National Defense ISAC

Embedding Security Controls in a DevOps Pipeline

June 10, 2025

Authors:
Raul Barreras
Alex Greene
Terence Ho
Wayne Leeke
Waldemar Pabon
Andrew Zuehlke

Reviewers:
Keith Kernes
Will Jimenez
Renee Stegman

About the Authors

Dr. Waldemar Pabon, ND-ISAC Lead & Senior Contributor

Dr. Waldemar Pabon is a Cyber Security Architect with over 29 years of experience in software engineering. Dr. Pabon leads the Application Security Working Group, Software Security Automation, and the COTS Software Assessments Subgroups at the ND-ISAC. Under his leadership, the Software Security Automation Working Group has published eight white papers. The papers provide ND-ISAC members and the industry with a roadmap on how to adopt application security best practices while leveraging automation as a catalyst to achieve efficiencies. Dr. Pabon has a Doctor of Science in Cybersecurity degree from Capitol Technology University and a Risk Management certificate from the Harvard University.

Raul Barreras, ND-ISAC Senior Contributor

Raul is an Information Security Professional with more than 20 years of experience. His multiple roles throughout his career include information security officer, systems administrator, developer, teacher, and occasional pen tester. In recent years, Raul has had the opportunity to use the accumulated experience in a new role: application security. This role has allowed him to discover how much he enjoys being a developer advocate and how fun and useful it is to build a community of security champions while improving the quality of the organization's software.

Alex Greene, ND-ISAC Contributor

Alex Green is an Application Security Manager responsible for validating SDLC security processes, Infrastructure as Code, Applications, API's, which may include attacking products and services like an outsider to help developers better defend and follow AppSec best practices. Over 23+ years of experience in IT at some capacity in various disciplines ranging from Consulting, Network and Security Administration to AppSec Support/Site Reliability Engineer and Penetration Tester. Advocate for DevSecOps and "Everything as Code“. Co-authored "Embedding Security Controls in DevOps" ND-ISAC AppSec whitepaper.

Terence Ho, ND-ISAC Contributor

Terence Ho is a Cloud Application Cybersecurity Specialist with 5 years of experience in the Information Security field with a primary focus on developing automation, integrating security tooling, testing, and standardizing innovative solutions to enhance the security of applications in the Enterprise. He joined ND-ISAC in December 2022 and became a member of the Application Security working group to learn more from the Information Security community and contribute his expertise to the group. Terence graduated from the University of Washington with a Bachelor’s Degree in Computer Science and Software Engineering with a focus on Information Assurance and Cybersecurity.

Wayne Leeke, ND-ISAC Senior Contributor

Wayne Leeke is a Senior Cyber Security Researcher with 10 years of combined experience in application development and vulnerability research. Wayne holds a Master of Cyber Security and Bachelor of Science in Computer Information Technology from Northern Kentucky University. He has authored multiple posters and participated in both private and public research. He joined the ND-ISAC in 2024 to exchange knowledge with fellow researchers.

Andrew Zuehlke, ND-ISAC Senior Contributor

Andrew Zuehlke is a Cyber Security Architect with seven years of experience in the Information Security domain. A 2017 alumnus of Appalachian State University, Andrew holds a Bachelor of Science in Computer Science and Computational Mathematics. At his current employer, Andrew supports the global Research & Development division as well as an emerging Sustainability & Circularity division. His professional resume includes a tenure as a Cyber Security Analyst, where he managed SIEM and EDR solutions. Since joining ND-ISAC in December 2020, Andrew has been an active participant and values the exchange of knowledge with Information Security experts. His engagement with the Application Security working group has led to the co-authorship of four previously published white papers.

Executive Summary

Organizations have evolved over time to identify and adopt different strategies that will enable the Software Development Lifecycle (SDLC) to be more agile and efficient. One such strategy that helps reduce the amount of time it takes to create software is DevOps. The National Institute of Standards and Technology (NIST) defines DevOps as a set of practices intended to embed automation to enable efficiencies in the different activities performed by software developers and Information Technology operations (n.d.). These activities have the objective of delivering features, fixes, and updates more frequently which helps enrich the user adoption of software (NIST, n.d.).

A related strategy has also emerged as a key requirement to ensure risk is detected across the multiple layers of the technology landscape used to build software. As organizations include multiple layers of technology (containers, cloud, API services, etc.), the risk associated with the software in the Software Development Lifecycle (SDLC) also increases. Application Security (AppSec) offers a great opportunity for organizations to embed security as a standard component of DevOps, enforcing a DevSecOps strategy that will help identify risk at the different layers of technology.

Adopting a “shift-left” approach ensures risk is identified proactively as early as possible in the SDLC. In a DevSecOps strategy, security controls are added at different stages to ensure all the components involved in the software are covered from a risk detection perspective. Containers have a different set of risks than cloud and

Infrastructure as Code (IaC). This difference requires specific security controls at the appropriate stages to ensure that risks are identified as efficiently and completely as possible.

As technology advances, risks in the runtime environment and the components used to build software become more apparent. It is important to decide at which stage in the DevOps process an organization will incorporate security measures. Whether the security control is implemented after the software is built or at the point of deploying the software to the runtime environment, automation plays a crucial role in reducing human errors. Automation helps prevent organizations from overlooking security requirements, which could otherwise lead to misuse, software weaponization, or compromise of the runtime environment.

This paper presents the incorporation of security controls for these emerging technologies:

- Containers
- Cloud
- Infrastructure as Code

The paper also discusses how to take advantage of the “Risk Gate” approach to decide when to allow DevOps to continue to deploy software to the runtime environment and when to stop because risk has been identified. To support better-secured software designs, the paper also covers how to embed Application Threat Modeling at an early stage in DevOps. Embedding Threat Modeling will enable organizations to have a clearer picture of the potential risks that could impact the software implementation.

Understanding threats allows the developer to incorporate proactive mechanisms to verify required countermeasures are implemented in the runtime environment.

No matter what emerging technologies an organization uses to create and implement software, integrating security controls in DevOps is crucial for effective risk detection. These controls improve the organization's ability to manage vulnerabilities, detect, and eliminate risks that could be exploited by attackers. By utilizing automation, all protections and detection can be standardized, making DevOps a key component in strengthening the security of software throughout the Software Development Life Cycle (SDLC).

Table of Content

Introduction

Objective

Emerging technologies have the potential to enhance business performance, resilience, and processes. However, security remains a major concern due to the inherent risks associated with these technologies. Organizations must thoroughly evaluate these risks to minimize the impact of potential cyber-attacks. It is crucial to comprehend how different technologies introduce varying levels of risk. For instance, cloud environments and containers present unique risk scenarios. To effectively address these risks, organizations should integrate security checks at different stages of their Software Development Lifecycle (SDLC).

Application Security

In the SDLC, it is important to implement security controls such as Static Application Security Testing (SAST), Software Composition Analysis (SCA), etc. These controls help to detect threat vectors that may have been introduced by software engineers, risks associated with dependencies, and malicious behaviors. The goal of these traditional security controls is to promote a "shift-left" approach, which means accelerating the detection of risks as early as possible in the SDLC. Unfortunately, SCA or SAST cannot identify risks in the container runtime environment and have no mechanisms to detect security misconfigurations in cloud environments. These traditional controls will not help to minimize risk in emerging technologies such as Infrastructure as Code, cloud, and containers or support a comprehensive risk detection plan.

Infrastructure as Code (IaC) provides a mechanism for organizations to standardize the way resources are deployed to the environment. The use of code for the deployment of resources reduces the risk of missing configurations that otherwise may expose the organization to risk. At the same time, the standardization provided by IaC will help organizations navigate regulatory compliance requirements by ensuring all resources adhere to the expected settings and configurations. Nevertheless, the lack of monitoring of the changes made to IaC could introduce security misconfigurations and non-compliance issues.

On the other hand, containers have multiple layers of risk. The host operating system supplies the necessary services for the container environment. The images used by running containers consist of dependencies required to provide various features to the end user. The registry contains an inventory of all the images available to the container environment. The runtime is used to execute each image to make the applications operational. An attacker can exploit any of these layers to compromise the environment and potentially the infrastructure.

Cloud environments can be used to scale up or down resources as needed in support of better performance. These environments will also accelerate the time-to-market with its many services and features while minimizing the level of responsibility by the customers based on a shared responsibility model. Despite this, security misconfigurations and non-compliance issues can also impact cloud environments and expose internal resources to the Internet. This is one of the most problematic scenarios in the cloud.

As organizations leverage emerging technologies and combine them with automation, it's crucial to understand how to integrate security as an essential component from the start.

The necessary security checks at various stages of the DevOps cycle will depend based on the technology. This paper aims to explain how security can be integrated into these stages, emphasize the significance of a risk gate approach, and demonstrate how automated threat modeling can proactively detect risks, and contribute to a robust security posture overall.

Audience

The audience for this white paper includes security engineers, software engineers, cloud architects, and product managers responsible for securing the environments where software will execute or be provisioned.

Structure of the paper

This paper introduces the importance of implementing additional security controls as part pf the SDLC practice to provide comprehensive risk detection in DevOps. The paper first introduces the concept of embedding security controls in the SDLC as a standard practice. Infrastructure-as-Code is presented; benefits and challenges are discussed as well as the strategy needed to detect security misconfigurations and non-compliance. A discussion on how to secure the four layers of risks in containers follows, which also includes strategies on how to detect and manage risk.

Best practices and compliance concerns are discussed to cover additional important aspects of securing Cloud resources by leveraging automation in a DevOps pipeline. Feedback loop integration is introduced as a mechanism to provide constant feedback to the development team regarding security risks detected during the execution of the security controls in the DevOps pipeline. The risk gate approach is also presented as a mechanism to stop the execution of a DevOps pipeline for those critical conditions that are identified as critical conditions to stop the automation process. Finally, the automation of Threat Modeling in a DevOps pipeline is introduced along with approaches on how to monitor for architecture drift and risk in software architectures.

Embedding Security Controls in a DevOps Pipeline

Overview

More organizations are seeing the benefits of adopting DevOps automation. Utilizing Continuous Integration, Continuous Delivery/Deployment (CI/CD), development teams can reduce the friction and frustration associated with manual legacy Software Development Lifecycle (SDLC) processes. With all the benefits automation brings to the speed and velocity developer teams' can deliver code, why do some organizations still encourage security teams to operate in siloes separated from development?

Figure 1 Security controls used to protect data and systems. [Image], by Gate For Security, n.d., (https://gateforsecurity.com/en/end-user-security-systems)

Opportunity for Change: Embedding Security into the SDLC (DevSecOps)

DevSecOps is the practice of integrating security testing at every stage of the software development process. It includes tools and processes that encourage collaboration between developers, security specialists, and operation teams to build software that is both efficient and secure (Amazon, n.d.).

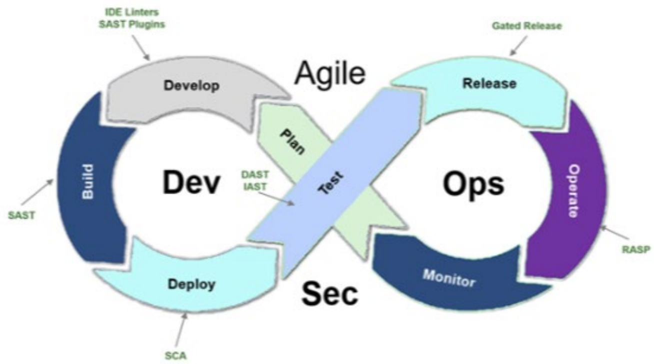


Figure 2 DevOps methodology to achieve efficiencies through automation. [Image], by Heim et al., 2020, (https://ndisac.org/wp-content/uploads/ndisac-security-automation-white-paper.pdf)

Transitioning from a legacy SDLC approach to include DevSecOps can be difficult. Organizations interested in adding security automation to their existing traditional waterfall and agile scrum methodologies should take the gradual organic approach to avoid business continuity issues. The white paper published by ND-ISAC "Software Security Automation: A Roadmap toward Efficiency and Security" offers guidance for organizations transitioning from legacy into a DevSecOps SDLC approach. DevSecOps is more than just tools and automation. DevSecOps is unlikely to succeed without a corporate culture shift that includes connecting teams, sharing knowledge and processes, and security tooling.

Connected Teams: Continuous Communication and Sharing of Processes

Although individualism, ownership, separation of duties and RACI (Responsible, Accountable, Consulted, Informed) are important, we should not overlook the benefits of information sharing. Continuous communication allows developers, security, and operation teams to collaborate on ideas and decisions, similar to how application components share data/procedures within loosely/tightly coupled systems that operate holistically.

Creating a culture of sharing security related ideas/tooling/monitoring/reporting and vulnerability remediation advice, will help developers and security teams stay connected.

AppSec teams should try to understand the development team workflow and existing tooling in order to assist or advise the placement of security controls in a frictionless manner. Staying informed of the developer's project deadlines and prioritizing what security vulnerabilities need to be fixed before release, shows empathy and builds strong work relationships.

Security Knowledge Sharing: Continuous Knowledge Enablement (Security Champions)

Application security hardening is a tough task for everyone. The moving targets and constantly changing landscapes provide obstacles that make it difficult to stay ahead of the curve and achieve the ideal security posture. The mindset to keep your organization secure should be shared with everyone and not just reserved to the limited number of individuals on the security team.

Figure 3 Putting a Face to Software Security Champions. [Image], by Beckers et al., 2019, (https://safecode.org/blog/putting-a-face-to-software-security-champions/)

Embedding security champions within different application teams is a good way of sharing knowledge and expertise. Security champions are usually also developers with passion/knowledge and interest in cyber security. They can assist with collaboration efforts by proxying security awareness and risk knowledge to development teams.

Technical Tools Sharing: Embedding Security Controls into the DevOps Pipeline

Embedding security controls directly into the various stages of the DevOps pipeline can be beneficial to everyone. Sharing tools and technical processes creates a cohesive atmosphere where teams work together for a common company goal. It provides a faster security feedback loop for developers, security and operation teams. Including security controls into all stages of the automation pipeline helps organizations lower the risk of major vulnerabilities ending up in production.

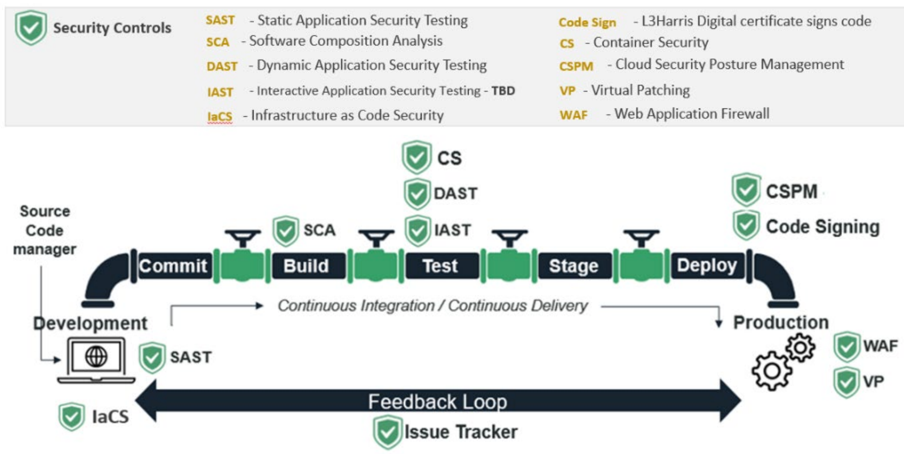


Figure 4 Standard DevOps pipeline with Security Controls

Infrastructure as Code (IaC)

What is Infrastructure as Code (IaC)?

Infrastructure as code is a mainstream pattern for managing infrastructure with configuration files rather than through a graphical user interface or through manual command line script sequences (Hashicorp, 2023).

How is it used?

By using a configuration specific language, a DevOps engineer can write code to provision, modify and manage the infrastructure components in a codified consistent manner avoiding manual ClickOps mistakes.

How is IaC beneficial?

Automation: Converting infrastructure, policy and configuration management as code allows organizations to utilize automation pipelines to deploy scalable secure solutions. Codifying workflows can help the business quickly create/deploy and maintain consistent infrastructure, reducing costly human errors from repetitive UI click operations (ClickOps).

Repeatable: A DevOps engineer can reuse the same code to provision multiple environments in parallel.

Readable: Configuration code is usually easier to read and write, which encourages teams from various technical backgrounds to start coding their specific processes quickly.

Traceable: Code can be stored in source code repositories to track, control and scan changes for security and compliance purposes.

Visibility: When Operations and Security teams store their IaC projects in source control, it brings a better awareness of how their processes change over time, what they are creating and how it can help the business.

Share/Reuse: Development, Security, and Operations teams can create and share reusable modules that could potentially be consumed into present or future team projects.

Some common use cases for infrastructure as code are:

- Infrastructure deployments
- CI/CD application deployments
- Configuration management
- Public/Private or Hybrid cloud automation
- Security automation
- Network automation
- Support Compliance and Standardization

IaC Approaches and Industry Formats

Infrastructure as code is written in configuration files that contain specific syntax and structure. The various IaC tools process the instructions/declarations contained in the configuration files to create or update the intended target.

Table 1 provides some common IaC file formats used in industry.

Table 1

Common IaC Formats

FORMAT	DEFINITION
HCL	HashiCorp Configuration Language
JSON	JavaScript Object Notation
YAML	Yet Another Markup Language
XML	Extensible Markup Language

Infrastructure as code tools use a declarative or imperative approach to process the code. The declarative IaC approach declares the infrastructure an organization would like provisioned/updated while tracking and changing the current state to reach the desired state defined in the configuration file. The imperative IaC approach defines scripts or configuration code, to control and execute each procedural step, needed for provisioning/updating the infrastructure.

Table 2 summarizes some common IaC tools used in the industry for Multi-Cloud/Single-Cloud provisioning.

Table 2

IaC Tools

ENVIRONMENT	TOOL	COMMENTS
Multi-Cloud	Terraform	Specializes in various public cloud infrastructure provisioning, using the declarative approach which tracks the system state.
Multi-Cloud	Ansible	Specializes in configuration management using the imperative approach for step-by-step control. Ansible can also handle provisioning infrastructure and various automation tasks effectively in the public cloud or private cloud/datacenter.
Single-Cloud	AWS CloudFormation	CloudFormation is used to provision AWS cloud services.
Single-Cloud	Azure ARM	Microsoft Azure Resource Manager is used to provision Azure cloud services.
Single-Cloud	Google CDM	Cloud Deployment Manager is used to provision Google Cloud Platform services.

Here is a small snippet of code for a popular IaC tool named Terraform utilizing the HCL syntax:

```
resource "azurerm_linux_virtual_machine" "kali" {
  name                = var.osName
  resource_group_name = data.azurerm_resource_group.rg_existing.name
  location            = data.azurerm_resource_group.rg_existing.location
  size                = var.vmSize
  admin_username      = var.adminUser
  network_interface_ids = [
    azurerm_network_interface.kalboxnic.id,
  ]

  admin_ssh_key {
    username   = var.sshUser
    public_key = file("~/.ssh/id_rsa.pub")
  }
```

Figure 5 IaC code snippet example.

Infrastructure as Code can be written in a format that supports compliance and standardization. This can be accomplished by following industry best practices along with existing organizational policies and standards. Remembering industry best practices and compliance standards can be difficult for any team while coding infrastructure. As humans, we may on occasion forget something important while trying to meet business deadlines and accidentally put our organization at risk. Is there something organizations can do to make sure the IaC is compliant? Policies are enforced? The answer is using Compliance as Code (CaC) and Policy as Code (PaC).

What is Compliance as Code?

Compliance as Code treats compliance requirements as code, which you can manage, version, and automate along with the rest of the software development process.

How is this any different from Policy as Code (PaC)?

There is overlap between policy as code and compliance as code, but there are key differences—the most notable being that compliance as code focuses on enforcing regulatory requirements, while policy as code can enforce any type of organizational policy (Trend Micro, 2023).

To learn more about Policy as Code and how it applies to cloud computing read the Codifying Cloud Infrastructure and Security Controls section of this paper.

How and where do we integrate IaC Security in DevOps?

IaC security can be applied at different stages of the SDLC. Developers can run IaC security audits locally or remotely combining different tools in their CI/CD pipelines. If the infrastructure is live and running in production, automated policy violation scans can trigger controls to block or adjust non-compliant infrastructure.

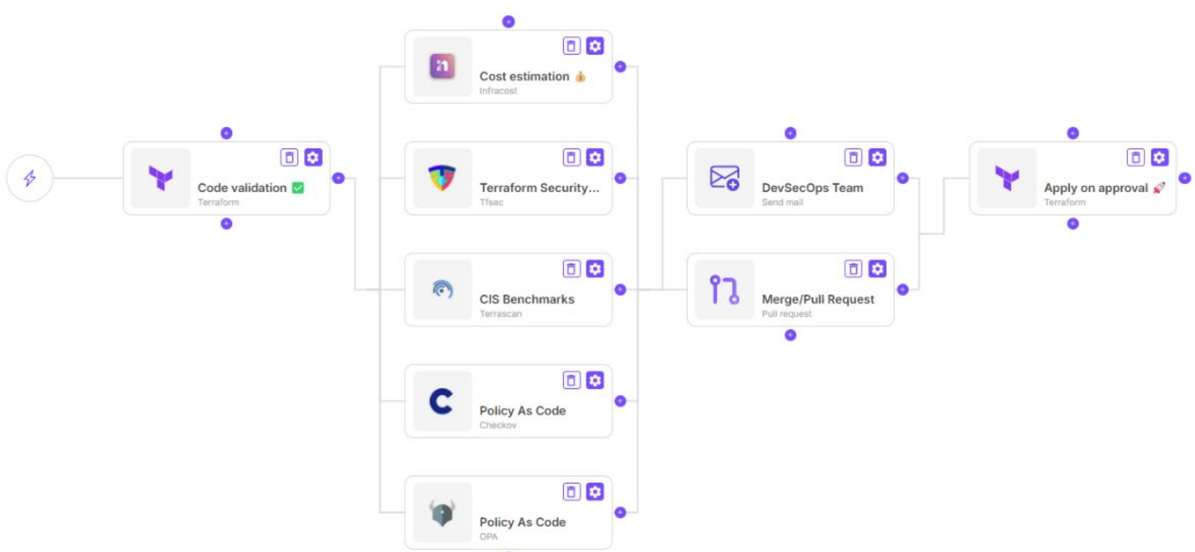


Figure 6 IaC security automation pipeline with manual delivery approval.

There are various open-source and commercial IaC security tools that help identify compliance violations and security risks in IaC or deployed infrastructure. Table 3 provides some common IaC security scanning tools used in the industry.

Table 3

Software supporting IaC Security Scanning

Open-Source	Commercial
Checkov	Snyk
TfSec	Aqua Security
TerraScan	VeraCode

It is good practice to run your security checks early in the SDLC which is referred as "Shifting Left" in the industry. There are various methods to apply security checks early which can provide a faster feedback loop for developers and reduce the likelihood of security misconfigurations entering production. With the power of modern laptops and the wide adoption of containers, some developers can run the same security automation checks locally. This process can potentially provide the fastest feedback loop and reduce risks before pulling/merging into the remote repositories. For instance, developers can use client-side Git hooks with security focused pre-commit scripts or use a security scanner plugin directly in their integrated development environment (IDE) to scan the IaC. Figure 7 provides some examples of IaC security scanners that could be installed as VS Code IDE extensions.

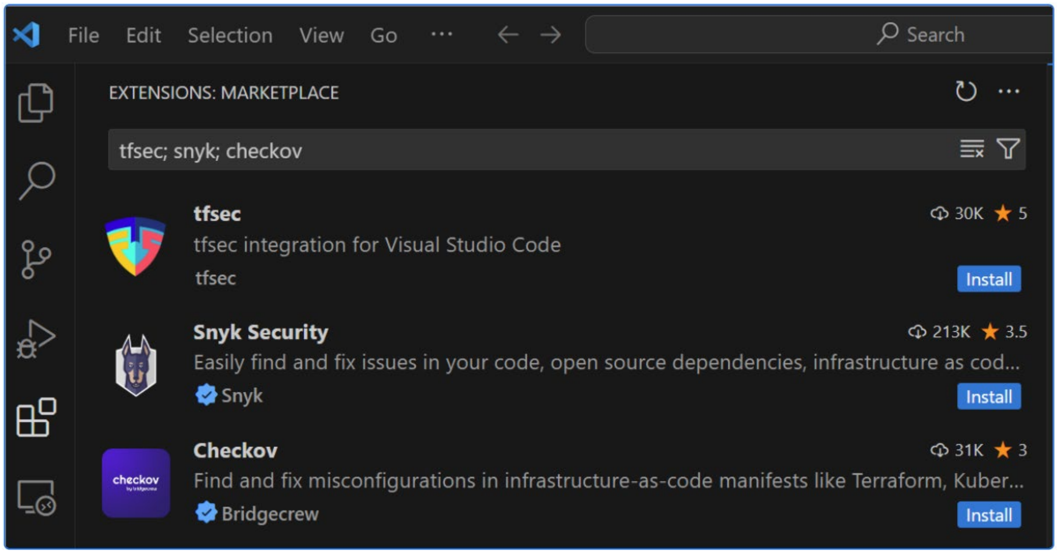


Figure 7 IaC security automation pipeline with

VS Code Dev Containers allow developers to create and quickly switch to container development environments. You can open an IaC code project and run security scans inside an isolated Docker container, without worrying about making changes to your laptop. Figure 8 shows the Dev Container feature which is available in the Visual Studio Code IDE.

Figure 8 VS Code Dev Containers Option in the IDE

Another option available to developers to trigger the validation of code is Git hooks. Git hooks are scripts that execute when a received Git event occurs. Git hooks have two categories, client-side and server-side. The client-side hook scripts are triggered during GIT merge/commits and server-side on pushed commits.

Client-side pre-commit hooks can be used to perform security scans locally on the developer's workstation and can block the commit if the audit checks fail. Below is a simple example of a client-side pre-commit hook script placed in the directory .git/hooks. When the script runs it will perform a security scan with two different scanners and exit with a non-zero status code if either scanner fails.

Container Security

Technological innovation has delivered containers as the strategy of optimizing resource utilization has grown in the industry. Instead of having competing memory and processing power requirements with operating systems services that are not always needed for the application to operate, containers provide a streamlined approach to enable features to the end user in an optimized manner. This creates greater efficiency from an operational standpoint, but there are security risks involved.

Containers, by their own nature, have 4 layers of risks:

- Host running the container environment
- Images
- Image Registry
- Container Runtime

Figure 9 provides the relationship between the layers in a container environment and the associated risk of each layer.

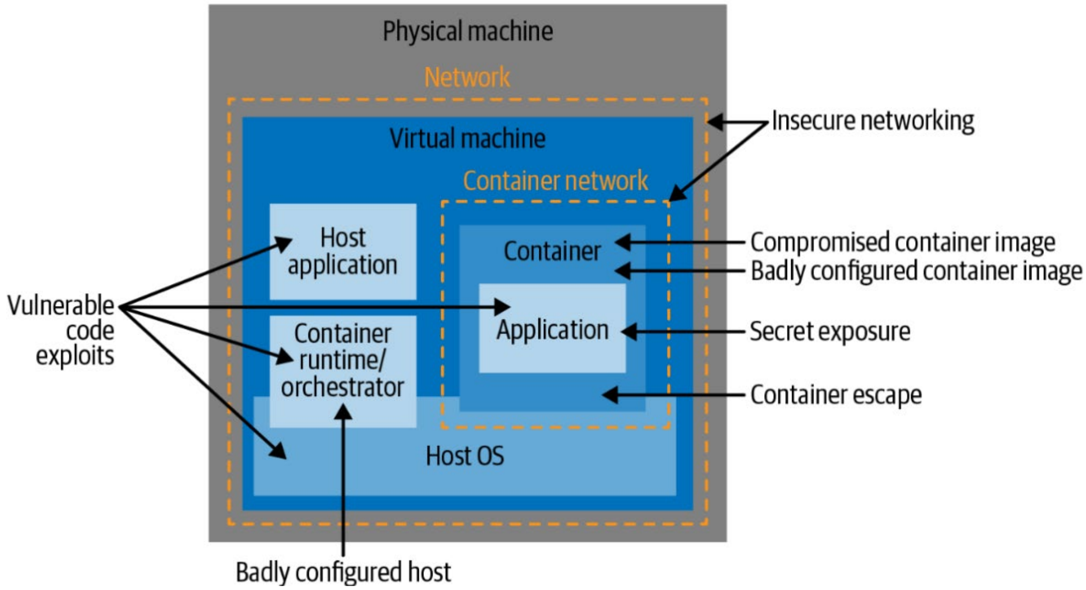


Figure 9 Containers layers and associated risk. [Image], by Rice, 2020, (https://www.oreilly.com/library/view/container-security/9781492056690/ch01.html)

Understanding the use of the container layers is important to define the DevOps strategy. In DevOps, the stages in the pipeline are created to support a build cycle. In that process, the production environment very often becomes the last destination for the deployment target. Teams have the option of replicating the production environment as part of the test environment included in the testing stage. This will guarantee that risks can be identified before the final production stage is reached.

To enhance the security of containers, it is crucial to comprehensively understand the various use cases and how to determine the specific security measures required in DevOps. Different layers of security risk for container security may necessitate different strategies or the use of different stages (i.e., build stage vs. deploy stage) for risk detection. As shown in Figure 9, certain container layers are linked to the runtime environment, while others affect the components used in the runtime.

Risk Layer #1: Images

Container images are the basic unit used during the container runtime to provide the features and capabilities to the end user. The images contain all the different dependencies and components needed for software implementation to execute. Due to the existence of these components, an image could hold risks associated with its dependencies just like a web application could hold risks in its dependencies. When an organization evaluates the right stage in DevOps to detect the potential risk of these dependencies, the right stage would be after the image build.

The immutable nature of images requires proactive risk detection to be integrated into the security scanning process after the build stage. Once the image is created, the container security platform will have all the necessary components to assess the potential level of risk. Figure 10 illustrates when security scanning should be performed in DevOps to identify risks in images. Depending on the results of the risk assessment, the DevOps pipeline will need to determine the appropriate action to take. The "Risk Gate Approach" section in this paper provides detailed information on controlling the flow of the DevOps pipeline and its impact on execution.

Figure 10 Embedding the container image scanning in a DevOps pipeline.

Risk Layer #2: Container Runtime

The runtime environment in a container implementation will have similar layers of risk as any other production deployment. Malicious behaviors and newly detected risks in dependencies are among several examples that will help understand why it is important to scan the container production environment. It is also important to understand how the runtime environment will be deployed and implemented. For example, if the organization has a fully functional container environment, the most logical step to detect risk is to verify during the deployment stage that the security monitoring required by the container runtime is deployed and functioning.

From that perspective, after the deploy stage in the pipeline, organizations can include a “Verify” stage to check if the runtime has the continuous monitoring needed for the container environment and if not, deploy what is needed. Many of the container security platforms provide API Services that can be used to validate if an environment has already been monitored as well as to gather the requirements to deploy the monitoring capabilities if needed. Figure 11 depicts the stage to use when checking for runtime detection in a container environment.

Figure 11 DevOps stage to verify runtime detection requirements are in place.

After the "Deploy" stage, having a "Verify" stage will allow the organization to confirm that all necessary countermeasures and protections are in place to maintain a strong security posture. Utilizing API services and automation will eliminate the need for manual validation and establish a self-sustaining environment, ensuring standardization of security controls and detection mechanisms in the container environment.

Risk Layers #3 & #4: Host OS and Container Registry

The services, software, and components deployed in the host OS of the container environment represent the third layer of risk. All these technological components could enable an attacker to compromise the container environment if risk is present. Because the main purpose of the host OS is to provide all the capabilities needed by container implementations to operate, the requirement to review the risk is part of the continuous monitoring strategy.

From that perspective, the approach to follow in DevOps is to verify the security requirements are in place in the host OS after the “Deploy” stage. Just like the strategy used for the container runtime, Figure 11 in the section “Risk Layer #2: Container Runtime” depicts the stage in DevOps where organizations have an opportunity to verify through automation the enforcement of a strong security posture and detect risk in the host OS.

The container registry operates similarly to the host OS. This part of the container environment is not built during the image creation process in the DevOps pipeline; instead, it is an integral part of the container environment itself. Therefore, the verification process for assessing risk within the registry should mirror the verification process for the host OS in the DevOps pipeline.

In the industry, there are various options for detecting risks in container environments. DevOps aims to eliminate human friction. Consequently, organizations should assess how container security solutions can contribute to streamlining processes through automation. They should choose security platforms that are in line with the organization's security strategy and objectives.

Cloud Security

Cloud Security is used to describe a collection of methodologies to reduce the risk of both internal and external threats to resources deployed into Cloud environments. Some methods include and are not limited to:

- Restricting port access to cloud resources such as virtual machines
- Validating that only users and service accounts that require access to cloud resources have the least privilege needed to perform their tasks
- Performing Infrastructure as Code (IaC) scanning
- Detecting configuration drift
- Containerized deployments
- Container scanning
- Secrets Vaulting e.g. using a secure secret manager
- Compliance with security frameworks and government regulation
- Policy enforcement and auto-remediation

Cloud Security is essential to protect cloud computing resources, and the data stored in Cloud environments. Cloud computing has many layers of risk such as security configurations, infrastructure, containers, services, compliance, secrets management. The technological layers used to build an application will help understand how to protect Cloud applications. Once we have an understanding of these layers, then we can understand the areas of risk and ways to address them in each layer.

As described in a previous ND-ISAC whitepaper on "How to Protect Cloud Native Applications", there are 3 main cloud services model as shown in Figure 12 below:

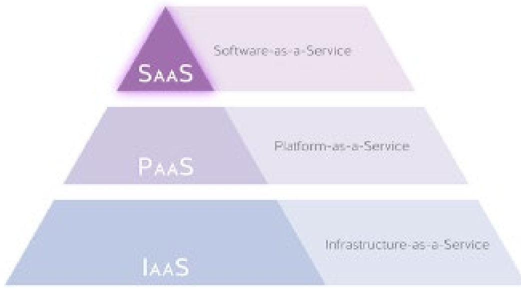


Figure 12 Main Cloud Services. [Image], by Sandoval, 2015, (https://nordicapis.com/living-in-the-cloud-stack-understanding-saas-paas-and-iaas-apis/)

Software as a service (SaaS) is a cloud computing model where software applications are provided over the internet. The software provider hosts and maintains the application, manages the underlying infrastructure and handles software updates and upgrades. They are responsible for the application's security, while the user is responsible for its data and granting proper access. Examples of Saas applications include email, customer relationship management (CRM) software, project management tools, and accounting software.

Platform as a service (PaaS) is a cloud computing model where users can rent and access a complete platform for developing, running, and managing their applications over the internet without worrying about the underlying infrastructure. PaaS platforms typically provide features such as automatic scaling, load balancing, and application monitoring, which can help improve applications' availability, performance, and scalability. In this mode, users are responsible for the application running on the platform and their data. Examples (but not limited to) of PaaS are AWS (Amazon Web Services) Elastic Beanstalk, Heroku, and Red Hat OpenShift.

Infrastructure as a service (laaS) is a cloud computing model where users can rent and access computing infrastructure resources, such as virtual machines, storage, networking, and other computing resources, on a pay-as-you-go basis. The cloud provider hosts and manages the underlying physical infrastructure, such as servers, storage devices, and networking equipment. Users control those virtual machines' operating systems, applications, and other software. Some laaS providers include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and Oracle Cloud.

In the Open Systems Interconnection (OSI) model by ISO, the lowest layer is the physical layer, then the data link layer and network layer. These lowest layers in the OSI model would be within the Infrastructure as a Service layer for Cloud applications. The network and transport layer would be within the Platform as a Service layer. The Application layer is where the application and user interface live, equivalent to Software as a Service (SaaS). Figure 13 provides a breakdown of the OSI layers.

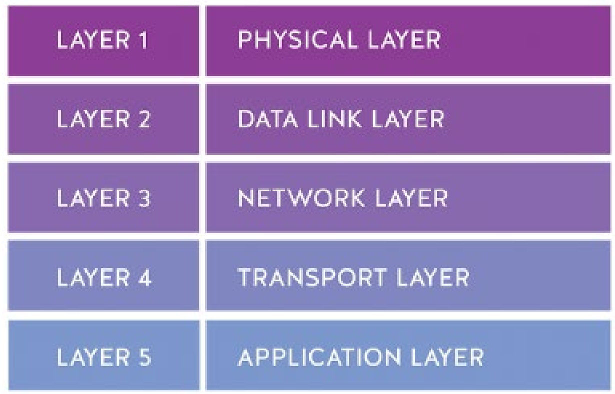


Figure 13 OSI Layers. [Image], by Sandoval, 2015, (https://nordicapis.com/living-in-the-cloud-stack-understanding-saas-paas-and-iaas-apis/)

Codifying Cloud Infrastructure and Security Controls

Codifying and automating security processes are essential in embedding security controls in DevOps. This is because code can be version controlled and easily be redeployed with an exact configuration, reducing the opportunity for security misconfigurations. For instance, in the IaaS layer, Infrastructure as Code (IaC) can be used to codify configurations of resources to ensure specific security requirements are built-into IaC files. The PaaS layer’s network and transport configuration could also be codified using IaC.

Policy as Code (PaC) can be used to enforce security controls across multiple layers of the cloud application’s stack. For example, PaC can be used to detect and report security misconfigurations to developers. Similar to how SAST is run, PaC can be run on IaC as a codified policy check to determine if an application is compliant with security controls. This compliance information can be reported immediately in a pipeline to developers with a pipeline evaluating a PaC bundle on the application code repository, providing a quick feedback mechanism to developers to remediate their code and even providing leaders with more accurate risk assessments of applications in an enterprise.

Another way PaC can be used is to enforce security configurations of an application by blocking services that violate the policy based on their IaC or run-time configurations. For example, many organizations put controls in place to determine how compute instances are provisioned. This helps prevent unwanted changes, regardless of the users' intentions or experience and knowledge levels. It is common practice to see controls that prevent compute instances from existing if they are associated with public IP addresses. The controls act as guardrails (Ray, 2024, p. 1). Figure 14 showcases the structure of Policy as Code.

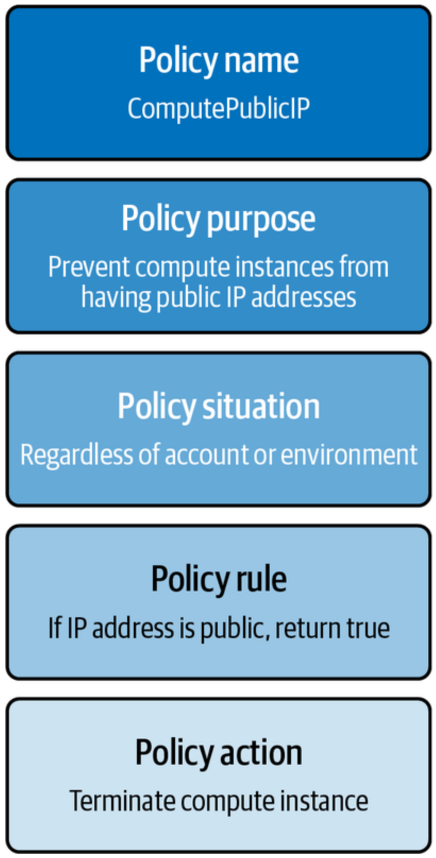


Figure 14 Policy as Code Structure. [Image], by Ray, 2024, (https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781098139179/files/assets/paco_0101.png)

This approach would immediately reduce the attack surface if the PaC is triggered during the run-time of an application and detects non-compliance or security misconfigurations.

Finally, PaC can also be used to trigger modifications to resources in run-time to adjust the security misconfigurations to become compliant again using a remediation mechanism. This would reduce the window that attackers could use to exploit an application if it was misconfigured. Figure 15 exemplifies the activity diagram showing how unwanted configurations can be stopped using PaC to determine the decision.

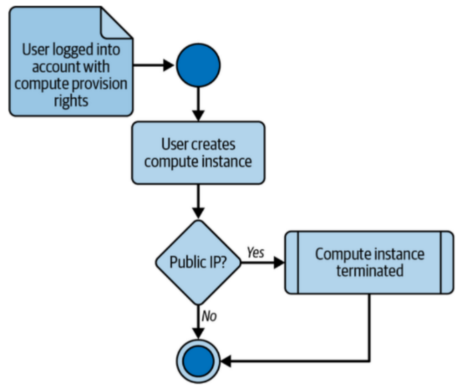


Figure 15 Policy as Code Structure. [Image], by Ray, 2024, (https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781098139179/files/assets/paco_0101.png)

Shifting Cloud Security Left in the Development Cycle

Cloud Security can be shifted left in the development cycle by adding processes earlier in the DevOps cycle such as adding IaC scanning in the CI/CD pipeline to scan any code being developed. This can provide near real-time feedback when writing code, for example using an IDE (Integrated Development Environment) plugin with Static Application Security Testing (SAST). When code updates are made, and changes are pushed out into a code repository, a full SAST scan of the code repository could be performed to detect application code security vulnerabilities immediately. Similarly, IaC scanning can be performed in the pipeline and using an IDE locally during development can provide a quicker feedback loop to developers for them to resolve any security misconfigurations. A bundle of PaC based on security frameworks and security best practices would help to detect any IaC security misconfigurations and compliance. PaC can also be used to trigger automations in run-time to remediate or block resources with security misconfigurations.

Other run-time evaluations could be run on a temporary instance of a build so that techniques such as DAST, penetration testing, run-time log analysis could help to detect any run-time vulnerabilities. Figure 16 shows how a build pipeline can include code analysis, but also trigger an external AppSec pipeline that might take more time.

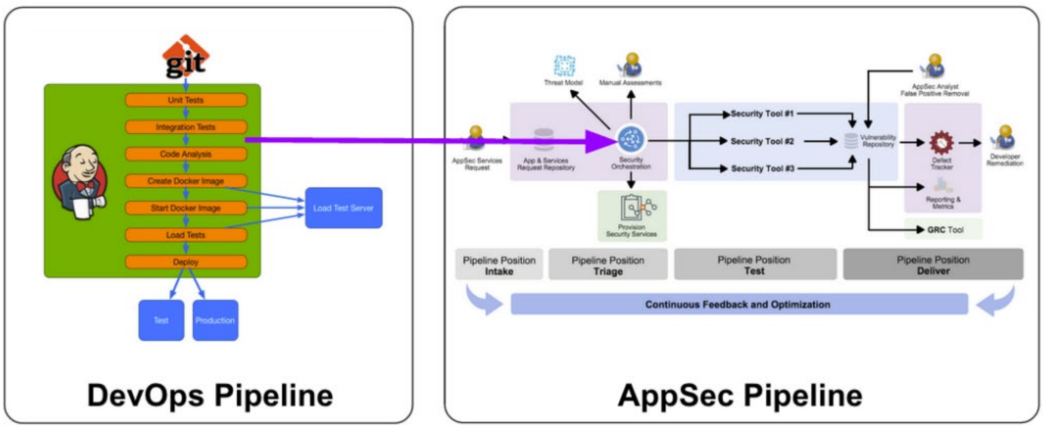


Figure 16 DevOps pipeline triggering an AppSec pipeline. [Image], by OWASP, 2020, (https://github.com/OWASP/www-project-appsec-pipeline/blob/master/assets/images/DevOps_AppSec_Pipline_Integration.png)

This enables additional more detailed security analysis, providing an opportunity for penetration testers to dive deeper into the vulnerabilities that may be found in a more in-depth analysis. Any findings throughout the various phases of security testing during the development process should be fed to the developer's backlog to be resolved.

Feedback Loop Integration

Many scanning tools and services have a means of integrating into issue trackers, providing a degree of parity with existing user stories and other application defects.

Integrating provides the immediate benefit of ease of use, which helps drive adoption of a tool. More importantly, findings now factor in the curation and selection of user stories and the handing of tech debt. The goal of many development teams is to select additions, fixes, and features that provide the most value during a sprint. Modern issue tracking provides seamless experiences with existing processes, rather than let findings be an afterthought.

Dependencies and code are introduced at different stages of development and at each stage feedback should be provided. Those interested in the security of software supply chains specifically should consult a framework such as: Supply-chain Levels for Software Artifacts, or SLSA. These frameworks shift focus to the components and processes that comprise the act of deploying an application. Techniques like integrating scanning tools into existing CI/CD pipelines can provide 'near' immediate feedback when building applications. Shifting more left, feedback can be provided directly when creating pull requests to merge code. Many tools can now analyze a software bill of materials (SBOM) for known vulnerabilities in third party components. Continuous scanning at all layers of an application can alert on new and known vulnerabilities during all phases of the SDLC process.

Organizations should decide on their risk tolerance and tune reporting to match what they decide is unacceptable risk. Risk calculation usually requires some contextual information that scanning tools lack. It is possible that although a finding is low, it may still need to be remediated due to compliance reasons (CMMC, PCI, etc.). Whether an application is internet facing typical increases overall risk of vulnerability. A security analyst can review findings using a calculator like the OWASP risk rating system before releasing findings to an issue tracker or can configure scanners to use a different default risk for CVEs. If practicing DevSecOps, communication between teams is necessary and security experts should participate in the feedback loop.

Production, internet facing environments should block violations rather than simply reporting and providing feedback. Technical controls can be leveraged to prevent the accidental deployment of unsafe code and artifacts to critical environments. Internal, or developmental environments may not need such strict scrutiny. It may be ill advisable to mandate anything that would slow the speed of development. By providing immediate, or near immediate feedback only, this allows multiple developers and security experts to freely work features and defects simultaneously. All members of the team should draft and agree upon appropriate procedure and sign off process for each environment.

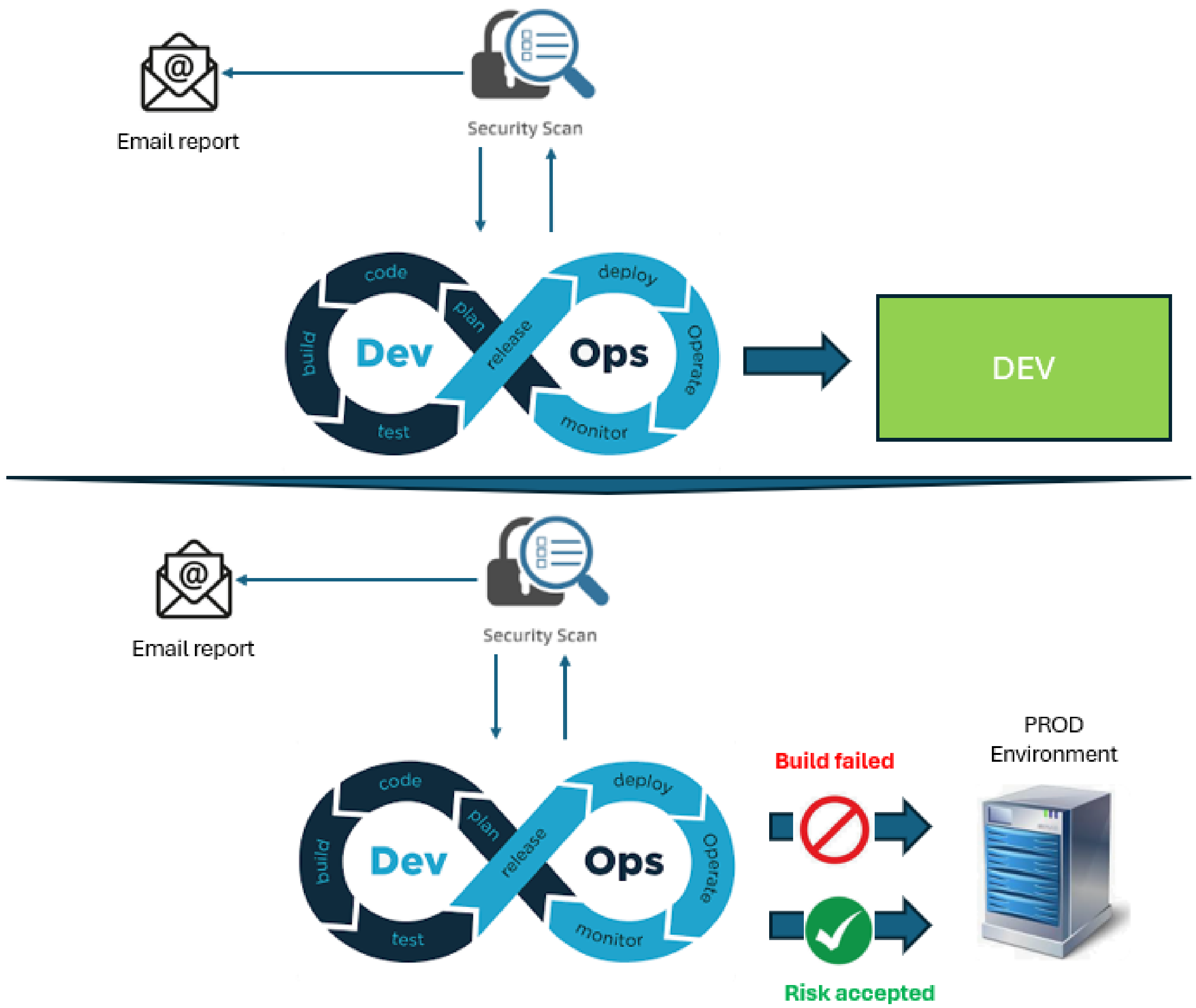


Figure 17 Having a nuanced strategy for deployments is key to balancing security, with impact on ease of development.

Risk Gate Control

What are gates?

Gates are a common part of our daily lives, though we may not always refer to them as such. In its simplest form, a gate is a barrier—logical or physical—that serves to protect a boundary and prevent the progression of potential risks. Consider your morning routine: you encounter a logical gate when you look at your calendar and determine how busy your day is and whether you will need to pack lunch. As you leave your home, you navigate through a physical gate—your front door, which requires a specific key. Then, you might open a garden gate that simply requires you to be tall enough to reach the latch. Your car presents another physical gate, which you unlock before a logical gate requires you to check the fuel level to ensure you can reach your workplace without needing to stop. Gates are everywhere and present challenges or requirements that must be met, prior to allowing an object forward.

What is a Risk Gate Control in DevOps?

In a DevOps pipeline, Risk Gate Controls are checkpoints that manage the development flow by introducing criteria-based barriers—“gates”—at strategic points. These gates receive inputs from various sources, and if all the necessary conditions are met, the code is allowed to progress further in the pipeline. Conversely, if any condition fails or does not respond in time, the code is held back pending additional review or correction. With this context, Risk Gate Controls ensure that only code satisfying a predetermined set of criteria advances through the pipeline.

Traditionally, gates in the DevOps pipeline have been used to maintain code quality across release cycles. However, their application can be applied to other areas, including code security and infrastructure health management, among others that will be discussed in upcoming sections. Risk Gates can be either manual—requiring human approval, often after manual tests—or automated—utilizing tools for testing and approval. Manual gates, while potentially more time-consuming and less efficient, have the advantage of detecting issues that automated tests may miss.

Automated gates, on the other hand, enhance efficiency by enabling automated testing and approvals. However, the overuse of automated gates can lead to a complex pipeline. To prevent overuse, a gate should be implemented only when there is a clear, repeatable, and quantifiable requirement. This ensures a balanced and effective use of gates within the DevOps pipeline.

Why Use Risk Gate Controls in DevOps?

The goal of a well-designed DevOps pipeline is to facilitate continuous integration, deployment or delivery, and monitoring of code throughout the development process. Gates can be introduced to this workflow to ensure all code meets baseline criteria before moving forward in the pipeline. However, when integrating gates into a workflow, there is a required balance between enabling efficiency and ensuring compliance with standards, and organizations must define gates in accordance with individual requirements.

When correctly implemented, gates proactively find issues and prevent poor quality code —whether it is vulnerable or not adhering to best practices—from reaching production. The earlier a gate is incorporated into the pipeline, the more efficient it becomes at identifying issues early on. Without such gates, errors might go unnoticed until later stages, potentially requiring significant rework to address and mitigate. Gates can also be used to ensure a healthy code environment. In situations where developers worked on individual components of code in isolated environments, gates can be used to ensure appropriate interoperability between code modules when they are merged. Additionally, gates can test and verify the infrastructure's readiness before deployment, ensuring a smooth transition to production.

The Importance of Quality Gates in a Pipeline

Quality gates in a DevOps pipeline must strike a balance between speed and thoroughness. While quality gates alone cannot achieve this equilibrium, they contribute to an efficient framework. Each gate should be designed with a distinct purpose and support a swift and secure transition of code into production. For a gate to function effectively, it must have access to all necessary information at the time of execution to accurately determine if the baseline criteria have been met. This ensures that the pipeline remains streamlined and that the gates serve their intended function without causing unnecessary delays or complications.

How Risk Gate Controls Affect the DevOps Pipeline?

Risk Control Gates in a DevOps pipeline are as varied as the gates we encounter in everyday life, differing in their testing scope, timing, and response mechanisms. Gates can be defined at the start or end of a given stage. They can run in parallel or be contingent upon specific prerequisites for progression. Gates may test for a binary status or assess overall coverage. Gates can be strategically placed throughout a CI/CD pipeline, from the initial code checkout, ensuring a consistent development environment—to post-deployment, verifying proper code function in production.

In a standard CI/CD pipeline, gates can be utilized with static code analysis to ensure that coding and security best practices are adhered to; as detailed in ND-ISAC's white paper "Software Security Automation: A Roadmap toward Efficiency and Security," these tests are run against static code, prior to it being deployed. Further along in the pipeline, gates can also be utilized with dynamic code analysis, testing live code for additional quality and security assurances.

While many gates operate on binary decisions—true or false—some support thresholds customized to an organization's specific needs. These thresholds might evaluate overall code coverage or pass rate percentage rather than individual test outcomes. A gate opens only when the minimum criteria are met, which could be a 90% pass rate for one organization and 99% for another.

Risk Gate Controls in a DevOps pipeline can be context aware, making decisions based on the application's deployment environment. For example, a gate might evaluate whether an application is internet-facing or internal. An internet-facing application could prompt a gate to halt the pipeline, while an internal application might only generate an alert, allowing the pipeline to continue. Some security controls can automatically signal to stop the pipeline, but others may need manual intervention, which can increase management overhead.

Incorporating automated tests within gates enhances the efficiency and effectiveness of a CI/CD pipeline. Manual processes often slow down the pipeline, but there are instances where manual intervention may be necessary. Therefore, it is important to have the capability for manual overrides on automated gates for urgent issues. To maintain accountability, organizations can consider implementing additional conditions for manual overrides, such as requiring approval from at least two individuals from different teams before proceeding.

These capabilities enable Risk Gate Controls to allow them to not only be effective in maintaining code quality and security but also flexible enough to adapt to urgent needs without compromising the integrity of the pipeline.

Common Scenarios for Gates

Gates within a DevOps pipeline can fulfill various functions, but some common scenarios include:

- **Incident Management Gates**: These gates ensure that all testing criteria are met before deployment is allowed. For instance, a gate might check for unresolved tickets before code is moved to production. This helps catch any recent bugs or incidents that developers might not be aware of, preventing the code with potential issues from being released.

- **Approval Management Gates**: This type of gate requires consent from specific teams before the development process can proceed. For example, an application might need approval from both legal and public relations teams before it can be deployed to production.

- **Security Management Gates**: These gates require security scans are completed before code deployment can continue. A gate might, for example, require a vulnerability scan with no findings before allowing deployment. Security scans are crucial for ensuring coding best practices, compliance with organizational policies, checking for outdated or vulnerable packages, and verifying code signatures. Due to their significant impact, Security Management Gates should be implemented early and throughout the pipeline.

- **Infrastructure Management Gates**: Infrastructure Management Gates ensure that the deployed infrastructure is healthy, within an expected utilization, and appropriately deployed prior to allowing an application to be deployed. These gates can also monitor infrastructure post-deployment to confirm that resource usage remains within healthy and expected ranges. Regular checks throughout the build process are beneficial to spot anomalies that might not be evident in early testing stages.

- **User Experience Management Gates**: User Experience Management Gates verify that the application's user experience meets end-user expectations before the code is promoted to production. If a code modification impacts the user experience, this gate assesses whether the change has caused the user experience to fall below a predefined standard.

There are numerous other types of gates that can be integrated in DevOps, each potentially more applicable to an organization's specific requirements. Developers must select gates that align with the organization's objectives, ensuring a smooth and secure development pipeline.

Risk Gate Control Example

Building quality gates into a pipeline will vary depending on the tools utilized in the pipeline. Below is an example using YAML; the logic used here can be extrapolated and applied to other CI/CD implementations as needed. The example consists of five stages: PreDeploymentCheck, Build, SecurityScan, Deploy, and PostDeploymentCheck.

- **PreDeploymentCheck**: This stage ensures the environment is healthy and available before proceeding. It uses an HTTP status check and halts the deployment if the check fails.

- **Build**: This stage depends on the successful completion of PreDeploymentCheck. During this stage, the application is built.

- **SecurityScan**: After the Build stage, the SecurityScan stage runs. It checks the security report for any "FAIL" entries. If any are found, the deployment is aborted.
- **Deploy**: Once the SecurityScan stage completes without issues, the Deploy stage triggers the application deployment.

- **PostDeploymentCheck**: This final stage verifies that the environment remains healthy and available after deployment using another HTTP status check.

While this example configuration is basic, it can be expanded to include more specific tests and conditions, such as code coverage, depending on the specific environment.

```yaml
###############################################
# Example Risk Gate Control YAML Configuration #
###############################################

stages:
  # Pre-Deployment Check Stage
  - name: PreDeploymentCheck
    jobs:
      - job: PreDeploymentCheckJob
        steps:
          # Step to check if the environment is healthy and available
          - script: echo "Checking if the environment is healthy and available..."
            displayName: 'Pre-Deployment Environment Check'
          # Step to verify environment health using an HTTP status check
          - script: |
              if [ $(curl -s -o /dev/null -w "%{http_code}" http://your-environment-url) -ne 200 ]; then
                echo "Environment is not healthy. Aborting pipeline."
                exit 1
              else
                echo "Environment is healthy."
              fi
            displayName: 'Verify Environment Health'

  # Build Stage
  - name: Build
    dependsOn: PreDeploymentCheck
    jobs:
      - job: BuildJob
        steps:
          # Step to build the application
          - script: echo "Building the application..."
            displayName: 'Build Application'
```

```
# Security Scan Stage
- name: SecurityScan
  dependsOn: Build
  jobs:
    - job: SecurityScanJob
      steps:
        # Step to run security scans
        - script: echo "Running security scans..."
          displayName: 'Run Security Scans'
        # Step to check the results of the security scans
        - script: |
            if [ $(cat security_scan_report | grep "FAIL") ]; then
              echo "Security scan failed. Aborting deployment."
              exit 1
            else
              echo "Security scan passed."
            fi
          displayName: 'Check Security Scan Results'

# Deployment Stage
- name: Deploy
  dependsOn: SecurityScan
  jobs:
    - job: DeployJob
      steps:
        # Step to deploy the application
        - script: echo "Deploying the application..."
          displayName: 'Deploy Application'

# Post-Deployment Check Stage
- name: PostDeploymentCheck
  dependsOn: Deploy
  jobs:
```

Figure 18 Example of Gate included to stop pipeline if security scan fails.

Embedding Threat Modeling in DevOps

What is Threat Modeling?

Threat modeling involves analyzing system representations to identify and address security and privacy concerns. Integrating threat modeling into DevOps is crucial for proactively identifying and mitigating potential security issues throughout the software development lifecycle (SDLC). Methodologies like STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) and PASTA (Process for Attack Simulation and Threat Analysis) offer structured approaches to identifying and assessing potential threats based on different attack vectors. Addressing potential threats early in the SDLC, before they can manifest into vulnerabilities, is a proactive measure the team can take to ensure the system's security. This early identification leads to cost savings, as addressing vulnerabilities during design or development is significantly less expensive than post-deployment remediations. Furthermore, integrating threat modeling into the DevOps pipeline helps comply with regulatory requirements by providing documented evidence of security practices.

In traditional practices, threat modeling activities are often siloed and isolated from the broader project, making integrating them with DevSecOps initiatives challenging. This approach is static and not aligned with agile methodologies, hindering its adaptability to the fast-paced nature of development. Threat modeling should provide continuous value, aligning with the development cycle and incorporating threats and mitigations into a backlog. However, they are frequently not cross-functional or updated as part of the release cycle and often rely heavily on security experts. Moreover, focusing too much on specific tools rather than on holistic integration limits the usability of the security output by DevOps teams, as the results may not be immediately actionable or valuable within the development workflow.

Traditional methods and tools like whiteboards do not align well with a DevOps environment. Changes in the infrastructure, emergence of new threats, evolving regulatory requirements, adoption of new technologies, and new services or features could affect the security of the application, requiring an update to the threat model. Threat modeling should be ongoing throughout development, not just a one-time exercise. This process should be included during the architectural phase and revisited as the application evolves, to answer the following critical questions:

- What are we building?
- What can go wrong?
- What are we going to do about it?

The principal output of threat modeling is a list of potential threats and their impact on the system and recommended security controls to mitigate the risks. Common artifacts include data flow diagrams, attack trees, and threat matrices. These artifacts help identify potential vulnerabilities and prioritize security controls based on the severity of the threats. These artifacts should be closely tied to the application's code and updated as the code evolves.

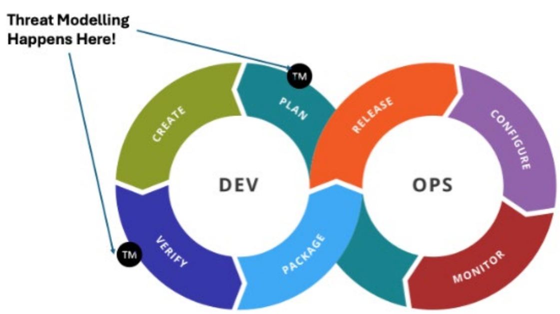


Figure 19 Where should we perform Threat Modeling?

Autodesk Continuous Threat Modeling

Ideally, a threat model should be performed during the design phase; however, in many cases, it may be necessary to revisit and update the threat model later in the development process or even after deployment. An example could be a new vulnerability or threat that takes us to the planning phase again. Several tools and practices can help integrate threat modeling into a DevOps pipeline. Because of its agnostic approach, we will discuss Continuous Threat Modeling (A-CTM), a methodology developed by Autodesk (Millighan and Tarandach, 2019).

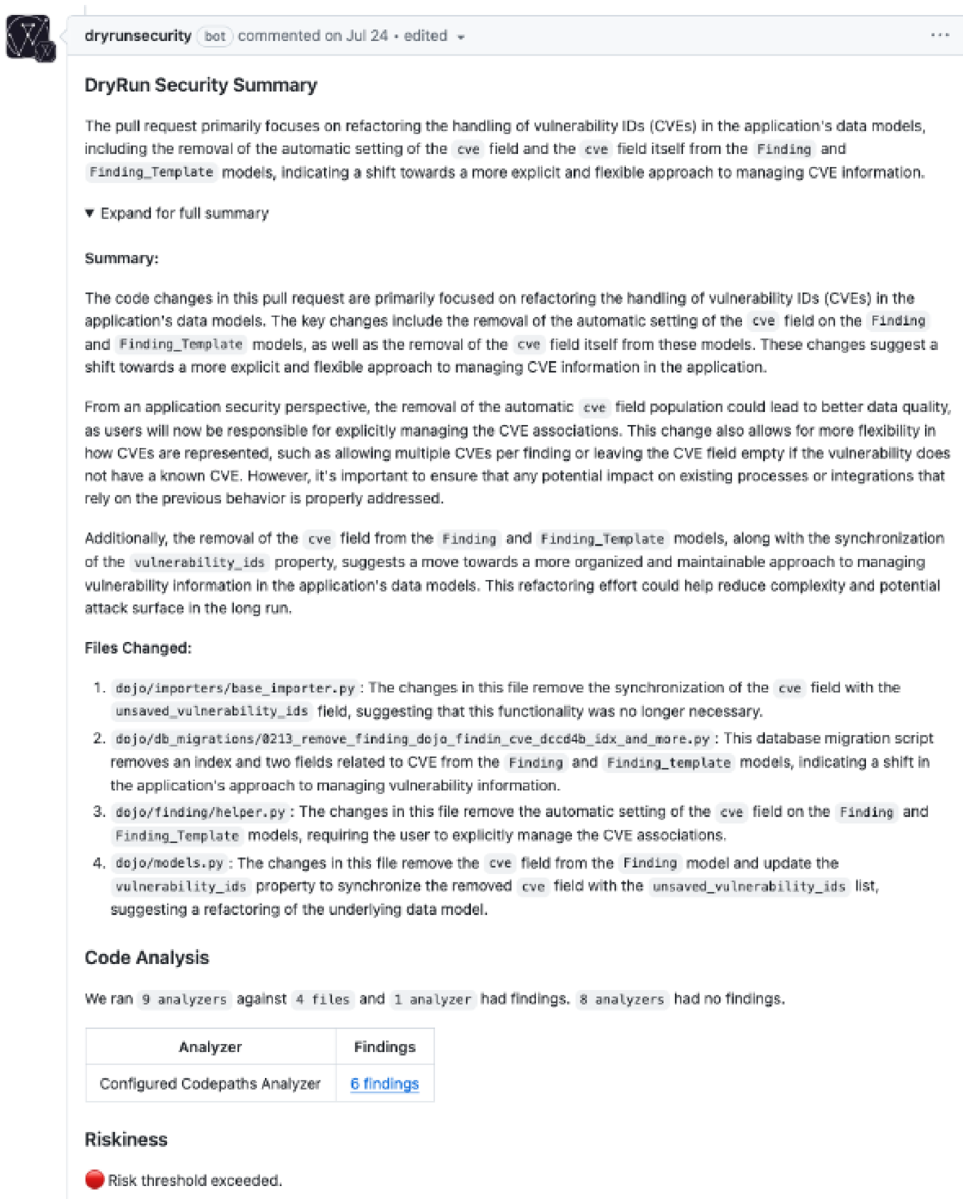
We start by identifying all the critical assets and drawing diagrams of the system to be modeled. Tools like OWASP PyTM (https://github.com/izar/pytm) can help us programmatically create a threat model using Python. You can find more info about PyTM, other threat modeling tools, and examples in the ND-ISAC Application Threat Modeling Whitepaper.

Subsequently, threat modeling will generate data flow diagrams (DFD) of the system and identify potential threats using a customizable threat library. The threat modeling exercise should identify possible vulnerabilities and weaknesses once the initial threat model is ready. A-CTM's documentation provides a good starting point. However, the process can also use other resources, such as the Common Weakness Enumeration (CWE) list, the OWASP Top 10, or the MITRE ATT&CK framework, to identify potential threats.

A complete threat model will include the following components: Data Flow Diagrams, network traffic requirements (ports in use, requirements from firewalls, etc.), a list of potential threats, the "crown jewels" that we want to protect, uses of cryptography, the stakeholders involved, and the security controls that we will implement to mitigate each risks. Each finding in the threat model should be associated with a ticket, and the team should prioritize these findings based on their risk level. It is also crucial to clearly define the security controls implemented to mitigate the identified risks. This approach ensures that the most critical threats are addressed first and that the chosen security controls are appropriate for the risks.

In a fast-paced DevOps environment, it can be challenging to keep up with the frequency of releases while maintaining the security posture of our applications. As emphasized before, each new feature or change that could impact the application's security should be threat-modeled, and the threat model updated as the system evolves. However, determining when to review the threat model can be time-consuming and exhausting. Here is where AI-powered tools, like DryRun Security, come into play. These tools leverage large language models (LLMs) to automatically detect potential security risks associated with code changes and recommend updates to the threat model or a security review. By automating this process, we can significantly reduce the administrative burden on our teams while ensuring that our applications remain secure throughout their development lifecycle. The integration of

AI-driven insights enables organizations to make more informed decisions about when to review the threat model.



dryrunsecurity bot commented on Jul 24 • edited

DryRun Security Summary

The pull request primarily focuses on refactoring the handling of vulnerability IDs (CVEs) in the application's data models, including the removal of the automatic setting of the `cve` field and the `cve` field itself from the `Finding` and `Finding_Template` models, indicating a shift towards a more explicit and flexible approach to managing CVE information.

▼ Expand for full summary

Summary:

The code changes in this pull request are primarily focused on refactoring the handling of vulnerability IDs (CVEs) in the application's data models. The key changes include the removal of the automatic setting of the `cve` field on the `Finding` and `Finding_Template` models, as well as the removal of the `cve` field itself from these models. These changes suggest a shift towards a more explicit and flexible approach to managing CVE information in the application.

From an application security perspective, the removal of the automatic `cve` field population could lead to better data quality, as users will now be responsible for explicitly managing the CVE associations. This change also allows for more flexibility in how CVEs are represented, such as allowing multiple CVEs per finding or leaving the CVE field empty if the vulnerability does not have a known CVE. However, it's important to ensure that any potential impact on existing processes or integrations that rely on the previous behavior is properly addressed.

Additionally, the removal of the `cve` field from the `Finding` and `Finding_Template` models, along with the synchronization of the `vulnerability_ids` property, suggests a move towards a more organized and maintainable approach to managing vulnerability information in the application's data models. This refactoring effort could help reduce complexity and potential attack surface in the long run.

Files Changed:

1. `dojo/importers/base_importer.py` : The changes in this file remove the synchronization of the `cve` field with the `unsaved_vulnerability_ids` field, suggesting that this functionality was no longer necessary.
2. `dojo/db_migrations/0213_remove_finding_dojo_findin_cve_dccd4b_idx_and_more.py` : This database migration script removes an index and two fields related to CVE from the `Finding` and `Finding_template` models, indicating a shift in the application's approach to managing vulnerability information.
3. `dojo/finding/helper.py` : The changes in this file remove the automatic setting of the `cve` field on the `Finding` and `Finding_Template` models, requiring the user to explicitly manage the CVE associations.
4. `dojo/models.py` : The changes in this file remove the `cve` field from the `Finding` model and update the `vulnerability_ids` property to synchronize the removed `cve` field with the `unsaved_vulnerability_ids` list, suggesting a refactoring of the underlying data model.

Code Analysis

We ran `9 analyzers` against `4 files` and `1 analyzer` had findings. `8 analyzers` had no findings.

Analyzer	Findings
Configured Codepaths Analyzer	6 findings

Riskiness

🔴 Risk threshold exceeded.

Figure 20 DryRun Security bot informs that the Risk threshold was exceeded and notifies the developers (Anderson, 2024)

Another example on how to integrate threat modeling in DevOps is presented in Appendix A.

Conclusion

The race against a cyber-attack can only be won if we allow technology to help us identify risk as part of a cohesive strategy in the SDLC. This paper describes the importance of embedding security controls after the build stage to ensure risk associated with the components of a container image are properly detected. The “Deploy” stage in DevOps also provides a great opportunity to verify whether all the security requirements are in place to protect the runtime environment. The combination of detecting risk in the runtime environment as well as the build process provides coverage for the four (4) layers of risk in container environments (host, images, registry and runtime).

Embedding IaC security scanning serves as a similar control you will get with SAST. Because IaC is code, it is important to identify as early as possible if the code has the potential to lead to security misconfigurations or non-compliance issues. The ability to detect these issues and notify the team as part of the DevOps pipeline enable software engineers to improve their deployment environments by reducing potential risk.

Deploying resources to cloud environments without proper vetting could expose resources that are intended to remain private. Implementing proactive detection through automation in the DevOps pipeline ensures a high degree of risk visibility.

Combining cloud with other technologies, such as IaC, will help the organization minimize risk, as deployments to a runtime environment will follow a standard process aimed at consistently minimizing risk. The standardization of resource creation and deployment creates positive effects by enforcing compliance requirements more easily.

To proactively identify potential risks in the design of a solution, it is beneficial to incorporate risk detection through threat modeling in DevOps. This will allow teams to pinpoint possible entry points that an attacker could exploit to compromise the environment. Additionally, it will enable the organization to implement technical controls that can verify the presence of necessary countermeasures for a specific threat or detect any deviations from the original design. Utilizing automation alongside Threat Modeling in DevOps streamlines the process of identifying non-compliance and deviations from the security baseline, can result in significant efficiency gains.

Security controls integrated within a DevOps pipeline can alert the automation process if there are conditions that may warrant halting the deployment to a live environment due to detected risks. This capability could mean the difference between creating a new issue in the issue tracker and experiencing a cybersecurity incident or compromise. The risk gate approach acts as an enforcing authority enhancing the organizations' ability to ensure robust security measures are consistently followed throughout the SDLC, regardless of the mix of emerging technologies being utilized.

Appendix A: Embedding Threat Modeling in DevOps Example

Let’s consider a simple CI/CD job using LLMs that checks a pull request / merge request and recommends performing an update of the threat mode, if needed. For sake of simplicity, we will focus only on the command executed, and we will not present the implementation of the GitHub Action workflow, GitLab CI/CD pipeline, or Jenkins pipeline. The output of this command will be added as a comment to the issue, acting as a code reviewer. Another approach could be to use a git hook to test the code before commit.

The example below is using ollama v0.3.6 (https://ollama.com/), using the model codellama:13b (id 9f438cb9cd58, https://ollama.com/library/codellama:13b) running on an Apple M3 Pro, 36 GB RAM, macOS Sonoma 14.6.1 (23G93). The application is a very simple application to monitor a list of machines included in a text file.

For the exercise, the following use case is considered:

"You are an experienced appsec practitioner. Given the following git diff, decide if we need to update the threat model of the application. Changes in the infrastructure, emergence of new threats, evolving regulatory requirements, adoption of new technologies, and new services or features could affect the security of the application. Changes to one or more of these parameters would require updates to the threat model. If you consider we must update our threat model, include the security risks associated to the change $(git diff)"

The diff file analyzed in this example is:

```
diff --git a/app.py b/app.py

index f88af07..14830be 100644
--- a/app.py
+++ b/app.py
@@ -1,23 +1,42 @@
-from flask import Flask, render_template
+from flask import Flask, render_template, request, redirect, url_for
import os
 app = Flask(__name__)

+# Ensure the uploads directory exists
+os.makedirs('uploads', exist_ok=True)
+
+app.config['UPLOAD_FOLDER'] = 'uploads'
+
def get_machine_status(machine_name):
# This is just a placeholder function. You can replace it with actual logic
# to check the machine status. For now, it will return "Online" for all machines.
+    # Placeholder function to determine machine status
     return "Online"

-@app.route('/')
+@app.route('/', methods=['GET', 'POST'])
def index():
     machines = []

# Read the list of machines from the file
if os.path.exists('machines.txt'):
with open('machines.txt', 'r') as file:
+    # Handle file upload
+    if request.method == 'POST':
+        # Check if a file is uploaded
+        if 'file' not in request.files:
+            return "No file part", 400
+        file = request.files['file']
+        if file.filename == '':
+            return "No selected file", 400
+
```

Figure 21 Threat modeling Diff file analyzed.

Knowing that LLMs are not deterministic, we ran the command several times, and the results were similar. Even when some answers were too verbose and in some cases flagged inexistent issues, in general the results were good and proved to be useful as a first step, helping developers to decide if they needed to update the threat model of the application. We are including two responses.

Answer 1:

This diff introduces several changes to the application's functionality and infrastructure. The most significant change is the addition of file upload functionality, which allows users to upload a list of machines to the server. This could potentially introduce new security risks, such as:

- Malicious code execution: If an attacker uploads a maliciously crafted list of machines, it could lead to code execution on the server, potentially allowing for unauthorized access or data breaches.

- Data tampering: If the uploaded file contains sensitive information about the application's users or infrastructure, it could be used as a means to manipulate the system or gain unauthorized access.

- Denial of Service (DoS): An attacker could upload a large list of machines, potentially causing the server to become unresponsive and unable to process requests.

To mitigate these risks, it is recommended to update the threat model of the application by considering the following new threats:

- Malicious file uploads: The addition of file upload functionality allows for the possibility of malicious users uploading files containing malware or viruses.

- Unauthorized access to uploaded files: Users may not have proper permissions to access uploaded files, potentially allowing unauthorized parties to access sensitive information.

- Denial of Service attacks via uploaded files: Large files uploaded by attackers could cause the server to become unresponsive and unable to process requests.

It is also recommended to implement additional security measures such as:

- Implementing input validation and sanitization for user-provided data, including file uploads.
- Enforcing proper access controls for uploaded files and ensuring that only authorized users can access them.
- Limiting the size of uploaded files to prevent potential DoS attacks.
- Implementing a content-aware firewall to detect and block malicious traffic, such as attempts to execute malware or exploit vulnerabilities in the server's code.

Answer 2:

Yes, we need to update the threat model of the application. The changes in the infrastructure and emergence of new threats require a reevaluation of the potential security risks associated with the application. The addition of a file upload feature introduces a new attack surface that must be addressed in the threat model.

The current threat model may not adequately account for the following security risks:

- File upload vulnerabilities: Uploading malicious files can lead to code execution, data tampering, and other types of attacks. The addition of a file upload feature introduces new attack vectors that must be addressed in the threat model.

- Cross-site scripting (XSS): If user input is not properly sanitized or validated, an attacker could exploit this vulnerability to steal sensitive information or execute malicious scripts.

- Insecure direct object reference: If a file upload feature is not implemented securely, an attacker could potentially access files outside of the intended directory structure.

Therefore, organizations should update the threat model to include these new risks and ensure that appropriate countermeasures are in place to mitigate them.

References

Amazon. (n.d.). *What is DevSecOps? - Developer Security Operations Explained*. https://aws.amazon.com/what-is/devsecops/

Anderson, G. (2024). *DefectDojo*. https://github.com/DefectDojo/django-DefectDojo/pull/9908

Beckers, K., Martin, J. & Ozmore, N. (2019, January 8). *Putting a Face to Software Security Champions*. https://safecode.org/blog/putting-a-face-to-software-security-champions/

CAB. (2023, December 7). *Baseline Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates*. https://cabforum.org/wp-content/uploads/Baseline-Requirements-for-the-Issuance-and-Management-of-Code-Signing.v3.5.pdf

ClickSSL. (2023, November 27). *What is Code Signing Certificate? How Does Code Signing Certificate Work?* ClickSSL. https://www.clickssl.net/blog/what-is-code-signing-certificate

Digicert. (2023, May 1). *Code Signing Changes In 2023*. https://knowledge.digicert.com/alerts/code-signing-changes-in-2023

Docker. (n.d.). *Content trust in Docker*. Docker. https://docs.docker.com/engine/security/trust/

Entrust. (n.d.). *What is a Hardware Security Module (HSM)*? Entrust. https://www.entrust.com/resources/hsm/faq/what-are-hardware-security-modules

Gate For Security. (n.d.). *Security Systems*. https://gateforsecurity.com/en/end-user-security-systems

GitHub. (n.d.). *Signing commits*. GitHub. https://docs.github.com/en/authentication/managing-commit-signature-verification/signing-commits

Hashicorp. (2023). *Infrastructure as Code: What Is It? Why is It Important?* https://www.hashicorp.com/resources/what-is-infrastructure-as-code

Heim, M., Keim, P., Munsch, J. & Pabon, W. (2020). *Software Security Automation: A Roadmap toward Efficiency and Security*. https://ndisac.org/wp-content/uploads/ndisac-security-automation-white-paper.pdf

Mehta, J. (n.d.). *How Does the Code Signing Process Work? – Understand the Code Signing Architecture*. Signmycode. https://signmycode.com/resources/code-signing-architecture-how-does-it-work

Millighan, S. & Tarandach, I. (2019). *Continuous Threat Modeling*. https://github.com/Autodesk/continuous-threat-modeling

Miller, Z., Ramani, R. (2022, June 23). *Cryptographic Signing for Containers*. AWS. https://aws.amazon.com/blogs/containers/cryptographic-signing-for-containers

Moriarty, K. (2023, August 3). *ACME End User Client and Code Signing Certificates*. https://datatracker.ietf.org/doc/html/draft-ietf-acme-client-07

National Institute of Standards and Technology. (2022). *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities*. https://doi.org/10.6028/NIST.SP.800-218

OWASP. (2020). www-*project-appsec-pipeline*. https://github.com/OWASP/www-project-appsec-pipeline/blob/master/assets/images/DevOps_AppSec_Pipline_Integration.png

Proska, K., Hidelbrandt, C., Zafra, D. K., & Brubaker, N. (2021, October 27). *Portable Executable File Infecting Malware Is Increasingly Found in OT Networks*. Mandiant. https://www.mandiant.com/resources/blog/pe-file-infecting-malware-ot

Ray, J. (2024). Policy as Code: Improving Cloud Native Security. O'Reilly.

Rice, L. (2020). Container Security: Fundamental Technology Concepts that Protect Containerized Applications. O'Reilly. https://www.oreilly.com/library/view/container-security/9781492056690/ch01.html

Sandoval, K. (2015, July 8). *Living in the Cloud Stack – Understanding SaaS, PaaS, and IaaS APIs*. https://nordicapis.com/living-in-the-cloud-stack-understanding-saas-paas-and-iaas-apis/

Trend Micro. (2023, March 16). *Policy as Code vs Compliance as Code*. Policy as Code vs Compliance as Code | Trend Micro (CA)

Vehent, J. (2018). *Securing DevOps: Security in the Cloud*. Manning Publication Co.