi.org/project/bandit/>

³⁰ <https://github.com/pmd/pmd>

- of materials (SBOM). [CycloneDX](#) and [SPDX](#) are two leading software BOM formats.
- Compare the software BOM against databases, such as the National Vulnerability Database (NVD), to identify vulnerabilities.
 - Provide information on the component's licensing requirements, including attribution.
 - Enable governance teams to develop a consistent set of open-source and third-party usage policies that apply across the company.
 - Automatically enforce the policy throughout the SDLC process (e.g., failing a build when it violates the policy, checking for vulnerabilities within the developer's IDE, and triggering an approval process for a new open-source or third-party component).

SCA solutions are automated and are available as open-sourced or licensed offerings. An example of an open source SCA tool is the [OWASP Dependency Check](#), and [OWASP Dependency Track](#). Licensed offerings are provided by vendors like [Synopsys](#), [Snyk](#), [Dependabot](#), and [Veracode](#).

Static Application Security Testing

Static Application Security Testing (SAST) is a methodology and tool that analyzes source code to find security flaws and vulnerabilities. SAST is also referred to as "white box" security testing. SAST helps developers identify vulnerabilities in the coding phase by providing them real-time feedback enabling them to fix issues before these vulnerabilities make it to the next phase of the SDLC. The NIST Source Code Security Analysis Tool Functional Specification v1.1³¹ identifies the classes of source code vulnerabilities that SAST tools must detect: input validation, range errors, API abuse, security features, time and state, code quality, and encapsulation.

As SAST tools are designed to scan source code before being compiled, it provides an opportunity to perform security scans earlier, hence shifting left. SAST scans can either be initiated manually once code is committed to an SCR or, most commonly, triggered as an automated pipeline action. Establishing a good SAST process involves:

1. Identify a SAST tool that works in the respective development environment. The tools should support the development environment's programming languages, libraries/frameworks, and binaries. It must be able to detect vulnerabilities based on the OWASP Top Ten and/or NIST specification Annex A.
2. Deploy the SAST tool in the respective development organization by establishing the infrastructure, access, and integrations with the IDE and the CI/CD pipeline.
3. SAST scans are run either in the IDE or at different stages in the CI process (e.g., pre-code check-in, post-commit).
4. Analyze scan results to remove false positives. If these issues are detected in the CICD pipeline, they should stop executing further, and the issues should be logged within a central repository.
5. Perform quick-fix changes and govern the development team's remediation of more time-consuming changes.

31 [NIST SP 500-268, Source Code Security Analysis Tool Function Specification Version 1.1](#)

SAST tools can scale well; they can be executed against source code repeatedly (e.g., check-ins, nightly builds). They can identify a large class of source code vulnerabilities with feedback that can save time and effort by catching vulnerabilities earlier in the SDLC. These tools can point out the location of the vulnerability in the source code and guide it to fixing the issue. SAST tools can also result in many false positives meaning the tool reports a weakness where no weakness is present. Analyzing test results and adjusting the policies/configuration for the tool can help with that.

Although SAST tools are effective, they have difficulty with certain classes of vulnerabilities (e.g., authentication and access control issues), especially issues related to the environment, configuration, and runtime. This may be remediated by adopting [Dynamic Application Security Testing \(DAST\)](#) as SAST and DAST complement each other. See **Figure 13**.

SAST vs. DAST

Static application security testing (SAST) and dynamic application security testing (DAST) are both methods of testing for security vulnerabilities, but they're used very differently.

Here are some key differences between the two:













White box security testing <ul style="list-style-type: none"> The tester has access to the underlying framework, design, and implementation. The application is tested from the inside out. This type of testing represents the developer approach. 		Black box security testing <ul style="list-style-type: none"> The tester has no knowledge of the technologies or frameworks that the application is built on. The application is tested from the outside in. This type of testing represents the hacker approach. 	
Requires source code <ul style="list-style-type: none"> SAST doesn't require a deployed application. It analyzes the source code or binary without executing the application. 		Requires a running application <ul style="list-style-type: none"> DAST doesn't require source code or binaries. It analyzes by executing the application. 	
Finds vulnerabilities earlier in the SDLC <ul style="list-style-type: none"> The scan can be executed as soon as code is deemed feature-complete. 		Finds vulnerabilities toward the end of the SDLC <ul style="list-style-type: none"> Vulnerabilities can be discovered after the development cycle is complete. 	
Less expensive to fix vulnerabilities <ul style="list-style-type: none"> Since vulnerabilities are found earlier in the SDLC, it's easier and faster to remediate them. Findings can often be fixed before the code enters the QA cycle. 		More expensive to fix vulnerabilities <ul style="list-style-type: none"> Since vulnerabilities are found toward the end of the SDLC, remediate often gets pushed into the next cycle. Critical vulnerabilities may be fixed as an emergency releases. 	
Can't discover run-time and environment-related issues. <ul style="list-style-type: none"> Since the tool scans static code, it can't discover run-time vulnerabilities. 		Can discover run-time and environment-related issues <ul style="list-style-type: none"> Since the tool uses dynamic analysis on an application, it is able to find run-time vulnerabilities. 	
Typically supports all kinds of software <ul style="list-style-type: none"> Examples include web applications, web services, and thick clients. 		Typically scans only apps like web applications and web services <ul style="list-style-type: none"> DAST is not useful for other types of software. 	
SAST and DAST techniques complement each other.		Both need to be carried out for comprehensive testing.	

Figure 13: SAST and DAST comparison³²

32 [What Is SAST and How Does Static Code Analysis Work? | Synopsys](#)

While SAST methods can be performed manually by security engineers, the effort and quality of manual code reviews are far too expensive compared to automated SAST tooling solutions. While SAST tooling selection depends on language support and the volume of false positives, [Synopsys](#), [Veracode](#), and [MicroFocus](#) are popular examples of licensed offerings. The OWASP community has also provided a list of open source and commercial tools available in the industry.

Container Hardening

Containerized applications and microservice architectures (with each microservice being containerized) are a common combination. This is because containers provide a platform for fast and repeatable development and facilitate dependency management within the same infrastructure, reducing the effects of application multi-tenancy and potential dependency clashes. In general terms, containers become key in enabling DevSecOps workflows.

A container can be understood as a process isolated from all the other processes in the machine³³. It is important to understand that container isolation is not the same as that provided by virtual machines and their corresponding hypervisor. Therefore, the attack surface of a containerized application may, in some cases, be bigger than that of virtual machines. For this reason, it is important to approach container security slightly differently than one would approach VM security.

Whenever containers are used, it is important to consider the notion of container orchestration (e.g., Kubernetes³⁴, Nomad³⁵, OpenShift³⁶). This type of software allows to control (or orchestrate) hundreds of containers so that configurations, resource scheduling, networking, isolation, and security configurations are to some degree centralized and become simpler to manage. Nevertheless, using orchestrators often introduces an additional layer of complexity in the system, which needs to be managed and whose security needs to be considered. Security considerations reflect the areas of a container that need to be hardened, the following security principles for container hardening are:³⁷

- **Least privilege:** Containers privileges should be minimized to the smallest set of permissions required to function.
- **Defense in depth:** Containers become an additional layer when applying defense in depth.
- **Reducing the attack surface:** A balance within the container orchestration layer and the simplicity of microservice architectures must be considered. This is because containerized microservice architectures may increase simplicity. Container orchestrators such as Kubernetes add a layer of complexity.
- **Limiting the blast radius:** Ensuring that a compromised container does not open the door for compromising additional system parts is critical in limiting the blast radius.
- **Segregation of duties:** Permissions and credentials need to be given according to specific segregation of duties strategy, this means that compromising a container will not give access to every resource and every secret within the system.

33 <https://www.redhat.com/en/topics/containers/whats-a-linux-container>

34 <https://kubernetes.io/>

35 <https://www.nomadproject.io/>

36 <https://www.redhat.com/en/technologies/cloud-computing/openshift>

37 Rice L. *Container Security : Fundamental Technology Concepts That Protect Containerized Applications* / Liz Rice. 1st edition. O'Reilly Media; 2020.

There are several container hardening guidelines. Some examples are the CIS Docker benchmark³⁸, the U.S Department of Defense container hardening process³⁹, and the U.S National Security Agency Kubernetes hardening guide.⁴⁰ Based on this, we have identified the following common aspects to focus on: container isolation, container networking, container image security, and container orchestration security.

- **Container isolation:** Ensuring containers are properly isolated is critical to ensure that the container cannot compromise the host (the machine hosting the containers). This means taking into account the following practical considerations:
 - *Privileged containers* have root access to all the devices in the host. This means that if you are root in the container, you are effectively root in the host. Thus, privileged containers should be treated carefully.
 - *Capabilities*⁴¹ allow Linux to have granular control over the different permissions that processes can have to perform certain actions in the system. Containers usually are assigned more capabilities than the ones they require. Thus, make sure that the containers you use only have the enabled capabilities they require.
 - *Mounted Volumes* allow containers to have access to storage within the host. However, mounting certain volumes (e.g., the `/`` volume) can allow a container to access the host filesystem completely.
 - *System calls* provide an interface between the kernel and the application. However, it is important to restrict the system calls that the container can perform. A tool that can be used for this is Seccomp.⁴²
 - *File access and permissions* within the container should also be controlled to ensure that high privileged binaries are not exploited to escalate privileges. Ways of doing this include AppArmor⁴³ and SELinux.⁴⁴
 - In case more isolation requirements are needed, it may also be interesting for developers to look into Kata Containers⁴⁵ and Firecracker.⁴⁶
- **Container networking:** Containers will often need to interact with each other as that is the nature of microservices. Therefore, it is critical to ensure that network isolation is properly managed. Using Web Application Firewalls, routing tables, service meshes, and network policies (in container orchestrators) ensure that services that should not communicate with each other are completely isolated.
- **Image security:** When considering image security, the focus should be on the integrity of the image to be deployed. That is, there must be assurances that the images used have not been modified or have not been compromised by a potential attacker. There are several ways to achieve this. In general, there are two cases to consider:
 - **Dockerfile security and build process:** In case a Dockerfile is being used to build the image from scratch, it is important to consider the following recommendations:

38 <https://www.cisecurity.org/benchmark/docker>

39 Container Hardening Guide Version 1, Release 1 15 October 2020 Developed by DISA for the DoD

40 https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/0/CTR_Kubernetes_Hardening_Guidance_1.1_20220315.PDF

41 <https://man7.org/linux/man-pages/man7/capabilities.7.html>

42 <https://man7.org/linux/man-pages/man2/seccomp.2.html>

43 <https://apparmor.net/>

44 <https://www.redhat.com/en/topics/linux/what-is-selinux>

45 <https://katacontainers.io/>

46 <https://firecracker-microvm.github.io/>

- Use a trusted registry (e.g., private Azure Container Registries, or Elastic Container Registries).
- Keep in mind if the security requirements of the application call for using tags or digests in the base images.
- Use multi-stage builds.⁴⁷
- Use rootless containers.
- Access control to Dockerfile editing.
- Check volume mounts to avoid mounting sensitive directories.
- Do not include any sensitive data in the Dockerfile.
- Avoid SETUID binaries, as they could allow attackers to escalate privileges within the container.
- Avoid unnecessary code in the Dockerfile.
- Ensure that the access to the Docker daemon is controlled and that not every developer has access to the building machines.
- Use daemonless builds for images: buildah,⁴⁸ podman,⁴⁹ and buildkit,⁵⁰ for example.
- **Image scanning and signatures:** Image scanning aims to catch application-level vulnerabilities and malicious code that could have been introduced in the images being used as bases. The goal then is to regularly scan images and sign them to ensure that no unauthorized changes are allowed once scanned and built. Some good practices are:
 - Run a private registry.
 - Sign images using tools such as Notary⁵¹ or Harbor.⁵²
 - Scan images regularly for application-level vulnerabilities or malicious packages.
- **Container orchestration security:** The topic of container orchestration security is extremely broad. As it encompasses the security of the infrastructure underlying the orchestrator and the security of the software-defined infrastructure layer that it manages and orchestrates. Our recommendation is, therefore, to use complete frameworks such as the CIS benchmark for Kubernetes⁵³ or the U.S National Security Agency hardening guide for a complete guide. Some general recommendations can help secure orchestrators such as Kubernetes. Ensure:
 - An admission controller (e.g., OPA Gatekeeper⁵⁴) enforces the creation of objects in the cluster.
 - All accounts have a properly defined RBAC and that `star` permissions are not granted whenever they are not required.
 - The exposure of node ports to the public internet has been avoided, unless necessary.
 - All containers deployed in the cluster follow the aforementioned recommendations.
 - All used images follow the aforementioned recommendations.
 - Service meshes such as Istio⁵⁵ or Linkerd⁵⁶ are used to guarantee mutual TLS authentication between microservices whenever sensitive information is

47 <https://docs.docker.com/develop/develop-images/multistage-build/>

48 <https://buildah.io/>

49 <https://podman.io/>

50 <https://github.com/moby/buildkit>

51 <https://www.cncf.io/blog/2021/07/28/enforcing-image-trust-on-docker-containers-using-notary/>

52 <https://goharbor.io/docs/1.10/working-with-projects/project-configuration/implementing-content-trust/>

53 <https://www.cisecurity.org/benchmark/kubernetes>

54 <https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/>

55 <https://istio.io/>

56 <https://linkerd.io/>

- communicated between containers
- Control plane isolation from the data plane and do not allow the deployment of application containers in the control plane nodes.

Infrastructure as Code Analysis

Infrastructure as Code (IaC) is managing infrastructure through code, using the same tools as developers, and maintaining consistency and repeatability of provisioning and deprovisioning of infrastructure. IaC configuration files contain your infrastructure specifications, which ensures usability and consistency in editing and distributing configurations. IaC is a key part of DevOps practices. It reduces the work required for developers and DevOps engineers to build infrastructure by allowing scripts to automate infrastructure builds and changes. It enables engineers to version control, deploy, and improve cloud infrastructure while performing security verification checks. This also presents an opportunity to proactively improve the posture of cloud infrastructure and reduce the burden on security and operations teams. There are primarily two types of IaC; scripting and declarative:

- Scripting Types:** These are scripts such as Bash, Go, Python, or any other languages that use the different clients (SDKs) provided by the cloud provider; for example, you can script the provisioning of Cloud infrastructure resources with the AWS CLI or Azure PowerShell. This type requires more lines of code as it is necessary to manage the different states of the manipulated resources and write all the steps to create or update the desired infrastructure.⁵⁷
- Declarative Types:** These are languages in which it is sufficient to write the state of the desired system or infrastructure in the form of configuration and properties. This is the case, for example, for Terraform and Vagrant from HashiCorp, Ansible, the Azure ARM template, PowerShell DSC, Puppet, and Chef. The user only has to write the final state of the desired infrastructure, and the tool takes care of applying it. This allows infrastructures to be defined declaratively and versioned using the same source code control tools used for the application code (i.e., GitOps).⁵⁸ After a declarative state has been established, “drift” is established as a state that is not corresponding to the defined desired state. Configuration management tools have historically been used to enforce a state derived from a point-in-time golden image.

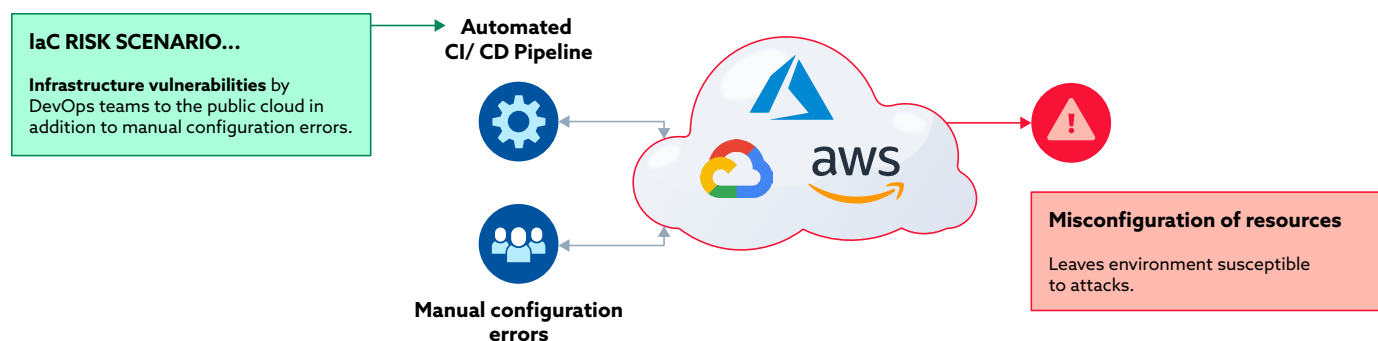


Figure 14: IaC misconfiguration workflow

⁵⁷ <https://www.techtarget.com/searchitoperations/feature/Understand-the-role-of-infrastructure-as-code-in-DevOps>

⁵⁸ Chandramouli, Ramaswamy. "Implementation of DevSecOps for a Microservices-based Application with Service Mesh." NIST Special Publication 800 (2022): 204C.

It is important to maintain controls that preserve its integrity consistently. Defining a range of allowed behaviors in policies defined as code can help detect “drift” or deviation from the desired configuration of allowed behaviors. Defining pre-deployment security configuration baseline checks are what secure by-default frameworks are built on. The ability for security teams to scale multiple applications grows more difficult with the necessary velocity.

Defining pre-deployment configuration safety checks are what secure by-default frameworks are built on. Defining security policy-as-code as configuration safety checks democratize security and compliance. It creates a self-service ecosystem where engineers can safely deploy resources without unintentionally breaking a service or deploying resources insecurely. Re-deploying resources after time dedicated to triage, code refactoring, follow-up, and testing can take days or weeks or resources. According to a study regarding the economic impact of technical and insufficient testing produced by NIST⁵⁹,

Within the scope of Infrastructure-as-code security, defining security policies are precursors to achieving compliance-as-code. IaC practice is taken one step further from declarative infrastructure-as-code to the desired state with codified compliance requirements/controls into actionable Policy as code (PaC). It creates a self-service ecosystem where engineers can safely deploy resources without unintentionally breaking a service or deploying resources insecurely. Re-deploying resources after time dedicated to triage, code refactoring, follow-up, and testing can take days or weeks or resources.

[The Open Policy Agent](#) (OPA)--a method of achieving PaC--is a policy engine that automates and unifies the implementation of policies across cloud-native environments. Organizations use the OPA to automatically enforce, monitor, and remediate policies across all relevant components. The OPA can be used to centralize operational, security, and compliance across services like orchestrators, application programming interface (API) gateways, and continuous integration/continuous delivery (CI/CD) pipelines. OPA can support your creation, deployment, and management of IaC by providing capabilities for enforcing policies. Some examples include⁶⁰:

- Application authorization: OPA uses a declarative policy language that enables the creation and enforcement of rules, including permissions to contribute policies for tenants.
- Kubernetes admission control: Kubernetes comes with a built-in feature that provides capabilities for enforcing admission control policies. OPA helps extend these by pointing container images at corporate image registries, injecting pods with sidecar containers, add specific annotations to your resources.
- Service mesh authorization: OPA can help regulate and control service mesh architectures by adding authorization policies directly into the service mesh. This can help limit lateral movement across a microservice architecture and enforce compliance regulations.

Code scanning (e.g., IaC analysis) on tools like Terraform is effective during the build stage. It can, in turn, reduce the likelihood of vulnerabilities and security weaknesses being written into code. However, it must be understood that scanning at build does not always yield a complete and holistic set of findings compared to post-deployment scanning (e.g., [Cloud posture management](#)). [Tfsec](#),

59 Planning, Strategic. “The economic impacts of inadequate infrastructure for software testing.” *National Institute of Standards and Technology* (2002): 1.

60 [Open Policy Agent: Authorization in a Cloud Native World](#)

[Kics](#), [Checkov](#), [Snyk](#), and [Terrascan](#) are all examples of IaC scanning tools for Terraform deployments. Once a security baseline is established, infrastructure or cloud resources are deployed according to policies defined in a separate policy engine. Each policy requires IaC resource definitions to comply with your security baseline (see [Guardrails](#)).

Although IaC analysis is recommended in 'coding' during infrastructure build, IaC security tools can provide advantages when used in additional stages like pre-deployment and runtime.

- **Pre-deployment:** Configuration Safety Checks can be conducted as hardening checks on cloud resources/services, scale set templates before making a VM or container golden image available for use. It is important to test IaC and security configuration before deployment regularly. Open source tools such as Conftest⁶¹ can be leveraged for structured checks or tests for Terraform code, Serverless configs, Kubernetes configurations, or CI/CD pipeline definitions. Once the desired state is declared in the policy code, a regression can regularly be detected among deployed cloud resources.
- **Runtime:** Defined policies can then be synchronized across resources running in the cloud. In running workloads, policies can be defined and handled using policy tools such as Open Policy Agent or Kyverno (specifically for Kubernetes security context).

Security Unit Testing

Security unit testing is the fine-grained verification of non-functional behaviors for security-related functionality. The practice of security unit testing encompasses both unit and integration tests, as most automated testing frameworks can be used to write and execute both. Development teams create these tests to evaluate application behavior at different levels during the implementation and testing phase.

Unit tests aim to test the behavior of isolated components within the code, including classes or methods. From the developer's perspective, the objective of security unit testing is validating the code toward secure coding requirements. On the other hand, the purpose of integration tests is to verify interactions between multiple units or components. For security integration testing, developers validate their own code components like functions, methods, classes, APIs, and libraries before being integrated with other software components.

A pragmatic approach for unit testing is to focus on software components that are prone to security vulnerabilities as part of a generic security test suite.

- **The OWASP Security Testing Guide**⁶² describes an effective security unit testing approach based on a generic security test suite as part of the wider unit testing framework. Using this approach, developers can build a collection of security test cases derived from the generic security testing suite.
- **InSpec:** A framework that can be used to create a generic for security unit and integration

⁶¹ See <https://www.conftest.dev/> for more information.

⁶² Owasp.org. 2022. OWASP Web Security Testing Guide | OWASP Foundation. [online] Available at: <<https://owasp.org/www-project-web-security-testing-guide/latest/>> [Accessed 7 August 2022].

testing for cloud infrastructure and system configuration is InSpec⁶³, which is an open-source testing framework for infrastructure with a human- and machine-readable language for specifying compliance, security, and policy requirements. Threat modeling also aids in identifying use and misuse cases for the generic test suite. The test suite can include relevant test cases against application security controls and requirements from standards like the OWASP Application Security Verification Standard. The software components and associated functions, methods, and classes for testing include commonly following areas covered in the OWASP, the Web Security Testing Guide (WSTG):

- **Identity, authentication & access control:** Authentication, identity, and access control are software components where security unit testing can assure vulnerabilities like broken access control as well as identification and authentication failures. Security unit tests for authentication include tests for failures or defects related to token abuse due to insufficient validation and modification.
- **Input validation & encoding:** For input validation and encoding, components include classes and methods that validate and sanitize user-provided input, like user input forms and user request parameters. Any components that take in user input may be potential entry paths for code injection or code injection like SQL or cross-site scripting.
- **Encryption:** Software components for encryption include components for transmitting encrypted data. Unit tests related to encryption include validation of encoding techniques, encryption libraries, and hashing algorithms. These tests cover attack cases against weak cryptographic controls, cleartext transmission of sensitive information, and improper signature validation for certificates and file hashes.
- **User and session management: security unit cases:** The unit and integration test in this area includes checks for account enumeration and brute forcing prevention. For session management, unit tests cover each user's session status and access rights for the duration of multiple requests. This aids assurance against broken access control and security misconfiguration.
- **Error and exception handling:** Unit tests for error and exception handling check for the correct status code is returned after an error and error messages do not contain sensitive information. Validating for such aids in preventing unexpected failures or validating potential information disclosure that could otherwise allow further exploitation.
- **Auditing and logging:** This category covers software components related to auditing and logging features. For logging, tests validate that logs do not include sensitive information and are properly formatted and encoded. These tests also validate that auditable events like failed logins and high-value transactions are not logged. These tests reduce the risk of security logging and monitoring failures.

Security unit testing may bring several benefits. Particular benefits are ease of use, advantages over scanning tools, and delivery speed⁶⁴.

63 GitHub. 2022. GitHub - inspec/inspec: InSpec: Auditing and Testing Framework. [online] Available at: <<https://github.com/inspec/inspec>> [Accessed 7 August 2022].

64 Gonzalez, Danielle. The State of Practice for Security Unit Testing: Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflows. Diss. Rochester Institute of Technology, 2021.

- **Ease of Use:** The main benefit is an adaptation of already-standard practices and infrastructure, whose benefits can be inherited to improve the security posture of software components with minimal disruption to the developer's current ways of working. That's because unit testing does not require extra tools, and developers can run and update tests as part of manual or automated security testing. Also, unit testing inherits the benefits of functional testing approaches, where developers can independently and continuously test small components. Unit testing is a familiar practice for most developers, as the teams have the required tools, skills, and infrastructure to run unit and integration tests.
- **Advantages over scanning tools:** It produces fewer false negatives and higher accuracy for test cases. It produces lower overhead and considers the developer's intent when creating unit and integration tests. Like scanning tools, developers can use unit tests to debug security exploits for further investigation, allowing more detailed vulnerability analysis and allowing developers to create fixes and confirm that it is not exploitable.
- **Delivery speed:** Security unit and integration also enable developers to reproduce security findings directly on the code level, allowing speed, lower cost, and earlier defect detection. This benefits increased security test coverage and traceability to test requirements. Test cases identify potential security issues that have root causes in source code. It also allows the detection of security defects and vulnerabilities early and in a repeated manner as part of the generic test quality assurance practices.

Peer Reviews

While often overlooked, peer review and validation are highly recommended before committing changes to your repository master branch. However, as human time is expensive and limited, it is important to peer review changes to source code and design, such as evaluating "entry points" and attack surfaces across trust boundaries (see [Threat Modeling](#)).

To enforce peer review on critical code components, most code repository software, such as [GitHub](#) and [Gitlab](#), provide branch protection features. By enabling branch protection on your project master branch, you can, for instance:

- Require pull request reviews before merging
- Require a certain number of approvals before merging
- Dismiss stale pull request approvals when new commits are pushed
- Require status checks to pass before merging
- Require branches to be up to date before merging

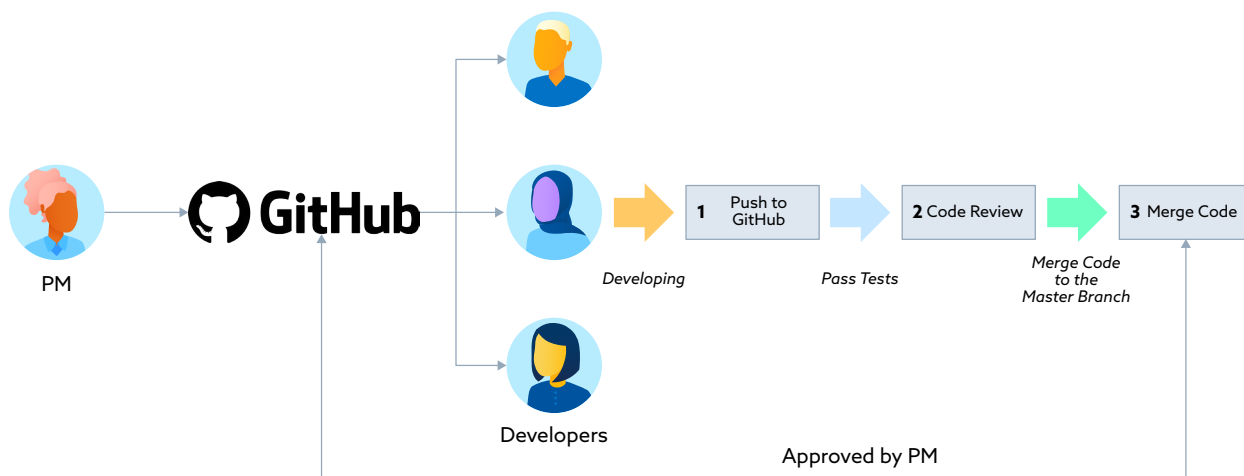


Figure 15: Source code peer review

Integration and Test

Integration and test includes the tools and processes to security test the functionality of an application/product. Without the tools and methods, security vulnerabilities and weaknesses will be the source of exploitation and result in data breaches and availability issues.

Dynamic Application Security Testing

Dynamic Application Security Testing (DAST) is a testing methodology that analyzes a running application to identify security vulnerabilities by simulating external attacks against the application's external interfaces. Unlike SAST, DAST tools will scan an application without the knowledge of the application's source code, internal workings, or design. Hence, it is referred to as "black box" testing. DAST tools are typically used to test web applications and services, though some solutions can cover API scanning, pen testing, and fuzz testing.⁶⁵

The most common application vulnerabilities DAST addresses are cross-site scripting, SQL injection, insecure server configuration, path disclosure, denial-of-service, code execution, memory corruption, PHP injection, JavaScript injection, local and remote file inclusion, and buffer overflows⁶⁶. DAST tools can include testing techniques observed in [fuzzing](#).

DAST tools run independently of the application. They can be run constantly as part of your testing suite. As new vulnerabilities are discovered, companies can find and patch vulnerabilities before they are exploited². DAST tools typically are programming language agnostic since it does not depend on the source code. Since DAST tools do not have access to source code, they cannot point to the location of the vulnerability in the code base, making it time-consuming to interpret reports. The tester must understand the application and its attack surface well to ensure that the DAST tool was configured

⁶⁵ [What is Dynamic Application Security Testing \(DAST\) and How Does it Work? | Synopsys](#)

⁶⁶ [Dynamic application security testing - Wikipedia](#)

correctly². DAST tools run later in the CI/CD pipeline as part of the Integration and Test phase, making it more expensive to fix vulnerabilities. Establishing a good DAST process typically involves:

1. Identifying a DAST tool that works in your environment. The Web Application Vulnerability Scanner Evaluation Project (WAVSEP)⁶⁷ and OWASP Benchmark⁶⁸ project are available to help establish the selection criteria and evaluation of DAST tools. The selection criterion needs to include the following⁶⁹:
 - a. A low number of false positives and negatives
 - b. Detailed reports can be generated automatically with a high-level summary and technical detail
 - c. Ability to model user journeys
 - d. Ability to support a range of scripting/programming languages, frameworks, and all web applications
 - e. Can identify previously unknown or undisclosed flaws, known as zero-day vulnerabilities
 - f. It can repeatedly run in an automated manner.
2. Understanding user workflows in the application – working with software developers and solution architects to understand how they interact with the application and/or by recording their actions.
3. Automating user interactions with scripts incorporated into the CI/CD pipeline as part of the integrate and test phase.
4. Identifying and remediating issues and vulnerabilities to prevent exploitation.

Compared to SAST tools, DAST tools generate fewer false positives because it simulates real-world scenarios⁷⁰. Also, unlike SAST tools, DAST tools can detect runtime and environment-related issues and require a running application⁷⁰. Acunetix, StackHawk, and OWASP ZAP are popular examples of DAST testing tools.

Interactive Application Security Testing

Interactive Application Security Testing (IAST) is a method that combines the techniques of SAST and DAST. Like SAST, it can identify problematic lines of code to help the developer with remediation. In addition, IAST also provides dynamic input detection. Therefore, IAST combines the capabilities of SAST and DAST; IAST differs from DAST in that it runs within the system under test. It must therefore be integrated into the code base before deployment. However, IAST can detect vulnerabilities during development and after release. IAST testing is typically implemented using an agent hosted in the compiled code base and the web server under test. The detection mechanism scans the source code for coding errors when the code is compiled.

Like DAST, IAST is a form of security testing that occurs while the application runs, typically in a QA or test environment. IAST contains a scanning agent and software libraries within the application's code base. The agent monitors how the application responds to the tests. IAST uses internal application flows, traffic flows, and analytics to run tests. The tool can perform static source

⁶⁷ [Security Tools Benchmarking](#)

⁶⁸ [OWASP Benchmark](#)

⁶⁹ [How to choose your DAST Tool | AppCheck](#)

⁷⁰ [Dynamic Application Security Testing \(DAST\) | Snyk](#)

code visibility and dynamic tests. The agent has visibility into the compiled code, including traffic requests, third-party libraries, and communication with the operating system.

Detectors compiled within the codebase analyze how the application responds to inputs and unit tests. The application server hosts the detection component during runtime. IAST runs in either active or passive scanning mode:

- **Passive:** Unlike DAST tools, IAST does not actively analyze the traffic flow but instead uses legitimate traffic for analysis, resulting in fewer false positives. During the passive scan, the IAST runtime agent scans all incoming traffic to the application and detects the data flow. Depending on how the application responds, the IAST tool can identify the vulnerability within the source code by looking at the URL and the included request parameters. It is recommended for IAST to enable passive interactive analysis as part of the unit testing phase, allowing the tool to act as a crawler and detect vulnerable third-party components.

The advantage of passive scanning is that it analyzes traffic generated by unit tests. On the other hand, a passive scan relies on real-time traffic and traffic generated by unit tests and only provides limited test coverage of the source code. Hence, the tool may miss out on vulnerabilities outside this traffic, mainly when a user or tester only tests a limited set of functionality.

- **Active:** An active IAST uses a DAST scanner in the background to create traffic and perform active scanning techniques which the IAST tool can detect. Fewer vulnerabilities are detected during an active IAST run as only traffic from the DAST scan is in scope. During an active run, the IAST can only detect vulnerabilities triggered by the DAST scan. On the other hand, this may also result in a lack of detection for vulnerabilities, particularly when the application modifies the request or the tested code sections of the DAST tool.

It is also recommended to have a testing environment to perform most of the tests to detect more vulnerabilities. The advantage of interactive analysis is that it combines the abilities of SAST and DAST and may replace one of these tools if required. The agent-based approach allows the detection of vulnerabilities across different environments like development and production during the execution.

Due to the combined capabilities of dynamic and static testing, IAST tooling also provides wider security testing coverage across the OWASP Top 10⁷¹. The interactive analysis also enables analysis of actual executions and can hence perform a more accurate security analysis during runtime. Compared to DAST, interactive analysis is more accurate in identifying and mapping the vulnerable sections of the code and may require less effort to remediate a finding. Interactive Analysis helps to overcome the challenges with DAST tools as it aids in identifying and mapping the section of the code tested by the DAST tool and requires less effort for remediation.

71 Mateo Tudela, Francesc, et al. "On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications." *Applied Sciences* 10.24 (2020): 9119.

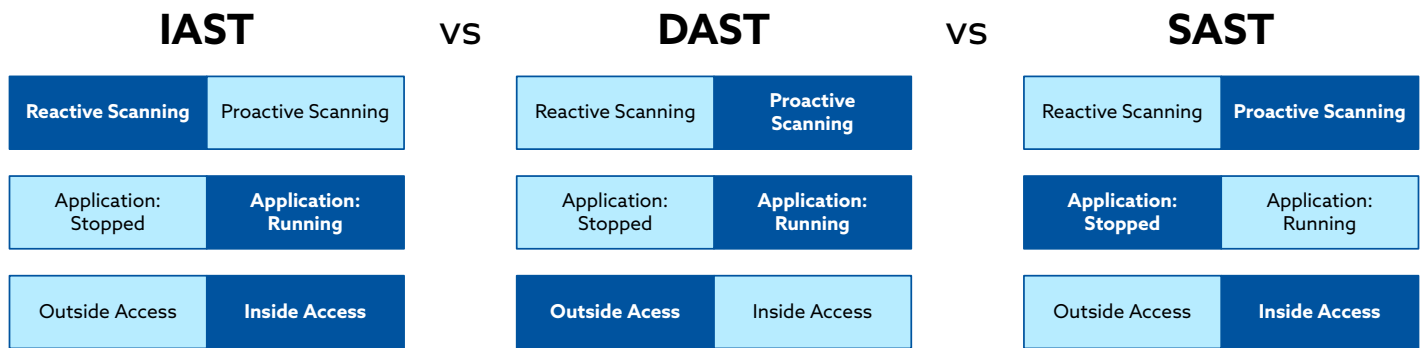


Figure 16: IAST compared to DAST and SAST

Most IAST tools are proprietary and commercial. However, some community editions exist. Open source IAST projects are available, like the [Contrast Community Edition](#), which provides IAST capabilities for Java and .NET Core. Commercial products for IAST are available through vendors like [Contrast](#), [Checkmarx](#), [Synopsys' Seeker](#), and [HCL's ASoC](#). As mentioned in this document, the [SAST](#) and [DAST](#) areas of product selection should depend on factors like language support and the volume of false positives.

API Testing

Application Programming Interface (API) enabled digital ecosystems are proliferating as a business can package tasks and functions as an API product or attribute. A good example in the financial services industry is open banking. APIs come in many shapes and forms; they may be internal, external, REST, SOAP, or some form of a procedure call.

Security is a key component of successful APIs. APIs expose data, transactions, and services to risks; sensitive data and IP may be compromised due to insecure APIs. Commonly observed API vulnerabilities are the lack of authentication and encryption, compromised client credential keys, and insecure protection of storage assets, for example, AWS S3 buckets/folders, and human intervention attacks by API injections (such as XSS, SQLi, and DDoS).

Understanding the API lifecycle helps decompose what needs to be secured, including identifying APIs consumed and testing against vulnerabilities. The [OWASP Top 10 API](#) vulnerabilities identify exploitable vulnerabilities to test against:

1. Broken object level authorization: APIs object identifiers, creating a wide attack surface - object-level authorization checks should be considered in every function that accesses a data source using input from the user.
2. Broken user authentication: Authentication implemented incorrectly allows attackers to compromise authentication tokens, compromising a system's ability to identify the client/user.
3. Excessive data exposure: object properties are commonly exposed without considering their sensitivity.
4. Lack of resources and rate limiting: the absence of restrictions on the size or number of

resources requested by the client/user can impact the API server performance and lead to Denial of Service (DoS) and brute force attacks.

5. Broken function level authorization: complex access control policies with different groups and roles, and an unclear separation between administrative and regular functions, lead to authorization flaws where attackers gain access to administrative functions.
6. Mass assignment: without proper filtering based on an allowlist, it usually leads to Mass Assignment, allowing attackers to modify object properties they are not supposed to.
7. Security misconfiguration: a result of insecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods, permissive Cross-Origin resource sharing (CORS), and verbose error messages containing sensitive information.
8. Injection: Injection flaws, such as SQL, NoSQL, and Command Injection, occur when untrusted data is sent to an interpreter as part of a command or query to trick the interpreter into executing unintended commands.
9. Improper asset management: Proper hosts and deployed API version inventory play an important role to mitigate issues such as deprecated API versions and exposed debug endpoints.
10. Insufficient logging and monitoring: this allows attackers to maintain persistence and pivot to more systems to tamper with, extract, or destroy data.

Organizations that need to add a layer of protection will utilize a Security API gateway. These perimeter gateway products provide a range of security services, including authorization and authentication of APIs, logging, and monitoring. The API gateway can be tested the same way as any other endpoint for authentication to enter a corporate network. If the gateway is a high end product, it will perform similarly to a firewall and will have very few ports and services exposed or available. Almost all cloud providers offer API gateways and may perform some authentication, but they have limited security functions.

While API security testing assists in building more secure and hardened API services, some common design considerations can help further secure your use of API's:

- **Using security policies:** Applying API like rate limiting at the IP address, quota limits to control API usage, and API key verification.
- **Bot detection:** Services to monitor traffic and identify illegitimate requests.
- **Payload validation:** For inbound traffic applying input and request validation on languages like XML and JSON.

Tools are available to test against API features and vulnerabilities. These can form part of unit tests, fuzz testing, or testing in runtime with solutions like a [DAST](#); some DAST vendors support API testing by inputting malicious content. Tooling specific to API testing is available like [BurpSuite](#), [Postman](#), and [Hoppscotch](#).

Fuzz Testing

Fuzz testing⁷² is a software testing method that injects malformed/unexpected inputs into a system to reveal software defects and vulnerabilities. A fuzzing tool injects these inputs into the system and then monitors for exceptions such as crashes or information leakage and can be used to test applications in runtime and API services. The most common forms of fuzz testing are:

- **Mutation-based fuzzing** which mutates existing data samples to create test data.
- **Generation-based fuzzing** produces new test data based on input models.

This testing method allows finding programming errors such as memory leaks or buffer overflow, which are rarely caught by developers' usual tests such as unit, integration, or system tests. Attackers frequently target these bugs to disrupt or gain access to production systems. For instance, the vulnerability Heartbleed on the cryptography library OpenSSL could have been detected using fuzz testing. Bugs usually found through fuzz testing are:

- Bugs specific to C/C++: Use-after-free, buffer overflows, uninitialized memory, and memory leaks
- Arithmetic bugs: Div-by-zero, int/float overflows, and invalid bitwise shifts
- Plain crashes: NULL dereferences and exceptions
- Concurrency bugs: Data races and Deadlocks
- Resource usage bugs: Memory exhaustion, hangs or infinite loops, and infinite recursion (stack overflows)
- Logical bugs: Discrepancies between two implementations of the same protocol ([example](#)), round-trip consistency bugs (e.g., compress the input, decompress back; compare with the original), and assertion failures

Fuzz testing is particularly beneficial in software projects consuming untrusted or complicated inputs such as file parsers, media codecs, network protocols, data compression, code compilers and interpreters, databases, browsers, text editors, or any other kind of user interface. Fuzzing provides a good overall picture of the quality of the target system and software. Using fuzzing, you can easily gauge the system and software's robustness and security risk posture under test.

Fuzzing is typically automated and can be performed using open source tools like [Clusterfuzz](#), [Wfuzz](#), [OSS-Fuzz](#), and [OWASP ZAP](#). Vendors like [Beyond Security](#) and [Synopsys](#) are examples of commercial products for fuzz testing.

Penetration Testing

Penetration testing is a type of technical assessment that simulates cyber-attack scenarios to evaluate the maturity of an organization's information security capability. These scenarios target defects or weaknesses in networks, systems, and application components. Identify whether malicious actors can exploit these vulnerabilities to compromise the resources within the environment. Techniques and toolsets are configured with a limited scope of operation to minimize disruption to the organization's business and operations.

⁷² <https://github.com/google/fuzzing/blob/master/docs/why-fuzz.md> partly inspired from this publication

The amount of information shared before the penetration test can influence the outcome of the assessment. Three methodologies are widely used to conduct this type of test, each with different goals and attack simulations:

- **Black box:** Method of testing that simulates an attack performed by an unprivileged user or outsider. This methodology assumes no prior knowledge of the systems, network, or resources within the environment is given to the tester as part of the assessment.
- **Gray box:** Method of testing that simulates an attack performed by an unprivileged user or insider. This methodology assumes knowledge of the environment's systems, networks, or resources. Or limited information such as credentials has been shared with the tester as part of the assessment.
- **White box:** Method of testing that simulates an attack performed by a privileged user or insider. This methodology assumes explicit and substantial knowledge of the systems, network, and resources within the environment are given to the tester as part of the assessment.

DevSecOps principles intensify the need to continuously perform these assessments to match the pace of the constant development and changes which are implemented on an ongoing basis. However, it is important to carefully plan your approach and target areas not contractually restricted by the rules of engagement.⁷³ Planning is especially needed for resources hosted in the cloud, where you will have to coordinate the test with the Cloud Service Provider (CSP) due to the shared responsibility model. The [CSA Cloud Penetration Testing Playbook](#) provides additional guidance and testing methodologies.

Penetration testing scope

The concept of shared ownership for cloud-hosted systems can greatly impact the scope of activities that can be performed during a penetration testing assessment. Shared ownership varies depending on the adopted cloud service model (e.g., SaaS, IaaS, PaaS) and the security controls that fall under the responsibility of the CSP, which are not usually within the scope of the assessment.

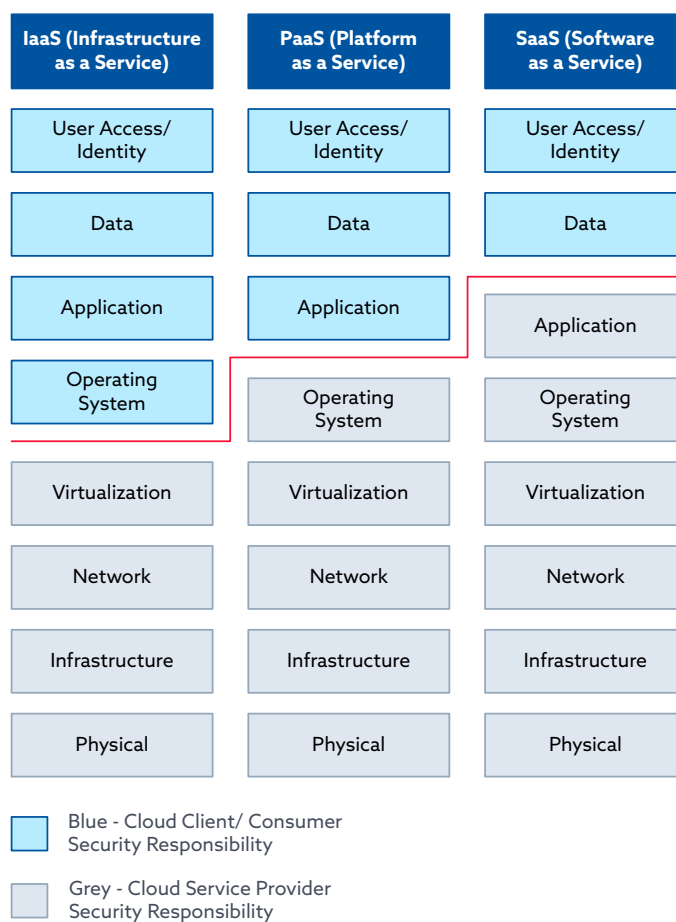


Figure 17: Shared responsibility model

73 [Microsoft Cloud Penetration Testing Rules of Engagement](#)

- **Software as a Service (SaaS):** The testing scope for SaaS environments is minimal. However, testers may be able to assess data or user access controls and exploit excessive user permissions as long as any test of the underlying infrastructure is excluded and explicit permission from the CSP has been granted.
- **Platform as a Service (PaaS):** The testing scope for PaaS environments includes the application layer, all available interfaces, and some platform configurations on top of what is allowed for the SaaS deployment model. All other layers are excluded. However, similar to SaaS environments, explicit permission from the CSP must be granted, especially for public cloud deployment models. Exploiting a flaw may impact other tenants or the provider, if the tester can compromise access. In this case, focus on code reviews and white box testing.
- **Infrastructure as a Service (IaaS):** The testing scope for IaaS environments includes the operating system layer on top of what is allowed for both the PaaS and SaaS deployment models. This scope includes testing any gold standard images used to create VMs, exploiting unpatched vulnerabilities, and abusing remote administration protocols left open to gain access. It's common to use SSH and RDP to access or pivot to other VMs. Still, it can be avoided by coordination with the respective CSP to confirm the rules of engagement. For white box testing, be conscious of your IP address ranges as they may change periodically as the VMs are provisioned and deprovisioned.

Penetration testing steps and defense in depth

As companies adopt a defense in depth strategy to protect their data and most critical assets, attackers have to circumvent many security controls to gain access to the resources in your environment. Implementing multiple layers of defense and countermeasures is a good practice that reduces the likelihood of a single point of failure and protects the confidentiality, availability, and integrity of systems and data.

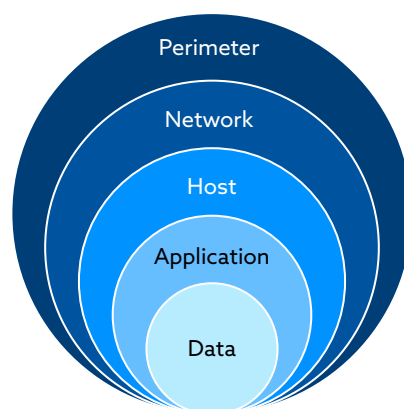


Figure 18: Layers of defense

To achieve your security objectives and protect your data, consider the following steps and targets when pentesting your cloud-hosted resources:

- **Reconnaissance:** Collect information from Open Source Intelligence (OSINT) sources. Be mindful of credentials that might have been hard-coded and stored in code repositories (e.g., GitHub). Social media is often a source of information that can be used to facilitate

sophisticated and targeted social engineering attacks. Inspect the content cloud administrators share on LinkedIn or other platforms to confirm they do not disclose information that malicious actors could use to compromise your environment.

- **Scanning:** Numerous data breaches have been linked to publicly exposed storage containers, exposing sensitive customer and proprietary information to the world on the internet and causing brand and financial damage. Consider scanning for credentials and misconfigured cloud security settings, especially AWS S3 buckets, Azure Blob storage, or similar type of storage that may have been set to “public”.
- **Exploitation:** Exploiting a vulnerability to gain access to a system or resource needs to be thought of as a well-planned strike to confirm an exploitable flaw. Benefit from focusing on cloud-specific technology and authentication attacks. Perform brute force and session attacks on exposed management interfaces, API consoles, web-based elements, and instance services to yield the highest value. Target credentials and cryptographic keys to gain access and bypass security controls.
- **Pivoting:** Pivoting is fundamental to a successful cyber attack. After gaining initial access, a malicious actor may need to move through the network to access sensitive data or high-value assets. Be mindful of subnets’ non-stateful access control lists, which may be over permissive by default and ease lateral movement as they allow resources to communicate with other subnets in the environment.
- **Exfiltration:** Data exfiltration is usually the primary goal of malicious actors. Stealing or encrypting data is usually related to financial gain, while disclosing it is meant to damage reputation. When conducting a black box or red team assessment, consider exfiltrating the data to a storage container in the cloud to reduce the risk of exposing sensitive information or storing it in an uncontrolled environment your organization may not have visibility.

Penetration testing benefits, frameworks, guidance, and playbooks

This type of technical assessment is often a requirement for regulatory compliance. It is used to assess the resilience of systems and applications and to evaluate the effectiveness of an organization’s security policies, security awareness, and ability to detect, prevent or reduce the impact of security incidents. It is an essential part of any security program that can improve the reliability, integrity, and security of on-premise and cloud-hosted systems.

Vulnerabilities identified as part of a penetration test assessment need to be qualified and prioritized for the more serious threats to be treated as a priority pragmatically and cost-effectively. Before advising whether a risk should be avoided, remediated, mitigated, transferred, or accepted, the tester needs to evaluate the associated impact. As mentioned in the [Threat Modeling](#) section, the [DREAD](#) model can be used to assess the severity of a computer security threat. In contrast, the [STRIDE](#) model can be used to classify and define it and answer the question of “what can go wrong?”. Consider the following penetration testing guidance and frameworks to gain as many benefits from this activity as possible:

- **NCSC [Secure Development and Deployment Guidance](#):** Contains high-level principles intended to establish a set of working practices to embed security and make code more stable and easy to maintain. This guidance also includes a range of security tools that can transform into an automated testing suite to assess builds as part of a deployment pipeline,

allowing for some test scenarios malicious actors could use continually. The NCSC also developed a [Penetration Testing](#) guidance that defines how to use penetration testing effectively to get the most out of this activity.

- **MITRE [Common Attack Pattern Enumeration and Classification](#):** CAPEC is an attack library that provides a dictionary of known patterns employed by malicious actors to exploit computer system vulnerabilities. Understanding how malicious actors to design and execute cyber attacks is essential to defending your environment effectively. This library is organized into attack domains, including supply-chain, software, and hardware attacks. Each pattern describes attack strategies, technical requirements, and ways to mitigate the attack, along with related weaknesses.
- **CSA [Cloud Penetration Testing Playbook](#):** This playbook focuses on the testing of systems and services hosted in public cloud environments. It provides a foundation for a public cloud penetration testing methodology and promotes the study of the [Security Guidance for Critical Areas of Focus in Cloud Computing](#) published by the CSA to support the management and mitigation of the risks associated with the adoption of cloud computing technology.
- **CREST [Penetration Testing Programme](#):** This guide provides practical advice for conducting a well-scoped, cost-effective penetration testing activity. It is designed to help organizations test consistently with a three-stage program: preparation, testing, and follow-up. The preparation stage details the testing drivers, purpose, and targets. The testing stage establishes an effective testing process, planning, and activity management tips. The follow-up stage provides guidance to remediate identified vulnerabilities actively and delivers the agreed action plan.
- **PTES [Penetration Testing Execution Standard](#):** This standard provides a structure that covers seven sections, initial communication, intelligence gathering, threat modeling, vulnerability analysis, exploitation, post-exploitation, and reporting. This guidance captures the entire penetration testing process to help better understand the organization and provide directions to support the testing execution. This standard may be used with the PTES [Technical Guidelines](#) to help identify the right procedures and tools depending on the scenario.
- **NIST [Information Security Testing and Assessment](#):** This guide provides examination methods and techniques to conduct technical testing as part of an information security assessment. It offers suggestions to prepare a robust planning process, root cause analysis, and reporting. It also includes insights on the execution process for the tester to identify, validate, and assess the exploitable weaknesses of a system and verify security controls in place.

Special use cases to consider in a DevSecOps environment

When conducting penetration testing in the context of DevSecOps, it is also important to consider specific scenarios which may be interesting from the point of view of the assessment. Consider the following scenarios as they may be overlooked or often not considered:

- **Attacking the CI/CD orchestrators:** With automation and using CI/CD orchestrators, it is important to consider penetration testing on the orchestrator itself. The scope of the test will be the orchestrator itself and the source code control application. The goal of this type of test is the following:
 - Confirm access control in the CI/CD orchestrator does not allow unauthorized jobs/runs which may deploy malicious infrastructure.

- Verify that secrets cannot be extracted by running malicious jobs in the orchestrators.
- Confirm branch policies do not allow unauthorized commits and pushes.
- Evaluate vulnerabilities within the orchestrator.
- **Attacking container orchestrators:** When pentesting container orchestrators, the scope of the test should include all the containers in the orchestrator, and the ultimate goal of this test should be, at the least, to perform a container breakout or to extract secrets and sensitive information.
- **Attacking cloud infrastructure:** When attacking cloud infrastructure, it is important to consider that although escalating privileges to administrators can be a goal, considering the characteristics of cloud infrastructure, compromising sensitive data, finding public storage or creating persistence from external accounts are also variables to consider. Therefore, it is important to agree on the penetration testing goals accordingly.

Penetration testing tools

Numerous tools and automation engines can support the identification and analysis of security weaknesses in computer systems. Incorporate controls for the [OWASP Top 10](#) as part of your penetration testing assessment to defend your systems from the most critical security risks. The CSA's [Top Threats to Cloud Computing](#) provides a list of relevant cloud security threats and their direct business impact. Consider some of the following tools to automate your testing activities and evaluate your DevSecOps program against the most common and severe security weaknesses:

- [bucket finder](#) - Brute force tool that scans across the Amazon S3 domain to determine if a bucket name exists and identifies whether it is public, private or a redirect.
- [Chef InSpec](#) - Open-source tool used to automate security testing and validate compliance against custom or pre-defined tests from the Chef Supermarket to audit your applications, dependencies, and infrastructure.
- [Gauntlt](#) - Automated attack platform that enables the creation of actionable tests that can hook into your deployment pipeline to assess your code using common penetration testing tools.
- [pacu](#) - Open-source AWS exploitation framework designed for offensive security testing to exploit configuration flaws and vulnerabilities within AWS environments.
- [MicroBurst](#) - Collection of scripts used to evaluate the security of Azure environments by assessing and exploiting weak configurations to enumerate services, resources, and credentials.
- [OWASP OWTF](#) - Collection of offensive web testing tools integrated to automate and improve the efficiency of penetration testing assessments in alignment with industry-recognized standards such as the OWASP Testing Guide, PTES, and NIST.
- [prowler](#) - Open-source AWS security best practice assessment tool used to evaluate compliance against various frameworks such as CIS, PCI-DSS, ISO27001, GDPR, HIPAA, FFIEC, SOC2, AWS FTR, and ENS.

A more comprehensive register of example DevSecOps testing tools is available on the [Awesome/DevSecOps](#) GitHub repository.

Container Testing

This section highlights the importance of testing the security of containers and container orchestrators in a production-like environment, as configurations related to container orchestrators and containers may be easily missed. Thus, it is important to develop a set of tests that can help to verify the configuration of the specific containers and container orchestrators. In particular, for testing containers, we recommend creating test cases that allow testing the configuration of container hosts and container orchestrators for potential security misconfigurations. Some of the tests we recommend are:

- Verify that insecure containers cannot be deployed in a production cluster. An example for Kubernetes could be using the bad pod's scenarios⁷⁴ to ensure that privileged pods cannot be created.
- Test containers with root users.
- Run Docker bench⁷⁵ and Kubebench⁷⁶ to ensure that best security practices from the CIS benchmark are run.
- If there are specific requirements for the containers which require elevated privileges, such as network or sysadmin capabilities, make sure that the applications hosted in the container are not vulnerable by following the testing guidelines in this document.
- Ensure that runtime monitoring is enabled on clusters and containers to understand the processes running in each component of the application infrastructure.
- Whenever penetration tests are scheduled, ensure that if the containerized infrastructure is involved, scenarios involving container escapes are tested to ensure that compromised containers cannot easily be escaped.

Effective container testing is reliant on understanding hardening requirements addressed in [Container Hardening](#). While container hardening and testing can take time to implement, using vulnerability scanning on containers will help identify exploitable vulnerabilities on hosted images and libraries.

A container vulnerability scanner specifically targets container images, filesystems, Git repositories, and Kubernetes clusters to identify OS packages and software dependencies, known vulnerabilities, Infrastructure as Code (IaC) misconfigurations, and hard-coded secrets. Cloud native services on AWS are an example of achieving this on its Elastic Container Registry, equally licensed vendors like Trivy can provide a similar service.

While [container hardening](#) can be challenging to perform and obscure in measuring the effectiveness of container security controls, the isolation of testing allows teams to understand the level of the exploitability of their containers. Tests can either be run manually as isolated activities or can be integrated into penetration tests.

⁷⁴ <https://github.com/BishopFox/badPods>

⁷⁵ <https://github.com/docker/docker-bench-security>

⁷⁶ <https://github.com/aquasecurity/kube-bench>

Integrity Checking

Integrity checking is managing, signing, and verifying artifacts (e.g., containers and software packages). The components used as the output of a build should be signed and trusted. Integrity checking guarantees consumers the integrity of these components. Signing artifacts and metadata allows consumers to better manage vulnerabilities in components deployed in their systems more efficiently and helps generate accurate SBOMs. Organizations typically manage the integrity of artifacts, sign container image manifest files, configuration files, and package and application artifacts.

While using artifact registries and managers don't provide direct security controls, they provide the basis to manage the integrity of artifacts and enable teams to target security activities like hardening and vulnerability scanning and remediation. For instance, vulnerability scanned and hardened instances like containers, manifest, and configuration files are uploaded to the artifact repository as a secure package and can be used as a reusable approved template.

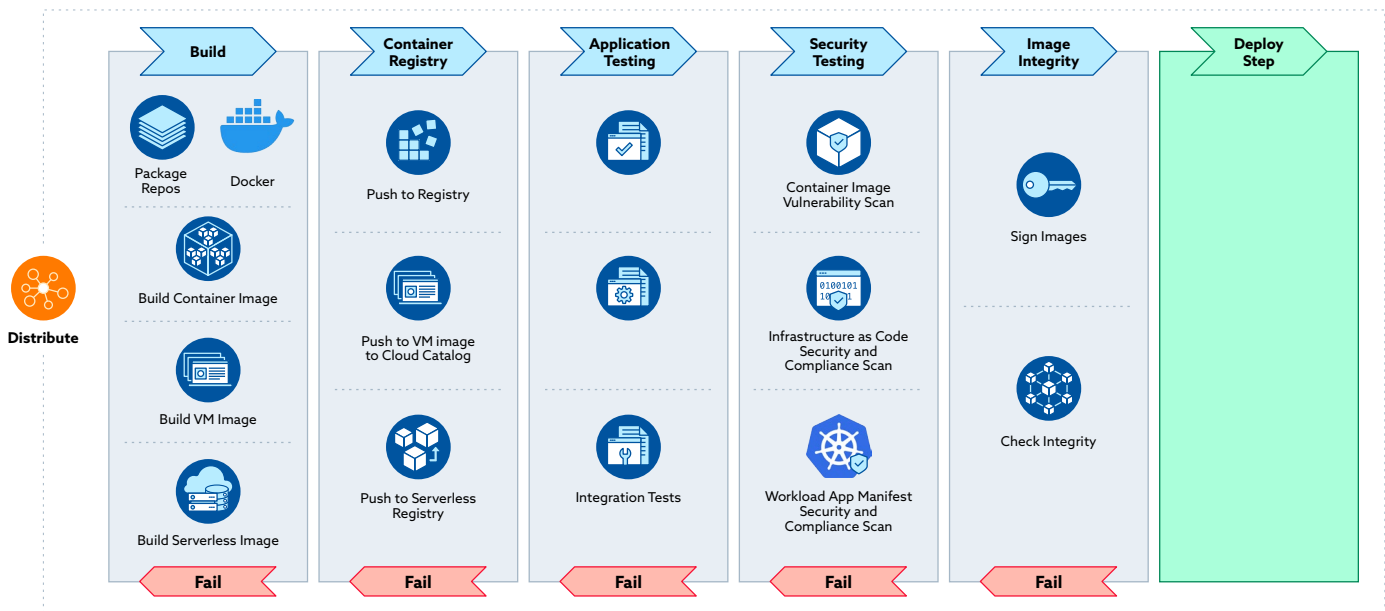


Figure 19: Example workflow of using an artifact registry⁷⁷

[Sigstore](#) and [JFrog Artifactory](#) are example tools to build an artifact registry to ensure the integrity of components. Sigstore is a public software and transparency service, similar to Certificate Transparency for SSL certificates, which can sign containers using Cosign and store the signatures in an OCI repository⁷⁸. JFrog Artifactory hosts and manages artifacts, binaries, packages, files, containers, and components.

Another alternative proprietary solution is [Google Cloud's Binary Authorization](#) service which allows the signing of container images and admission control to prevent unsigned, out-of-date, or vulnerable images from reaching production at runtime.

⁷⁷ [tag-security/CNCF cloud-native-security-whitepaper-May2022-v2.pdf](#) at main

⁷⁸ See the Open Container Initiative website for more information: <https://opencontainers.org/>



Delivery and Deployment

Pre-deployment safety checks to ensure that an application/product is deployed onto secure infrastructure. Without including security in deployment, there is a risk that vulnerabilities and poor security practices weaken an application/product, exploiting it for attacks in production environments.

GitOps⁷⁹

An evolution of IaC; GitOps is a paradigm for implementing continuous deployment that takes version control, collaboration, compliance, and CI/CD tooling and applies them to infrastructure automation.

While key SDLC activities have been automated, infrastructure has remained a largely manual process that requires specialized teams. With the demands made on today's infrastructure, it has become increasingly crucial to implement infrastructure automation. Modern infrastructure needs to be elastic so that it can effectively manage cloud resources that are needed for continuous deployments.

GitOps is used to automate the process of provisioning infrastructure. Similar to how teams use application source code, operations teams that adopt GitOps use configuration files stored as code (infrastructure as code). GitOps configuration files generate the same infrastructure environment every time it's deployed, just as application source code generates the same application binaries every time it's built.

GitOps = IaC + MRs + CI/CD

IaC: GitOps uses a Git repository as the single source of truth for infrastructure definitions.

MRs: GitOps uses merge requests (MRs) as the change mechanism for all infrastructure updates. A merge commits to your main (or trunk) branch and serves as an audit log.

CI/CD: GitOps automates infrastructure updates using a Git workflow with continuous integration (CI) and delivery (CI/CD). When new code is merged, the CI/CD pipeline enacts the change in the environment. Any configuration drift, such as manual changes or errors, is overwritten by GitOps automation, so the environment converges on the desired state defined in Git.

Traditional processes mostly rely on human operational knowledge, expertise, and actions performed manually, but in the case of GitOps, all changes are interactions with the Git repository. The Git repository and GitOps process become crucial to secure. The immutability of infrastructure protects teams from making changes from outside the main deployment process. This makes it easier to detect and reverse environment changes based on the declarative state in the Git repository. Using GitOps, teams reduce the number of people and components that have access to production infrastructure.⁸⁰

⁷⁹ <https://about.gitlab.com/topics/gitops/>, Entire indent section from gitlab

⁸⁰ <https://github.com/cncf/tag-security/tree/main/security-whitepaper/v2> page 43, last paragraph

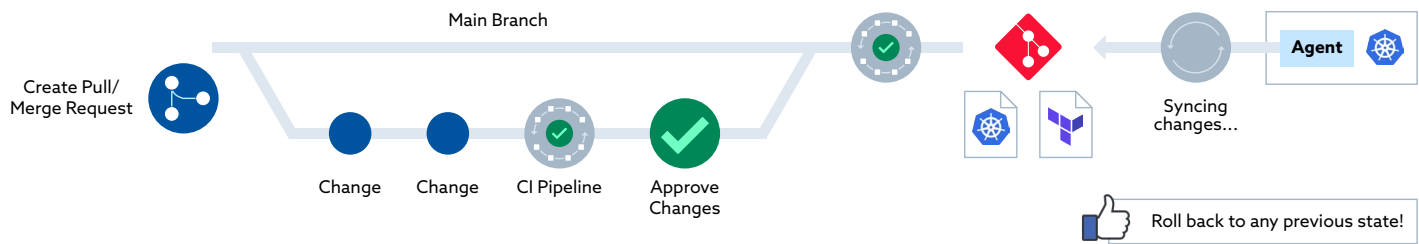


Figure 20: GitOps workflow

GitOps increases the management of securing infrastructure by:

- **Limiting manual operations:** less manual work when deploying infrastructure changes.
- **Auditing changes:** changes are tracked in the respective git repo.
- **Providing a single source of truth:** centralized management and deployment of IaC.
- **Enforcing policy:** controls and gates on processes for security requirements.

Guardrails

A guardrail is a high-level rule that provides a baseline for your cloud environment. Guardrails are implemented preventive or detective controls that help govern resources and monitor compliance across resources. The concept of guardrails is discussed in greater detail in the *DevSecOps Pillar 4 - Bridging Compliance and Development* paper.

Guardrails are integrated tools and, in the best case, automated in the software development process to ensure alignment with the organization's goals and objectives. Guardrails establish a baseline that can constantly monitor deployments. These baselines represent high-level rules with detective and preventive policies applied. Guardrails may be implemented as a means of compliance reporting (e.g., number of machines running OS in the currently-approved image list) or as an enforced/auto-remediating set of controls (e.g., devices running any OS other than those approved are automatically shut down).

A focus on in-line guardrails and feedback within the DevOps pipeline—achieved through tightly integrated tools and processes—can shift the approach to compliance from point-in-time to continuous. Consideration must be given to ensure the tools and techniques are implemented to align with the agreed compliance objectives. Evidence must be produced easily to achieve external validation of these controls through internal and external governance. Guidance for building effective guardrails⁸¹ is as follows:

1. **Sensible:** Confusing or misunderstood guardrails could cause the guardrails to be ignored or to be followed in a checklist manner.
2. **Transparent:** Standards and policies governing how teams should behave need to be explicit. Guardrails that are unknown will only be followed by accident and will not be effective.
3. **Challengeable:** Guardrails make sense within the context they were established. When the

81 <https://tcagley.wordpress.com/2019/03/12/five-guardrails-for-guardrails/>

context changes, the teams using the guardrails need to be able to challenge the validity/ need of the guardrail.

4. **Enforced:** Breaking or ignoring guardrails needs to have consequences. Guardrails exist to keep teams aligned to the goals of the organization and channel behavior in the right direction.
5. **Tone at the Top:** Guardrails provide a safe place for teams to make decisions without micromanaging. Once guardrails are established, leaders need to respect the “void” and let the team decide and self-organize to solve the problem they are addressing.

“The Six Pillars of DevSecOps: Automation” discusses “the implementation of a framework for security automation and programmatic execution and monitoring of security controls to identify, protect, detect, respond, and recover from cyber threats” as it pertains to securing code, applications, and environments. The output of these automated processes provides the evidence required to meet compliance objectives. In DevSecOps, the deployment pipeline can consume or even build environments that align with compliance objectives. This may be a combination of monitoring and enforcing controls and will likely vary depending on the environment, data classification, and business risk (i.e., development vs. production). When building guardrails⁸², consider the following:

- **Define the problem:** Define the business outcome - preventing it from becoming a blocker - and then move on to the technical problem.
- **Set the scope:** Multiple cloud accounts are now the standard. Determine guardrail scope and technical specification. Multiple accounts may affect where you run the guardrail and manage the required IAM privileges.
- **Pick the deployment model:** There are technical specifics regarding where and how to run your guardrail. Balancing centralized and localized guardrails requires key decisions for a larger, enterprise-class deployment.
- **Define filters:** Basic filters might work on a simple resource tag, while more complex filters might evaluate a combination of include and exclude rules for greater flexibility.
- **Determine triggers:** The main categories are time-based assessments and event-based triggers. Event-based for more deterministic events and time-based sweeps to catch conditions missed purely on an event trigger. Based on the trigger, collect the information needed and, in combination with the scope and filters, make a decision.
- **Send notifications:** Establish alerts for guardrail creation as email/text/Slack/ticketing system or export to a SIEM.
- **Validate and log:** Reviewing and verifying the guardrails’ effectiveness is vital. Log guardrail metrics and incidents, even if the guardrail fixed everything within seconds. Validating and logging keep a consistent record and measure mean resolution times.

Security objectives should be integrated directly into the deployment pipeline, with a target state defined in code, templates, and guardrails. The goal should be an automated audit trail for ease of review and relevant security metrics for continuous review by both the DevOps and compliance teams to drive security decisions. An example of applying guardrails to cloud environments is seen in **Figure 21** below. Cloud constraints are applied (detective and preventative) before building and provisioning accounts and environments.

82 <https://devops.com/building-great-cloud-security-guardrails/>



Figure 21: Cisco's example of AWS guardrails⁸³

Tailor guardrails to an organization's compliance requirements and risk appetite. The success criteria for guardrails rely on three fundamental principles identified below, supported by an example set of control categories in Table 2. For a lower-level set of cloud security controls, see the Cloud Control Matrix v4⁸⁴.

- Understand the varying degrees of assurance from informational/feedback, vulnerability reporting, or tracking compliance deviation on infrastructure deployments (e.g., using the approved image for a VM) to enforce/blocking (guardrails blocking unapproved VM image to run). This understanding helps strike a balance between security and developer flexibility based on context (i.e., development or production environments) and associated risk appetite.
- Define a desired target security state or Key Performance Indicators (KPIs); for example, patching, Service Level Agreements (SLAs), vulnerability assessment, and remediation time. The desired target state will be dependent on the organization's security and risk appetite, the context of its cloud and application landscape, and regulatory requirements. The "Pillar 6: 'Measure, Monitor, Report, and Action' paper" will go into more detail on defining a desired target security state.
- Establish Segregation of Duties (SoD) for production environments where at least two individuals are required to make changes to a critical feature of an application to prevent fraud and error. For environments where proof of concepts are performed (i.e., development and test), SoD controls can be relaxed to allow developers to freely create features with fewer security controls that could inhibit productivity. This developer freedom assumes that the associated environments do not have access to production data. The implementation of SoDs can begin in pre-production and quality assurance environments to ensure that

83 [DevSecOps: Security at the Speed of Business - Cisco Blogs](#)

84 <https://cloudsecurityalliance.org/artifacts/cloud-controls-matrix-v4/>

security controls and application features work as intended. Application owners will also need to ensure application availability can be restored using an SoD approach in an incident.

While guardrails can be logistically challenging to design and implement, they offer the following benefits that transform an organization's approach to cloud security,^{85 86}. Guardrails:

- Define a range of acceptable behavior. The range between the guardrails allows teams and individuals to determine their course.
- Speed up the decision process. Guardrails provide a range of solutions to a question and a starting point for decisions and cut down the time needed to make a design or security decision allowing developers to operate at speed without requiring active involvement from the security team.
- Provide teams and individuals independence to make business decisions based on context without sacrificing alignment. One standard tenet of efficient software development (all forms) is that the business defines what needs to be done. The "what" defines the business problem the developers (broadest usage) need to solve; these are guardrails. The technical team then defines how they will solve the business problem. Technical and architectural standards and guardrails provide an environment that allows the team to make decisions without asking for permission.
- Allow product teams to scale by automating security controls in developer workflows. This automation assures enforcement without having to perform reviews and track remediation of vulnerabilities manually.
- Reduce risk. The architectural guideline provides the same service for software developers. The goal of the guardrail is to reduce the possibility of rework based on research and experiments performed by others so that as much time as possible is spent solving business problems.
- Reduce the risk that development teams will ignore and bypass security when the controls are baked into the development process.

Example cloud native guidance is available; for example, [AWS](#) provides mandatory, elective, and strongly recommended guardrails. Whereas licensed offerings like [Guardrails.io](#) help establish guardrails in application development and build.

85 <https://tcagley.wordpress.com/2019/03/14/guardrails-in-decision-making/>

86 <https://www.secureworld.io/industry-news/security-guardrails-matter-appsec>

Environment Separation

Environments in the context of this white paper are collections of different technological components that serve a specific purpose. Examples of environments include different network segments and, for example, the existing development environments used in development (e.g., development, testing, staging, and production). Acknowledging the broadness of this definition, a relevant categorization is defined from the point of view of DevSecOps in the cloud.



Figure 22. Environment Layers

Figure 22 categorizes the different environments in a modern cloud-based DevSecOps environment. The above characterization explains each layer's separation (or isolation) requirements. For example, the segregation required between development environments (development, testing, staging, production, sandboxes), networking segments, logical resource containers (e.g., resource groups in the most common CSPs), and any multi-tenancy requirements that need to be satisfied by microservice-supporting infrastructure (e.g., Kubernetes, Nomad).

Following the model in **Figure 22**, specific separation baselines and considerations are presented for each layer, providing practical guidelines and considerations that may help enterprise architects, developers, and organizations implement an environment separation strategy tailored to cloud environments DevSecOps.

Organizational Hierarchy: At the outermost layer considered in an organizational hierarchy. This environment represents the organization at the highest level and its components will be the different organizational entities that compose it. In practice, this layer is implemented by cloud providers to ensure a modularized structure that allows for segregation of duties, controlled billing, and flexibility when implementing security controls, such as access requirements.^{87 88 89} At an organizational level, the main concern is establishing the boundaries between different organizational units, sub-organizations, and third-party organizations. The goal is to ensure that each organizational unit can only access what it should be able to access (i.e., the least privilege principle). Coordinating properly with each organizational unit and defining roles and responsibilities is important. Once

87 <https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations>

88 <https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/organize-subscriptions>

89 <https://aws.amazon.com/organizations/getting-started/best-practices/>

these are defined, processes and technical means (i.e., automatically enforced policies and access controls) should be developed and implemented within the organizations to ensure that the defined boundaries are respected. Notice that although this level of separation may be a better fit in enterprise architecture guidelines, it is impossible to provide appropriate segregation on the lower layers if these components have not been appropriately defined. Separation can be ensured using the technical means provided by different cloud providers. For example, in AWS, organizational units⁹⁰ and Service Control Policies⁹¹ (SCPs) can be used to manage permissions within the organization as well as to create hierarchical organization structures; using the case of Azure, management groups⁹² can be used to model hierarchical organizational structures and to ensure that policies are applied depending on the requirements of each group. In Google Cloud, folders⁹³ allow organizations to create hierarchical structures and organize different projects.

Development Teams: Developers will create and host and operate their own applications. This layer is composed of the different development environments used throughout the end-to-end development lifecycle. In the case of development teams, isolation guarantees full ownership over their managed applications and systems. This level of separation is to ensure that each team has its working area that should not be affected by the work of other teams. Do keep in mind that this recommendation does not advocate for siloed teams. Rather, the different levels of permissions between teams' environments are granted appropriately. Similar to the layer above, the separation here consists of defining the specific roles and responsibilities for each developer team and ensuring the processes and technical means are defined to ensure the appropriate level of separation is maintained and enforced. These teams can be modeled as specific organizational units which contain specific resources.

Development Environments: Represents the usual environments used for developing applications: dev, test, staging, production, sandboxes, and shared. These environments will have distinct features that enable developers and ensure that software development occurs as intended. Segregation at this level aims to introduce isolation so that workloads can be tested throughout the SDLC without compromising production data and infrastructure. Mature organizations typically have six different (shared) environments:

- **Development:** Where most of the development occurs. In this environment, developers can modify, test, and try code changes without breaking any functionalities in live applications and systems. New features and updates are first tried in this environment to ensure the stability of live applications. From a security perspective, changes to this environment must affect no live customer data or infrastructure. For example, a dev web application should not be able to modify production databases.
- **Testing:** Where most of the testing happens. While some tests can be run in the development environment as part of the building phase (e.g., unit tests), most integration and functional tests happen in the testing environment. As with the development environment, it is recommended that no live customer data or infrastructure is stored or modified in this environment.

90 https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_ous.html

91 https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html

92 <https://docs.microsoft.com/en-us/azure/governance/management-groups/overview>

93 <https://cloud.google.com/resource-manager/docs/creating-managing-folders>

- **Staging:** A replica of the production environment. This environment should be used to perform any additional tests that may be required (end-to-end tests, regression tests, load tests, and some penetration tests). This environment aims to provide an isolated environment as close to production as possible to ensure that the developed software artifact behaves as intended.
- **Production:** Production is the environment where customer data and the application live. The isolation requirements for this environment are critical to ensure security. For this reason, it is recommended that human access to this environment is limited to “break glass” emergency procedures. All environment changes are performed via automated, auditable tools like CI/CD orchestrators.
- **Sandboxes:** Special environments that focus on experimentation and testing new technologies and architectures. These environments are usually volatile, and isolation is required to ensure that no other environments are affected by any potential changes introduced in the environment.
- **Shared resources** - An environment that will host specific to one needs sharable resources, such as authentication or web proxies, CI/CD tools, specific security controls, VPN endpoints, internal API gateways, or acting as a data collector account for logs or events from other accounts.

Different approaches can be used to ensure that this separation level is maintained. For example, different accounts or subscriptions within an organizational unit can be used to represent each of the environments. In this way, logical isolation can be maintained for the different environments. However, it is also important to remember that separating these development environments should also consider segregation at the lower layers. For example, connecting a production database with a test web application should not be possible.

Workload Accounts: CSP accounts can also provide hard boundaries for isolating workloads that process data of different sensitivity levels. Using separate workload accounts ensures that dev workloads resources can’t access prod workload resources and vice-versa and minimizes possible blast radius to that workload account. Workloads are the collection of resources and code that delivers business value, such as a customer-facing application or a backend process.⁹⁴ The accounts may differ in complexity, scope, and purpose. Workload accounts will help support SDLC processes by providing an account level separation between different types of environments: dev, test, staging, or production.⁹⁵

Resource groups: In the cloud, resources are assets that can represent infrastructure, servers, and hosting runtimes. Resource groups are a way of logically grouping artifacts, applications, systems, and different components. Notice that we are using a generic definition for the resource groups that, although related, should not be confused with the ones used by different cloud providers such as AWS⁹⁶ and Azure⁹⁷. Segregation between resource groups allows for keeping track of clusters of resources that are related. It is a logical layer of separation, but one that allows to keep track and ensure

94 <https://docs.aws.amazon.com/wellarchitected/latest/userguide/workloads.html>

95 <https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/descriptions-of-example-purposes-of-workload-environments.htm>

96 <https://docs.aws.amazon.com/ARG/latest/userguide/resource-groups.html>

97 <https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal>

visibility of the different components of the designed solution. As with the development environments above, it is a good practice to ensure that resource groups are isolated (at a network level).

Resources: We consider resources the base layer for applications and solutions. As mentioned above, they represent databases, serverless functions, web apps, and all the different types of services offered in the cloud (be they private or public). Segregation at this level considers aspects such as network-level segregation, access controls of human and non-human accounts to different resources, and tenancy requirements of the implemented solution.

- **Network segmentation:** The decisions and activities performed to segregate different systems at a network level using different techniques ranging from physically separating them (e.g., firewalls, routers, etc.) or using software-defined solutions like web application firewalls (WAFs), container network interfaces (CNIs). Security practices on this level include approaches such as micro-segmentation and zero-trust at a network level.
- **Access controls:** Ensures proper authorization and authentication to resources ensures separation of duties. The goal is to follow least privilege, ensuring that access is only granted as needed and when needed (i.e., just in time). There are several ways to ensure access control at the level of resources. From a high-level perspective, some approaches include role-based access control⁹⁸, where roles with specific permissions are defined and assigned, and attribute-based access control⁹⁹, where specific subject attributes are evaluated to obtain the permissions.
- **Tenancy requirements:** In containerized infrastructures hosted using container orchestrators (e.g., Kubernetes, Docker Swarm, Nomad, Openshift, among others). It is often common to have multi-tenant environments where solutions that may be unrelated are hosted in the same cluster with some sort of logical separation used (e.g., namespaces). On top of this, these applications often have specific security requirements, which may require specific segregation techniques related to CNIs, or Web Application Firewalls (WAFs).

Applications: Applications are software programs hosted in resources or live in third-party infrastructure and consumed as Software as a Service. Separation at this level requires that applications cannot access unrelated resources or other applications and ensure that access is only granted as required by the application. In this sense, access control mechanisms play a critical role.

With environment separation being process and design leveraged, it does not require proprietary tooling to implement exclusively. Additionally, with strong environment separation, organizations can consider Zero Trust Tenets¹⁰⁰ when defining access to these environments for users, groups, resources, and assets. While there are several reasons for this, the most compelling one tends to be that, in most cases, organizations will use commercial CSPs (traffic travels over the internet and becomes public), increasing the attack surface of the organization's IT infrastructure. This is coupled with new challenges to identity and access control in authentication and authorization, as well as the trust requirements for developers and non-human accounts.

98 <https://csrc.nist.gov/glossary/term/rbac>

99 https://csrc.nist.gov/glossary/term/attribute_based_access_control

100 <https://cloudsecurityalliance.org/artifacts/towards-a-zero-trust-architecture/>

The aim is to avoid inherited environmental trust by implementing Zero Trust principles: micro-segmentation through least-privilege, assume breach and continuous verification. The access should only be granted after positive authentication and authorization for the fixed duration based on a dynamic policy and other behavioral and environmental attributes.

Secrets & Key Management

Secrets Management: Secrets management is the practice of managing access credentials for secrets. A secret is a key-value pair that users and system components use to authenticate and authorize services. A major issue with secrets management is exposed secrets in version-controlled source code and configuration files. Secrets are often both generated and encrypted with various cryptographic functions. Secrets management systems aim to reduce the risk of exposing secrets. A secrets management system manages the lifecycle of secrets and provides secure storage and transfer of secrets¹⁰¹. Examples of secrets are user credentials and machine-to-machine access credentials. User credentials authenticate users to systems, including passwords, SSH keys, and one-time passwords. Machine-to-machine credentials include API access tokens and web tokens.

Key Management: Secrets management also encompasses key management. Key management is the practice of creating, storing, using, rotating, and destroying cryptographic key material. A cryptographic key is a string of bits used by a cryptographic algorithm to transform plain text into cipher text or vice versa. Symmetric keys are used to symmetrically encrypt plaintext data and decrypt ciphertext data. Asymmetric keys (also called public keys) use private and public keys to create, manage and revoke digital certificates, signing and verifying digital signatures for encryption in transit like TLS/SSL. Within cloud environments, a key management system provides a mechanism for securely storing encryption keys for encryption and digital signature algorithms. The NIST SP 800-57¹⁰² series provides key management guidance for cloud environments. The publication describes recommendations and best practices for cryptographic algorithms and usage of public key infrastructures, including definitions of the security services that may be provided when using cryptography and the algorithms and key types employed. The recommendation also applies to managing keys.

Secrets & Key Management Lifecycle: Within cloud environments, secrets and keys are often managed in a vault. A vault provides secure storage of generic secrets like passwords, databases, keys, and certificates. Within a vault, a secret may undergo one or several stages during the secret management lifecycle, which has these phases:

- **Creation:** The phase when a secret or key is generated. In this phase, secrets management systems ensure that secrets meet sufficient complexity requirements.
- **Activation:** A secret management system may have the secret in a pre-activated state activation state before activation. During the phase, it has been generated but not been authorized for use. In the activation phase, secrets are then enabled for usage.
- **Rotation:** A secret is updated or replaced during the rotation phase. A secret management

101 Blomqvist, Markus, Lauri Koivunen, and Tuomas Mäkilä. "Secrets Management in a Multi-Cloud Kubernetes Environment." (2021).

102 Barker, E., Barker, W., Burr, W., Polk, W., Smid, M., & Gallagher, P. D. (2020). NIST Special Publication 800-57 Recommendation for Key Management—Part 1: General.

- system allows automatically rotating secrets without user interaction during a certain period.
- **Revocation:** When a secret has been compromised or is no longer needed before its expiry date, it is revoked.
- **Expiration:** Within a secret management system, a secret can be configured to be deactivated to prevent the secrets from being used for authentication. Within cloud environments, secrets can be short-lived and expire after a defined time.
- **Destruction:** Destruction is the phase of destroying and removal of secrets or keys by the secrets management system. Secrets management tools ensure that actions are audited per secret, including destroying a secret.

The management of secrets through a secrets management system has several benefits for the pragmatic implementation of DevSecOps practices¹⁰³:

- **Centralization:** Secrets management system supports centralizing the administration of various secrets distributed across cloud environments. Secrets management aims to secure the secrets at rest and while in transit.
- **Short-lived keys & secrets:** Secrets, keys, and certificates can be configured to be short-lived with a defined expiration time and be revoked after its expiration or in case of compromise.
- **Auditing & Logging:** Secrets management ensures that all actions across the lifecycle are audited and logged. Storing audit logs of the secrets management solutions helps to locate unauthorized access issues and keep important information for incident analysis.
- **Cross-platform access:** secrets can be made available across different platforms and all phases of systems development, including local development, automated builds, staging, and production environments. Using multi-cloud capable secrets management solutions enables a complete, managed solution to handle the entire lifecycle of the secrets and their assignments to users and machines.

A pragmatic approach for effective secrets and key management is to use the secrets management solution that CSPs offer; however, licensed services can help reduce CSP risks of managing and hosting secrets. Cloud service providers offer secret and key management systems or both as a stand-alone solution and for multi-cloud purposes. A secret management system allows the centralization of secrets and provides granular role-based access control and capabilities to configure secret access policies to trusted identities only. Some example services are:

- **GCP Secrets Manager:** In GCP, the Secret Manager service allows managing secrets for API keys, passwords, certificates, and other sensitive data.
- **AWS Secrets Manager:** In AWS, the Secrets Manager encrypts the protected text of a secret using the AWS Key Management service for key storage and encryption.
- **Azure Key Vault:** For Azure, Microsoft provides the Azure Key Vault service for managing and rotating secrets, certificates, and cryptographic keys in Azure. A popular stand-alone solution for managing secrets across multiple cloud providers and on-premise infrastructure is Vault by Hashicorp. Vault particularly aims at managing and rotating machine-to-machine secrets across multi-cloud deployments.
- **HashiCorp Vault:** Hashicorp Vault is a multi-cloud secret & key management solution that an organization can use to centralize secret management across hybrid and multi-cloud

¹⁰³ <https://www.hashicorp.com/resources/unlocking-the-cloud-operating-model-security>, "What is Identity-based Security", 11/05/2019

environments. A major feature of HashiCorp Vault is managing dynamic secrets, which are secrets that are generated when they are accessed. Vault enables the management of dynamic secrets and keys along the secrets management lifecycle.

- **Cloud HSM services:** Cloud Hardware security modules (HSM) are key management services that allow hosting encryption keys and performing cryptographic operations in a cluster of FIPS 140-2 Level 3 certified HSMs. These services allow the creation and usage of cryptographic keys within the respective cloud provider.

Overall, using secret and key management systems for cloud environments are efficient DevSecOps practices and allow a shift from manual security operations to a more automated culture. For more on key management, refer to the CSA's release on [Key Management on Cloud Services](#).

Securing the CI/CD Pipeline

Continuous integration, delivery, and deployment are approaches to the delivery and deployment of products. These approaches allow regular code commits that trigger builds and run tests with the deployment minimizing the need for manual processes.

CI/CD pipelines allow faster, more diverse, and flexible delivery. However, adding the CI/CD pipeline to the technology stack introduces an attack surface that can be exploited, for example:

- Insecure code imported into a CI/CD pipeline from a third-party source
- Unauthorized access to source code repositories
- A breach of a dev/test environment where an attacker disables security tests
- Insecure secrets management within the pipeline.

The security of this process is critical if you need to protect the integrity of your code and the systems it builds. Security should, however, work with this process, not hinder it. CI/CD security risks can be addressed in 4 areas; governance, identity management, deployment configuration, and git repository management.

Governance

Governance for third-party services needs to encapsulate pipeline dependency approval, management, and decommissioning. For approval, vetting processes for third parties should include access to resources, while for management and decommissioning, visibility should be maintained over the method of integration, permissions, access to secrets, and ingress/egress network connectivity to all third parties integrated into CI/CD systems.

A potential breach may involve any of the systems which take part in the CI/CD processes, including SCR, CI, Artifact repositories, package management software, container registries, CD, and orchestration engines. Further gains in the management and governance of your CI/CD pipeline can be applied by:

- Pipeline attestation through signing resources (software and infrastructure), used in development and production environments to validate resource integrity against a signing authority - see [Integrity Checking](#). Artifact verification tooling can be used as a way to help prevent unverified software from being delivered down the pipeline.

- Understanding the application ecosystem with visibility capabilities can help familiarize all systems involved. A fully mapped application ecosystem can help security reviewing and [threat modeling](#) during the design phase.
- Identifying log sources for all relevant systems ensures that relevant logs are enabled and centralized. This should include people and programmatic access. The aggregation and centralization of logs (e.g., SIEM) should be the foundation to support the detection of anomalies and potentially malicious activity.

Governance can also be achieved through effective Git repository management by applying Git limitations and rules of engagement for developers where a Git repository connects to a pipeline deployment:

- Limit auto-merge rules and manage drift between production and source code repositories. See [GitOps](#). Configuration drift detection measures aimed at detecting configuration drifts (e.g., resources in cloud environments that aren't managed using a signed IAC template), potentially indicative of resources deployed by an untrusted source or process.
- Detect secrets uploaded/pushed to, and stored on, code repositories. See [security hooks](#).
- SCM solutions provide the ability to sign commits using a unique key for each contributor. This measure can be leveraged to prevent unsigned commits from flowing down the pipeline.

Identity Management

The management of identities for CI/CD pipelines and dependent services (e.g., source code repositories), supports the principle of least privilege to programmatic and human access. For protecting CI/CD pipelines, a role-based access controls (RBAC) design should be considered to refrain from granting base permissions in a system to all users. Ensure subsequent access to network resources and pipeline nodes reflects the RBAC design and is aligned with the principle of least access. Complementing RBAC, the CI/CD pipeline should address:

- Removing user accounts in an SCM that can push unreviewed code to a repository and on the CI/CD pipeline subsequently deploy pipelines.
- Removing local user accounts for each tool with different levels of access. Instead, manage identities using a centralized identity provider using SSO.
- Consider credentials bound to predetermined conditions (e.g., IP address, FQDN, geolocation, or VPN binding) to ensure that compromised credentials cannot be used outside your environment.

Deployment Configuration

The pipeline management, design, and configuration also bear importance to how a CI/CD pipeline can be exploited. [Cider's Top 10](#) represents the key security risks within CI/CD pipeline. For each risk, a set of recommendations are provided. We've grouped those into the following changes that can be made in the configuration of your pipeline.

- Packages:
 - Enable checksum verification and signature verification for pulled packages.
 - Prevent pulling/fetching code packages directly from the internet or untrusted sources.
 - Configure all clients to pull packages from internal repositories containing pre-vetted packages and establish a mechanism to verify and enforce this client configuration.
 - Ensure clients are forced to fetch packages from your organization's internal registry.
 - Evaluate the need for triggering pipelines on public repositories from external contributors.
 - Avoid configuring clients to pull the latest version of a package.
- Pipeline secrets:
 - For pipelines exposed to secrets, ensure that each branch configured to trigger a pipeline in the CI system has a correlating branch protection rule in the SCM.
 - Ensure secrets that are used in CI/CD systems are scoped to only the secrets it requires.
 - Verify that secrets are removed from any artifacts, such as from layers of container images, binaries, or Helm charts.
 - Monitor for exposure of sensitive data in the pipeline logs, such as secrets, keys, and environment variables.
- Logical and network segmentation:
 - Identify and configure branch protection rules on branches hosting production code.
 - Pipelines running unreviewed code are executed on isolated nodes, not exposed to secrets and sensitive environments.
 - Do not use a shared node for pipelines with different levels of sensitivity / that require access to different resources.
 - Shared nodes should be used only for pipelines with identical levels of confidentiality.
 - To prevent manipulating the CI configuration file to run malicious code in the pipeline, each CI configuration file must be reviewed before the pipeline runs. Alternatively, the CI configuration file can be managed in a remote branch, separate from the branch containing the code built in the pipeline. The remote branch should be configured as protected.
 - CI and CD pipeline jobs should have limited permissions on the controller node. Where applicable, run pipeline jobs on a separate, dedicated node.
 - Ensure network segmentation is configured to allow the execution node to access only the required resources. Where possible, refrain from granting unlimited access to the internet to build nodes.
 - Ensure installation scripts are being executed as part of the package installation (separate context exists for those scripts), which does not have access to secrets and other sensitive resources available in other stages in the build process.

System Hardening

System hardening is any process, methodology, product, or combination used to increase the security of a system directly.¹⁰⁴ System hardening is the process of limiting potential weaknesses that make systems vulnerable to attacks. This process aims to reduce the attack surface on a system by securing the data, ports, components, functions, and permissions. After installation, common server operating systems often have vulnerabilities due to missing patches and insufficient security configuration. The process of system hardening significantly reduces the risk of known vulnerabilities. However, it is important to ensure that a system can still function after hardening. System hardening includes several tasks to reduce a system's attack surface, including the following¹⁰⁵:

- **Initial system setup:** Tasks include file system encryption, configuring secure boot mechanisms, file integrity checking, and login warning banners. For Linux, this task may also include enabling kernel security modules Security-Enhanced Linux (SELinux) and Advanced Intrusion Detection Environments (AIDE) for file integrity checking.
- **Service configuration:** System services that are not needed are deactivated to minimize the attack surface of operating system services and applications. Additional tasks may include installing endpoint protection and antivirus software.
- **Network configuration:** Tasks include configuring non-required network services like disabling ICMP redirects and non-required networking interfaces. A major task is the firewall configuration on the operating system level by configuring iptables, adding a default deny policy, and disabling IPv6 if not in use.
- **Logging and auditing:** Logging and auditing hardening measures include configuring system logs like Syslog, audit data retention, and modification to access controls. Login attempts, file access, and use of privileged actions also need to be logged.
- **Access, Authentication, and Authorization:** Tasks to ensure the allocation of rights on the operating system is minimized according to the least privilege principle. Further tasks include user account configuration, password complexity requirements, and configuration of authentication services like RDP and SSH.
- **System maintenance:** Tasks for system maintenance include configuring file permissions and group settings on file and directory and optionally multi-factor authentication for privileged administrative actions. Also, tasks for updates and patching to ensure the latest patches are applied.

There are various standards and recommendations for system hardening¹⁰⁶.

- **CIS Benchmark:** A pragmatic approach to system hardening uses the system security benchmark published by the Center of Information Security (CIS). CIS has released the CIS Benchmarks, a set of security recommendations for system hardening. There are currently more than 100 CIS across over 25 product families available. These include recommendations for operating systems, web servers, and cloud platforms to container orchestration environments.

104 Mourad, Azzam, Marc-André Laverdiere, and Mourad Debbabi. "Security hardening of open source software." (2006).

105 Rose, Tina, and Xiaobo Zhou. "System Hardening for Infrastructure as a Service (IaaS)." 2020 IEEE Systems Security Symposium (SSS). IEEE, 2020.

106 <https://www.cisecurity.org/cis-benchmarks/>

- **STIG and SCAP:** Security Technical Implementation Guide (STIG)¹⁰⁷ is another set of recommendations for systems hardening. STIG was originally developed for the U.S. Department of Defense to provide a configuration standard for systems and applications. These standards make it harder for an attacker to access the system. Standards are defined for various products, such as operating systems, network devices, databases, and mobile devices. The Security Content Automation Protocol (SCAP) supports automating STIG guidelines. SCAP is a method for using specific standards to help organizations automate vulnerability management and policy compliance evaluation. SCAP and STIG support automating the process of scanning systems for various operating systems, including cloud environments.
- **Golden Images:** A pragmatic approach to establishing a set of immutable hardened images in cloud environments is to create golden images. More secure than an out-of-the-box image, a golden image is a pre-configured virtual image template. A golden image provides a secure configuration baseline for deployment in cloud environments. A widely used tool to create golden images is Packer by Hashicorp.¹⁰⁸ Packer allows the creation of identical images for cloud infrastructure using a single source template. CIS provides a set of hardened server images which organizations can use for golden images. Within public cloud platforms, it is also possible to directly purchase CIS hardened images from the marketplace in the respective cloud provider.¹⁰⁹

System hardening reduces a system's attack surface and security risks by removing all non-essential software programs and utilities¹¹⁰. Particularly using security benchmarks like CIS or STIG with hardened golden images reinforces the security posture based on industry guidance without having to dedicate time and effort to fine-tune specific security settings in cloud systems and applications. Also, industry guidelines make the hardening processes robust and comprehensive without removing the flexibility to adapt to requirements and objectives for specific use cases and scenarios.

Overall, system hardening reduces the attack surface, which results in fewer opportunities to gain a foothold into the system for attackers. It also reduces the risk of reputation loss due to a compromised system or unauthorized access.

107 A. DHondt and H. Bahmad, "Understanding SCAP Through a Simple Use Case," Hakin9 - IT Security Magazine, vol. 11, no. 3, pp. 100-110, 2016.

108 <https://www.packer.io/docs>

109 <https://www.cisecurity.org/hardened-images>

110 Ferreira, Jéssica Pereira. Hardening de Sistemas com CIS Benchmark. Diss. 2021



The capabilities and practices that can be applied after an application/product has been released into production. Runtime security enables continuous improvement by identifying inefficiencies, vulnerabilities, and weaknesses and enabling incident response.

Chaos Engineering

The process of testing a computing system (e.g., a microservice architecture) to ensure it can withstand unexpected disruptions. The core idea of chaos engineering is that testing is experiment-driven. Chaos engineering means that specific scenarios are defined with concrete hypotheses. These hypotheses are then tested during the experiment itself. If a failure occurs, lessons learned are documented, and improvements are implemented for further iterations of the experiments. In a nutshell, the idea is to find potential points of failure that can completely disrupt the business for the organization's systems.

Although chaos engineering was conceived as a tool to test against potential availability risks such as denial of service or downtime of critical systems. The same concepts can be applied to the testing of security controls. In this way, security control failures/weaknesses are assumed due to erosion over time and proactively discovered in a controlled environment. Teams develop a habit of constant transparency and group analysis and depend less on limited and time-boxed root cause analyses. This type of experimentation is often called Security Chaos Engineering (SCE).¹¹¹

Identifying security gaps or security control erosion through proactive experimentation can be the introduction of unexpected variables or "fault injection" in a CI/CD pipeline or a Continuous Verification pipeline. SCE use cases may also be executed by executing fault injection in running workloads.

For complex systems that require each service to function properly, the interactions between those services can cause unpredictable outcomes. These unpredictable outcomes, coupled with disruptive events that affect production environments, make distributed systems inherently chaotic. Chaos events could derive from improper fallback settings when a service is unavailable; outages when a downstream dependency receives too much traffic; cascading failures when a single point of failure crashes. An empirical, systems-based approach addresses the chaos in distributed systems at scale and builds confidence in the ability of those systems to withstand realistic conditions.

- Define the IT system's steady-state and normal behavior as a measurable output. Measurements of that output over a short period of time constitute a proxy for the system's steady state. The overall system's throughput, error rates, and latency percentiles could all represent steady-state behavior metrics of interest. By focusing on systemic behavior patterns during experiments, Chaos verifies that the system does work, rather than trying to validate how it works.
- Introduce variables that reflect events like server downtime, hard drive malfunction, network

111 <https://www.oreilly.com/library/view/security-chaos-engineering/9781492080350/>

- connections issues, software failure, and traffic spikes.
- Chaos test in production by sampling real traffic. For authenticity and relevance, chaos prefers to experiment directly on production traffic. If testing in production is impossible, tests should be performed in an environment as close to production as possible and with realistic traffic.
- Limit scope and scale to avoid and reduce the impact of tests. Ensure the fallout from experiments are minimized and contained.

A Chaos experiment template as a process leveraged document should contain:

- Identification of the experiment with name, authors, and revision
- Notified personnel and teams
- Scope of the experiment
- Hypotheses of the experiment
- Metrics to be observed in the experiment
- Risk assessment for experiments done in production
- Guardrails to mitigate the aforementioned risks and minimize the blast radius of the experiment.

While chaos engineering can be difficult to achieve, it's a barometer for DevOps maturity and provides the following benefits:

- Resilience to applications and infrastructure, where weaknesses have been identified, design, process, and configuration improvements can better secure your products.
- Provides a working practice to improve systems used within the organization continuously.
- Ability to test chaos and restoration activities for complex architectures.

Tools can automate the Chaos Engineering experience without needing to define system steady states or hypothesize the experiment. Some example tools available are Chaos Monkey¹¹², AWS Fault Injection Simulator¹¹³, Azure Chaos Studio¹¹⁴, and Chaos Toolkit¹¹⁵.

Cloud Security Posture Management

Cloud Security Posture Management (CSPM) is primarily used for monitoring cloud compliance, vulnerabilities, and resource misconfiguration in a cloud account or subscription. Cloud misconfigurations are typical errors that expose data or resources to attack, which is common during the setup process for a cloud service. Many large data breaches have occurred because of misconfigured cloud resources like exposed storage buckets and leaving data exposed.

112 <https://netflix.github.io/chaosmonkey/>

113 <https://aws.amazon.com/fis/>

114 <https://azure.microsoft.com/en-us/services/chaos-studio/>

115 <https://chaostoolkit.org/>



Figure 23: Zscaler's findings on cloud security threats¹¹⁶

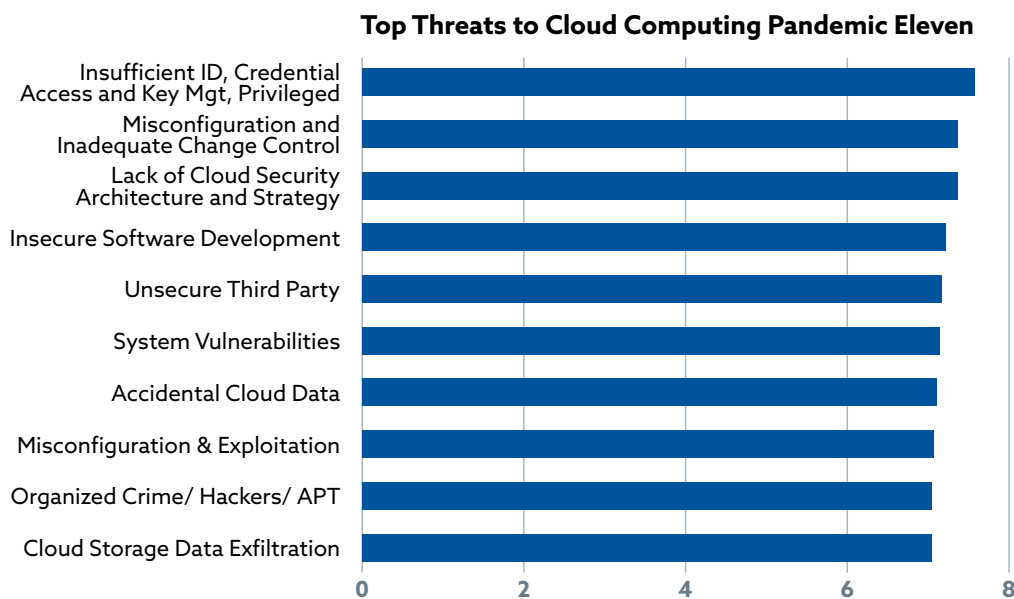


Figure 24: Top threats from CSA's Cloud Pandemic Eleven

The core purpose of a CSPM is the continuous monitoring of cloud resources for security policy enforcement gaps, security threats, and data security through a collection of capabilities. CSPMs accomplish this through API calls to the cloud services, accessing logging for event-triggered alerting within the CSPM. CSPMs are enablers in building security baselines (e.g., WAF and firewall rules), detecting security misconfiguration (e.g., risky access and permissions), and applying automated correction controls (e.g., automated patching and image refresh).

116 <https://www.zscaler.com/resources/>

Organizations need to comply with strict requirements for protecting data and controlling access to that data. Regulated industries have to adhere to specific benchmarks; for example, PCI DSS for retail, HIPAA for healthcare, FFIEC for financial services, NIST, GDPR, and ISO27001. A CSPM automatically scans for and detects violations which can help organizations better comply with regulations.

While compliance gains can be realized through the use of a CSPM (see [Pillar 4: Bridging Compliance and Development](#)), controls can be applied that complement the operating effectiveness of cloud engineering - CSPM benefits are increased in combination with the following activities recommended in this paper:

- **[IaC Analysis](#)**: Fix misconfigured and vulnerable cloud resources at the build phase to reduce the volume of security issues identified by the CSPM once cloud services are deployed.
- **[GitOps](#)**: Synchronize security changes identified from the CSPM on IaC to auto-deploy as live cloud services
- **[Guardrails](#)**: Use the pre-set rules and feedback from CSPM findings to define consistent guardrails across multi-cloud environments.
- **[System Hardening](#)**: Use the CSPM to educate infrastructure hardening decisions on cloud resources.

While CSPMs are expensive to procure and implement; they do introduce benefits beyond just improving cloud security:

- **Secure baseline configuration of cloud services**: Based on a compliance framework mapping, such as a CIS benchmark, developers can adopt an industry-standard set of guardrails, or can tailor that framework to align with corporate and regulatory policy and adjust the controls and monitoring as appropriate.
- **Real-time cloud asset inventory**: Regardless of whether or not an organization adopts DevSecOps, discovery, and inventory of all software and cloud resources is critical to a mature security capability. CSPM solutions routinely scan cloud environment(s) and maintain that list of services and cloud resources.
- **Data asset storage**: CSPM solutions may also be able to inventory cloud objects and services and enumerate data objects stored in storage containers, databases, and other data repositories (e.g., PII.) With knowledge of the type of data processed or persisted by a system, an organization may choose to adjust the security controls applicable to that system and/or refine the risk calculation for that object or component, resulting in security controls that evolve and are commensurate with risk.

Cloud Security Posture Management tools are typically available through licensed providers. Some examples of CSPM products in the market are [Wiz](#), [Palo Alto's Prisma Cloud](#), [Checkpoint](#), [CrowdStrike](#), and [Zscaler](#).

Monitoring and Observability

Monitoring is the process of collecting, aggregating and analyzing data to track applications and infrastructure. Monitoring helps:

- Guide business decisions
- Provide feedback to quickly find and fix problems early
- Communicate information about your systems to other parts of the business
- Detect and respond to suspect activity, cyber incidents and events.

Monitoring is capturing and displaying data, whereas observability can discern system health by analyzing its inputs and outputs. For example, we can actively watch a single metric for changes that indicate a problem—this is monitoring.

Monitoring can include the underlying hardware resources, network transport, applications/ microservices, containers, interfaces, normal and anomalous endpoint behavior, and security event log analysis. **Table 3** is the monitoring tools and phases summary from the US Department of Defense's DevSecOps reference architecture¹¹⁷.

Activities/ tools	Description/ features	Input	Output
Logging	Logging events for all user, network, application, and data activities	All user network, application and data activities	Logs
Log analysis and auditing	Analyze and audit to detect malicious threats / activity; Automated alerting and workflows for response Forensics for damage assessment	Logs	Alerts and remediation report
System performance monitoring	Monitor system hardware, software, database, and network performance; Baseline system performance; Detect anomalies	Running system	Performance KPI measures; Recommended actions; Warnings or alerts
System security monitoring	Monitor security of all system components; Security vulnerability assessment; System security compliance scan	Running system	Vulnerabilities; Incompliance Findings; assessments and recommendations; Warnings and alerts.

¹¹⁷ [DoD Enterprise DevSecOps Reference Design](#)

Asset inventory	Inventory IT assets	IT assets	Asset inventory
System configuration monitoring	System configuration (infrastructure components and software) compliance checking, analysis, and reporting	Running system configuration; Configuration baseline	Compliance report; Recommended actions; Warnings and alerts
Database monitoring and security auditing	Baseline database performance and database traffic; Perform user access and data access audit; Detect anomalies from events correlation; Detect SQL injection; Generate alerts	Database traffic, events and activities.	Logs; Warnings and alerts

Table 3: Monitoring tools and phases summary

A system is observable if it emits useful data about its internal state, which is crucial for determining root cause¹¹⁸. Observability is the ability to measure a system's current state based on the data it generates, such as logs, metrics, and traces. The goal of observability is to understand what's happening across all environments so issues can be detected and resolved to keep systems efficient and reliable. Organizations also adopt observability to help detect and analyze the significance of events to their operations, software development lifecycles, application security, and end-user experiences¹¹⁹. The measurements of logs, metrics, distributed traces and user experiences are the key pillars to achieving observability success:

- **Logs:** Structured or unstructured text records of discrete events that occurred at a specific time.
- **Metrics:** The values represented as counts or measures that are often calculated or aggregated over a period of time. Metrics can originate from a variety of sources, including infrastructure, hosts, services, cloud platforms, and external sources.
- **Distributed tracing:** Activity of a transaction or request as it flows through applications and shows how services connect, including code-level details.
- **User experience:** Extends traditional observability telemetry by adding the outside-in user perspective of a specific digital experience on an application, even in pre-production environments.

The DevOps Research and Assessment ([DORA](#)) at Google Cloud has published a set of metrics that organizations can leverage to assess and measure performance quantitatively.

- **Deployment Frequency:** Deployment frequency defines the rate of successful production.

¹¹⁸ [Observability vs. monitoring: What's the difference? | Dynatrace news](#)

¹¹⁹ [What is observability? Not just logs, metrics, and traces | Dynatrace news](#)

Increased deployment frequency enables to rapidly deploy changes multiple times a day instead of only weekly or monthly releases. Organizations can increase deployment velocity by continuous delivery & continuous integration pipeline that allows automated testing and fast feedback

- **Lead Time for Changes:** This metric defines the time required to get a change into a production environment. Automated performance testing and small iterative changes are pragmatic approaches to improve overall lead time, enabling organizations to remediate defects faster due to accelerated feedback. Trunk-based development also allows shorter lead times, automating testing by working on separate feature-based branches.
- **Change Failure Rate:** A failure rate of deployments in production in percentage. The same approach to shortening lead times applies to change failure rates, such as automating tests, working with trunk-based development, and releasing changes in small batches. Increased deployment frequency enables efficient defects detection and faster remediation. Reporting and notifying upon failures allows for fixing bugs and ensuring that new releases meet security requirements.
- **Time to Restore Service:** This metric is defined as the required recovery time from production failures, allowing organizations to quickly recover from failure shortly after when the failure has occurred. Reduced time to restore service also allows rolling back any changes causing the failure more quickly. Continuous performance management through cloud-native monitoring and alerting allow notifying stakeholders in the case of a failure.

Baselining an organization's performance on DORA metrics helps improve IT operations' efficiency and effectiveness. Applying measurements like DORA for cloud applications and infrastructure can help organizations understand cloud systems' performance and resiliency. An organization can gain additional visibility and insight through reports around these metrics. Organizations benefit from a metrics-centered performance management strategy. This strategy enables quantitative assessments to gain insights into the efficiency of workload transactions and business workflows. Metrics enable organizations to understand quality deficiencies and improve release cycles for workloads, including their overall resiliency and availability¹²⁰.

Implementing a monitoring and observability system allows tracking of internal metrics to determine performance. Whilst we'll explore security scenarios and use cases for observability metrics in "Pillar 6: Measure, Monitor, Report, Action," some example metrics recognised by Google Cloud provides an indication for how this analysis can occur¹²¹:

- **Changes made to monitoring configuration:** How many pull requests or changes per week are made to the repository containing the monitoring configuration?
- **"Out of hours" alerts:** What percentage of alerts are handled at night? While some global businesses have a follow-the-sun support model which makes this a non-issue, it can be an indication that not enough attention has been paid to leading indicators of failures. Regular night-time alerts can lead to alert fatigue and burned-out teams.
- **Team alerting balance:** If teams are in different locations, are alerts fairly distributed and addressed by all teams?

120 Tom Hall, Atlassian, DevOps metrics - Why, what, and how to measure success in DevOps, August 2022, Available @ <https://www.atlassian.com/devops/frameworks/devops-metrics>

121 [DevOps measurement: Monitoring and observability | DevOps capabilities | Google Cloud](#)

- **False positives:** How many alerts resulted in no action, or were marked as “Working as Intended”?
- **False negatives:** How many system failures happened with no alerting, or alerting later than expected? How often do postmortems include adding new (symptom-based) alerts?
- **Alert creation:** How many alerts are created per week (total, or grouped by severity, team)?
- **Alert acknowledgement:** What percentage of alerts are acknowledged within the agreed deadline (such as 5 minutes, 30 minutes)?
- **Alert silencing and silence duration:** How many alerts are in a silenced or suppressed state per week? How many are added to this pool, how many removed?
- **Unactionable alerts:** What percentage of alerts were considered «unactionable»? That is, the engineer alerted was not able to immediately take some action, either out of an inability to understand the alert implication, or due to a known issue. Unactionable alerts are a well known source of toil.
- **Usability:** Alerts, runbooks, dashboards. How many graphs and dashboards? How many lines per graph? Can teams understand the graphs? Is there explanatory text to help out new engineers? Do people have to scroll and browse a lot to find the information they need? Can engineers navigate from alert to playbook to dashboards effectively? Are the alerts named in such a way to point engineers in the right direction? These might be measured by surveys of the team, over time.
- **Mean-Time-To-Detect, Mean-Time-To-Respond, impact:** The bottom line is time to detect, time to resolve, and impact. Consider measuring the “area under the curve” of the time that the outage was affecting customers times the number of customers affected. This can be estimated or done more precisely with tooling.

For monitoring and metric collection, an organization can use cloud-native solutions like AWS CloudWatch, GCP Operations Suite, and Azure Monitor centrally to collect, access, and correlate performance metrics visibility and to understand performance issues. Applying cloud-native monitoring metrics allows fine-grained monitoring of the control flow of cloud applications and platforms, including timings of individual transactions.

By tracking metrics, a better understanding of monitoring and observability will be ascertained. Breaking these measurements down by product, by operational team, or other methods will provide insight into the health and security posture of products and teams. Organizations implementing monitoring and observability can use open source tooling like OpenTelemetry, Jaeger and Zipkin. Organizations can leverage commercial tools to improve performance management based on defined metrics. Datadog, Dynatrace and Splunk are example commercial solutions for observability, providing security-related events, cloud integration including containers, and asset discovery.

The DevSecOps working group (WG) will provide a white paper on the topic of DevSecOps monitoring metrics in “Pillar 6: Measure, Monitor, Report, and Action.” The paper describes some of the most critical metrics to monitor in a DevSecOps environment. It will include metrics to measure the mean time to remediate vulnerabilities and recover from an incident, software and infrastructure reproducibility, security control volume, and defense-in-depth control averages.

Attack Surface Management

Attack Surface Management is an emerging cybersecurity discipline that refers to the processes and technology necessary to discover, identify and manage the risks presented by the IT systems of an organization. Attack Surface Management includes External Attack Surface Management (EASM) which involves conducting attack surface management activities on internet-facing assets of an organization, and Internal Attack Surface Management (IASM) which includes conducting attack surface management activities on IT assets only accessible from within an organization. The attack surface is comprised of four main components¹²²:

- **On-premises assets:** servers and hardware.
- **Cloud assets:** cloud resources, such as servers, workloads, SaaS applications, or cloud-hosted databases.
- **External assets:** external vendor assets that store and process data or are integrated with the network.
- **Subsidiary networks:** Networks that are shared by more than one organization (e.g., in the event of a merger or acquisition).

Number of Assets Comprising the Attack Surface

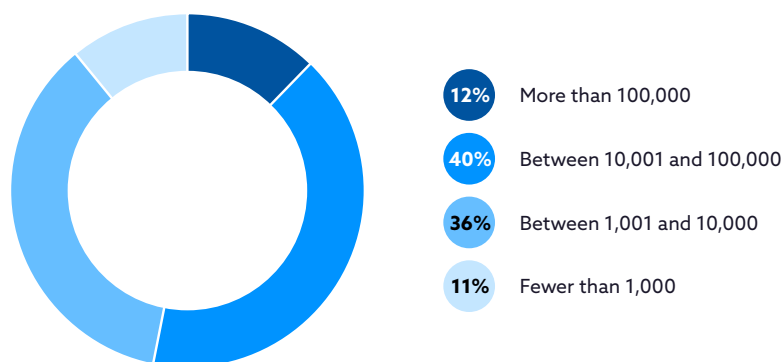


Figure 25: Qualys, volume on the attack surface¹²³

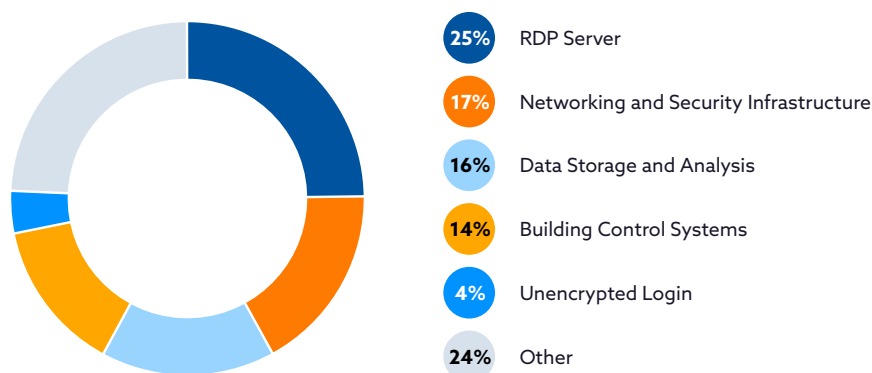


Figure 26: Palo Alto, risks across the global attack surface¹²⁴

¹²² <https://www.crowdstrike.com/>

¹²³ <https://blog.qualys.com/qualys-insights/2022/07/28/attack-surface-management-a-critical-pillar-of-cybersecurity-asset-management>

¹²⁴ https://www.paloaltonetworks.com/content/dam/pan/en_US/assets/pdf/reports/cortex_xpanse-attack-surface-threat-report.pdf

Attack surface management typically comprises the following capabilities:

- **Discovery:** Enumerate all infrastructure, network connections, applications, APIs, credentials, digital certificates and domains, which are part of an organization's technology stack, including dependencies with 3rd-party services, components, and APIs.
- **Inventory and Classification:** Build an inventory of organization IT assets and classify and prioritize them based on various criteria including asset type, technical properties, business criticality, and compliance requirements.
- **Risk Analysis and Scoring:** Correlate asset information with threat intelligence information and asset classification to calculate risk scores.
- **Continuous Security Monitoring:** The continuous monitoring of assets involves monitoring and analyzing logs and security events from assets for suspicious and malicious behavior detection and vulnerabilities. The key is continually evaluating all aspects of the solution, not just performing periodic analysis.
- **Remediation:** Remediate risk assets and weakness in exposed assets. It's important to share the findings and remediations with the product teams and technology owners to improve understanding and awareness of the issues to improve the design continually.

Prioritizing security findings with the additional context of data that could be lost helps to focus personnel on the higher-risk findings and also identifies those areas of the deployment pipeline and processes which may need to be improved. By assuming the mindset of the attacker and mimicking their toolset, organizations can improve visibility across all potential attack vectors. The benefits of an effective attack surface management tool can enable organizations to:

- Automate asset discovery, review, and remediation
- Identify and disable shadow IT assets and other previously unknown assets
- Remediate known weaknesses like weak passwords, misconfigurations, and outdated or unpatched software

Organizations can continuously monitor assets and identify vulnerabilities by observing resources outside of the organization's network boundary, at the point of discovery by cybercriminals. Licensed tools and services which can be used for attack surface management are available, with examples including [CyCognito](#) and [Palo Alto Cortex Xpanse](#).

Attack surface management solutions continuously monitor an organization's public IP address space and that of connected organizations on a regular basis. This includes identifying rogue assets created as shadow IT like the exposed servers (e.g., Jenkins server, AWS S3 bucket) with default credentials posing a risk to the organization. While attack surface management is achieved by tooling, it's important to note that an organization's attack surface will evolve as devices are constantly added, new users are introduced and business needs change.

Postmortems

A project postmortem is a ceremony used to identify the causes of a project failure, incident, or significant business-impairing downtime, and review future preventative measures. Project postmortems are intended to inform process improvements that mitigate future risks and promote iterative best practices.

An incident postmortem brings teams together to take a deeper look at an incident and figure out what happened, why it happened, how the team responded, and what can be done to prevent repeat incidents and improve future responses. Blameless postmortems do all this without any blame games. In a blameless postmortem, it's assumed that every team and employee acted with the best intentions based on the information they had at the time. Instead of identifying and punishing the source of human error, blameless postmortems focus on improving performance moving forward.

[Atlassian's Incident Management Handbook](#)

When things go wrong, looking for someone to blame is a natural human tendency. It's in Atlassian's best interests to avoid this, so when you're running a postmortem, you need to overcome it consciously. We assume good intentions on the part of our staff and never blame people for faults. The postmortem needs to honestly and objectively examine the circumstances that led to the fault so we can find the true root cause(s) and mitigate them.

A postmortem includes the incident impact, mitigating actions, and root cause, with the critical part being the follow-up to prevent the incident from recurring. The goal of the postmortem is to continuously learn and improve by ensuring that all root causes and the corresponding preventative actions are well understood. Teams have some internal flexibility, but common postmortem triggers include:

- User-visible downtime or degradation beyond a certain threshold
- Data loss of any kind
- On-call engineer intervention (release rollback, rerouting of traffic, etc.)
- A resolution time above some threshold
- A monitoring failure (which usually implies manual incident discovery)

Google's Postmortem strategies:

- Ease postmortems into the workflow. A trial period with several complete and successful postmortems may help prove their value, in addition to helping to identify what criteria should initiate a postmortem.
- Make sure that writing effective postmortems is a publicly rewarded and celebrated practice, both through the social methods mentioned earlier and through individual and team performance management.
- Encourage senior leadership's acknowledgment and participation.

Although the postmortem process does present an inherent cost in terms of time or effort, it provides the benefits of continuous improvement for ways of working, fosters a positive culture to help incentivize transparency, innovation, and change and improves communication channels across teams.

Google has provided an [example](#) output of the postmortem process, while outages and real postmortem events are available within the links of the [danluu/post-mortem](#) Github page. Other postmortem frameworks available are [Atlassian's blameless postmortem](#), which can also be complementary to incident management processes from [CSA's cloud incident response framework](#).

Purple Teaming

Cybersecurity simulation is a way to test technologies and processes in response to simulated cyberattacks. This typically requires a replicated environment to effectively “war-game” against potential attacks in realistic scenarios. This gives rise to so-called colored teams. In this section, the focus is on *red and blue teams*, and their combination, which yields a purple team. In context, the red team is the offensive team that is in charge of conducting penetration tests, vulnerability assessments, and full-fledged simulated cyber attacks. Sometimes, these simulations will not be informed to the blue team, the defensive specialist team. This team comprises members of the organization's Security Operations Center and is the first level of response and identification of security incidents. Purple teaming, in contrast, is an approach in which members of both teams come together to ensure that both teams can learn from each other, increasing the value of the exercise for the organization.

Purple team exercises can occur in two different phases¹²⁵:

- **Purple team exercises at the testing time**
 - In this case, different types of purple team exercises can be performed in the event of a red team turning into a purple team exercise:
 - **Catch-and-release:** this type of test is optimal for ending stages of a test, where the red team has been caught by the blue team repeatedly and changing TTPs is not a possibility anymore. In this case, a communication channel between the blue and red team is established (moderated by the white team - the team in charge of managing the red team exercise). Using this communication channel, the blue team will communicate the Indicators of Compromise whenever the red team activities are identified. Assuming these are correct, specific machines or accounts can be released as part of the test if they have been blocked.
 - **Collaborative proof-of-concept:** Whenever a specific test or scenario is not possible due to, for example, the systems being out of scope or having a high-risk of impact in critical systems, a collaborative approach can allow the red team and blue team to work on a proof-of-concept of the attack in which all offensive and defensive aspects are considered.
 - **War game:** Specific type of purple team tests in which the blue team is completely aware of the presence of the red team and the nature of their activities. In these scenarios, specific flags are assigned to target assets, and the red team must capture these, whereas the blue team must defend the systems.
- **Purple teaming at the closure of the test**

Whenever the purple teaming is performed at the end of a red team exercise, there are several possibilities:

125 https://www.ecb.europa.eu/pub/pdf/other/ecb.tiber_eu_purple_best_practices.20220809~0b677a75c7.pl.pdf

- **Tabletop discussion:** In this case, potential scenarios are discussed and addressed by the blue team, the red team, as well as management (optionally). Some ways for this discussion to happen are the role-playing of a specific scenario, a theoretical discussion, and a business continuity discussion.
- **Re-exploration of scenarios of the test:** In this case, some of the scenarios analyzed in the red team can be revisited as walkthroughs for the blue team to understand where the security gaps exist and how potential mitigations could be developed and implemented.
- **Exploration of new scenarios:** If the red team has new ideas which may be explored during the closure of the test, this could bring interesting results in collaboration with the blue team. The idea would be to walk through potential new scenarios with the blue team that may fill gaps not addressed by the initial test.

Purple teaming, successfully achieved by effective team structures and processes, does not aim to replace a red teaming exercise - the test is kept confidential and the blue team is not informed to increase the realism of the simulation. However, in specific cases, it may be more beneficial that both teams can learn from each other. Such is the case, for example, when the red team has not been able to evade identification and understanding the SOC structure can help them improve their approach for more successful simulations or vice versa when the blue team has not been able at all to catch the red team activities. A purple teaming exercise will then help improve the detection capabilities of the blue team by understanding the red team's tactics, techniques, and procedures (TTPs).

Vulnerability Management (Post-identification)

The increasing growth of cyberattack and publicity of cyber-related breaches resulted in the further need for these practices to be adopted, especially with DevSecOps, where changes are implemented continuously at a rapid pace. To address these challenges, consider the following steps as part of your vulnerability remediation process. These will need to be fine-tuned to your environment and continuously improved through lessons learned:

- **Identification:** Identify vulnerabilities through infrastructure and application testing and continuous vulnerability scanning. Consider monitoring public and vendor security advisories and cyber threat intelligence to identify additional vulnerabilities relevant to your organization.
- **Evaluation:** Evaluate, qualify and triage vulnerabilities to determine their applicability and severity. This will enable the organization to prioritize remedial activities and focus treatment efforts on the more significant risks.
- **Treatment:** Remediate the vulnerabilities affecting systems by applying a fix through patching, configuration change, code modification, or a process change. Develop countermeasures for risks that can be mitigated, allowing them to be deployed quickly in the event of an active attack where remediation cannot be accomplished.
- **Reporting:** Collect data indicators and compare them against established metrics used for management reporting. Continuously assess the effectiveness of the vulnerability remediation process by evaluating your skill set capabilities and available technological solutions.

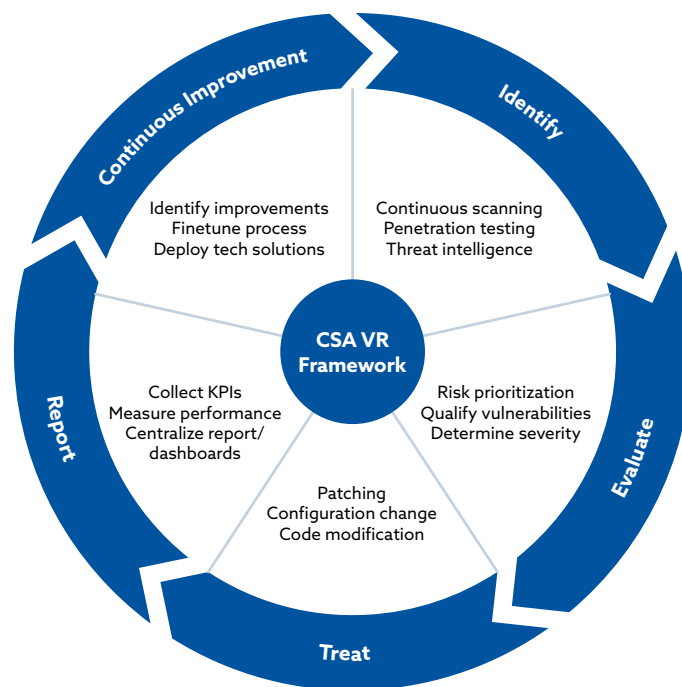


Figure 27: CSA vulnerability remediation framework

While security activities in the stages for Coding and Testing help identify vulnerabilities in application and infrastructure, vulnerability remediation evaluates, prioritizes, and treats the defects and weaknesses found in networks, systems, and application components.

Remediating vulnerabilities can be challenging and can be perceived as an anti-pattern to agile development methods. Traditional patching methods are usually manual, time-consuming, and costly, requiring a lot of expertise to evaluate the impact of a fix. DevSecOps principles have revolutionized these practices by offering ways to continuously test software and embedded security and quality throughout the development cycle. Remediation is performed to mitigate the risk and likelihood of vulnerabilities being exploited. It is an essential part of the broader Vulnerability Management process, which manages vulnerabilities throughout their lifecycle from initial identification to risk classification, prioritization, remediation, and reporting. These vulnerabilities can be created during development activities and manifest as software defects requiring patches or configuration issues requiring administrative action to be resolved.

Vulnerability remediation ties into other practices, such as attack surface management. Efforts to remediate vulnerabilities should be prioritized within the context of the overall attack space and the possibility of exploitation. Teams should be split and focused on the network layer, infrastructure (virtual machines, containers, operating systems), and application layer (applications, APIs, microservices). As mentioned in other areas of this paper, remediation should be automated where possible.

Not all vulnerabilities are the same. By shifting security left, teams can identify vulnerabilities early and often and remediate them in the same development sprint using tools like SAST and

SCA. Equally, by identifying vulnerabilities through a CSPM, DAST, or penetration testing, more analysis and rigor needs to be applied when evaluating findings and determining risk profiles and exploitability. It is essential to evaluate vulnerabilities before triage and then agree on the actions to take regarding each identified vulnerability:

- **Fix:** Agree on whether a vulnerability should be remediated depending on its severity, complexity, and dependencies to assess treatment impact and feasibility. For severe vulnerabilities that cannot be patched, consider a system redesign or replacement to eliminate the risk associated with the vulnerability.
- **Accept:** Acknowledge the vulnerability is present and evaluate whether a risk acceptance approach is tolerable for the vulnerability. Factors influencing this decision can be cost, patch availability, or skilled resources.
- **Investigate:** Perform further investigation on the applicability, severity, and likelihood of exploitation to understand better the risk associated with the vulnerability and the possible treatment solutions and technical complexity.

A robust vulnerability remediation process is essential to minimize your organization's risk profile. Key driving factors for the implementation and enforcement of a strong vulnerability remediation process includes:

- **Security benefits:** Allows for the efficient assessment, prioritization, and remediation of the vulnerabilities identified within your environment. Reducing the risk of your organization being compromised by a cyber attack, avoiding operational and business disruptions or financial and reputational damages. This practice can also identify other improvement areas such as patch management and asset management to drive technological improvements.
- **Compliance:** Mandatory requirement as part of most security best practice industry standards, guidelines, and frameworks, including CIS, PCI-DSS, ISO27001, GDPR, HIPAA, and FFIEC. A strong process may be essential to avoid financial penalties for non-compliance due to obligations and to reduce the likelihood of being impacted by a security breach which may bring additional fines on top of reputational damages.

The management and remediation of vulnerabilities are considered critical to reducing the likelihood of compromise and, as such, are a common compliance requirement for most regulators. In today's environment, remediating vulnerabilities has become even more relevant with the zero trust approach to security which aims to shift the focus away from the network perimeter and more toward proactive protection of resources. The continuous changes and agile development of DevSecOps practices require the adoption of a strong vulnerability remediation process to reduce the risk of introducing vulnerabilities. Consider the following frameworks and guidelines to manage, triage efficiently, and remediate vulnerabilities while minimizing operational disruption:

- **CISA [CRR Volume 4 Vulnerability Management](#):** This guide was developed by the Department of Homeland Security's Cyber Security Evaluation Program (CSEP) to help organizations manage operational risks and implement a mature Cyber Resilience Review (CRR) process. The Vulnerability Management activity takes an approach derived from the CERT-RMM and focuses on the process of identifying, analyzing, and managing vulnerabilities. This includes steps to define a vulnerability resolution strategy and convert it into a structured plan with rules and guidelines to reduce exposure and mitigate vulnerabilities.

- **Hyperproof [Vulnerability Management Guide](#):** This guide promotes the importance of vulnerability management for corporate security and regulatory compliance. It encourages an organized, coordinated, and step-by-step approach to manage vulnerabilities across your organization effectively. It includes seven components: the establishment of foundational guidelines, the management of assets, configurations, and patches, and the identification and assessment of vulnerabilities through scanning and penetration testing. The article also includes nine tips to enhance your patch management process and a list of vulnerability scanners that can help you identify vulnerabilities.
- **NCSC [Vulnerability Management](#):** This guide was developed to help organizations assess and prioritize vulnerabilities and identify the most relevant patches with the available resources and funds. The guidance focuses on managing vulnerabilities in widely available software and hardware, mostly deploying patches and modifying weak configurations rather than niche software issues. This includes metrics and factors influencing your vulnerability triaging and prioritization methodology.
- **NIST [Guide to Enterprise Patch Management Planning](#):** This guide provides recommendations to help conduct preventive maintenance through enterprise patch management. It provides suggestions to simplify and operationalize patching to improve the reduction of risk and help prevent system compromises, data breaches, and disruptions. The process includes steps to identify, prioritize, acquire, verify and monitor the installation of patches, updates, and upgrades. Promoting the protection of business assets to avoid costly breakdowns and reliably achieve your organization's mission.
- **OWASP [Vulnerability Management Guide](#):** This guide breaks down vulnerability management into three manageable and repeatable functions: Detection, Reporting, and Remediation. These functions consist of four main processes, which can be viewed as tasks that include a to-do list. The document provides recommendations to incrementally and continuously refine each task to fit your objectives and make your organization more resilient.

Incident Response¹²⁶

Preventive security controls cannot completely eliminate the possibility of critical data being compromised in a cyberattack. Therefore, organizations must ensure that they have a reliable incident response processes strategies in place. Incident response is an organized approach to addressing and managing the aftermath of a security breach or cyberattack. The goal is to handle the situation in a way that limits damage and reduces recovery time and costs.

Incident response can be defined as the process designed to manage cyberattacks in a cloud environment and comprises four phases (illustrated in **Figure 28**):

- Phase 1: Preparation
- Phase 2: Detection and Analysis
- Phase 3: Containment, Eradication, and Recovery
- Phase 4: Postmortem

¹²⁶ [Cloud Incident Response Framework | CSA](#)

Phase 5.1 Preparation	Phase 5.2 Detection and Analysis	Phase 5.3 Containment, Eradication, and Recovery	Phase 5.4 Postmortem
CSA Sec. Guidance v4.0 9.1.2.1 Preparation	CSA Sec. Guidance v4.0 9.1.2.2 Preparation and Analysis	CSA Sec. Guidance v4.0 9.1.2.3 Containment, Eradication, and Recovery	CSA Sec. Guidance v4.0 9.1.2.4 Postmortem
NIST 800-61r2 3.1 Preparation	NIST 800-61r2 3.2 Detection and Analysis	NIST 800-61r2 3.3 Containment, Eradication, and Recovery	NIST 800-61r2 3.4 Post-Incident Activity
TR 62 0.1 Cloud Outage Risks	TR 62 4.2 COIR Categories 5.1 Before Cloud Outage: CSC 6.1 Before Cloud Outage: CSP	TR 62 5.2 During Outage: CSC 6.2 During Outage: CSP	TR 62 5.3 After Outage: CSC 6.3 After Outage: CSP
FedRAMP Incident Comm. Procedure 5.1 Preparation	FedRAMP Incident Comm. Procedure 5.2 Detection and Analysis	FedRAMP Incident Comm. Procedure 5.3 Containment, Eradication, and Recovery	FedRAMP Incident Comm. Procedure Post-Incident Activity
NIST (SP) 800-53 r4 3.1 Selecting Security Control Baselines Appendix F-IR IR-1, 1R-2, 1R-3, IR-8	NIST (SP) 800-53 r4 AT-2, 1R-4, IR-6, 1R-7, IR-9, SC-5, SI-4	NIST (SP) 800-53 r4 Appendix F-IR 1R-4, IR-6, IR-7, IR-9	Incident Handlers Handbook 7 Lessons Learned 8 Checklist
Incident Handlers Handbook 2 Preparation 8 Checklist	Incident Handlers Handbook 3 Identification 8 Checklist	Incident Handlers Handbook 4 Containment 5 Eradication 6 Recovery 8 Checklist	
ENISA Cloud Computing Security Risk Assessment Business Continuity Management, page 79			

Figure 28: CSA incident response

There are several key aspects of a cloud incident response system that differentiate it from a non-cloud incident response system, notably in the areas of governance, shared responsibility, and visibility.

- **Governance:** Data in the cloud resides in multiple locations, perhaps with different CSPs. Getting the various organizations together to investigate an incident is a significant challenge. It is also resource-draining on large CSPs that have a colossal client pool.
- **Shared responsibility:** Cloud service customers, CSPs, and/or third-party providers all have different roles to ensure cloud security. Generally, customers are responsible for their data and the CSPs for the cloud infrastructure and services they provide. Cloud incident response should always be coordinated across all parties. The domains of shared responsibilities are also different between the CSPs and CSCs depending on the model of cloud services chosen, such as software-as-a-service (SaaS), platform-as-a-service (PaaS), and infrastructure-as-a-service (IaaS). This idea has to be well understood. For example, in IaaS, managing the operating system (OS) lies with the CSC. Therefore the IR responsibilities for the OS also lie with the CSC. Organizations should have a consistent and well-defined multi-cloud strategy/framework towards engaging with CSCs, CSPs, and/or third-party cloud providers. Any organization going with an 'all-in' strategy with a single CSC, CSP, or third-party cloud provider is indirectly introducing a single point of failure in case of an

outage at the service providers' end. A single CSP approach to the supply of cloud services may result in a situation where the organization's business could suffer a sustained outage in case of any failures introduced at the CSC/CSP over which the organization does not have control. This scenario will impact business operations substantially and raises the possibility of a business continuity plan (BCP) strategy unable to recover—resulting in a systemic CIR event. When approaching service provider diversity from the CIR perspective, organizations are also encouraged to consider aspects of digital service sovereignty (e.g., data residency, data sovereignty) in their plans.

- **Visibility:** A lack of visibility in the cloud means that incidents that could have been remediated quickly are not addressed immediately and are at risk of further escalation. The cloud can make for a faster, cheaper, and more effective IR if appropriately leveraged. There are already many built-in cloud platform tools, information sources, services, and capabilities provided by CSPs and their partners to significantly enhance detection, reaction, recovery, and forensic abilities. Care must be taken when developing the IR process and documentation when leveraging cloud architectures instead of traditional data center models. The CIR must be proactive and architected to sustain against failure throughout the process.

Responsibility	On-Prem	IaaS	PaaS	SaaS	
Data classification & accountability risks	●	●	●	●	Requires Internal Trust
Client & endpoint risks	●	●	●	● ●	
Identify & access risks	●	●	● ●	● ●	
Application risks	●	●	● ●	●	Requires External Trust
Network risks	●	● ●	●	●	
Host risks	●	● ●	●	●	
Infrastructure risks	●	●	●	●	

● Cloud Provider is responsible
 ● Cloud Customer is responsible

Figure 29: Cloud provider / customer responsibility

References

- Cloud Security Alliance, The Six Pillars of DevSecOps: Pillar 4, Bridging Compliance and Development, February 2022, Available @ <https://cloudsecurityalliance.org/artifacts/devsecops-pillar-4-bridging-compliance-and-development/>
- Cloud Security Alliance, The Six Pillars of DevSecOps: Pillar 5, Automation, July 2020, Available @ <https://cloudsecurityalliance.org/artifacts/devsecops-automation/>
- Cloud Security Alliance, The Six Pillars of DevSecOps: Pillar 1, Collective Responsibility, February 2020, Available @ <https://cloudsecurityalliance.org/artifacts/devsecops-collective-responsibility/>
- Cloud Security Alliance, Information Security Management through Reflexive Security: Six Pillars in the Integration of Security, Development and Operations, August 2019, Available @ <https://cloudsecurityalliance.org/artifacts/information-security-management-through-reflexive-security/>
- Cloud Security Alliance, The Six Pillars of DevSecOps, Achieving Reflexive Security Through Integration of Security, Development and Operations, August 2019, Available @ <https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/>
- Cloud Security Alliance, Cloud Penetration Testing Playbook, July 2019, Available @ <https://cloudsecurityalliance.org/artifacts/cloud-penetration-testing-playbook/>
- Cloud Security Alliance, Cloud Threat Modelling, July 2019, Available @ <https://cloudsecurityalliance.org/artifacts/cloud-threat-modeling/>
- Patel, S.: 2019 Global Developer Report: DevSecOps finds security roadblocks divide teams (July 2020), <https://about.gitlab.com/blog/2019/07/15/global-developer-report/>, [Online; posted on July 15, 2019]
- Assal, H., Chiasson, S.: 'Think secure from the beginning' A Survey with Software Developers. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. pp. 1-13. CHI '19, Association for Computing Machinery, New York, NY, USA (2019)
- Gasiba, Tiago Espinha, et al. "Awareness of Secure Coding Guidelines in the Industry-A first data analysis." 2020 IEEE 19th International Conference on Trust, Security, and Privacy in Computing and Communications (TrustCom). IEEE, 2020.
- Blomqvist, Markus, Lauri Koivunen, and Tuomas Mäkilä. "Secrets Management in a Multi-Cloud Kubernetes Environment." (2021).
- Barker, E., Barker, W., Burr, W., Polk, W., Smid, M., & Gallagher, P. D. (2020). NIST Special Publication 800-57 Recommendation for Key Management-Part 1: General.

HashiCorp, Unlocking the Cloud Operating Model: Security - What is Identity-based Security, May 2019, Available @

<https://www.hashicorp.com/resources/unlocking-the-cloud-operating-model-security>

Rose, Tina, and Xiaobo Zhou. "System Hardening for Infrastructure as a Service (IaaS)." 2020 IEEE Systems Security Symposium (SSS). IEEE, 2020.

Mourad, Azzam, Marc-André Laverdiere, and Mourad Debbabi. "Security hardening of open source software." (2006).

GitHub, Chef InSpec: Auditing and Testing Framework, August 2022, Available @

<https://github.com/inspec/inspec>

OWASP Foundation, OWASP Web Security Testing Guide (WSTG), The OWASP Testing Project, July 2022, Available @

<https://owasp.org/www-project-web-security-testing-guide/latest/2-Introduction/>

Gonzalez, Danielle. The State of Practice for Security Unit Testing: Towards Data-Driven Strategies to Shift Security into Developer's Automated Testing Workflows. Diss. Rochester Institute of Technology, 2021.

Planning, Strategic. "The economic impacts of inadequate infrastructure for software testing." National Institute of Standards and Technology (2002): 1.

Tómasdóttir, Kristín Fjóra, Mauricio Aniche, and Arie Van Deursen. "The adoption of javascript linters in practice: A case study on eslint." IEEE Transactions on Software Engineering 46.8 (2018): 863-891

OWASP Foundation, OWASP DevSecOps Guideline Project, June 2022,

Available @ <https://owasp.org/www-project-devsecops-guideline/>

GitHub, Checkstyle, August 2022,

Available @ <https://github.com/checkstyle/checkstyle>

Douglas Crockford, JSLint, The JavaScript Code Quality and Coverage Tool, July 2022, Available @

<https://www.jshint.com/>

Nicholas C. Zakas, ESLint, Find and fix problems in your JavaScript code - eslint - pluggable JavaScript linter, August 2022,

Available @ <https://eslint.org/>

GitHub, Terraform-Linters/tflint: A pluggable terraform linter, August 2022,

Available @ <https://github.com/terraform-linters/tflint>

Kubelinter, KubeLint analyzes Kubernetes YAML files and Helm charts and checks them against various best practice, August 2022,

Available @ <https://docs.kubelinter.io/>

PyPI, Bandit - Security oriented static analyser for python code, August 2022,
Available @ <https://pypi.org/project/bandit/>

PMD, PMD - An extensible cross-language static code analyzer, July 2022,
Available @ <https://pmd.github.io/>

Mateo Tudela, Francesc, et al. "On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications." Applied Sciences 10.24 (2020): 9119.

Brunnert, Andreas, et al. "Performance-oriented DevOps: A research agenda." arXiv preprint arXiv:1508.04752 (2015).

Schulz, H., Okanović, D., van Hoorn, A., & Tůma, P. (2021, April). Context-tailored Workload Model Generation for Continuous Representative Load Testing. In Proceedings of the ACM/SPEC International Conference on Performance Engineering (pp. 21-32).

Datadog, Performance Monitoring (APM), August 2022,
Available @ <https://www.datadoghq.com/product/apm/>

Tom Hall, Atlassian, DevOps metrics - Why, what, and how to measure success in DevOps, August 2022,
Available @ <https://www.atlassian.com/devops/frameworks/devops-metrics>

Acronyms

AI	Artificial Intelligence
CISA	Cybersecurity and Infrastructure Security Agency
CSA	Cloud Security Alliance
CSA CCM	Cloud Security Alliance Security, Cloud Control Matrix
CSA GSD	Global Security Database
CI/CD	Continuous Integration, Continuous Delivery
CIO	Chief Information Officer
CIS	Center for Internet Security
CISO	Chief Information Security Officer
CSC	Cloud Service Customer
CSP	Cloud Service Provider
CTO	Chief Technology Officer
DAST	Dynamic Application Security Testing
DORA	DevOps Research and Assessment
EASM	External Attack Surface Management
FFIEC	Federal Financial Institutions Examination Council
GDPR	General Data Protection Regulations
HIPAA	Health Insurance Portability and Accountability Act of 1996
IaaS	Infrastructure as a Service
IAC	Infrastructure as Code
IAM	Identity and Access Management
IDE	Integrated Development Environment

IDS	Intrusion Detection System
INVEST	Independent, Negotiable, Valuable, Estimable, Small, Testable
ISO	International Standards Organization
KPI	Key Performance Indicator
ML	Machine Language
NCSC	National Cyber Security Center
NIST	National Institute of Standards and Technology
OPA	Open Policy Agent
OS	Operating System
OWASP	Open Web Application Security Project
PaaS	Platform as a Service
PCI-DSS	Payment Card Industry Data Security Standard
PII	Personally Identifiable Information
RBAC	Role-based access control
REST	Representational state transfer
SaaS	Software as a Service
SAST	Static Application Security Testing
SBOM	Software Bill of Materials
SCA	Software Composition Analysis
SCM	Supply Chain Management
SCR	Source Code Repository
SDLC	Software Development Life Cycle
SLA	Service Level Agreement

SIEM	Security information and event management
SOAP	Simple Object Access Protocol
SOAR	Security Orchestration, Automation and Response
SOC	System and Organization Controls
SOC 2	System and Organization Controls Two
SOD	Segregation of Duties
SPII	Sensitive Personally Identifiable Information
VPC	Virtual Private Cloud
VPN	Virtual Private Network
VM	Virtual Machine
XDR	Extended detection and response