The Six Pillars of DevSecOps: Automation

SAFECode
Software Assurance Forum for Excellence in Code
Driving Security and Integrity

CSA cloud security alliance®

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

© 2020 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 Authors:

Souheil Moghnie
Theodore Niedzialkowski
Sam Sehgal

Contributor:

Michael Roza

CSA Analysts:

Sean Heide

Special Thanks:

Ankur Gargi
Raj Handa
Manuel Ifland
John Martin
Kamran Sadique
Charanjeet Singh
Altaz Valani

© Copyright 2020, Cloud Security Alliance. All rights reserved.

Table of Contents

© Copyright 2020, Cloud Security Alliance. All rights reserved.

Foreword

The Cloud Security Alliance and SAFECode are both 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 used by its authors 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: Training and Process Integration
> - Pillar 3: Pragmatic Implementation
> - Pillar 4: Bridging Compliance & and Development
> - Pillar 5: Automation
> - Pillar 6: Measure, Monitor, Report & and Action

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 second of those follow-on publications.

Introduction

0.1 Background

The DevSecOps Automation pillar advocates the use of security automation to achieve reflexive security, specifically, 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.

Automation is a critical component of DevSecOps because it enables process efficiency, allowing developers, infrastructure, and information security teams to focus on delivering value rather than repeating manual efforts and errors with complex deliverables.

This paper focuses on a risk-based security automation approach that strings automated security actions throughout the continuous software development deployment cycle. Examples of activities that can be automated include application, host, and container vulnerability scanning.

DevOps teams utilizing best practices including continuous integration, continuous delivery and infrastructure as code are agile, with the ability to address user requirements dynamically, release features incrementally, and deliver at a faster pace.

In order to integrate information security controls into this process, automated security capabilities are crucial in order to provide timely and meaningful feedback.

The complexity of cloud infrastructure today means that small code changes may lead to a disproportionate downstream impact; therefore security checks need to be integrated throughout the software development and deployment lifecycle, through design, implementation, testing, and release, and monitored in production.

In alignment with the Pragmatic Approach pillar of reflexive security, adopting tentative and modest automation of security capabilities will already enable rapid feedback and can potentially eliminate whole classes of risk, for example, container scanning to ensure OS hardening or software composition analysis for known Common Vulnerabilities and Exposures (CVEs).

0.2 Purpose

This document provides a framework to enable automation to transparently integrate security into the software development lifecycle by:

- Enabling a fast, reciprocal flow of security-related information to DevOps teams to create and validate secure code by design in a continuous manner, rather than deferring security into a single stage impediment at delivery.
- Allowing a balanced approach to software development and security needs such that delivery efficiency is improved through automated integration.

© Copyright 2020, Cloud Security Alliance. All rights reserved.

0.3 Audience

The target audience of this document includes those involved in the management and operational functions of risk, information security and information technology. This includes the C-suite (CISO, CIO, CTO, CRO, COO, CEO), and especially the individuals involved in the following functional areas: DevSecOps, automation, DevOps, quality assurance, information security, governance, risk management, change management and compliance.

1. Scope

This document describes the automation pillar of DevSecOps: the necessity of security automation, security test automation techniques, and mechanisms to achieve it.

This document specifically provides a holistic framework for facilitating security automation within DevSecOps and best practices in the automation of security controls and provides clarification of common misconceptions about security testing in the context of DevSecOps.

2. Normative References

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

> - ISO/IEC 27000:2018, Information technology -- Security techniques -- Information security management systems -- Overview and vocabulary
> - CSA Information Security Management through Reflexive Security
> - CSA The Six Pillars of DevSecOps

3. Terms and Definitions

For the purposes of this document, the terms and definitions given in ISO 27000, CSA Information Security Management through Reflexive Security and the following apply.

3.1 Software Composition Analysis (SCA)

Security testing that analyzes application source code or compiled code for software components with known vulnerabilities

Note 1 to entry: software components in software composition analysis may include open source, libraries and common code.
Note 2 to entry: known vulnerabilities may be discovered via vulnerability databases such as CVE.

© Copyright 2020, Cloud Security Alliance. All rights reserved.

3.2 Static Application Security Testing (SAST)

Security testing that analyzes application source code for software vulnerabilities and gaps against best practices

Note 1 to entry: Static analysis can be performed in multiple environments including the developer's IDE, source code, and binaries.

Note 2 to entry: Also called "white box testing"

3.3 Dynamic Application Security Testing (DAST)

Security testing that analyzes a running application by exercising application functionality and detecting vulnerabilities based on application behavior and response

Note 1 to entry: Also called "blackbox testing"

3.4 Interactive Application Security Testing (IAST)

Software component deployed with an application that assesses application behavior and detects presence of vulnerabilities on an application being exercised in realistic testing scenarios

3.5 Runtime Application Security Protection (RASP)

Security technology deployed within the target application in production for detecting, alerting, and blocking attacks

Note 1 to entry: Similar to a WAF but instrumented within the application

3.6 Web Application Firewall (WAF)

Application firewall that monitors, alerts, and blocks attacks by inspecting HTTP traffic

3.7 Cloud Security Posture Management (CSPM)

Security technology that can discover, assess, and resolve cloud infrastructure misconfigurations vulnerable to attack

3.8 Cloud Security Monitoring and Compliance

Security technology that monitors virtual servers and assesses data, applications, and infrastructure for security risks

3.9 Threat Modeling

Methodology to identify and understand threats impacting a resource or set of resources

Note to entry: Common methodologies of threat modeling include STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) and OCTAVE (Operationally Critical Threat, Asset, and Vulnerability Evaluation).

3.10 Manual Security Code Review

Human process of reading source code to identify security issues

3.11 Software Delivery Pipeline

Set of automated processes used for delivering software from conception to deployment

3.12 CSA DevSecOps Software Delivery Pipeline (CDDP)

Security-enabled software delivery pipeline aligned with DevSecOps principles

4. CSA DevSecOps Software Delivery Pipeline

4.1 General

Every piece of software goes through the stages of conception, design, develop, build, and test prior to deployment in production environments. In the modern software delivery pipeline, tools are heavily automated to guard stage checkpoints. At every pipeline stage, automated checks are performed in order to prevent quality issues slipping into the next stages.

Multiple processes are involved within a software delivery pipeline, including software development, change management, configuration management, and service management, potentially with the concepts of continuous integration, continuous delivery and continuous monitoring applied. Tools used by development and operations include source control, build, continuous integration, containerization, configuration management, orchestration, and monitoring.

Integration of security testing within the software development pipeline requires security activities to be fully automated and bridged with the native tools and processes already employed by teams within the delivery pipeline. Activities required by checkpoints but whose outcome requires manual intervention, called asynchronous testing in this document, shall be specially considered so that the benefits of automated security checkpoints are not lost to them.

© Copyright 2020, Cloud Security Alliance. All rights reserved.

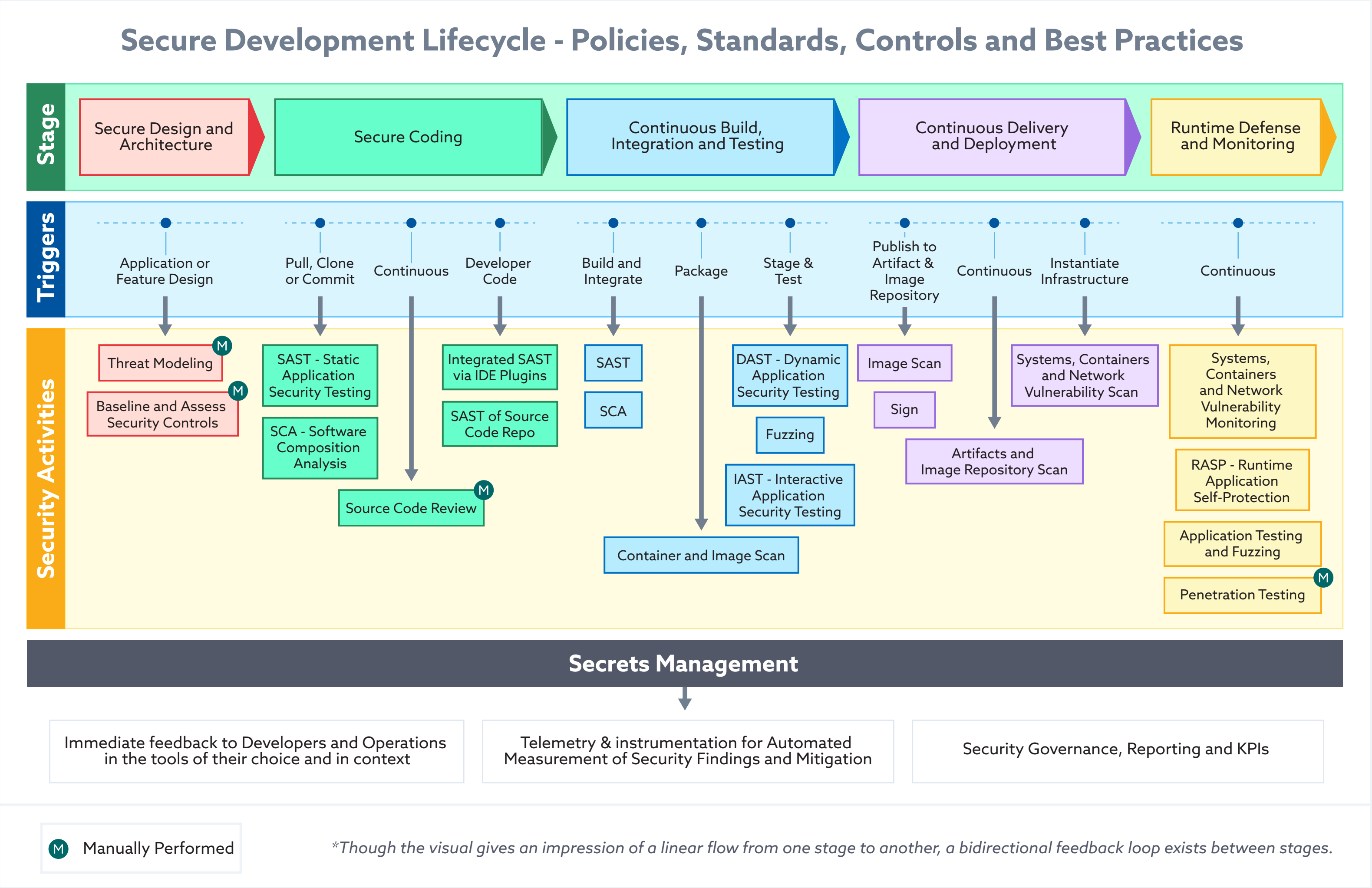
Secure Development Lifecycle - Policies, Standards, Controls and Best Practices
Stage
Secure Design and Architecture
Secure Coding
Continuous Build, Integration and Testing
Continuous Delivery and Deployment
Runtime Defense and Monitoring
Triggers
Application or Feature Design
Pull, Clone or Commit
Continuous
Developer Code
Build and Integrate
Package
Stage & Test
Publish to Artifact & Image Repository
Continuous
Instantiate Infrastructure
Continuous
Security Activities
Threat Modeling
M
Baseline and Assess Security Controls
M
SAST - Static Application Security Testing
SCA - Software Composition Analysis
Source Code Review
M
Integrated SAST via IDE Plugins
SAST of Source Code Repo
SAST
SCA
Container and Image Scan
DAST - Dynamic Application Security Testing
Fuzzing
IAST - Interactive Application Security Testing
Image Scan
Sign
Artifacts and Image Repository Scan
Systems, Containers and Network Vulnerability Scan
Systems, Containers and Network Vulnerability Monitoring
RASP - Runtime Application Self-Protection
Application Testing and Fuzzing
Penetration Testing
M
Secrets Management
Immediate feedback to Developers and Operations in the tools of their choice and in context
Telemetry & instrumentation for Automated Measurement of Security Findings and Mitigation
Security Governance, Reporting and KPIs
M
Manually Performed
*Though the visual gives an impression of a linear flow from one stage to another, a bidirectional feedback loop exists between stages.

Figure 1: The CSA DevSecOps Delivery Pipeline

4.2 Structure

4.2.1 General

A security-enabled software delivery pipeline is considered to have three granularity levels:

- Stages
- Triggers and Checkpoints
- Activities

4.2.2 Stages

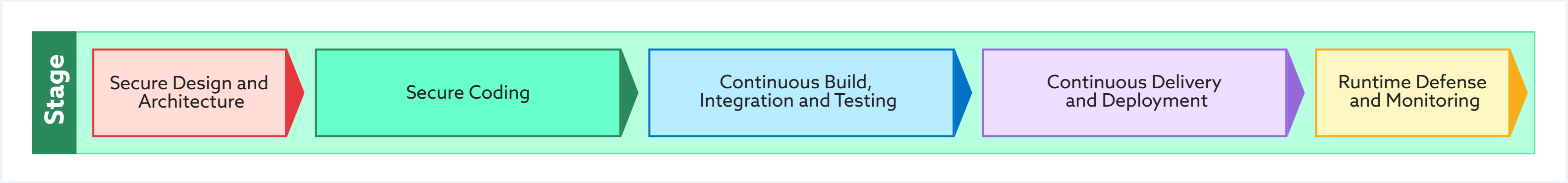
Stage
Secure Design and Architecture
Secure Coding
Continuous Build, Integration and Testing
Continuous Delivery and Deployment
Runtime Defense and Monitoring

Figure 1-1: Stages in the Delivery Pipeline

Stages represent an incremental maturity level of the deliverable product. During conception and architecture design, the maturity level is low. When the software is ready for continuous delivery and deployment, the software is clearly more mature.

Every software deliverable must go through stages from design, coding to testing before it can be finally deployed to production. This is independent of software development methodology (Waterfall, Agile).

There is an opportunity at every stage to embed and automate appropriate security controls.

This document identifies the following distinct stages:

> 1. Secure design and architecture
> 2. Secure coding (Developer IDE and Code Repository)
> 3. Continuous build, integration and test
> 4. Continuous delivery and deployment
> 5. Continuous monitoring and runtime defense

4.2.3 Triggers

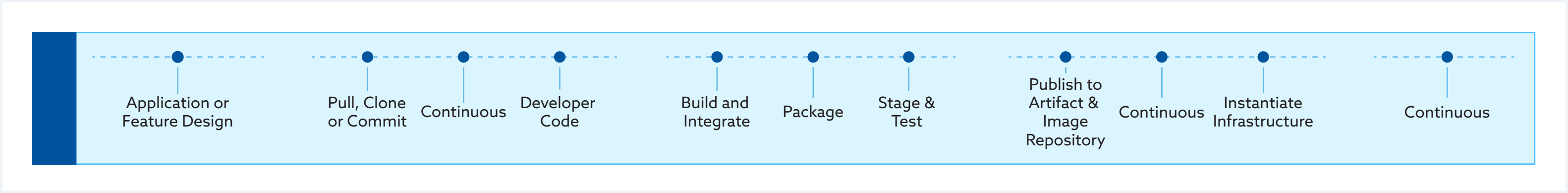

Figure 1-2: Triggers in the delivery pipeline

Triggers and checkpoints (see Figure 1-2) represent transitions within stages. When a trigger condition is met, one or more security activities are activated, and the outcomes of those activities determine whether the desired requirements of the checkpoint are satisfied.

If the outcome of security activities meets the desired requirements, the deliverable is transitioned to the next checkpoint (or the next stage if the checkpoint is the last one). Otherwise, the deliverable remains and is not allowed to advance to the next checkpoint.

EXAMPLE 1: Architecting a new feature can trigger the start of a threat modeling exercise, which can be used to identify general or sector-specific security controls applicable to that feature.

EXAMPLE 2: A code commit or pull/merge request by a developer may automatically trigger a source code scan and a code review process.

EXAMPLE 3: A push of a container image to a container registry may trigger a vulnerability scan before the image becomes available in the registry.

There are two types of triggers:

> - Change-based triggers: these triggers are set off by change events, such as a code push.
> - Recurring triggers: these triggers are set off by time events, such as a daily timer.

Change-based triggers are used to guard checkpoint transitions. They are generally used for short-running security checks that can be run in an event-based manner.

Recurring triggers are usually used for security checks that run for longer times, such as those that involve batch processing or integration between components.

EXAMPLE 4: A recurring trigger is preferred for batch scanning of multiple source code repositories seeking for misplaced secrets.

4.2.4 Activities

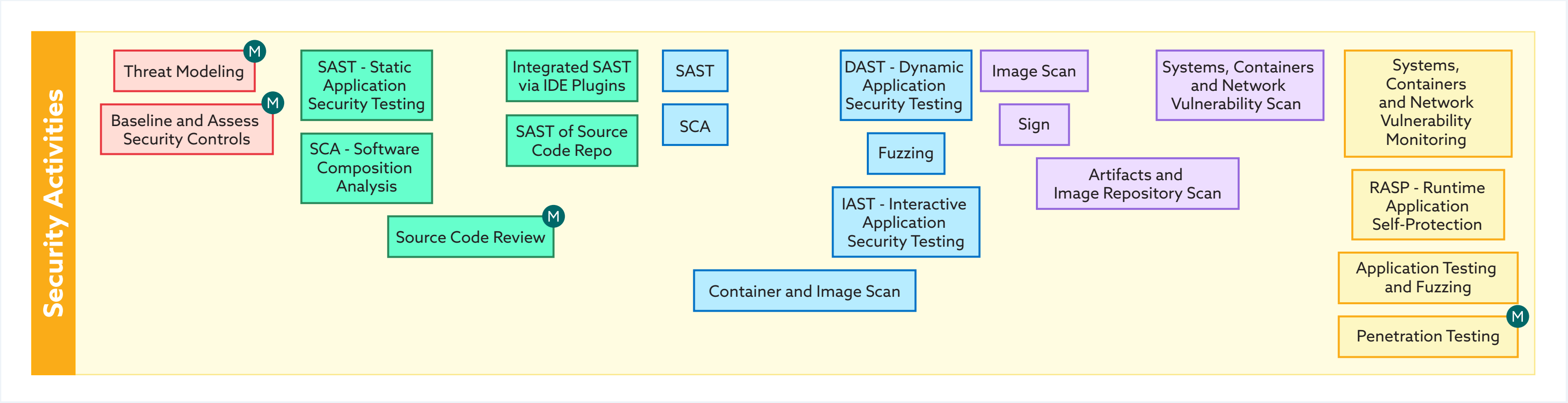
Security Activities
Threat Modeling
M
Baseline and Assess Security Controls
M
SAST - Static Application Security Testing
SCA - Software Composition Analysis
Source Code Review
M
Integrated SAST via IDE Plugins
SAST of Source Code Repo
SAST
SCA
Container and Image Scan
DAST - Dynamic Application Security Testing
Fuzzing
IAST - Interactive Application Security Testing
Image Scan
Sign
Artifacts and Image Repository Scan
Systems, Containers and Network Vulnerability Scan
Systems, Containers and Network Vulnerability Monitoring
RASP - Runtime Application Self-Protection
Application Testing and Fuzzing
Penetration Testing
M
M Manually Performed

Figure 1-3: Security Activities in the delivery pipeline

Activities are individual processes that take the deliverable as input and produce a positive or negative outcome. If a deliverable meets the criteria of the activity, the outcome is considered positive. Otherwise, the outcome is negative.

Specifically, each activity represents the implementation of a security control -- a testing or verification task performed to detect security vulnerabilities in code, library, applications or the environment in which an application is going to run.

Activities are independent of stages and triggers: some activities can be run in all stages, or bound to a single stage. Multiple triggers may rely on the same activity outcome. Detailed description and guidelines of each security activity are available in referenced documents.

4.3 Setting Up and Maintenance of Pipeline

To support DevSecOps, the delivery pipeline should be built into the native tools and processes already being used by DevOps teams. Security needs to be part of the normal release process to minimize friction and support efficient, fast release cycles.

From a pragmatic security perspective, pipeline security testing capabilities should be introduced incrementally with high value, low impact capabilities added first. Security checks from code quality or library management[1] tools already being used can generate quick wins with minimal operational impact or additional resourcing (identifying a vendor, acquiring a new technology, and instrumentation).

When introducing a new testing capability, e.g. Static Application Security Testing (SAST), the DevOps team, for example – should focus on a particular finding type first and tune the tool to prevent false positives and ensure maximum code coverage (eliminating false negatives). The team can now perform root cause analysis on missed vulnerabilities and false positives to gain the optimum efficiency out of automated testing before adding a new finding type or capabilities.

Large numbers of false positives can burden DevOps teams and harm the credibility of security tooling.

A good strategy is to "shift security left while accelerating right." Automated security controls and activities do not replace an organization's existing SDLC policies, standards and procedures. Rather, automation augments existing capabilities to secure processes and assets. Organizations should start by automating controls and activities that are being performed manually.

5. Risk-prioritized Pipelines

5.1 General

Not all applications have the same risk profiles. In order to effectively prioritize resources and to strike balance with deliverability, it is beneficial to implement a mechanism to provide more scrutiny to high-risk application deliveries while fast-tracking application deliveries with a stable history.

A risk based approach should be used to determine the overall rigor of security testing engaged when deploying a build. There are generally three types of risk factors to consider described below.

[1] https://www.csoonline.com/article/3398485/28-devsecops-tools-for-baking-security-into-the-development-process.html

© Copyright 2020, Cloud Security Alliance. All rights reserved.

5.2 Risk Factors

5.2.1 Application Risk

The inherent risk attributed to the application regardless of technical exposure, including consideration of the data it processes and its usage context

EXAMPLE: An application handling banking or healthcare data can be considered "high risk" while an application handling PII and credit cards can be considered "medium risk." Those that handle static content or public information can be considered "low risk."

5.2.2 Risk of Proposed Change

The risk introduced by the change itself, considering the potential security impact of the change, data the change affects and the nature of the change

EXAMPLE: Changes that affect core security components, such as authentication or access management, changes that introduce new features or technologies and changes that involve rearchitecting are more likely to impact the security posture of the application, and therefore should be tested more thoroughly.

5.2.3 Risk of Reliability History of Pipeline and its Deliverables

The risk attributed to the reliability and integrity history of the delivery pipeline and the application in question.

EXAMPLE: A stable delivery pipeline with a history of secure deliverables is understandably less risky than an unstable delivery pipeline with a history of vulnerable deliverables. An application that has seen stable secure releases is less risky than another with a string of security failures.

5.3 Prioritization of Pipeline Configuration

Delivery pipeline configurations can be prioritized according to an integral of risk factors. Every delivery pipeline and application is unique; in implementation it is necessary to consider whether the risk differentiation mechanism is appropriate and sufficient for a use case.

The existence of differentiated treatment for delivery pipelines provides a graded incentive for securely developed and deployed applications, leading to faster delivery times.

This clause utilizes the traffic light metaphor with three differently colored processing lanes: green, yellow and red, each offering a different level of security scrutiny:

> - Green lane: short running tests, optimized for continuous delivery
> - Yellow lane: may involve DAST, minor out-of-band activities of short processing times, continuous delivery possible
> - Red lane: may involve IAST, major out-of-band activities of longer processing times, continuous delivery difficult.

EXAMPLE 1: Delivery of applications that rank low with all three risk factors can be considered "overall low risk" and may continue along the "green lane."

EXAMPLE 2: A change to personal data handling code that does not impact personal data sharing in a low risk application can be considered "medium risk" and may continue along the "yellow lane."

EXAMPLE 3: Delivery of applications with a record of significant vulnerabilities should be subject to more security rigor using the "red lane."

EXAMPLE 4: If a proposed change introduces a new standard component for safe handling of data, or requires a developer to be trained on relevant security best practices, the "red lane" would apply.

Root cause analysis should be performed when a new vulnerability is detected to determine if the cause is related to people, process, or technology, and what can be re-architected or changed to prevent recurrence.

The table below provides a sample scoring of changes that combines these three factors to determine the target pipeline configuration. The numeric values of High (3), Medium (2), and Low (1) are used with a total score of 5 and above considered to be in a "yellow lane," and 7 and above considered to be in the "red lane."

	Application Risk	**Change Risk**	**Reliability Risk**	**Delivery Pipeline Configuration**
Change #1	Medium (2)	Low (1)	Low (1)	Green (4)
Change #2	Low (1)	High (3)	Medium (2)	Yellow (6)
Change #3	High (3)	High (3)	High (3)	Red (9)

Table 1: Build Impact and Resulting Release Pipeline

© Copyright 2020, Cloud Security Alliance. All rights reserved.

6. Delivery Pipeline Activity Framework

6.1 General

This section presents a framework (see Figure 2) that represents five classes of security activities used within the delivery pipeline, as well as guidelines for activities within those separate classes.

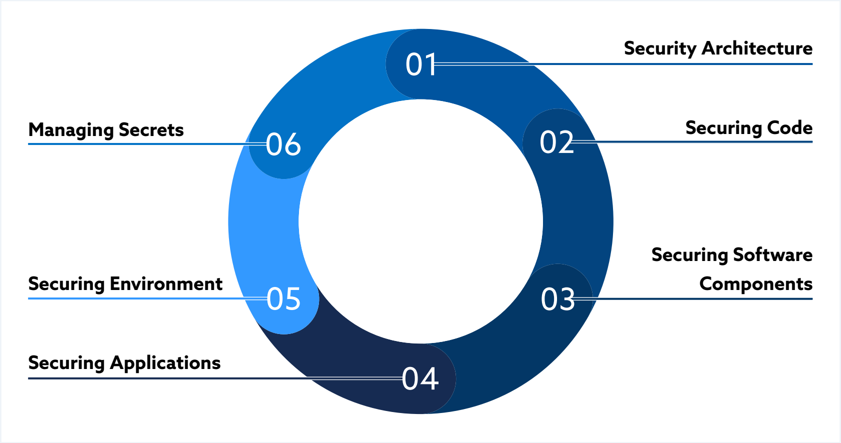

Figure 2: Security Activities in the Security-enabled Delivery Pipelines

6.2 Securing Design

Security can be achieved only when it has been designed in. Applying security measures as an afterthought is a recipe for disaster. Design flaws are known to account for a large portion of vulnerabilities [1].

While some consider manual processes to be incompatible with DevSecOps practices, it is not the case. There are core security design activities that require human intelligence, such as threat modeling, which must be performed to ensure establishment of a secure architecture.

Traditionally, threat modeling can be performed only manually, but there are tools today that help automate this process. Moreover, threat models typically do not require frequent updates unless new components are added to or security-sensitive components are changed in the application, so manual updates still remain a viable mechanism.

6.3 Securing Code

Vulnerabilities can arise from poor coding, regardless of how well thought-out and robust the security architecture of an application is. If the code written by developers contains security defects, it is only a matter of time before the application gets breached.

SAST tools can help scan source codes of the deliverables and report security (and generic) defects, preventing them from manifesting as security vulnerabilities.

© Copyright 2020, Cloud Security Alliance. All rights reserved.

The goal of DevSecOps when automating static code testing is to tighten the feedback loop to developers by providing code security alerts as early as possible, in the context and as prescriptive as possible to developers.

There are numerous opportunities in the delivery pipeline to perform security scanning, including:

- On the local machine: through the IDE or integrated test suites
- Continuous integration with code scanning: code scanning on code push, merges, releases on the continuous integration system
- Continuous delivery pre-deployment: scanning prior to deployment

6.4 Securing Software Components

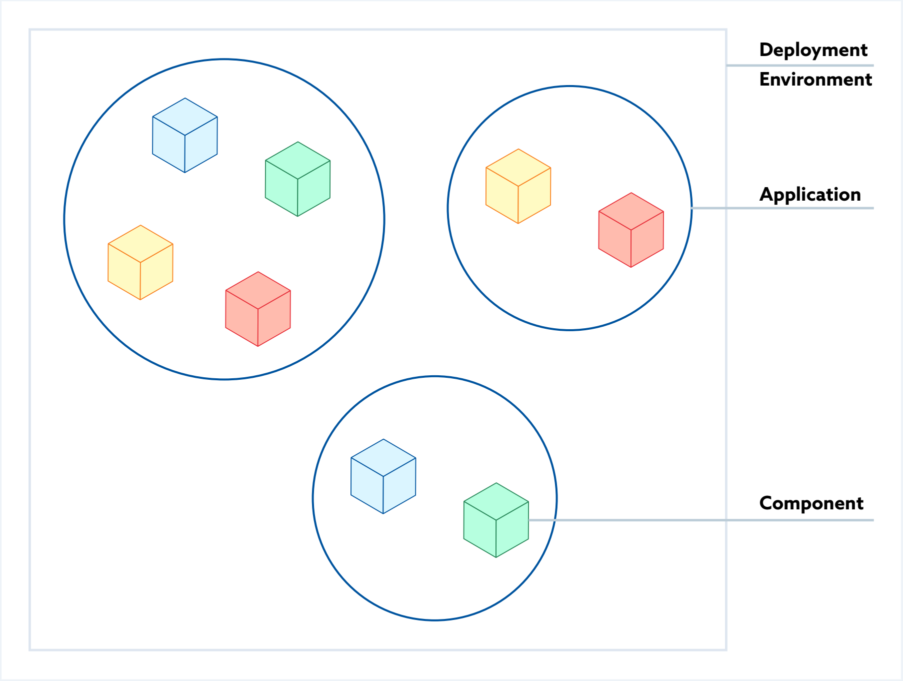

Figure 3: Relationship among Application, Application Components and Environment

An application can be only as secure as its components. Vulnerabilities of its components, whether produced in-house, by third-parties or from open source projects, will likely compromise the larger application.

It is crucial to manage these software components in a systematic manner, with measures such as:

- prompt detection and reporting of vulnerabilities within third-party software such as open source and containers
- detecting usage of unauthorized components

Vulnerability management of these components should be continuous, such that it is done not only during the development process, but also thereafter. It is critical to ensure that the third-party components used remain secure during the life of the application.

Upon the release of any public vulnerability of components of the application, a process should be triggered indicating the need to upgrade to a non-vulnerable version of the componentLibrary.

This process can be automated using source code scanning tools through the continuous integration process or runtime scans.

6.5 Securing Applications

An application must be security tested in a realistic environment before it can be trusted. Even with a solid architecture, secure code and vulnerability-free components, the integration of them does not necessarily lead to a bullet-proof application.

Dynamic application testing can be used to test applications in the staging, testing and production environments for assurance. DAST tools can discover a variety of security issues from fault injection, fault detection, to resource and secret leaks. (See Figure 1 for the appropriate placement(s) of DAST in the pipeline).

Fuzzing is also an effective method for identifying security vulnerabilities. Various types of automated fuzzing are available to be run within the delivery pipeline to ensure the application's resilience against erroneous and malicious input.

6.6 Securing Environments

General

A hostile environment renders a secure application insecure. The approaches of Infrastructure as Code (IaaC), containerized applications and continuous deployment present new possibilities for securing infrastructure and environments, including:

A hostile environment renders a secure application insecure. The approaches of Infrastructure as Code (IaaC), containerized applications and continuous deployment present new possibilities for securing infrastructure and environments, including:

1. Scanning of IaaC code and artifacts
2. Runtime defense within environments

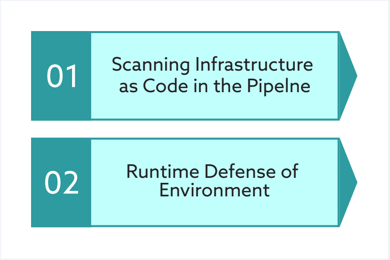
01
Scanning Infrastructure as Code in the Pipelne
02
Runtime Defense of Environment

Figure 4: Multi-pronged Approach to Securing Infrastructure

© Copyright 2020, Cloud Security Alliance. All rights reserved.

IaaC Security

IaaC code is used for declarative management of cloud infrastructure resources, and the IaaC configuration state can be an accurate reflection of the actual configuration of those resources. The security analysis of IaaC code and artifacts can be performed with two mechanisms:

1. Static scanning of IaaC templates and configuration states
2. Automated scanning of declaratively-deployed resources

Runtime Security

Deployed applications are subject to the security of their environment and therefore must be monitored. There are generally two approaches to this:

1. Runtime monitoring of deployed applications and environments are important measures in ensuring that both the environment and application are protected (see Figure 4).
2. Runtime defense ensures that discovered vulnerabilities and attacks at runtime are mitigated and reported as quickly as possible.

6.7 Managing Secrets

The management of secrets within the application and the delivery pipeline is paramount and impacts the entire delivery pipeline. The use of proper cryptographic algorithms does not necessarily provide sufficient security against adversaries. For instance, a developer may select the best encryption algorithm, but if the key is stored in an open location, the security becomes compromised.

Processes including key management and key rotation, must be properly handled. Cloud key vaults and key management infrastructures may be considered as part of an automated solution.

7. Automation Best Practices

7.1 General

Many security activities used in the security-enabled delivery pipeline can be used independently in any DevSecOps process. Most mature organizations have provisions for out of band (also known as asynchronous) testing. This section highlights best practices and considerations that are applicable beyond DevSecOps.

7.2 Mitigating Vulnerabilities

Vulnerabilities are easiest to address when they are just created. With this in mind, the strategy is to move the detection of newly created vulnerabilities towards the development process, prior to deployment. For example, tools that check for security issues within the IDE or prior to committing the code changes can be considered.

For vulnerable code detected at later stages, it is important that they are mitigated prior to production deployment. Based on the assessed risk of vulnerabilities, ones of low severity may be addressed later; those of high severity must be addressed as soon as possible.

7.3 Asynchronous Testing (Out-of-band Testing)

There are generally two categories of asynchronous testing:

> - Certain crucial security activities cannot be fully automated as they require human intelligence. These include threat modeling, penetration testing, and peer code review.
> - Heavyweight automated tests that take a long time to perform, such as SAST.

These activities can be triggered and performed "out of band" rather than inline with automated deployment so as to not disrupt the software deployment pipeline.

Issues identified out of band must be tracked by the delivery team to ensure visibility, prioritization, and potential rollback depending on the risks identified. If critical issues arising from out-of-band testing are not addressed in time, they should count as blockers to the continuous delivery process at the appropriate checkpoints.

7.4 Continuous Feedback Loops

The longer a security defect exists in the application, the more likely further changes and deployment will add to the complexity and cost of remediating the defect.

DevSecOps teams must be alerted to security issues as early as possible so that defects are prevented from going downstream.

Furthermore, timely feedback empowers individuals to diagnose and remediate changes that caused the vulnerability quickly, allowing them to perform root cause analysis while the accompanying context is still fresh in mind.

7.5 Breaking Builds

The organization needs to recognize that security is a business requirement for security activities to declare "broken builds" on a failure to meet security requirements while disrupting an automated deployment pipeline.

Breaking builds for security enforces quality, provides rapid feedback to DevSecOps teams, and prevents data exposure and potential exploits outside the organization's risk appetite, e.g. releasing code with a high severity, publicly known vulnerability with active exploits.

Activities that can break a build include:

> - High severity defect identified beyond outside organization's risk appetite
> - Specific vulnerability or vulnerability class identified that conflicts with organization policy
> - Number of high or critical findings crosses a certain threshold
> - Necessary security activities fail to execute effectively, e.g. automated scans are failing because of configuration errors

8. Conclusion

In order to benefit from DevSecOps and the reflexive security approach, one must rely heavily on automation.

The Automation pillar can be achieved using the security-enabled delivery pipeline and by the application of security controls within it. The risk-prioritized approach further allows optimization for continuous delivery and a mechanism for handling non-automated security checks.

© Copyright 2020, Cloud Security Alliance. All rights reserved.

References

1. 3 security risks that architecture analysis can resolve. Synopsys, 2016. Available at: https://www.synopsys.com/blogs/software-security/security-risks-that-architecture-analysis-can-resolve/

2. Tactical Threat Modeling. SAFECode, 2017. Available at: https://safecode.org/wp-content/uploads/2017/05/SAFECode_TM_Whitepaper.pdf

3. Managing Security Risks Inherent in the Use of Third-party Components. SAFECode, 2017. Available at: https://safecode.org/wp-content/uploads/2017/05/SAFECode_TPC_Whitepaper.pdf

4. Focus on Fuzzing. SAFECode, 2020. Available at: https://safecode.org/fuzzing/

© Copyright 2020, Cloud Security Alliance. All rights reserved.