GitGuardian

DevSecOps Blueprint:

Implementing Security-by-Design and Effective Vulnerability Management

By C.J. May

The DevOps movement has established a culture of automation as the heart and soul of the software development world. Over time, application security has similarly evolved into “DevSecOps” which promises to enhance security while maintaining high levels of velocity.

In this document, we will explore a holistic implementation guide for DevSecOps as a collection of technology-driven, automated processes.

- **What are the tools and technologies that play a role in DevSecOps?**
- **How can we use technology to set software engineering teams up for success?**
- **How do we align roles and responsibilities to ensure cohesion, safety, and velocity?**

These are some of the questions that will be answered as we progress through the architecture narrative. This is the mission statement we will be using as a guideline:

> The objective is to implement a secure-by-design software development process that empowers engineering teams to own the security for their own digital products. This is achieved through controls and training, reduced friction, and a focus on velocity through a technology-driven, automated DevSecOps architecture.

There will be examples in this ebook that may not apply directly to your company, but the themes and overall approach may still be valuable to you.

Table of contents

PART 1

Vulnerability Management Lifecycle

First, we will be examining how technology can support the “people and processes” side of DevSecOps. Controls and automation will increasingly play a role in this program architecture, but we need to start by defining the roles and responsibilities of the humans involved.

At its core, DevSecOps is about managing vulnerabilities within software products. Some of these vulnerabilities are introduced by the software engineers creating the product, and others come from third-party dependencies that teams have little control over. The work needed to remediate these vulnerabilities must compete with other work like new features and bug fixes. In this section, we will look at a technology-driven vulnerability management lifecycle that allows informed decisions to be made about work selection.

Stages of vulnerability management

The vulnerability management process can be broken down into 3 stages:

Identification — Observability — Management

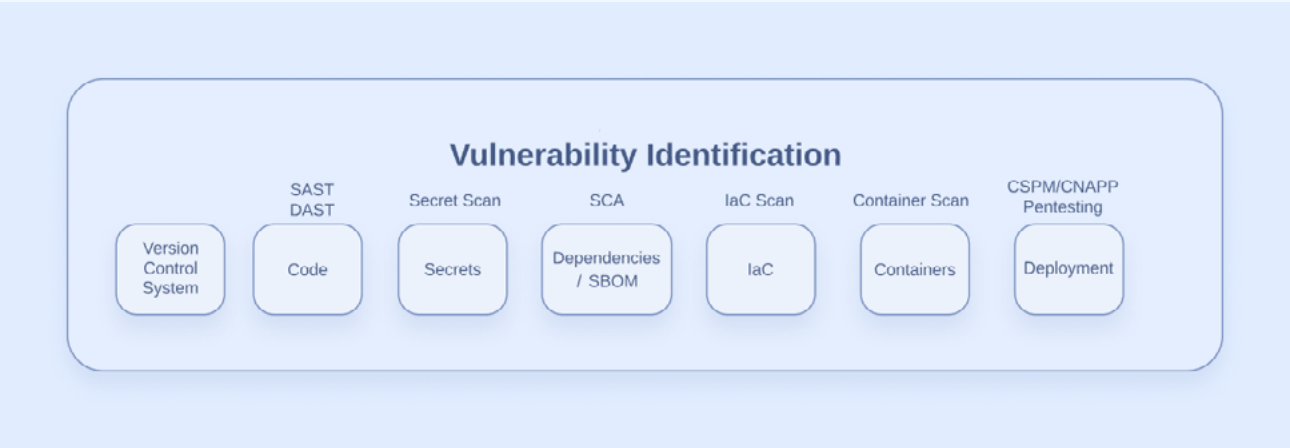
Each stage is critical for the success of the next one. Below is a simplified diagram that we will add to as we go through this section. Technology plays a central role in each stage, but human interactions should not be overlooked as well.

▶ *Figure 1: The 3 stages of the vulnerability management process*

Identification

This probably goes without saying, but you can't fix vulnerabilities that you aren't aware of. Identifying vulnerabilities is a complex challenge, though. It's not just a matter of scanning source code; there are many artifacts in the layers of a software product that require their own type of vulnerability scanner.

The first stage of the diagram is a list of places in the software development lifecycle (SDLC) where vulnerabilities or misconfigurations can arise, as well as the types of vulnerability scanning tools that can identify security issues in these areas. Here's a quick rundown of each area:



▶ *Figure 2: Identification stage*

VERSION CONTROL SYSTEM

GitHub is an example of a VCS, and it is often overlooked as an area that can create security risks. Misconfigured actions and poor access control can allow attackers to gain read access to internal code or even inject malicious code.

CODE

In this context, "Code" refers to the custom code that was written by the software engineers at your own company. Vulnerability scanners such as SAST and DAST (Static/Dynamic Application Security Testing) can identify various vulnerabilities that were accidentally created in the code. But they can't catch everything, which is why you need other specialized tools for vulnerability identification.

SECRETS

Finding and preventing leaked secrets is what GitGuardian is all about. Hackers and

red teamers regularly find secrets in plain text that allow them to elevate their access. Identifying leaks and prioritizing their cleanup is a critical piece of DevSecOps.

DEPENDENCIES/SBOM

SBOM stands for Software Bill of Materials, and it refers to a document containing all the third-party dependencies in your code. Why would you want to track software dependencies? New vulnerabilities emerge every day in open-source code. If your software uses a vulnerable dependency, it may also be exploitable. Software Composition Analysis (SCA) tools identify vulnerabilities in the dependencies you use.

IAC

Infrastructure-as-Code (IaC) refers to deployment code such as Terraform or Kubernetes that declaratively defines how your deployment will be configured. IaC scanners identify security misconfigurations and unsafe exposure in your planned deployments.

CONTAINERS

Containers add another layer of dependencies to your software, because they provide all the operating system programs that your code needs to run. These added dependencies can also introduce security issues into the runtime of your application. Container image scanners help you identify these kinds of issues.

DEPLOYMENT

Application deployments are the final area that we will cover. If you run your app in the cloud, you can use Cloud-Native Application Protection Platform (CNAPP) or Cloud Security Posture Management (CSPM) tools to identify misconfigurations and known vulnerabilities in your deployed application. Pentesting your live applications is another way to find previously undiscovered vulnerabilities. Human-operated security testing is the only way to find some kinds of vulnerabilities, such as business logic flaws.

This list may not be exhaustive, and new threats and tools are emerging all the time. But this highlights the plethora of vulnerability types that we're dealing with. The complexity of security scanners is the first major challenge of vulnerability management. There are many things that need to be evaluated for security issues, and no single tool can do it all. If you have coverage in all these areas, you will likely end up with a mix of sources that identify vulnerabilities. This can get messy, which is what leads us to the next stage of the process.

Observability

Once you have identified vulnerabilities, you need to make sense of the noise. The observability stage is where the magic happens in the process diagram. In this stage, you are translating the technical security issues into higher-level metrics and scores to inform your decision-makers who are not security experts.

As an aside, it's worth noting that the industry's overuse of the term "critical" to describe vulnerabilities has somewhat diluted its impact. With so many issues labeled as critical, the meaning has been diluted to the point that we no longer have a term that invokes a swift response without interpretation and guidance from a security professional. However, by layering additional context through the observability techniques described above, organizations can cut through the noise and focus on the most impactful remediation actions.

This leads us to the second major challenge of vulnerability management: translating risk for non-security professionals and prioritizing vulnerability findings. The observability stage of the vulnerability management process is where you attempt to do that translation.

▶ *Figure 3: Observability stage*

On the right side of the diagram, you can see that observability mainly happens in the places where software engineers do their work. By taking advantage of features like Pull Request checks and IDE plugins, you are making vulnerabilities relevant where they are relevant. Minimizing context switching is a key focus in DevSecOps architecture that aims to reduce friction for those involved.

On the management-focused side, there is a bit more going on. The Product Owners (POs) or Managers decide what gets worked on. For these people to make informed decisions, you need to present existing vulnerabilities in a way that is digestible for someone who is not a security expert.

Informing POs and managers about vulnerabilities can be as simple as granting them access to the dashboards of your vulnerability scanning tools. All security scanners have their own severity scoring system that attempts to rank the findings, but some tools are better than others. GitGuardian allows you to tune the severities yourself, which is a great feature.

Using individual tools to inform your less technical decision-makers isn't ideal, though. Decision-makers don't want to go to a bunch of different dashboards and learn how each tool works. They want a single place they can go to get a clear view of the security risk in the digital products they are responsible for.
To set your program for success, you need to distill the multitude of vulnerabilities into a prioritized list that is relevant for your business. A tool category that aims to do this is Application Security Posture Management (ASPM).

The term "ASPM" is relatively new. Garter defined it in May 2023 as a solution that "continuously manages application risk through collection, analysis, and prioritization of security issues across the software life cycle." The main idea is that you ingest the findings from your vulnerability scanning tools into an ASPM, and it does the prioritization and metrics for you. If you want to read more about ASPM, check out this article from GitGuardian.

While an ASPM isn't necessary for success, you will certainly need many of the capabilities associated with ASPM. For the sake of our process diagram, "ASPM" can refer to a process or solution that provides the following benefits for the observability of security issues:

- **"Single pane of glass" overview of your risk across multiple types of vulnerabilities**
- **Hierarchical grouping of projects, teams, product groups, etc.**
- **Context-based prioritization based on public exposure, known exploitation, business criticality, commit frequency, etc.**
- **Deduplication of vulnerabilities across multiple tools**
- **Generation of SLIs such as time-to-remediation, risk scores, etc.**

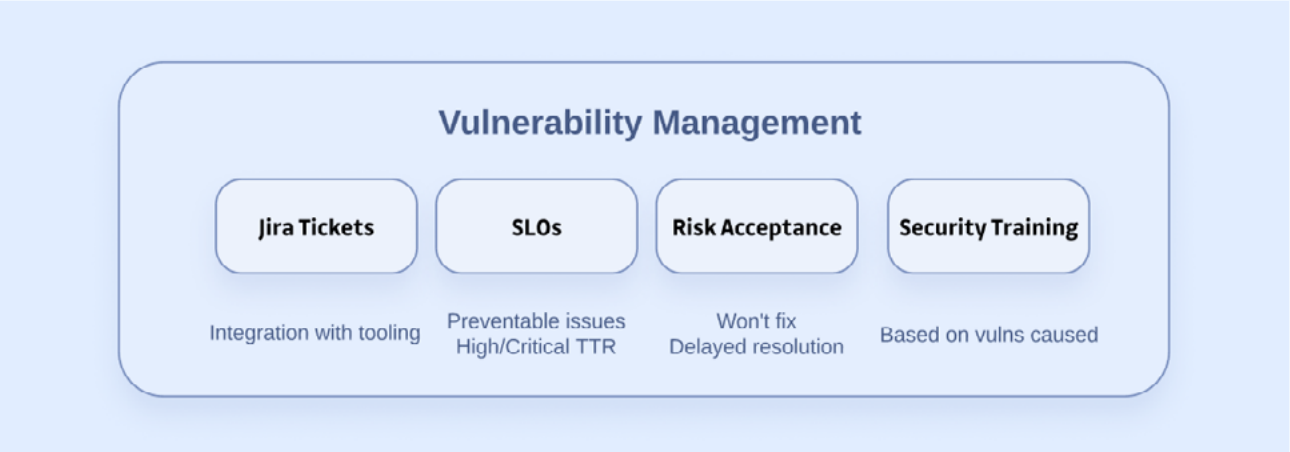
- **Team or developer-based metrics to identify training needs**
- **Corporate memory (who has fixed similar vulnerabilities that can help)**

You may be able to build your own solution for some of these features, or your existing tools might get you close enough. You could also hire someone whose job it is to organize and track vulnerability findings. In the end, the most important outcomes of the observability stage are the metrics and prioritization of your open vulnerability findings. If your vulnerability management program feels like a mess of findings, you're probably lacking some of the organizational capabilities listed above.

Lastly, as you tune the prioritization or scoring for your observability layer, remember the key warning from this section: if everything is critical, nothing is. Do your best to present the risk in a way that accurately reflects the likelihood and business impact of the vulnerabilities. In the next section, we will cover what it looks like for your product owners or business leaders to make decisions about vulnerability risk and remediation.

Management

Once the team leaders have access to high-fidelity data on their team's security risk, they can make informed decisions about work selection, risk management, and training opportunities. Below is the last stage of our process diagram.



▶ *Figure 4: Management stage*

Most use Jira or another ticketing system to plan and track work. Product owners or managers can use these systems to create issues for vulnerabilities that need to be remediated. If the observability stage has been successful in prioritizing risks, team leaders should be equipped to make informed decisions about this work themselves.

If more guidance is desired, Service Level Objectives (SLOs) are another way to think about work selection. Examples of SLOs could be:

- **Introducing no more than 1 preventable vulnerability per sprint cycle**
- **Remediating critical vulnerabilities within 7 days**
- **Remediating high vulnerabilities within 4 sprint cycles**

These are just made-up examples. Security, product teams, and decision-makers should collaborate on the creation of SLOs to find a balance that works best for the business. Additionally, the security team will continue to play an important role in auditing open vulnerabilities for imminent threats.

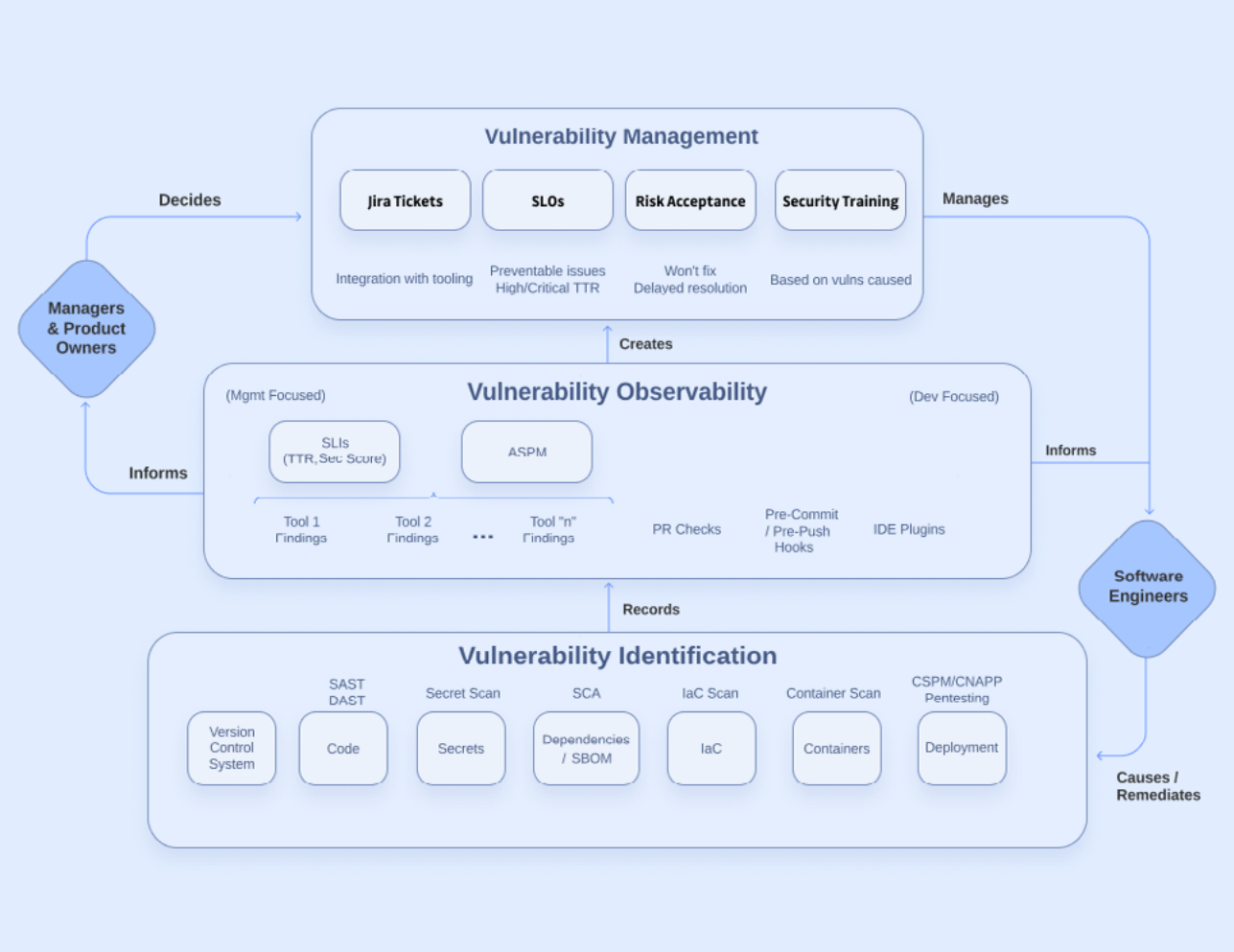
There may be times when an SLO cannot be met. In those cases, the security risk needs to be escalated along with the current workload of the product team. Security issues must fight for developer time with new features, bug fixes, and tech debt. Sometimes, business leaders may decide to accept the risk of a security issue or delay the fix because they believe it's in the best interest of the company. Other times, a critical security issue may need to delay the launch of a new feature, and timelines need to be shifted.

The last piece of vulnerability management is an often-overlooked area: security training for non-security personnel. Insights from your observability stage can highlight which types of vulnerabilities a developer or team is struggling with. These insights help security teams and leaders identify training needs.

The most important part of security training is creating a positive learning culture. Security training shouldn't feel like a punishment. If it does, then the implementation needs some work. A great example of security training is having a regular security segment in a meeting dedicated to sharing and learning. The security topics covered should prioritize the needs of the audience and be presented in a way that is positive, not pointing fingers.

Final process diagram

Now that we've covered each stage of vulnerability management, this is the final process diagram. Here's a link to a copyable version on Lucidchart that you can use to track your progress. You can color-code areas to mark them as having full, partial, or no coverage.



▶ *Figure 5: Vulnerability management process full diagram*

Roles and shared responsibilities

We've covered the whole process diagram, but we still need to define roles and responsibilities. The following section is my suggested role structure to support the success of the vulnerability management process that we've laid out in this section.

Software Engineers

Software engineers share the ownership of the security work for the business's software products. They are responsible for:

- **Preventing the introduction of new vulnerabilities in their team's projects**
- **Remediating existing vulnerabilities in their team's projects**
- **Adhering to security-related SLOs, if any**

Managers and Product Owners

Managers and product owners share ownership of the security work for the business's software products. They are responsible for:

- **Being aware of the existing vulnerabilities in their team's projects**
- **Managing the work related to resolving vulnerabilities**
- **Escalating risk when other work conflicts with vulnerability remediation**
- **Adhering to security-related SLOs, if any**

Directors and Tech Executives

The higher-level decision-makers own the security risk for the business's software products. They are responsible for:

- **Being aware of critical severity vulnerabilities in the business's digital products**
- **Accepting or escalating risk when prioritizing other work over vulnerability remediation**
- **Working with the business when prioritizing vulnerability remediation over other types of work**
- **Auditing security-related SLOs, if any**

Security Team

The software security team owns the work that supports the success of the vulnerability management process. They are responsible for:

- **Managing technical tools for vulnerability identification and prevention**
- **Providing conceptual and tool-based training for software engineers**
- **Managing technical tools for vulnerability observability**
- **Auditing open vulnerabilities for imminent threats**
- **Consulting for individual vulnerabilities as needed**
- **Human-operated penetration testing for additional vulnerability identification**
- **Working with other roles to establish security-related SLOs**

Vulnerability Management Lifecycle

Summary

We've covered a lot in this section, so let's do a quick recap. This section showed how technology can support the "people and processes" side of the vulnerability management lifecycle. In the context of DevSecOps, the vulnerability management process includes 3 major stages: identification, observability, and management. In each stage, a considerate implementation of the technologies we covered is critical for setting yourself up for success. Ultimately, your goal is to empower digital product teams to make informed decisions about how much work needs to be dedicated to remediating security risk.

In software development, security is a collective and collaborative effort. Security teams, developers, and decision-makers all play a role in preventing threats from impacting those who consume our software.

In this section, we covered how to approach DevSecOps from the vulnerability management side of software development. The next section will cover the software engineering side of things, where we will look at how to prevent the introduction of new vulnerabilities.

PART 2

Secure-by-Design Software

In this section, you will learn how to use controls and automation to create a secure-by-design software development pipeline. This part of DevSecOps is less about the "decision-making" side of things, and more about the developer experience. You will learn how to equip software engineers with the tools they need to successfully own the security for their own code, and how to support them through automation.

What does "secure-by-design" mean?

The primary goal of DevSecOps is to produce applications that are "secure by design." This means that your software development lifecycle (SDLC) includes security as part of the development process, guaranteeing a minimum level of security for every software product.

Through this strategy, your digital products will have gone through rigorous security testing by the time they are published, and you will have caught many vulnerabilities before they are made available to customers.

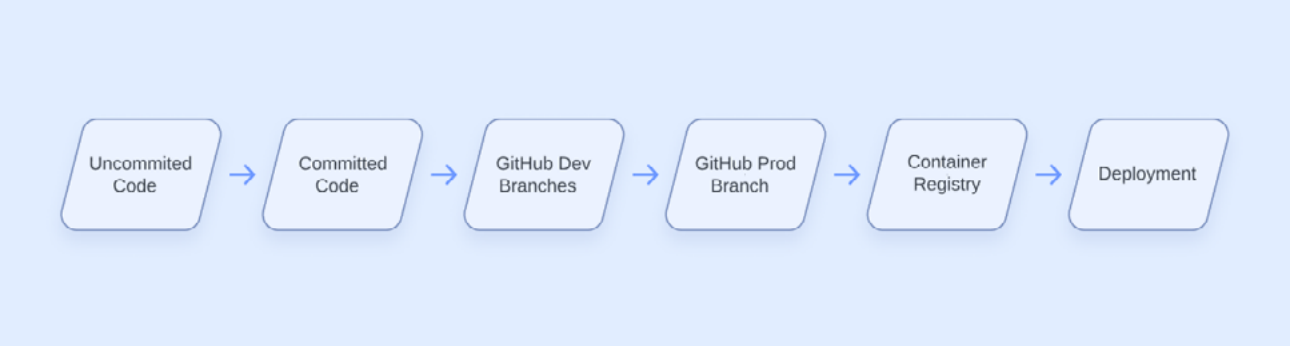
There are 2 key components in the software development pipeline that support a secure-by-design strategy:

Security Gates — Automation

These 2 components work together to enforce a security baseline while preserving efficiency. Just like last time, we will be building a diagram to visualize how this process works.

Software development pipeline

In DevSecOps, security procedures are built into the existing software development pipeline. In this section, we will look at an example organization that produces container-based apps deployed through Kubernetes. These are the high-level artifacts or steps in the example pipeline:



▶ *Figure 6: Example pipeline*

Your development process may look different, but as we go, you'll find that many of the concepts will be transferable to whatever you're working with. Let's break each stage down briefly before moving on.

Uncommitted Code

The software development pipeline begins with the software engineer drafting new code or modifying the application. This typically happens on the developer's local workstation. From there, the code is added to the git history using the "git commit" command.

Committed Code

Once the code has been committed to git, it will now exist in the repository's history forever. We can make additional changes, but the old commits are still able to track how the code was changed over time. At some point, we will "git push" our code to our git hosting provider. In this case, that's GitHub.

GitHub Dev Branch

It's not good practice to make changes directly on your production (prod) branch, so we have a different development (dev) branch to push our code to first. Once our code is ready to be published, we will open a Pull Request to review and merge the changes from the dev branch into our prod branch.

GitHub Prod Branch

If the code passes review in the Pull Request from the dev branch, it is ready to be published as a container. We will build a container image with our code and publish the image to our container registry.

Container Registry

In the container registry, our software sits frozen as an image waiting to be deployed. Our example organization uses Kubernetes for orchestration to deploy the container(s) to run our app.

Deployment

Once we've deployed our app, it is running and exposed to the internet. Years ago, this was where most organizations started doing vulnerability assessments of their applications. This means any major security issue would require you to start from the beginning and go through the entire process again.

That's obviously not very efficient, so over time, new tools were created to help developers shift security "left" in the pipeline and catch vulnerabilities earlier. Unfortunately, these tools have only been marginally effective, and vulnerable software is still extremely common. If we have the tools to tell us what is vulnerable and things still aren't getting better, what are we doing wrong?
Many of our challenges with secure software development lie within our requirements. The ONLY requirement for software development has traditionally been that the app meets our functionality needs.

To make software that is secure-by-design, you need to redefine what "production-ready" means for your digital products. You must agree upon some minimum level of security and guarantee that you are meeting those standards through security gates that prevent code from moving on until it has met your security requirements. Security gates are the first key component of a DevSecOps pipeline.

Security gates

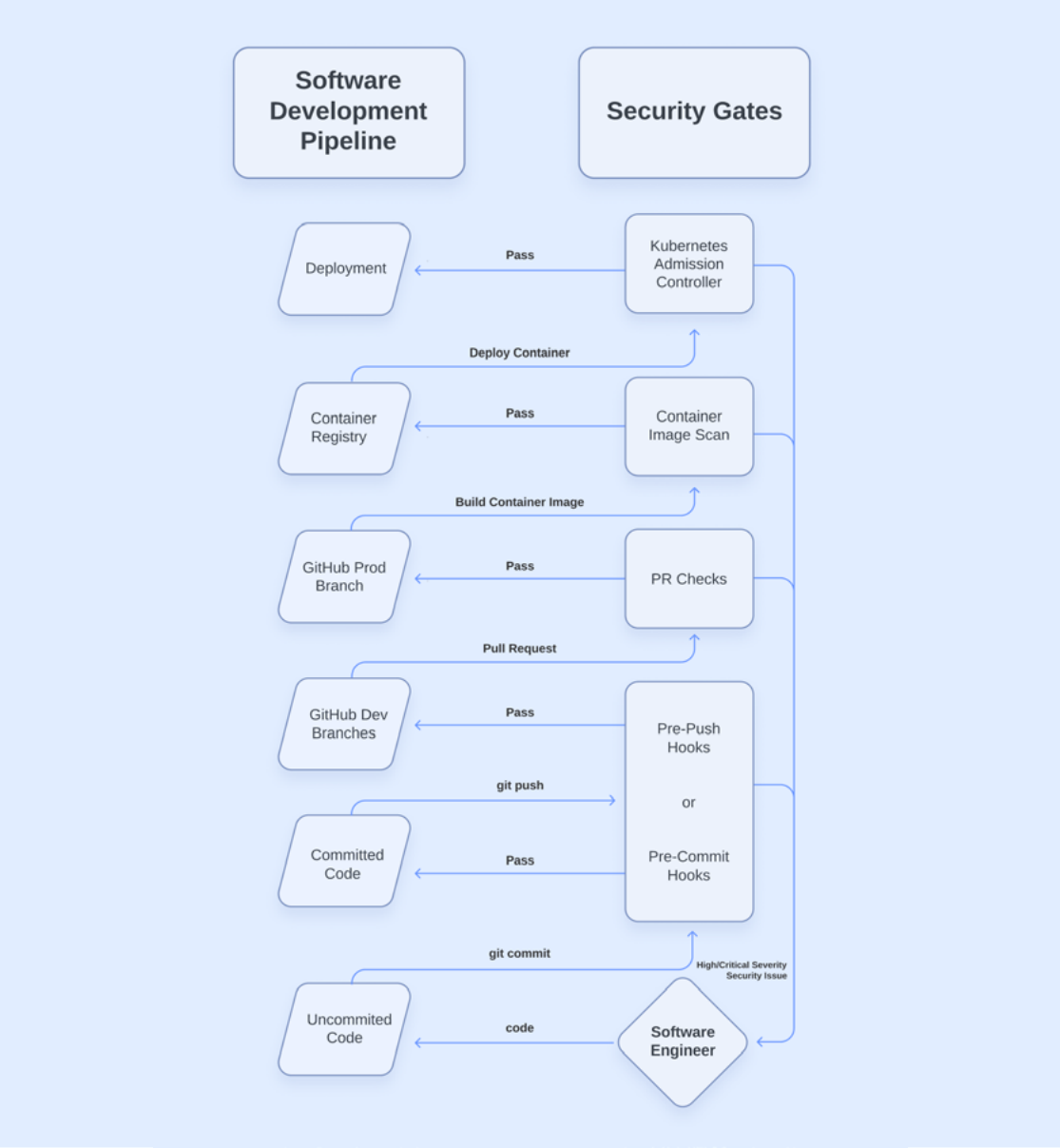
To guarantee security in your software development pipeline, you need to add security checks to each step of the process. Each of these checks will be automated to make sure that they can't be forgotten.

Scanner Types

Earlier in this document, we covered many types of vulnerability scanners in the "Identification" stage of our process diagram. The same types of scanners will be used in the security gates we are about to cover. There is a difference in how these scanners are used as security gates, though.

In the vulnerability management lifecycle from last time, the security scanners cataloged historical vulnerabilities previously introduced in our source code. In security gates, we are mostly concerned with preventing the introduction of new vulnerabilities.

There's one more consideration you should make for scanner implementation. Web dashboards are helpful for decision-makers and security personnel, but they aren't a good fit for software engineers. Rather than making engineers context-switch and go somewhere else to see their vulnerabilities, you should integrate your security scanners with the tools they already use. By doing this, you are showing them vulnerabilities *where they're relevant*, and *when they're relevant*.



▶ *Figure 8: Continuous automation added to the development pipeline*

With that out of the way, let's discuss the different types of security gates you can set up. Below is the next part of our diagram, which introduces security gates into the existing pipeline.

As you can see, rather than moving directly from one stage of your pipeline to the next, an automated security check must be passed before the code can progress. You can set whatever you want as your threshold for failure, but ultimately, a failure will require software engineers to fix vulnerabilities before they can move on.

Now, let's describe in detail each type of security gate. We're going to start slightly out of order, but it will be clear why in a moment.

Pull Request checks

Pull Request (PR) checks are one of the most critical pieces in a secure-by-design SDLC because PRs are the first place in your pipeline where you can truly guarantee that security checks are happening. Prior to the code being in GitHub, everything is happening in local development environments. Hopefully, you trust your software engineers, but you can never be 100% sure that they aren't disabling security checks or using personal devices to write code. You do have full control of the GitHub repository, though.

By configuring branch protection rules, you can force all code to go through a Pull Request before transitioning from a "development" branch to the "production" branch. In DevSecOps, PRs are an opportunity to run various security checks and require them to pass before the code can be merged.

Pre-commit and pre-push hooks

Even though the Pull Request is the first enforceable security gate, it's still important to shift checks further "left" to catch things even earlier in the process. Pre-commit and pre-push hooks are automated actions that run when you run a "git commit" or "git push" command.

As a security gate on software engineers' workstations, you can configure pre-commit or pre-push hooks. I emphasize the word "or" because the two hook types are redundant. At this level, all you need to do is pass security checks before pushing your code to GitHub.

There are reasons to use one or the other, depending on what type of scan we're talking about. For example, you can set up a hook to discover leaked secrets. A pre-

push hook won't detect a leaked secret until it has already been committed to the git history. At that point, it might be difficult to undo your commits and you will have to revoke and rotate the secret.

You can save yourself from all that extra work by configuring your secret scanner as a pre-commit hook, which prevents the code from entering the git history until it is free from secrets. This is a great example of why you need security gates at multiple steps of the software development pipeline. You can't guarantee or perform some scans until later, but it saves time by catching things as early as possible.

On the other hand, pre-push hooks might make more sense for SAST or SCA scanners. Overdoing pre-commit hooks can cause you to be unable to commit the code you are working on —especially when you are trying to commit a lot of new code all at once. By using pre-push hooks instead of pre-commit, you are allowed to commit code with vulnerabilities and still use the version control functionality of git. You just need to fix any scan failures before pushing to GitHub.

The debate of pre-commit versus pre-push hooks is arguable other than secret scans. The recommendation is to have security and software engineering teams meet to discuss and agree on what makes the most sense. The key thing to remember from this section is that performing the same scan at different stages of the SDLC is helpful, not unnecessary.

Container image scans

Now let's get back on track in the software development pipeline. We've pushed code to GitHub and opened a Pull Request that passed the security gates in our PR checks. Now the code has been merged into our "prod" branch, and we are ready to build the container image that will run our application. If you have multiple environments, this same thing might occur in your "dev" or "test" branch. This security gate will look the same either way.

Successful merges should be configured to build and publish your container image using an automated workflow. After the workflow step that builds the image, you should scan the container image for vulnerabilities. The scan should fail the build if it finds vulnerable dependencies in the image that exceed your severity threshold. If the scan passes, your workflow publishes the image to your container registry, where it will wait to be deployed.

Kubernetes admission controller

The last security gate in our example pipeline is a Kubernetes admission controller. Admission controllers are just plugins that validate or even modify the instructions given to your Kubernetes cluster. The capabilities of the admission controller depend on the plugins.

In our example pipeline, the admission controller scans the container images being deployed for critical dependency vulnerabilities that may have been introduced since the container was built. You could also enforce configuration policies that prevent unsafe settings from your Kubernetes infrastructure-as-code (IaC).

If the admission controller plugins pass all their validations, the application finally reaches deployment. Having gone through each of the security gates we covered along the way, you can now be confident that your deployed application meets the minimum security level that you configured in your security gates.

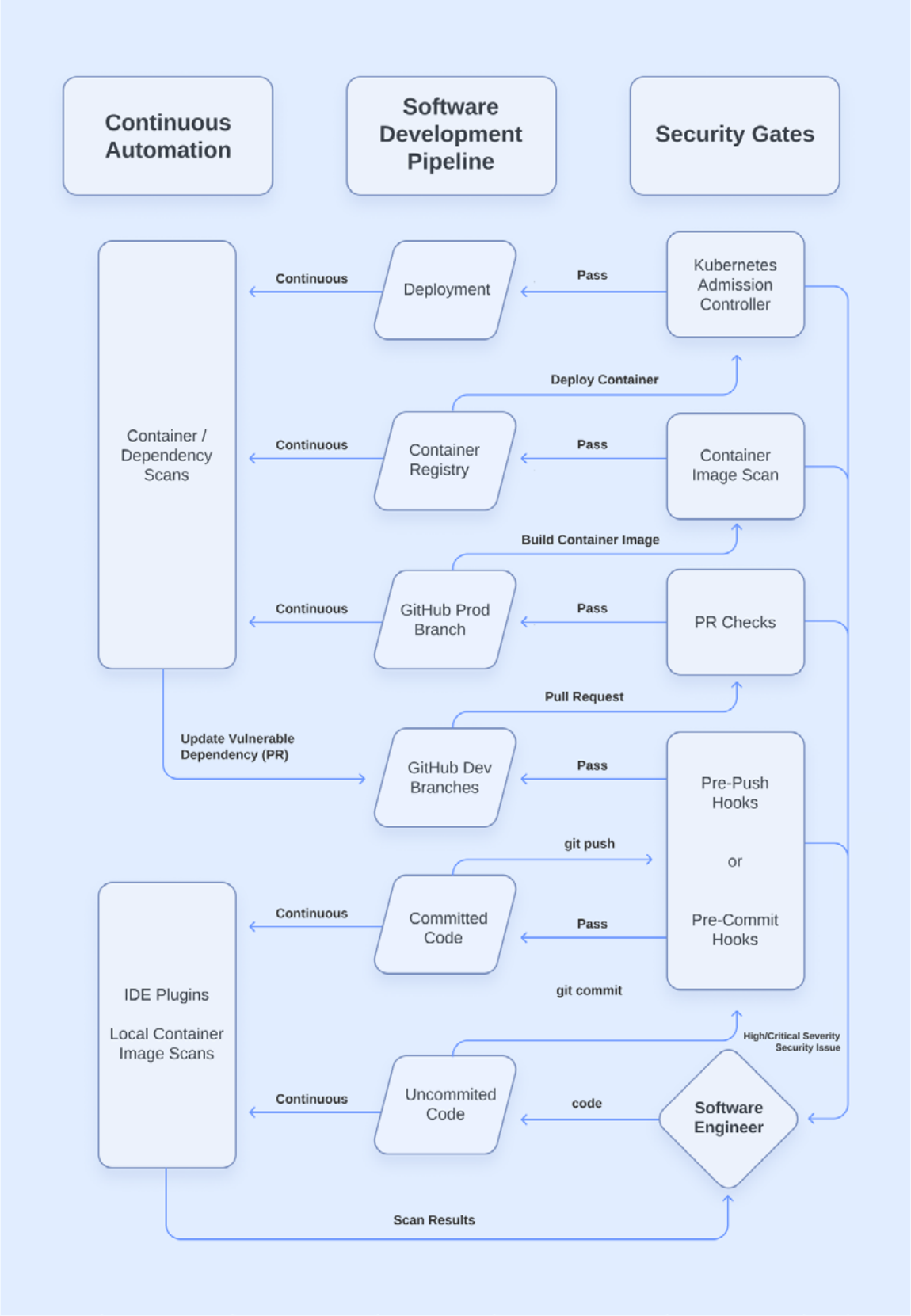
Assisting with automation

At this point, you're probably thinking, "This is adding a lot of barriers in the way of getting to deployment." You're not wrong, but you should remember that we are redefining what "production-ready" means. Is it beneficial to rush to deployment if your app is full of holes?
Luckily for us, there is another component to DevSecOps pipelines that aims to assist developers and protect velocity: Continuous Automation.

The goal of Continuous Automation is to provide software engineers with assistive technology that produces automated feedback and security fixes. This will reduce the time and cognitive load needed to meet your new security requirements. Just like the security gates, you should try to utilize technologies that operate where work is already being done to reduce context-switching. Below is the final process diagram, to which we have added the Continuous Automation column:

IDE Plugins

The IDE is as far "left" as you can get when providing vulnerability feedback—it's where software engineers are writing the code! By using security scanners as IDE plugins, you get instant feedback in the linter, which highlights issues in the code that are already being worked on. There are all sorts of security scans that can take place in



▶ *Figure 8: Continuous automation added to the development pipeline*

IDEs: SAST, SCA, Secrets, IaC, etc... These plugins will work the same whether the code has been committed or not.

Local container image scan

Because container images aren't built until the later stages of the software development pipeline, they don't get stopped by a security gate until you are quite far into the process. The further "right" you are when an issue is discovered, the more time-consuming it is for you to fix it.

To reduce the number of times you get stopped by a container image scan late in your security gates, you need to give your software engineers the ability to run the same container image scans locally on their workstations in the earliest stages of development.

Automated Dependency Updates

Out of everything in the DevSecOps pipeline, one of the most challenging things you'll deal with is the constant discovery of vulnerabilities in your dependencies. Software can pass through all your security gates and reach deployment, only to have a critical vulnerability discovered in some component that your app uses the very next day. While software engineers can prevent the introduction of many types of vulnerabilities, this will always be something that they have to deal with. This problem can be reduced by using minimal container base images that have fewer packages, but you will always have some dependencies.

To make this ongoing challenge easier and less time-consuming, you must use tools that automate dependency updates. SCA and Container Image scanners identify vulnerable dependencies in your runtimes, container registries, and source code. Robust dependency scanners will have the option to automatically create Pull Requests that update the vulnerable packages to a fixed version. Then, all you need to do is accept the fix and merge the PR.

Others

As technology evolves, there may be new ways to automate security fixes. For example, security researcher Jonathan Leitschuh has used CodeQL and OpenRewrite to automate vulnerability discovery and patches across many open-source repositories. There is also a chance that we can use future AI models to find and fix vulnerabilities in our code. Ultimately, a major goal of DevSecOps is to help developers work less while

maintaining security. Any chance you get to do that will be a huge win for DevSecOps and the security of your software.

Reducing friction

Now that we've covered the entire process of creating a secure-by-design software development pipeline, here is some final advice to help you in your implementation process.

Importance of consistency

Throughout this section and the last one, we've highlighted the importance of reducing context switching. Application security is a complex problem, even for people dedicating their entire career to it. To simplify things for non-security collaborators, the implementation of your DevSecOps architecture needs to be as consistent as possible. There are two main ways to accomplish this.

The first way to provide a consistent security experience for your software engineers is to ensure that your scan results are the same at every stage of the SDLC. If you use a specific container image scanner in your Kubernetes admission controller, you need to make sure that your software engineers have access to the same scan on their local workstation and build workflow. They need to be getting the same information in the same format so there are no surprises in the later stages of the software development pipeline.

The other way to drive consistency in your DevSecOps implementation is to consolidate scanning tools where possible. Previously in this document, we discussed how there is no one tool that is the best at scanning for every type of vulnerability. But if you have the option of using the same provider for a few of your scans (e.g. SAST, SCA, and Container) rather than having a different tool for each one, it will reduce management overhead and provide a more consistent experience for your software engineers.

Order of operations

Lastly, you need to know that the implementing order for the various components matters. The first thing you should do is help your software engineers become familiar with the security tooling and vulnerability scans. Take the time to help them set up their IDE plugins, show them how to address false positive findings, set expectations and timelines, configure automated dependency fixes, and gather feedback about all the tools. Once your software engineers are familiar with the tools and findings, you can start raising

warnings in your security gates. Don't just block things right away, just display warnings for the things that would cause blocks in the future. Gather feedback about the warnings. Finally, meet with software engineering leaders and agree upon severity thresholds that will be blocked based on the feedback from the warnings and the perceived workload increase. A good place to start would be PR checks and working on the rest from there.

Secure-by-Design Software

Summary

There are a lot of technologies involved in the implementation of a secure-by-design SDLC. As you evaluate new code security tools, you should be trying to answer these questions to find the right fit for your DevSecOps architecture:

- **How and where can we use this tool as a required security gate?**
- **How can we automate fixes or feedback with this tool to protect our velocity while meeting security requirements?**

If the answer to either of these questions makes it seem like the tool won't work well with your goals, it may be worth checking out alternatives to see if there are any that can do these things.

By leveraging technology and automation, you can guarantee that your digital products meet a minimum level of security without much drag on your ability to deliver. Here is the template for the process diagram from this section so you can use it to track your own progress.

That concludes part 2 of this document on DevSecOps architecture. Part 3 will be the final section. In it, we will be exploring how to protect the security and integrity of the systems involved in your SDLC.

PART 3

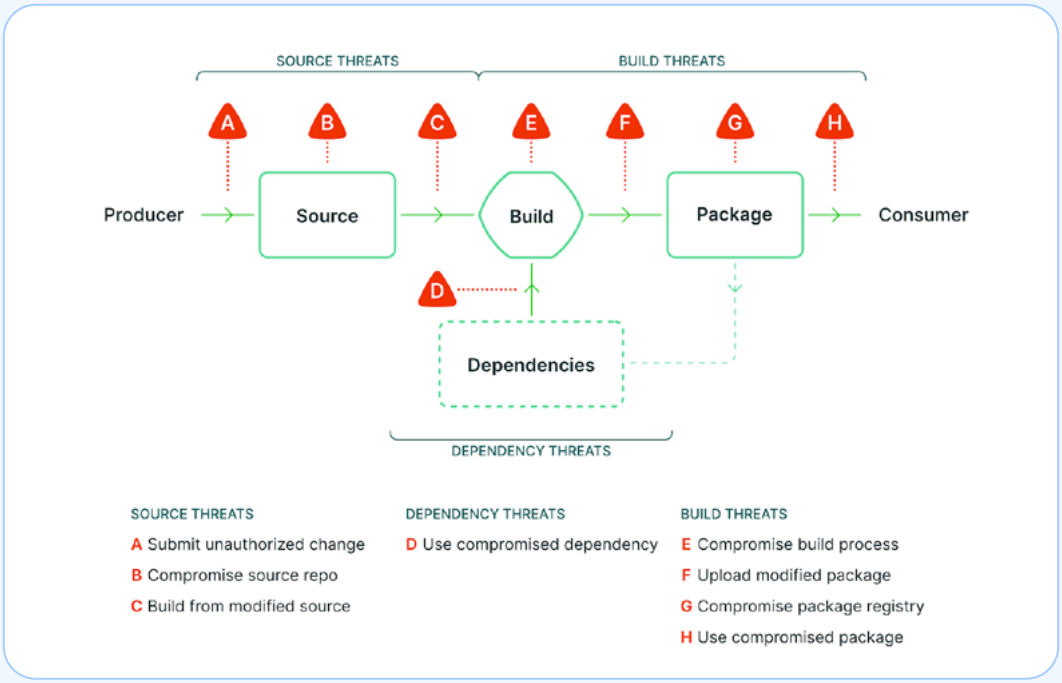
Pipeline Integrity and Security

At this point, we have covered how to manage existing vulnerabilities and *how to prevent the introduction of new vulnerabilities*. We now have a software development lifecycle (SDLC) that produces secure-by-design software. All we have left to cover is how to enforce and protect this architecture that we've built.

In this section, you will learn how to add integrity and security to the systems involved in your SDLC. These processes should be invisible to your software engineers. They should simply exist as guardrails that ensure the rest of your architecture is utilized and not interfered with.

Threat landscape

Whenever we talk about securing something, we need to answer the question, "From what?" Threat modeling is the practice of identifying things that could go wrong based on what you are trying to protect and who/what you are protecting it from. In the context of DevSecOps architecture there are a handful of threats that you should be considering broadly.



▶ *Figure 9: Supply Chain Threats v1.0 (source: OpenSSF)*

The diagram above is a threat model of the software development process from the Open Secure Software Foundation (OpenSSF):

Of the threats listed above, “use compromised dependency” (Threat D) is the most challenging to mitigate. We usually have little control over the external dependencies that we use in our code. The xz utils backdoor was an eye-opening spotlight on the widespread impact that a single compromised dependency can have. Unfortunately, malicious insiders in the open-source ecosystem are currently an unsolved problem.

Instead, we will focus on the things that we can control. For dependency threats, you can look out for malicious look-alike packages and use SCA tools to identify when you are using outdated, vulnerable versions of your dependencies. Ideally, your dependencies also provide ways for you to verify the integrity of the software you are consuming.

The following section will cover how to produce verifiable software for your consumers. We will explore ways to mitigate source threats and build threats through integrity checks. Then, we will examine the assumptions we are making about the integrity checks and discuss how you can use security to build trust in those assumptions.

Pipeline integrity

Signing binaries is the most well-known method of verifying software integrity. In our DevSecOps architecture, however, you need to go beyond verifying individual software artifacts. You need to be able to verify the integrity of your pipeline that produces the software. You might be wondering why you would need to verify the integrity of a pipeline you built from the ground up. It turns out that the assumptions we make about our own SDLC can be wrong.

In software development environments, there are usually ways to skip steps or bypass controls. For one, it’s common for software engineers to be able to publish artifacts like container images directly to your registry. Even innocent intentions can help vulnerabilities slip around security checks that would have otherwise caught them. In a worse scenario, a compromised developer account could allow a threat actor to push backdoored packages directly to your registry.

To verify that your software artifacts are the product of the DevSecOps systems that you have in place, you must improve the integrity of your software development pipeline through the practices outlined below.

BRANCH PROTECTION

One of the most important controls in the software development pipeline is branch protection. Branch protection rules protect against the source threats in the supply chain threat model (Figure 9).

By requiring a Pull Request (PR) to merge code into your production branch, you are ensuring that humans are authorizing changes (Threat A) and verifying that the source code is free from vulnerabilities and backdoors (Threat B). You should also trigger automatic builds when there are changes to the production branch, which will produce builds that come from the source code that has been reviewed (Threat C).

REPRODUCIBLE BUILDS

In the Solarwinds supply chain attack, it was the compromise of Solarwinds' build servers that led to the injection of the Sunburst backdoor. Injecting malicious code late in the software development pipeline is an effective way to reduce the chance that a human will catch the backdoor.

The best mitigation strategy you have against compromised builds (Threat E) is to make your software builds reproducible. Having reproducible builds means that you can run the same steps against the same source code on a different system and end up with the exact same binary result. You would then have a way to verify that a binary was not tampered with. If you rebuilt and the resulting binary was different, it would raise some questions and warrant investigation.

ARTIFACT SIGNING

Signing software has been a common practice for a long time because it provides a way for consumers to verify who the software came from. This protects software consumers even if your package registry is compromised (Threat G).

Unfortunately, this still leaves us with many assumptions because a binary signature doesn't say anything about how the software was built. For example, a software engineer or threat actor might have write access to your container registry and the ability to sign container images. By pushing a locally built container image directly to your registry, they would still be bypassing the automated checks and human reviews that happen in your PRs.

SLSA FRAMEWORK

To verify how and where a piece of software was built, the Open Source Security Foundation (OpenSSF) created the Supply-chain Levels for Software Artifacts (SLSA) framework. In practice, you can utilize the procedures outlined in the SLSA framework as a verification step before deployment. This would mitigate the compromised registry example we just covered because it would detect that the container wasn't built in your CI pipeline.
SLSA ranks your software build process on a 0-3 scale to determine how verifiable it is. A whole article could be written about SLSA, but to keep things short, here is a summary of the 4 levels and what they aim to protect against:

Level 0 - Nothing is done to verify the build process. You don't have any way to verify who built the software artifact or how they built it.

Level 1 – Software artifacts are distributed with a provenance that contains detailed information about the build process. Before using or deploying the software, you can use the information in the provenance to make sure that the components, tools, and steps used in the build are what you expect.

Level 2 – The provenance is generated at build time by a dedicated build platform that also signs the provenance. Adding a signature to the provenance allows you to verify that the documentation came from your build platform and hasn't been forged or tampered with.

Level 3 - The build platform is hardened to prevent the build process from having access to the secret used to sign the provenance. This means that a tampered build process cannot modify the provenance to hide its anomalous characteristics.

At SLSA level 3, you have a way to verify that you aren't falling for build threats E-H in the supply chain threat model (Figure 9). However, you might notice that SLSA is placing some trust in the build platform to be hardened adequately.

Trust in the platforms that make up the SDLC is one of the guiding principles of SLSA. The purpose of SLSA is to verify that your software artifacts came from the expected systems and processes rather than people with write access to your package registries. So, how do you build trust in the systems that produce your software? By securing them.

Pipeline security

DevSecOps architecture is technology-driven, which means there are many different systems involved in the software development pipeline. At this point in the strategy we have covered, the SDLC is producing software that is secure-by-design, and there are ways to verify that the pipeline is being used and not skipping steps. The final piece of this puzzle is preventing the compromise of the systems involved in the DevSecOps platform.

Securing development systems

If a system involved in development or CI gets compromised by a threat actor, they may be able to inject backdoors into your software, steal secrets from your development environment, or even steal data from the downstream consumers of your software by injecting malicious logic. The typical definition of a "production system" is the place where software is deployed, but you need to treat the systems that build your software like they are also production systems.

WORKSTATION SECURITY

On the "left" side of the SDLC, developers are writing code on their workstations. This isn't a guide on enterprise security, but endpoint protection solutions such as antivirus and EDR play an important role in securing these systems. If you are concerned about your source code being leaked or exfiltrated, you might also consider data-loss prevention (DLP) tools or user and entity behavior analytics (UEBA).

REMOTE DEVELOPMENT

If you want to protect your source code even further, you can create remote development environments that developers use to write the code. Development tools like Visual Studio Code and JetBrains IDEs support connecting to external systems for remote development. This is not the same thing as dev containers, which can run on your local host. Remote development refers to connecting your IDE to a completely separate development server that hosts your source code.

This isolation of the software development process separates your source code from high-risk activities like email and browsing the internet. You can combine remote development with a zero-trust networking solution that requires human verification (biometrics, hardware keys, etc) to connect to the remote development environment. If a developer's main device gets compromised, remote development makes it much

harder to steal or tamper with the source code without escalating privileges and gaining access to additional systems.

Remote development adds some friction to the software development process, but this is a very powerful way to protect your source code at the earliest stages of development.

BUILD PLATFORM HARDENING

The SolarWinds supply chain attack that we covered earlier is a prime example of why you need to treat build systems with great scrutiny. Reproducible builds are a way to verify the integrity of your build platform, but you still want to secure these systems to the best of your ability.
Similarly to workstation security, endpoint protection and other enterprise security solutions can help monitor and protect your build platform. You can also take additional steps like limiting administrator access to the build platform and restricting file system permissions.

Securing deployment systems

If your software is deployed as a service for others to use, you need to make sure that you are securing your deployment systems. A compromised service can leak information about your users and provide a threat actor with ways to pivot to other systems.

ZERO-TRUST NETWORKING

A powerful control against the successful exploitation of your applications is restricting outbound network access. Historically, it's been very common for public-facing applications to be in a DMZ, a section of the network that can't initiate outbound network connections to the Internet or any other part of your network (except for maybe a few necessary services). Inbound connections from your users are allowed through, but in the event of a remote code execution exploit, the server is unable to download malware or run a reverse shell.

If you use Kubernetes for your container workloads, you can utilize modern zero-trust networking tools like Cilium to connect your services and disallow everything else. Cilium comes with a UI called Hubble that visualizes your services in a diagram to assist in building and troubleshooting your network policies.

PRIVILEGE DROPPING AND SECCOMP

If you run your services inside Linux containers, you can easily limit their access to various system resources. Seccomp is a Linux kernel feature that works with container runtimes to restrict the syscalls that can be made to the kernel. In other words, it's a mechanism to ensure containers run according to certain rules, and that other interactions are forbidden (filtered). This feature is not enabled by default, but can be progressively leveraged to restrict the capabilities of your containers to the right level. The basic idea behind it is to prevent privilege escalation from the container to the host.

At a minimum, you can use the "RuntimeDefault" seccomp filter in Kubernetes workloads to utilize the default seccomp profile of your container runtime. For example, here you can see the list of syscalls blocked by Docker's default seccomp filter. This default filter is intended to be safe for most applications, but it might block some low-level or observability workloads, in which case a special policy can be implemented.

The best filter, though, is the one customized to your application's specific needs. This ensures containers adhere to the least privileges principle, which could possibly prevent the successful exploitation of a vulnerable system. Here is a tutorial series to get started with building custom seccomp filters in the CI. Beware, however, that this technique can introduce instability into your application if you aren't performing the necessary testing when creating the filter.

CONTAINER DRIFT MONITORING

Another powerful security feature that container deployments enable is container drift monitoring. Many container applications are "stateless," which means that they shouldn't be changing in any way. You can take advantage of this expectation and monitor your stateless containers for any drift from their default state using tools like Falco. When a stateless container starts doing things that it wouldn't normally do, it could indicate that your app has been exploited.

Identity

Lastly, let's look at a few identity-related practices that can meaningfully improve the security of the systems in your software development pipeline.

SECRET MANAGEMENT

There's a lot of complexity in DevSecOps around identity and access management because you are dealing with both human and machine identities at multiple stages

of your SDLC. When your services talk to one another, they need access to credentials that will let them in.

Managing the lifecycle of these credentials is a bigger topic than what we will be covering here, but having a strategy for secret management is one of the most important things you can do for the security of the systems in your SDLC. For detailed advice on this topic, check out GitGuardian's secret management maturity model framework.

LEAKED SECRET PREVENTION

No matter how mature your secret management process is, secrets always seem to find a way into places they shouldn't be. Whether they are in source code, Jira tickets, chats, or anywhere else, it's impossible to prevent all your secrets from ever being exposed. For that reason, it's important to be able to find secrets where they shouldn't be and have a process to rotate leaked secrets so they are no longer valid.

HONEYTOKENS

Leaked secrets are a very sought-after target for threat actors because of their prevalence and impact. You can take advantage of this temptation and intentionally leak special secrets called honeytokens that would never be used except by malicious hackers that are looking for them. By putting honeytokens in convincing locations like source code and Jira tickets, you are setting deceptive traps with high-fidelity alerts that catch even the stealthiest attackers.

Others

We could keep listing more ways to secure systems, but this would be outside the scope of our architecture. Most of the topics covered in this section were included because they are specific to the software development environment. Ultimately, collaboration between product and enterprise security is an important factor in protecting the integrity of your DevSecOps architecture.

Securing these systems builds trust in your software development pipeline, and adding ways to verify its integrity allows us to mitigate many of the supply-chain threats that you and your customers face.

Conclusion

Today's threat actors are increasingly targeting software vulnerabilities to infiltrate supply chains and carry out their attacks. Therefore, it is more important than ever to produce software that is secure by design. DevSecOps aims to include security in the efficiency gains that DevOps has brought to the software engineering world. However, application security is a multifaceted challenge that requires a comprehensive strategy.

An effective DevSecOps program demands a holistic approach that addresses every aspect of the software development lifecycle. This includes the people involved (developers, security teams, operations), the processes followed (secure coding practices, security testing, incident response plans), and the tools and technologies leveraged (code scanners, automated security checks, access controls). Only by considering security at each of these layers can organizations embed security seamlessly into their development workflows and ensure their software products are fundamentally secure.

Adopting DevSecOps is an ongoing journey of continuous improvement. New threats, tools and best practices will constantly emerge, so DevSecOps programs must be agile and adaptable. The program architecture outlined in this whitepaper provides a robust foundation to build upon. By implementing these core pillars, development teams will be well-positioned to create software that is resilient, trustworthy, and able to withstand the cyber threats of today and tomorrow.

GitGuardian

Level up your AppSec program with GitGuardian

Book a Demo

www.gitguardian.com