sysdig SECURE EVERY SECOND.

E-BOOK

Securing the Cloud: A Guide to Effective Vulnerability Management

Table of Contents

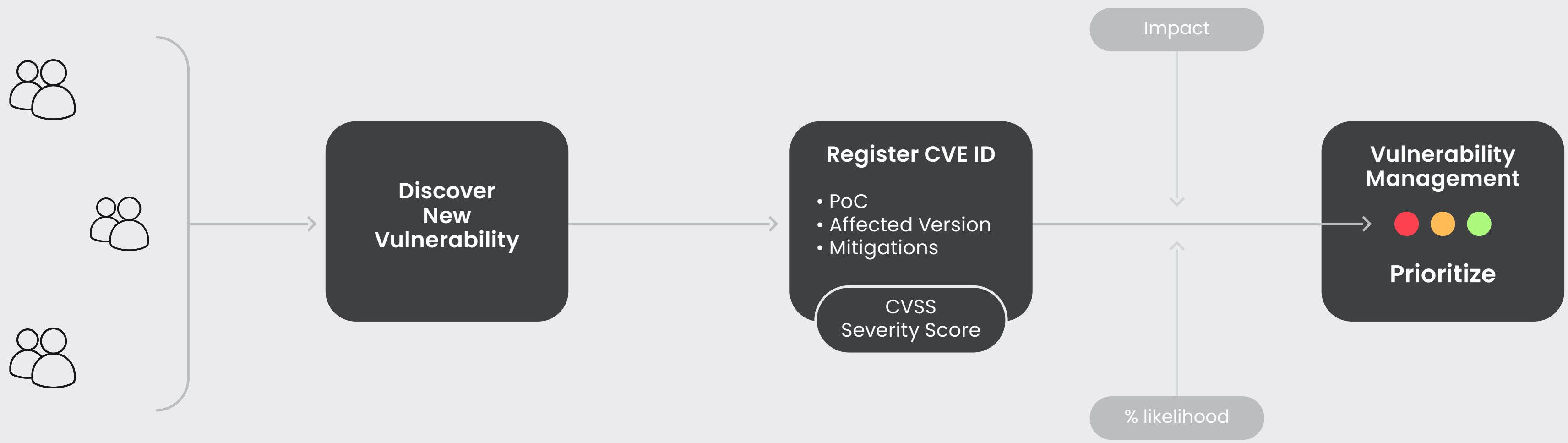
Vulnerability Management Then and Now

The approach to vulnerability management has evolved significantly over time. Previously, people relied on tools like Nessus to scan their data center networks and identify flaws in third-party software like operating systems, web browsers, and network device firmware. The quantity of flaws always exceeded the quantity of time available to fix them, and good approaches to prioritize the effort by actual risk to the organization did not emerge until fairly recently.

Today, the speed of cloud-based attacks has surged as adversaries leverage automation and AI techniques. This escalation has underscored the need to prioritize vulnerabilities based on their actual risk levels.

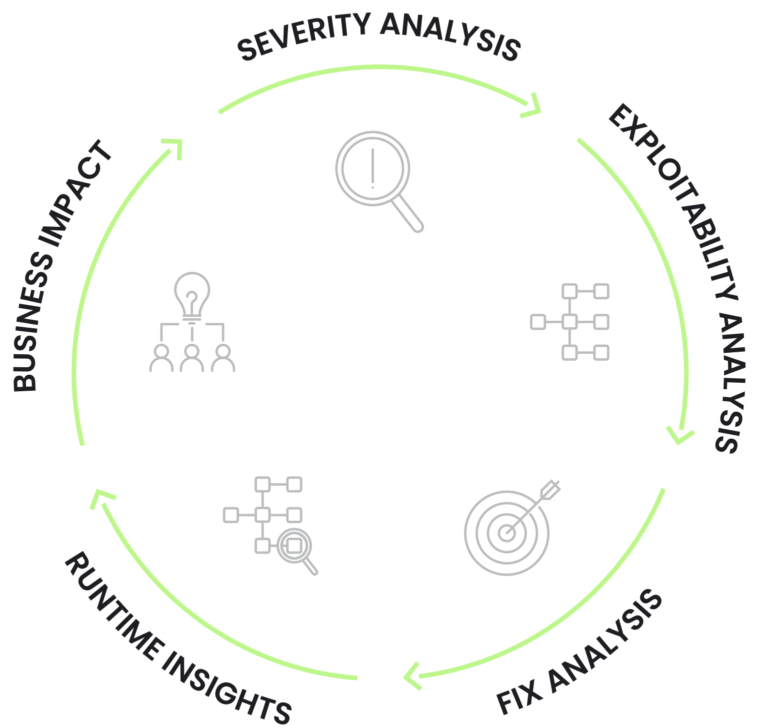
In-house software development is also becoming more prevalent. Businesses differentiate by creating better software, which gives rise to new challenges in vulnerability management. New stakeholders emerge as software development teams and DevSecOps teams enter the picture, and their incentives heavily prioritize speed of delivery. The number of flaws increases while the amount of time available to fix them decreases. Although conventional vulnerability management techniques are still applicable, this e-book primarily focuses on addressing the latest vulnerability management challenges associated with securing cloud-native applications.

Vulnerability Management Process



The framework

While the overarching framework for vulnerability management remains mostly unchanged, there are some notable differences in the modern software development landscape. The basic steps of vulnerability management still include asset identification, scanning, mitigation, remediation, and ongoing verification.



There are, however, several new factors in play when managing vulnerabilities in modern software development:

- **Speed of delivery is a business priority.**

  Speed is a crucial consideration. In the fast-paced world of software development, organizations need to keep up with the latest technologies and push out new products quickly. This means that vulnerability management must not hinder the development process or slow down releases.

- **Full-context risk prioritization is a Non-negotiable requirement.**

  Vulnerability management must be informed by a thorough understanding of the context in which the software operates. This includes factors such as the application's purpose, the types of data it handles, the running state of the application, and the potential impact of exploiting a vulnerability. Security teams must prioritize risk accordingly, and address vulnerabilities that pose the greatest risk to the organization first.

- **Developers are a key stakeholder in the vulnerability management program.**

  Vulnerability management in modern software development requires close collaboration with developers. Developers are the ones creating the software, so remediation often falls squarely on them, and any mitigation requires their input. It is crucial for security and development teams to collaborate in order to find and address vulnerabilities promptly. This relationship does not exist in traditional IT organizational structures.

While the basic framework for vulnerability management has remained largely unchanged over time, the modern software development landscape requires a more nuanced approach. Organizations need to balance speed with security, prioritize risk based on the context in which the software operates, and involve developers in the process without hindering their agility.

Vulnerability management for software that you build

Designing and executing applications at scale in cloud-native environments have introduced some new operating models. Many of the underlying concepts stem from familiar ideas like secure-by-design development strategies and defense in depth. We'll explore some of the unique nuances and challenges in this section.

Shift left

"Shift left" is a term used in the context of vulnerability management to refer to the practice of integrating security considerations to catch vulnerabilities earlier in the software development life cycle (SDLC), rather than waiting until later in the process (usually at runtime) when they can be more difficult and costly to fix. This can help improve the overall security posture of the organization and reduce the potential impact of security breaches.

Traditionally, the focus of vulnerability management has been to identify and remediate vulnerabilities in deployed applications and systems, often as late as in production environments. However, this reactive approach can result in vulnerabilities that are more difficult to remediate and can create significant security risks, especially at scale.

In rare cases, the remediation of a flaw can require significant rework of the application logic, placing a huge and avoidable burden on the engineering teams. By shifting left, organizations can address security issues earlier in the development process, which can help reduce the number and severity of vulnerabilities introduced into production environments.

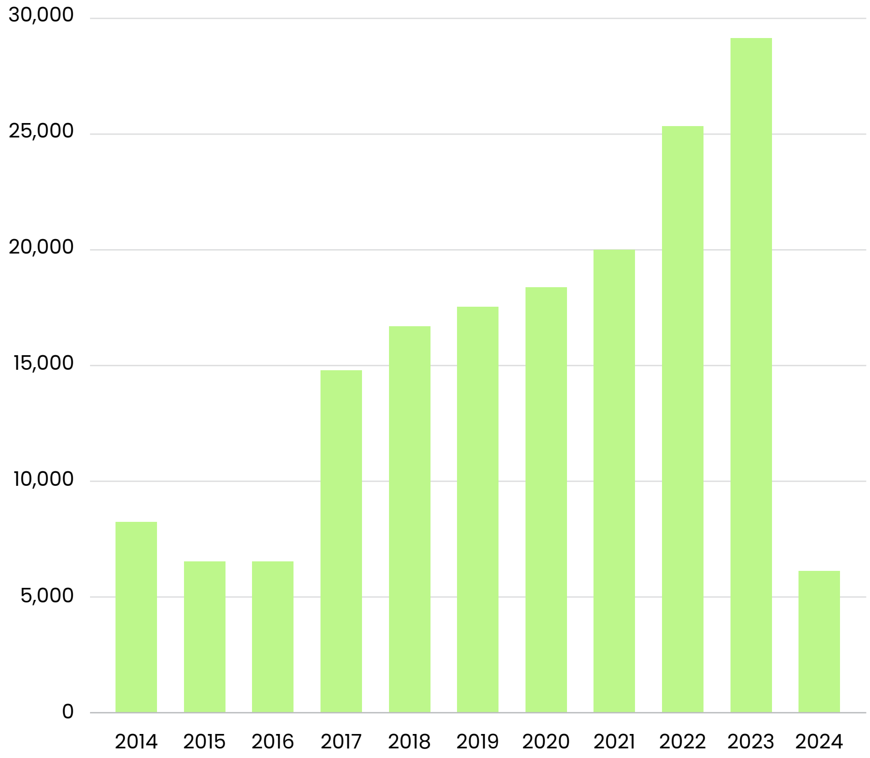
Full life-cycle vulnerability management

Full life-cycle vulnerability management is a crucial aspect of an organization's security posture, and it involves managing vulnerabilities throughout the entire application life cycle. This includes identifying, prioritizing, and remediating vulnerabilities in a timely and efficient manner.

Traditional vulnerability management programs tend to be scoped to servers, applications, network devices, and other data center assets. The program attempts to aggregate the findings from a single scanning tool (like Nessus) into a central dashboard for the security team to review. Remediation effort is handed off to the system owners via traditional workflow systems (like ServiceNow) used in various IT operations teams.

Vulnerability management programs scoped to include cloud-native or any homegrown software will need to use additional tooling, such as static application security testing (SAST), dynamic application security testing (DAST), software composition analysis (SCA), and developer-friendly workflow tools such as Jira.

Number of CVEs by year



It is always important to prioritize vulnerability assessment findings by risk because it's impossible to fix all of the flaws, especially with the discovery of new Common Vulnerabilities and Exposures (CVE) every day. The challenge of knowing where to send those findings and how to ensure that they are actionable is exacerbated by the complexity of modern architectures and the involvement of developers, who typically have very little security training.

Whenever possible, organizations should strive to prioritize vulnerabilities by risk within a scope of remediation accountability, rather than consolidating all findings into a single platform. Each application or asset owner should care about the highest risk vulnerabilities in their system. For example, software developers may not care about vulnerabilities on network switches unless they wrote the firmware.

Therefore, full life-cycle vulnerability management should not be a one-size-fits-all approach, but rather a customized and integrated approach for each software development organization. Integrating vulnerability management tools into the DevSecOps pipeline can help ensure the identification of vulnerabilities early in the development process for remediation before they become larger problems.

Full life-cycle vulnerability management should prioritize risk within a DevSecOps team's scope of control, and seamlessly integrate with DevSecOps processes to ensure a timely and efficient response.

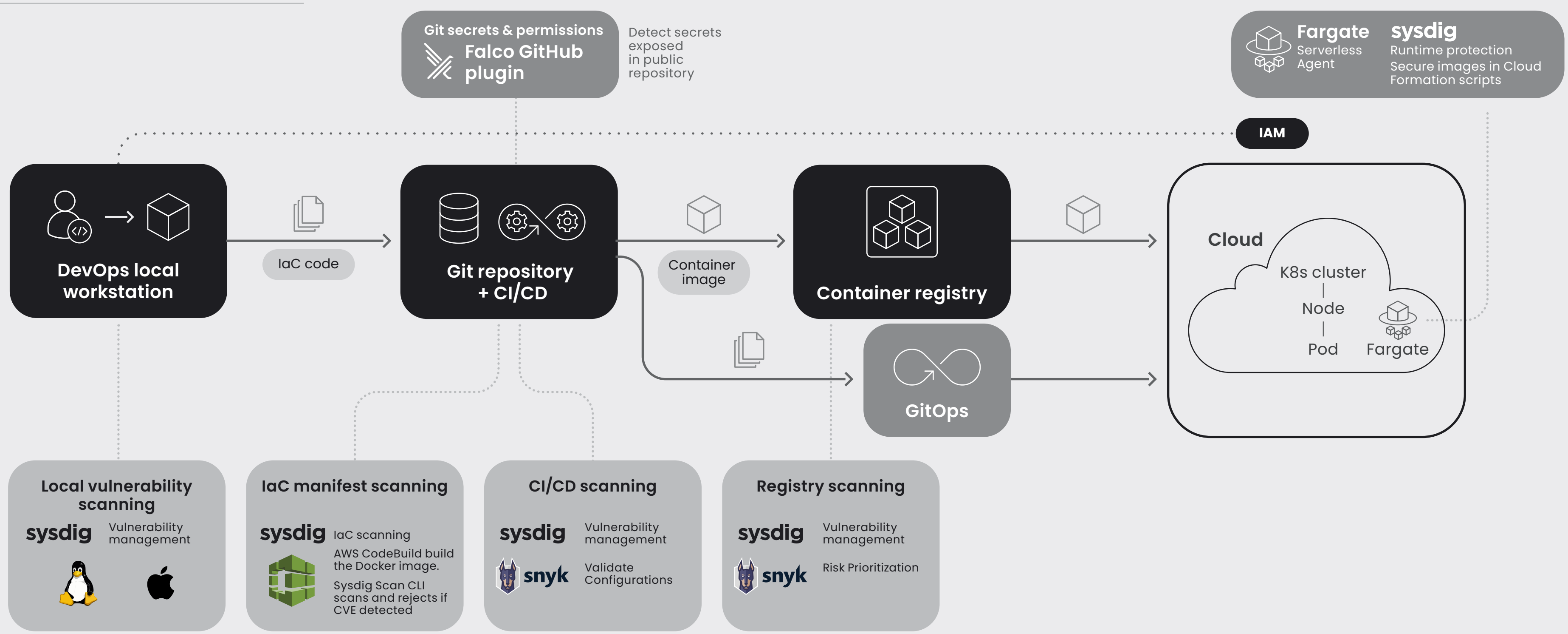
Where are the vulnerabilities?

Across the software development life cycle, there are a variety of areas where vulnerabilities exist. Some of these areas are either nonexistent or not accessible with traditional vulnerability management programs. A modern vulnerability management program must scan every possible source of vulnerability introduced into the software and help identify who is responsible for addressing these flaws.

Application code

The source code itself is where the vulnerability begins its life, and it is entirely in the software developer's scope of responsibility. Training, security champion programs, and the secure-by-design methodology can help developers write more secure code. Application security testing (AST) tools like linters and SAST provide assessment capabilities for the code itself.

Integrating Vulnerability Scanning in the Software Development Lifecycle



Source: CI/CD integrated vulnerability management

Application artifacts

These are vulnerabilities that exist within the artifact, generated after packaging the source code into its deployable form. For example, Java archive (JAR) files, which are commonly used to package Java applications, can introduce vulnerabilities if they contain outdated or vulnerable dependencies. Who owns the vulnerability assessment of this asset depends on the type of artifact in question and how the team is organized. Typically, developers are responsible for ensuring that the artifact they ship to staging environments is defect-free to the extent specified by security policies. However, security teams should use automated tools to scan the application components for vulnerabilities through all stages of testing and staging.

OSS dependencies

Many applications rely on third-party open source software (OSS) components to function. These components can introduce vulnerabilities if they are not properly maintained or updated. Developers are responsible for keeping track of these dependencies and ensuring that they are up to date and free from vulnerabilities. Security teams should scan for vulnerable OSS components throughout the continuous integration/continuous delivery (CI/CD) pipeline. Security and/or DevOps teams sometimes curate and maintain a registry of pre-approved OSS and commercial components for developers to use, disallowing the inclusion of any other packages.

Included libraries

Similar to OSS dependencies, without proper maintenance and updating, included libraries can also introduce vulnerabilities. Developers are responsible for keeping track of the libraries they use and ensuring that they are up to date and free from vulnerabilities. Security teams should scan for vulnerable libraries throughout the CI/CD pipeline.

Container images

Container images are a special type of application delivery vehicle that include everything an application needs to function in a single package. They allow for easy and rapid portability between platforms and alleviate "works on my machine" dependency problems. Vulnerabilities can exist within multiple layers of container images. Developers are responsible for ensuring that the image layers containing application artifacts and enabling architecture components are up to date and free from vulnerabilities before deployment.

However, IT operations, DevOps, or security teams may be responsible for maintaining the base image layers, which include the operating system. Security teams should also use dedicated container scanning tools to identify vulnerabilities in container images throughout the CI/CD pipeline. Most traditional vulnerability management tools do not provide any visibility into container images or running containers.

Hosts/nodes

The underlying infrastructure that supports cloud-native applications, such as Linux servers or infrastructure-as-a-service (IaaS) platforms, can also introduce vulnerabilities if security teams are not properly securing or maintaining these assets.

An improperly secured infrastructure supporting cloud-native applications, such as Linux servers or IaaS platforms, can introduce vulnerabilities. Breached containers usually have limited impact, but a breached host can be a huge security risk. Although container hosts are disposable, they remain a risk. Immutable operating systems like Container Linux (formerly CoreOS), RancherOS, Red Hat Enterprise Linux Atomic Host, and Ubuntu Core are image-based, safer, and more focused on running containers. Operations teams are typically responsible for securing the underlying infrastructure. Legacy tools may not be able to scan special operating systems for running containers.

Orchestration systems (Kubernetes)

In traditional monolithic architectures, each application typically runs on its own dedicated infrastructure with its own set of permissions. This means that if one application has a vulnerability, it is less likely to affect other applications running on the same infrastructure. In a Kubernetes environment, however, multiple applications may run on the same infrastructure, and a compromised plug-in can potentially affect multiple applications at once.

Kubectl plug-ins are a gateway for vulnerabilities

Kubectl plug-ins are external binaries that users can invoke through the kubectl command-line tool to extend its functionality. When running a kubectl plug-in, it runs with the same permissions as the kubectl command itself. If you run the kubectl command with root privileges, the plug-in will also have root privileges.

Having the same permissions can be a unique problem for the Kubernetes orchestration platform because Kubernetes is designed to run multiple containers on a single host, each with its own set of permissions. A compromised kubectl plug-in can potentially access or modify any resource within the cluster, including other containers or nodes. Who is actively maintaining these plug-ins? If there's a CVE in one of these plug-ins, how do you handle unpatchable vulnerabilities?

To address this challenge, it is important to carefully manage the permissions granted to kubectl and its plug-ins. This may involve limiting the use of plug-ins to trusted sources or auditing the code of plug-ins before using them in production environments. Additionally, vulnerability scanning in Kubernetes environments should be a continuous process, with regular scans to identify and mitigate any vulnerabilities discovered.

Unpatchable vulnerabilities

The issue with unpatchable vulnerabilities in Kubernetes, such as CVE-2020-8554, is that they represent a potential security risk that traditional patching or remediation methods cannot fully eliminate. These vulnerabilities are often the result of fundamental design flaws or limitations in the technology itself.

In cases where there are known, unpatchable vulnerabilities in Kubernetes, there are a few steps that organizations can take to protect themselves. One option is to use a tool like Open Policy Agent (OPA) to enforce strict policies around the use of Kubernetes resources and configurations. By setting up policies that restrict certain actions or configurations that could lead to exploitation of the vulnerability, organizations can reduce the risk of an attack.

Alternatively, you can monitor the environment for signs of exploitation or compromise using a tool like Falco. This can involve setting up alerts and monitoring systems to detect any unusual activity or changes in the Kubernetes cluster, as well as conducting regular security audits to identify potential vulnerabilities.

Unfortunately, the best approach will depend on the specific vulnerability and the risk it poses to the organization. In some cases, it may be possible to mitigate risk through policy enforcement and monitoring, while in other cases, it may be necessary to take more drastic measures, such as limiting or disabling certain features or functions within Kubernetes. It is important to work closely with security experts and the Kubernetes community to stay up to date on the latest vulnerabilities and mitigation strategies.

Shared responsibility model of IaaS offerings

The shared responsibility model in the cloud refers to the fact that cloud providers are responsible for the security of their underlying infrastructure, while customers are responsible for the security of their own applications and data that resides on the cloud infrastructure.

While this model can be beneficial in many ways, it can also lead to some challenges when it comes to vulnerability detection and mitigation. One example is the case of managed Kubernetes services, like Google Kubernetes Engine, Amazon Elastic Kubernetes Service, and Azure Kubernetes Service. While these services provide a convenient and scalable way to deploy and manage Kubernetes clusters, they also limit your control over the underlying infrastructure. This can make it challenging to detect and mitigate vulnerabilities that may exist in the infrastructure or in the Kubernetes cluster itself.

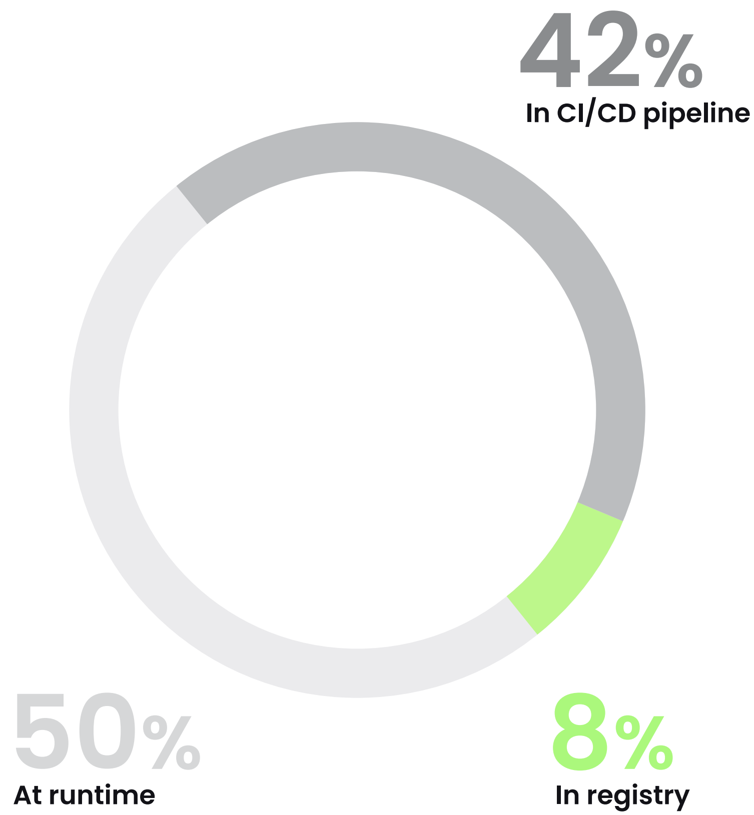
One approach to addressing this challenge is to use third-party vulnerability scanning tools specifically designed to scan Kubernetes clusters. These tools can help identify vulnerabilities in both the infrastructure and the applications running on the cluster, and can provide recommendations for how to mitigate those vulnerabilities.

We recommend working closely with the cloud provider to ensure that they are meeting their responsibilities for securing the underlying infrastructure. For example, you can ask them to perform regular security audits or penetration testing on the infrastructure to ensure that it is secure. In reality, the cloud provider can do a "best effort" at updating vulnerable components, but likely cannot make radical changes for all customers based on individual feature requests.

Third-party architecture components (middleware)

Cloud-native applications often rely on third-party OSS or commercial architecture components, which are essentially modern middleware and include things like load balancers and databases. Without proper maintenance and updating, these components can also introduce vulnerabilities. Developers are responsible for keeping track of these components and ensuring that they are up to date and free from vulnerabilities. Because they are not built in-house, these pieces of software often bypass CI/CD security controls (and thus the pipeline altogether) and may not be assessed until runtime. According to the "Sysdig 2023 Cloud-native Security and Usage Report," only 42% of vulnerability scans are performed at the CI/CD pipeline phase. Ideally, security teams will scan for vulnerable architecture components throughout the CI/CD pipeline.

Where images are scanned



Identifying and fixing vulnerabilities requires a collaborative effort between different teams, including developers, security professionals, and operations teams. By staying vigilant and regularly scanning for vulnerabilities, organizations can help ensure the consistent security of their cloud-native applications.

When are the vulnerabilities discovered?

Based on the above information, when is it appropriate to scan for vulnerabilities in cloud-native environments? The short answer is through every stage of the CI/CD pipeline, and then constantly in production to account for any disclosures that occur after deployment.

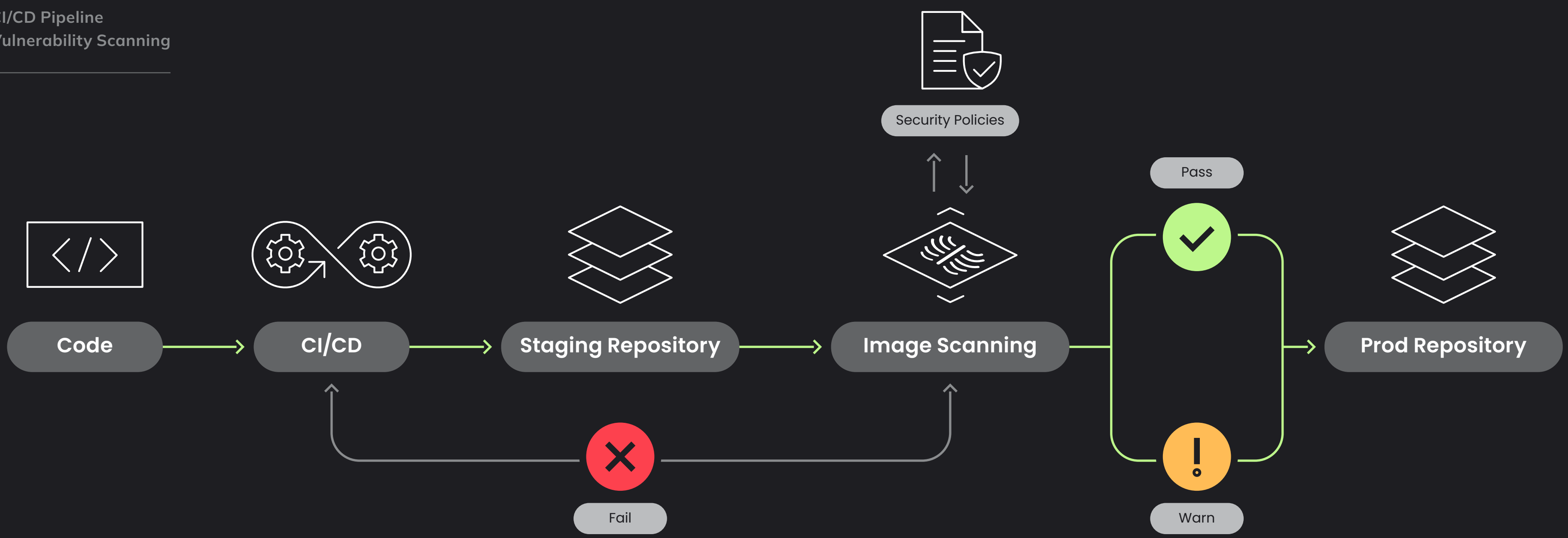
On developer machine before merge

One best practice is to scan for vulnerabilities on the developer's machine before merging the code with the main branch. This can help catch vulnerabilities early in the development process, when they are typically easier and cheaper to fix. Developers can use tools such as dependency checkers and static analysis tools to scan their code for vulnerabilities.

In the CI/CD pipeline

Another best practice is to incorporate vulnerability scanning into multiple stages of the CI/CD pipeline. This ensures the scanning of code for vulnerabilities each time it is built and deployed. Vulnerability scanning can introduce a high degree of automation to vulnerability discovery, which is good for supporting delivery speeds. However, the tailoring of pass/fail gates can pose difficulties and requires careful consideration in the context of each organization's specific risk appetite.

CI/CD Pipeline
Vulnerability Scanning



Registries

Cloud-native applications are often deployed using container images, which are stored in registries. Noncontainerized artifacts also live in registries. Most organizations have many different types of registries. Before deploying an artifact, it's important to scan it for vulnerabilities. Registry scans can occur multiple times in the life cycle, but it's absolutely critical to ensure that the production registry is clean.

Runtime

While it's best to catch vulnerabilities as early as possible, it's also important to scan for vulnerabilities at runtime. This can help identify vulnerabilities missed during earlier stages of the development process, vulnerabilities introduced during runtime, or vulnerabilities disclosed after the last scan occurred. Regular runtime scanning can ensure the identification and prompt addressing of any vulnerabilities. It's worth noting that "runtime" can mean staging or production, and ideally, the teams have full scanning capability in both types of runtime environments.

We cannot stress enough the importance of incorporating vulnerability scanning into the various stages of the development process. This will ultimately help ensure the security of your cloud-native environments. By catching vulnerabilities early and regularly scanning for them, organizations can help minimize the risk of cyberattacks and protect their sensitive data.

Prioritization of vulnerability assessment findings

Arguably, one of the hardest aspects of vulnerability management is prioritization, as not all vulnerabilities have the same level of risk, and you must allocate resources efficiently to address the most critical ones first. This guide provides a comprehensive framework for categorizing vulnerabilities based on their potential impact on your organization's assets, services, and operations. By following these guidelines, you can prioritize your vulnerability management efforts and ensure that your teams focus on the most critical issues.

Risk contextualization is an approach taking into account the specific context and environment in which vulnerabilities exist. It involves analyzing the severity of the vulnerability, the assets at risk, the potential attackers and their motivations, the available mitigation options, and more.

To highlight the importance of risk contextualization in prioritizing targeted controls to mitigate workload and cloud-based threats, we use a two-part framework to think about vulnerability risk:

- **Vulnerability threat context** comes from the threat intelligence surrounding the vulnerability itself, regardless of which assets, if any, it is affecting. It includes information like whether an exploit is publicly available or likely to become available soon, whether the exploit is easy to execute, and whether the exploitation of this flaw is popular among attackers today.
  Threat context data usually comes from third-party sources, and could be in machine-readable form or in the form of a report. Many vulnerability assessment tools integrate threat intelligence sources to help prioritize the scan results within their interface. Security teams can manually gather additional threat contexts from public and commercial sources like Exploit Database, Metasploit Framework, and others.

- **Affected asset business context** considers the importance of the assets within your organization that can be affected by any given vulnerability, and the potential impact to your business should that asset be abused. Additionally, it includes the details of your environment, such as the asset's accessibility from the internet or what mitigating controls may be in place to protect it.

Risk contextualization enables organizations to prioritize vulnerabilities based on their potential impact on the business and the likelihood of successful exploitation in the specific context of their environment. This approach helps optimize resource allocation, reduce risk exposure, and enhance the overall security posture of the organization.

There are dedicated frameworks such as the Stakeholder-Specific Vulnerability Categorization (SSVC), a vulnerability management methodology that assesses vulnerabilities and prioritizes remediation efforts based on exploitation status, impacts to safety, and prevalence of the affected product in a singular system.

There are also frameworks for calculating a risk score like the Exploit Prediction Scoring System (EPSS). EPSS is a scoring model that predicts the likelihood of a vulnerability being exploited. In general, commercial vulnerability assessment tools have risk-based prioritization built in and tied to an actionable remediation workflow.

Runtime risk intelligence for cloud-native applications

The methodology above applies to any vulnerability management program, but when handling homegrown software, there are some additional considerations. Building software packages with only necessary dependencies is important for reducing complexity and minimizing the risk of vulnerabilities. However, even with careful selection of dependencies, it's possible that some unnecessary ones may slip through the cracks. Operating system packages, in particular, include a lot of unnecessary components that may contain vulnerabilities but that your application never uses. In fact, the "Sysdig 2023 Cloud-native Security and Usage Report" stated that operating system packages contained 37% of vulnerabilities. This can lead to bloat in the package and make it difficult to identify which dependencies you are actually using.

A recent innovation in cloud-native risk prioritization is runtime insights, which provides visibility into which dependencies your application is actually using in runtime and prioritizes them based on their real risk. This allows your development teams to focus their efforts on addressing the most critical vulnerabilities and reducing the impact of cyberattacks on your business. By eliminating unnecessary dependencies and prioritizing those actually in use, you can streamline your software package and improve the security and stability of your applications.

Remediation

At some point, you will need to either fix or mitigate vulnerabilities in your organization's systems and applications. Effective remediation requires identifying and prioritizing vulnerabilities based on their severity and impact, as discussed in the previous section. We already mentioned that the teams responsible for performing the remediation can vary depending on which environments, applications, or even individual components the vulnerability affects.

Asset inventory

The first step to enabling a smooth workflow is maintaining an accurate asset inventory, with an owner assigned to every entry. Because cloud-native applications tend to be highly distributed and complex, keeping track of them isn't easy.

In the context of ephemeral workloads, tagging can help organizations keep track of their assets in real time. Ephemeral workloads create and destroy assets constantly, which can make it difficult to maintain an accurate inventory. By using tags, you can quickly identify which assets are currently in use and which ones have been decommissioned, as well as who the application owners are and what constraints or requirements may be associated with the workload.

You can also use tagging to enforce compliance checks. You could assign specific tags to assets that must comply with strict security requirements, such as access controls or encryption. In the case of vulnerable images, we can state that an image do not comply with those rigorous compliance standards, and therefore cannot be used in Federal Risk and Authorization Management Program environments.

One of the most important capabilities of the modern cloud-native application protection platform (CNAPP) is integrating different risk indicators into the asset inventory module. For example, with a very simple query, a user can list all resources vulnerable to a specific CVE, exposed to the internet, and have the vulnerable library in use.

Remediation responsibility

Who is actually capable of performing the remediation has never been simple to answer, but the nature of cloud-native applications exacerbates the problem further. For example, different layers of a container image can be the responsibility of different people or teams. An IT ops or DevOps team might own the operating system layers, but a development team might own the application layers.

Similarly, developers may own fully custom-built components of an application, while another team may own third-party components and middleware. It's critical that the security function that performs the scanning hands off remediation responsibility to the right people and in a rigorously risk-prioritized way, with the understanding that DevOps teams do not usually have the security expertise to properly prioritize remediation work.

Metrics and incentives

A key aspect of secure vulnerability management is ensuring that remediation efforts are timely and effective in addressing vulnerabilities that attackers could exploit. However, different technical teams involved in this process may have competing metrics and incentives. Businesses reward developers for shipping a lot of features as quickly as possible because that ultimately creates value.

Security, on the other hand, must protect the business from harm, so each flaw that slips into production is an additional source of risk. As such, modern organizations must leverage automation, risk-based prioritization, and new tools like admission control to optimize the release of safe software quickly. Ultimately, a culture shift is required, and security and engineering leaders alike must incentivize their teams to work together to avoid the adversarial relationships that plagued traditional vulnerability management.

Mitigation

The recent 5/5/5 benchmark report shed light on the time it takes attackers to cause harm once they exploit vulnerabilities. In some cases, the time to respond, whether by remediation or mitigation, should be less than 5 minutes. That's why vulnerability mitigation is an important step in the vulnerability management process, where identified vulnerabilities are temporarily addressed to minimize their impact while awaiting a more permanent solution. In some cases, remediation of the vulnerabilities may require significant disruption to a system or process; therefore, mitigation is a temporary holdover until it is possible to properly execute the remediation.

In an ideal world, this would mean **fixing it**: updating the package or library to a version that contains the fix for the aforementioned vulnerability in the source code (patch) and then triggering a new deployment for that particular piece of software, which could include building the artifact, performing the tests required, etc., and finally deploying it into the production environment.

A significant hurdle to overcome with respect to remediation is how to quickly patch every single potentially impacted or exploitable asset or dependency; these processes also need to be able to scale. It's not trivial to patch old versions that could impact new features or poor performance. Transitive dependencies exacerbate the problem; in other words, your code or system likely relies on many other codebases or systems, and dependency chains become quite nested in practice.

Including as much context as possible for developers to fix the vulnerability is important to avoid wasting time on figuring out who is responsible or even how to fix it.

Unfortunately, it is not always possible to fix a vulnerability. Maybe the fix is not available yet, requiring more time to test that it doesn't break anything else, or any other million reasons.

There are a few approaches if that's the case:

- **Downtime**. If the potential breach is big enough, perhaps the affected application shouldn't be running. When the Log4j vulnerability was found, Quebec shut down 3,992 websites as a "preventive measure."
  – If the vulnerability is focused on a specific feature, it may be possible to disable just that particular feature using feature flags.
- **Hardening**. You can try to prevent the exploitation of the vulnerability by making some changes to the application or the system if possible.
  – If the vulnerability can be network-exploitable, consider enforcing the security groups or Kubernetes network policies. For example, here is a network policy to block all of the egress traffic for a specific namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
Metadata:
  name: default-deny-egress
  namespace: default
Spec:
  podSelector: {}
  policyTypes:
  - Egress
```

- **Monitoring**. Monitor the system for suspicious activity and respond when necessary.

This last step overlaps with a whole security category called threat detection and response.

By using a runtime detection engine tool like Falco, you can detect attacks that occur in runtime.

Let's assume that the attacker exploits a vulnerability and wants to open a reverse shell on a pod. In this case, the Falco runtime policies in place will detect the malicious behavior and raise a security alert.

Validation

One process associated with remediation is confirming the successful remediation of a vulnerability. To do this, you will need to verify that the remediation efforts were successful such that the vulnerability is no longer present, and that the organization's systems and applications are secure.

Validation can occur through various means, such as manual testing, automated scanning, or third-party assessments. Most tools' reporting capabilities allow you to run a before-and-after report to verify that a CVE is no longer present in your environment.

Risk acceptance

Sometimes, it is not possible to fix a vulnerability, either because there may not be a fix available yet, it will take some time because of organizational processes, or simply because it is a false positive.

In those scenarios, it can be helpful to make a conscious decision to filter out (accept) those vulnerabilities so that they don't affect the report results. But it is important to have a clear understanding of the consequences, and to put in place some methods (such as reminders) to review whether you can remove the filter.

Reporting

Reporting the results of a vulnerability assessment is a key part of this process, and it will take on a different form depending on the target audience. All scanning tools should be able to create some kind of report, but their content and structure are often inconsistent.

One consideration is that reporting to senior leadership should be far less granular than reporting to technical teams. Executive reports must include high-level program metrics, preferably showing trends over time. Technical reports should be structured in a way that is most useful to the team receiving the information. Effective reporting requires clear and concise communication that highlights any problems within the vulnerability program, not just a list of identified CVEs.

The nuances of cloud-native architectures

Cloud-native architectures have several additional nuances when it comes to vulnerability scanning. These unique considerations stem from the nature of modern architectures and the increasing degree of automation that IT requires today. Here are some of the characteristics of cloud-native architecture:

- **A high degree of environmental complexity.**

  Cloud-native architectures rely on microservices, which are small, independent components that perform a specific function. This can make it more difficult to scan for vulnerabilities because the components may be spread across multiple hosts or data centers and may have different security requirements.

- **The ephemeral nature of infrastructures.**

  Cloud-native architectures are designed to be ephemeral, meaning that the clusters, hosts, pods, and containers are intended to be created and destroyed regularly and frequently. But the dynamic provisioning and deprovisioning of infrastructure components in response to system or customer demand can pose challenges for vulnerability scanning because the components may not exist long enough to be scanned. The IP address and container ID will change, and therefore will not be easily trackable.

- **Immutable components.**

  Cloud-native infrastructure and application components should be immutable. This means that changes to the infrastructure should be made by creating new instances rather than modifying existing ones. This approach can make it more difficult to scan for vulnerabilities because the infrastructure components are constantly being replaced.

- **Massive scalability.**

  Cloud-native architectures are designed to be elastic and scalable, which means that they can easily be expanded or contracted as needed. This can pose challenges for vulnerability scanning because the infrastructure components may be spread across multiple hosts or data centers.

- **Workflow automation.**

  Cloud-native architectures are designed for automation to manage infrastructure components quickly and scalably. This automation can make it more difficult to perform vulnerability scanning because it may be necessary to integrate the scanning process into the automation tools.

This section explored some additional details that can either impede or bolster your vulnerability management program. Not every program can address every nuance from the beginning, but as your organization increases its DevSecOps maturity, these elements will certainly become relevant.

Vulnerability management deployment methods

In cloud-native environments, there are two different ways to implement vulnerability management solutions: agentless-based deployment or agent-based deployment. Each method has its pros and cons.

Agentless scanning

Agentless vulnerability management solutions leverage existing cloud providers' APIs to discover and scan resources very fast. They create a snapshot of existing volumes and then run the vulnerability management analysis against the snapshot. In general, agentless scanning is fast, easy to onboard, requires less maintenance, and is less disruptive to the running workload.

However, in some cases, using agentless scanning alone comes with certain limitations in system visibility. For example, it does not provide insights into whether vulnerable packages are already in use and loaded into the system memory. Additionally, agentless scanning lacks real-time visibility, potentially causing it to miss information about intermediate states of the system between scans.

Agent-based scanning

On the other hand, deploying software agents on cloud-computing workloads allows organizations to gain more comprehensive insights into processes, users, file activity, network connections, and other system-specific details. This enables more effective cloud threat detection and response capabilities, including advanced techniques such as behavioral analysis and machine learning algorithms.

If incorporating both agentless and agent-based approaches, the initial deployment will be much easier via agentless, while using the agent will give you deep visibility and stronger security as you progress in your cloud security journey. This flexible and comprehensive approach allows for greater protection and enables organizations to effectively adapt to evolving security challenges as they mature their cloud presence.

Integrating into the development life cycle

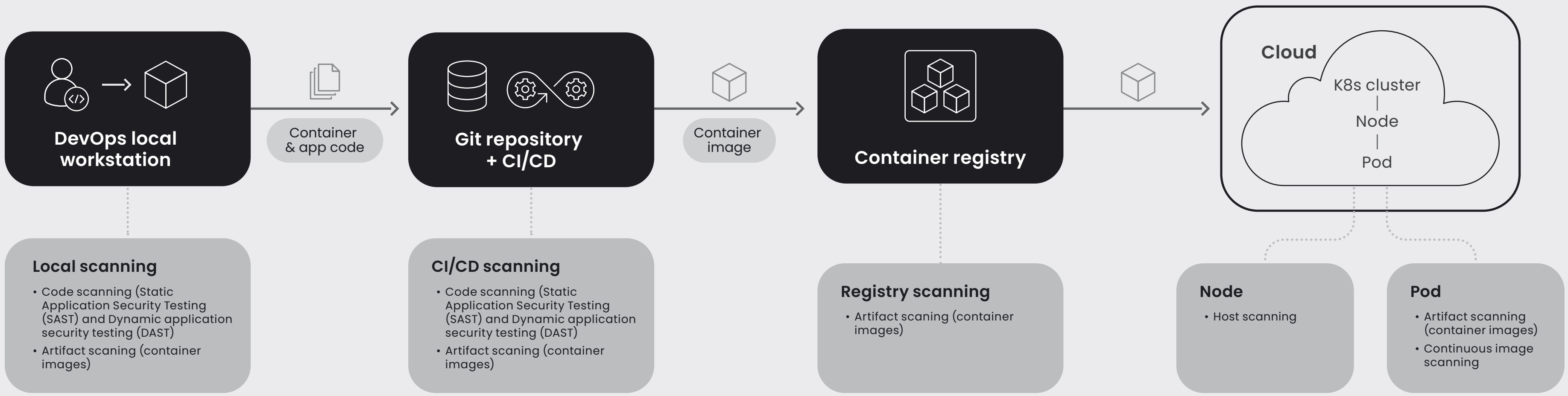
Scanning for vulnerabilities is a best practice and a must-have step in the application life cycle to prevent security attacks. It is also important where you perform this step, but why?

Application life cycles involve a number of steps, from the developer workstation creating fine art in the shape of lines of code to the final production environment where customers use a web application, mobile application, or anything else. Vulnerabilities can occur in any of these steps, so we highly recommend establishing some barriers to prevent them from ruining your environment.

The "defense in depth" concept recommends performing automatic vulnerability scanning on different steps of the application life cycle – sometimes even overlapping them – which will reduce the number of vulnerabilities introduced into your production environment.

In today's ever-evolving threat landscape, it's becoming increasingly clear that vulnerability scanning before production is not sufficient. That is because new vulnerabilities are constantly being disclosed, and unexpected behavior can occur during runtime. As such, vulnerability scanning processes should

Securing the Software Development Lifecycle



adopt a complete or comprehensive vulnerability assessment approach that emphasizes the importance of thoroughly assessing vulnerabilities without any preconceived trust assumptions.

What if someone bypassed the CI/CD and pushed the container image directly? What about those images scanned weeks ago? Were new vulnerabilities discovered since that last scan?

It is also important to note that fixing vulnerabilities earlier in the software development process is easier and that every step of the process makes things more complicated. That's why shifting left improves the overall security posture of the organization and reduces the potential impact of security breaches.

CI/CD

Let's assume that you already fixed all of your vulnerabilities by updating the libraries' dependencies and submitted the pull request. The next step in the build chain is usually to run a CI/CD pipeline to build the application, build the container image, run some tests, and check for vulnerabilities again. But why again?

- Who can guarantee that the developers performed the vulnerability scan religiously, locally at their workstations, before submitting the pull request?
- What if the developer performed the scan a couple of days ago, but there is now a new vulnerability?

CI/CD pipelines are basically made up of different steps. A very basic example can be something like:

- Check out the code.
- Run some linting to make sure the build won't fail.
- Build the artifact.
- Perform some unit tests.
- Deploy the artifact.

But these steps can also be complex; it depends on the requirements of the application or the environment itself. Fortunately, there can be as many steps as you need – even multiple steps happening at the same time. Adding as many security checks as possible in a CI/CD pipeline is a good idea.

Let's see a more complex example. This time, the application is packaged as a container image:

- Check out the code.
- Run some linting to make sure the build won't fail.
- Check for vulnerabilities on the dependencies.
- Check for misconfigurations or secrets.
- Perform a static code scan.
- Build the application and container image.
- Check for vulnerabilities on the container image.
- Perform some unit tests.
- Deploy the artifact.

Why check for vulnerabilities on the container image again? The answer is simple: What if the image used as a base already has vulnerabilities?

Ultimately, the goal is to deploy workloads to the production environment that are as clean as possible and within the company's security and compliance standards. CI/CD enables a lot of automation of the security assessment and remediation of any software your organization builds, at speeds that were previously unimaginable.

Binaries, packages, and language intricacies

Package managers are the common approach when installing software in container images or host operating systems. Tools like apt or dnf make it really easy to install, update, and manage the software. Most of those tools use a database to store the metadata about the packages, such as when it was installed, the dependencies, the versions, the files included in a package, etc.

Using the package version, it is trivial to check against the vulnerability databases to see if it is vulnerable or not. In the following example, let's manually check whether the curl version in a Debian system is affected by any vulnerability in the Debian vulnerability database:

Information on source package curl

debian

curl in the Package Tracking System | curl in the Bug Tracking System | curl source code | curl in the testing migration checker

Available versions

Release	Version
buster	7.64.0-4+deb10u2
buster (security)	7.64.0-4+deb10u5
bullseye	7.74.0-1.3+deb11u3
bullseye (security)	7.74.0-1.3+deb11u7
bookworm	7.88.1-1
sid	7.88.1-5

Open issues

Bug	buster	bullseye	bookworm	sid	Description
CVE-2023-23915	fixed	vulnerable (no DSA, ignored)	fixed	fixed	A cleartext transmission of sensitive information vulnerability exists ...
CVE-2023-23914	fixed	vulnerable (no DSA, ignored)	fixed	fixed	A cleartext transmission of sensitive information vulnerability exists ...
CVE-2022-43551	fixed	vulnerable (no DSA, ignored)	fixed	fixed	A vulnerability exists in curl <7.87.0 HSTS check that could be byp ...
CVE-2022-42916	fixed	vulnerable (no DSA, ignored)	fixed	fixed	In curl before 7.86.0, the HSTS check could be bypassed to trick it in ...

Open unimportant issues

Bug	buster	bullseye	bookworm	sid	Description
CVE-2021-22923	vulnerable	vulnerable	fixed	fixed	When curl is instructed to get content using the metalink feature, and ...
CVE-2021-22922	vulnerable	vulnerable	fixed	fixed	When curl is instructed to download content using the metalink feature ...

Resolved issues

Bug	Description
CVE-2023-23916	An allocation of resources without limits or throttling vulnerability ...

Source: Debian.org

But what about binaries that are not installed using package managers? Golang binaries are the classic example of applications that are just a single binary copied into a container image, like this:

```
## Build
FROM golang:1.16-buster AS build
WORKDIR /app
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
RUN go build -o /helloworld
```

```
## Deploy
FROM gcr.io/distroless/base-debian10
WORKDIR /
COPY --from=build /helloworld /helloworld
USER nonroot:nonroot
ENTRYPOINT ["/helloworld"]
```

There are no package managers involved; instead, a helloworld binary has been compiled from source code and then copied into the final container image. How can you check to see if it included any vulnerable dependency?

In this particular example of using go, the go build command embeds some information about the dependencies (debugging information) into the binary itself by default (it can be disabled) so that they can be extracted (using https://pkg.go.dev/debug/buildinfo, for example) and then performs the matching against a vulnerability database. For golang, that would be the Go Vulnerability Database.

Some programming languages don't include that information; they expect you to know the dependencies they are using implicitly (the package.json file for Node.js, for example).

The best approach to solve this problem would be for every piece of software included in the container image to have its own software bill of materials (SBOM) so that the vulnerability scanning tools could easily extract them and compare them against the vulnerability databases.

Unfortunately, the SBOM is frequently unavailable or incomplete. In such cases, you must rely on dependency mapping tools (such as Checkov) and ensure that your teams understand the particular behaviors of the languages and package managers that they use.

SBOMs

SBOMs are useful for vulnerability management because they provide an inventory of software components used in an organization's systems and applications, enabling the identification of vulnerabilities and the prioritization of remediation efforts. SBOMs include a detailed list of software components, including version numbers and dependencies, which can be integrated with vulnerability scanning tools to automatically identify vulnerable software components.

By having a complete inventory of software components, security teams can more accurately assess the risk of software vulnerabilities and identify potential vulnerabilities before they are exploited. Ideally, every piece of software should generate and publish its own SBOMs, but tools like syft can generate them from file systems or container images.

Creating and maintaining an accurate SBOM can be challenging in complex software environments with many dependencies. Some components may be buried deep within the software stack or may exist in multiple places, making it difficult to identify and track them all. Additionally, software components may change frequently, and it can be challenging to keep the SBOM up to date.

Update in flight or destroy and redeploy

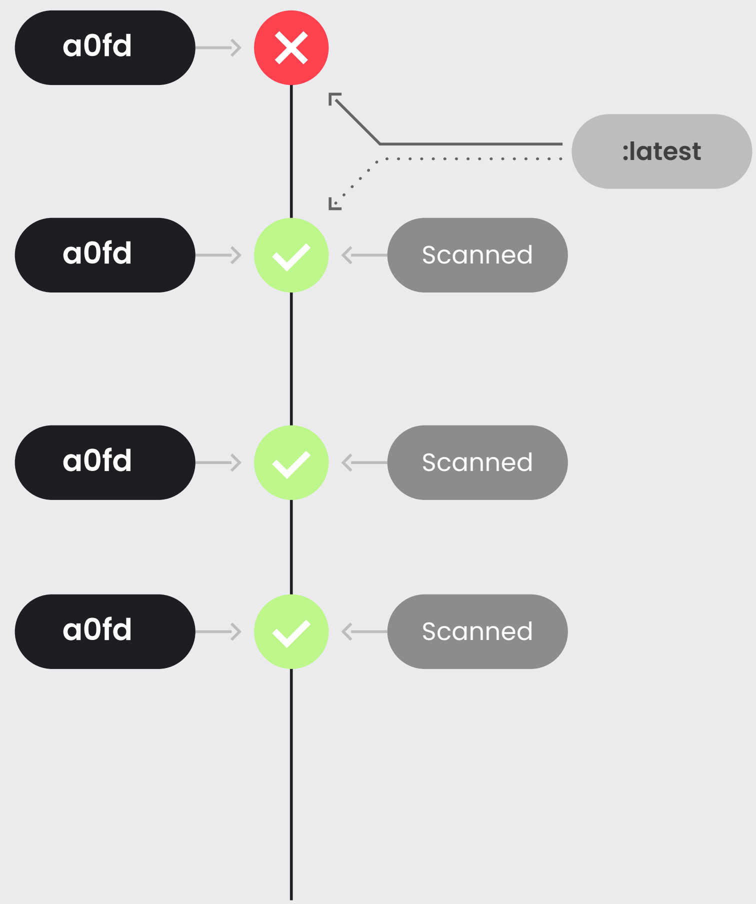
Traditional IT management likes to touch running systems to make necessary changes and updates at runtime. This goes against the immutable philosophy of the cloud, but not all applications in the cloud are fully cloud-native on the day they land there. For example, many cloud migrations include an application refactoring effort that can take a long time. In the early stages, the application may have a monolithic core with microservices around it as components are broken off into a more distributed architecture. In this case, the core may be managed in a "legacy" way, while the newer microservices are treated as immutable.

Containers, in particular, were intended by design to be strictly immutable, which is why images are rebuilt and redeployed any time a change needs to be made. But is this always the right approach?

Using the latest package versions is always a good idea because they should contain less vulnerable software. The main concern when updating the container image packages is if by updating the packages/dependencies, the container image behavior or application breaks or behaves differently. To avoid this, having a proper life-cycle process to conduct proper testing before running the container image in the production environment is a must.

Pinning image versions

Sometimes, the image you scan is not the same one deployed in your Kubernetes cluster. This can happen when you use mutable tags, like "latest" or "staging." Such tags are constantly updated with newer versions, making it hard to know if the latest scan results are still valid.



Source: Sysdig blog

Using mutable tags can cause the deployment of containers with different versions from the same image. Beyond the security concerns from the scan results, this can cause problems that are difficult to debug. Imagine the following scenario:

- An application was already deployed using the :latest tag two weeks ago, and it is running properly.
- Suddenly, a traffic peak requires deploying another instance of that same application on another node that didn't have the container image cached, so it will pull the :latest one, which depending on the timing of the latest build process will include different package versions.

For example, instead of using `ubuntu:focal`, you should enforce the use of immutable tags `like ubuntu:focal-20200423` when possible.

Keep in mind that for some images, version tags tend to be updated with minor, nonbreaking changes. So although it looks a bit verbose, the only option to ensure repeatability is to use the actual image ID:

```
 ubuntu:@sha256:d5a6519d9f048100123c568eb83f7ef5bfcad69b01424
f420f17c932b00dea76
```

```
 podman images --digests
REPOSITORY                TAG         DIGEST
IMAGE ID      CREATED       SIZE
docker.io/library/busybox  latest
sha256:7b3ccabffc97de872a30dfd234fd972a66d247
c8cfc69b0550f276481852627c  abaa813f94fd  2 months ago  3.96 MB
docker.io/kindest/node     <none>
sha256:f52781bc0d7a19fb6c405c2af83abfeb311f13070
7a0e219175677e366cc45d1  476b7007f4f5  4 months ago  828 MB
```

Container build caching

A fully immutable approach would necessitate rebuilding the entire image from start to finish every time something changes. However, some images are very large and have quite a complex build process. In such cases, image caching can avoid rebuilding all of the enabling layers that are not affected by the change. Caching increases how quickly you can update, but it can be dangerous.

Consider the following Dockerfile:

```
FROM ubuntu:focal
RUN apt-get update && \
 apt-get upgrade -y && \
WORKDIR /
COPY ./helloworld /helloworld
USER nonroot:nonroot
ENTRYPOINT ["/helloworld"]
```

The first problem is that if you don't re-pull the base image (ubuntu:focal), it will be reused on every build. Fortunately, the docker build command includes the --pull flag, which will always attempt to pull a newer version of the image.

The next problem is that the first build will download a few Ubuntu packages instructed by the apt-get command. However, the next runs of the build process will use the cached layers, including the one running the apt commands:

```
-zsh
❯ podman build .
STEP 1/6: FROM ubuntu:focal
STEP 2/6: RUN apt-get update &&     apt-get upgrade -y
--> Using cache 5cc82b66a2a5977d2283e510d24428bab8175adfc6fb6f534404c6024da18b28
--> Pushing cache []:0999c4195f9ff9db221acd6ec2aaa609db17c1c173c4d31e476407ccffa84ec9
--> 5cc82b66a2a
STEP 3/6: WORKDIR /
--> Using cache 326c30522ed042dce5b520373517195fb0b2b3de337cf6966fd7d8d513aa1de9
--> Pushing cache []:0ff449a04bfa425cdfb122335f4037b218057342f578e01a33a51d4bf4230263
--> 326c30522ed
STEP 4/6: COPY ./helloworld /helloworld
--> Pushing cache []:e38146533e68e8759859c4311e80f21f39458c64d6e8150ad6767909fee19670
--> dc512d2e412
STEP 5/6: USER nonroot:nonroot
--> Pushing cache []:14f5dd88ef552565a606a983cfcf873259faeb18c6c4358161bee972356edb51
--> edb751c934a
STEP 6/6: ENTRYPOINT ["/helloworld"]
COMMIT
--> Pushing cache []:5626cfd947ec8c3fa6744502995ea56657a89be44598c9f9dc5e3efdd7a87b92
--> d72457410a3
d72457410a30a70556aaca5bf75922a7d0377b5365abbb56b0b8ab0a53eef885
```

Unless you change the RUN command, the layer containing the updated packages will be reused.

Best practices for writing Dockerfiles from Docker say:

When processing a `RUN apt-get -y update` command, the files updated in the container aren't examined to determine if a cache hit exists. In that case, just the command string itself is used to find a match.

This is acceptable if there are no other packages that need updating, but otherwise you can end up with old and insecure packages.

Fortunately, the Docker build command includes the `--no-cache` flag to discard the cache when building the image. This can increase the build time, so it is a trade-off that you need to consider when building the images.

A reasonable policy would be something like:

- Use cache on regular builds (if that happens a few times a day in CI environment, for example) to speed up the build process.
- On a regular basis (daily, weekly, in the event of a major CVE), invalidate the cache using the –pull and –no-cache flags.

Image build caching can be powerful. It provides significant benefits in terms of speed and efficiency when building container images, particularly in large-scale production environments. However, image build caching also introduces potential risks.

One of the main risks is that outdated or vulnerable packages or dependencies may be cached, which can lead to security vulnerabilities in the resulting images. This is especially true in environments where images are built and deployed quickly, with little time for thorough testing and security reviews.

Similarly, image build caching can lead to inconsistency in the resulting images. If an image is built with cached layers that are not up to date, it may not behave as expected, or may not be compatible with other components in the system. This can lead to unexpected errors, downtime, or other issues.

Using admission controllers

Kubernetes is designed to provide a secure and scalable environment for deploying and managing containerized applications. One of its key features is the ability to prevent unauthorized software from running within the cluster. Kubernetes achieves this by using a combination of access control policies, network isolation, and container runtime security features. These measures help ensure the deployment of only approved software, reducing the risk of malicious actors gaining access to your environment.

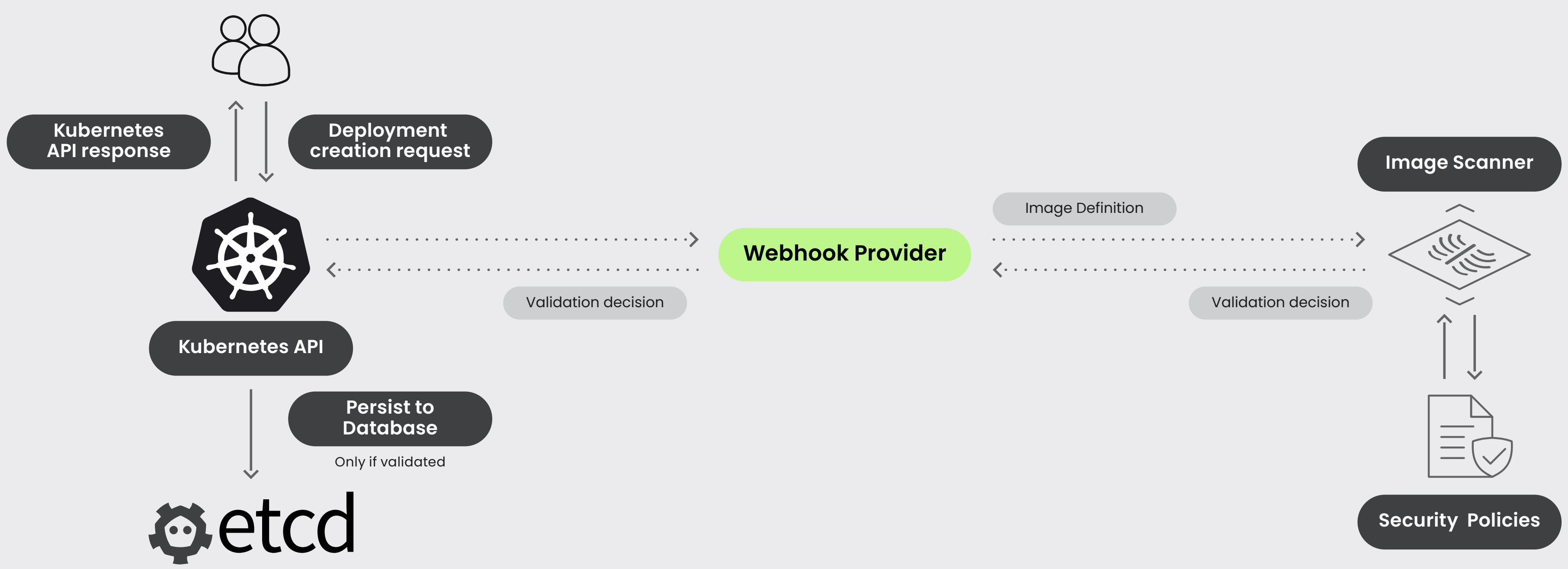
In addition, Kubernetes supports automation and cloud-native patterns, making it easier to manage and deploy applications at scale. This is in contrast to traditional software orchestration methods, which typically rely on manual configuration and maintenance. Kubernetes supports automation through its declarative configuration model and API-driven control plane, which enable you to define the desired state of your environment and have Kubernetes automatically manage the deployment and scaling of your applications to meet that state.

The automation capabilities of Kubernetes make it particularly well-suited for cloud-native environments, where applications are designed to be distributed, fault-tolerant, and scalable. By taking advantage of Kubernetes' features for preventing unauthorized software from running and automating key processes, you can ensure that your applications run smoothly and securely, while also reducing the amount of manual labor required to manage your environment.

Even if you identify vulnerable images in your CI/CD pipelines, there is often little stopping their deployment in production. Admission control is a Kubernetes mechanism for preventing workloads from deploying to clusters unless they comply with policy, and the policy can be related to the container image's vulnerability status. Ideally, you would like Kubernetes to check the images before scheduling them, blocking the deployment of unscanned or vulnerable images onto the cluster, especially a production cluster.

Kubernetes admission controllers are a powerful Kubernetes-native feature that help you define and customize what you allow to run on your cluster. An admission controller intercepts and processes requests to the Kubernetes API after authenticating and authorizing the request, but before the persistence of the object.

Kubernetes Admission Controller



Scanning tools usually offer a validating webhook that can trigger image scanning on demand and then return a validation decision.

An admission controller can call this webhook before scheduling an image. The security validation decision returned by the webhook will be propagated back to the API server, which will reply to the original requester and only persist the object in the etcd database if the image passed the checks.

However, the image scanner makes this decision without any context on what is happening in the cluster. You could improve this solution by using OPA.

OPA is an open source and general-purpose policy engine that uses a high-level declarative language called Rego. One of the key ideas behind OPA is to decouple decision-making from policy enforcement.

With OPA, you can make the admission decision in the Kubernetes cluster instead of the image scanner. This way, you can use cluster information in decision-making like namespaces, pod metadata, etc. An example would be having one policy for the "dev" namespace with more permissive rules, and then another very restrictive policy for "production."

OPA

OPA allows teams to define and enforce policies across their cloud-native environments. With OPA, teams can create policies to detect and respond to malicious or vulnerable images within their containerized applications. You can customize these policies to fit the specific needs of a team and include rules that identify specific vulnerabilities, block malicious images, or trigger alerts.

Here is an example of a policy using OPA that blocks images with known vulnerabilities:

```
package main

import data.vulnerabilities

deny[msg] {
 input.image.vulnerabilities[_].id == vulnerabilities[_].id
 input.image.vulnerabilities[_].severity >=
vulnerabilities[_].severity
 msg := sprintf("image contains high severity vulnerability %v
(%v)", [vulnerabilities[_].id, vulnerabilities[_].severity])
}
```

In this example, the policy defines a deny rule that triggers when an image contains a vulnerability with a severity level equal to or greater than a pre-defined threshold. The policy uses a data file called "vulnerabilities" that contains a list of known vulnerabilities and their associated severity levels.

Teams can integrate this policy into their CI/CD pipeline to automatically block the deployment of images with known vulnerabilities. This helps ensure the use of only secure images in production environments.

OPA gatekeeper

Tools like OPA and Falco can help teams respond to vulnerabilities and malicious images. In the case of Kubernetes, OPA Gatekeeper is an additional policy engine that can enforce policies on Kubernetes-specific resources.

With Gatekeeper, teams can define policies that enforce specific configurations, security requirements, or other constraints on Kubernetes resources. Here's an example policy for detecting vulnerable images using Gatekeeper:

```
package k8srequiredlabels
import data.k8srequiredlabels
violation[{"msg": msg}] {
 image := input.review.object.spec.template.spec.containers[_].
image
 vulnerabilities := data.vulnerabilities[image]
 vulnerabilities != null
 vulnerabilities[_].severity >= 7
 msg := sprintf("Vulnerable image %s found in container %s",
[image, }
```

This policy checks the severity level of vulnerabilities for a given container image. If it is equal to or greater than 7 (on a scale of 0 to 10), it generates a violation message indicating that a vulnerable image has been found in a container. This policy assumes the availability of a data source called "vulnerabilities" that maps image names to vulnerability information.

Constraints are defined using OPA Rego language and are customizable to meet the specific needs of an organization or application. They can enforce a wide range of policies, such as requiring all pods to run with specific security settings or, in this case, disallowing the use of certain container images.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: vulnerable-image
spec:
  match:
    kinds:
      - apiGroups: [“”]
        kinds: [“Deployment”]
  parameters:
    labels:
      requiredLabel: “true”
  enforcementAction: deny
  audit:
    namespaces:
      - default
  rego:
    content: |
      package k8srequiredlabels
      import data.vulnerabilities
      violation[{“msg”: msg}] {
        image :=
input.review.object.spec.template.spec.containers[_].image
        vulnerabilities := data.vulnerabilities[image]
        vulnerabilities != null
        vulnerabilities[_].severity >= 7
        msg := sprintf(“Vulnerable image %s found in container %s”,
[image, input.review.object.metadata.name])
      }
```

This constraint uses the policy defined earlier to deny deployments that use vulnerable images. It matches on deployments, audits the “default” namespace, and enforces the policy by denying noncompliant resources. With this constraint in place, Gatekeeper will monitor new deployments and prevent the use of vulnerable images at runtime.

Dedicated scanning tools

While image scanning is a common feature, it is typically considered part of SCA, which is in turn part of AST. To address the unique challenges of vulnerability management in cloud-native architectures, it is important to use tools and practices specifically designed for these environments. This may include SCA tools that can scan container images for vulnerabilities, as well as integration with orchestration platforms like Kubernetes to automate vulnerability scanning and patching processes. By taking advantage of these dedicated tools, you can more effectively manage vulnerabilities in your cloud-native environment and reduce the risk of security breaches.

As the demand for cloud-compatible vulnerability management tools grows, traditional vendors are expanding their offerings to include solutions that can scan IaaS, containers, and other cloud-native environments. However, these tools may have limitations when it comes to cloud-specific features, such as scan speed and visibility into serverless functions. In other words, while it is possible to adapt these tools for the cloud, they may not be optimized for the cloud, and may not be as effective as purpose-built cloud-native vulnerability management solutions.

Some of the open source tools available include:

- Trivy: A comprehensive and versatile security scanner.
- Clair: An app for parsing image contents and reporting vulnerabilities affecting the contents.

Additionally, it is possible to license commercial options to perform cloud-native vulnerability management at scale. You can learn more about Sysdig Secure here.

Conclusion

As cloud attacks continue to evolve, the need for effective vulnerability management becomes increasingly important. The dynamic and distributed nature of cloud-native architectures requires a new approach to vulnerability management that is tailored to the unique challenges of the cloud.

As a result, an effective vulnerability management strategy for cloud-native architectures requires a shift in mindset from traditional vulnerability management practices. It involves runtime insights and an assessment of the entire cloud environment, including containers, microservices, and serverless functions. It also requires automation and collaboration between DevSecOps teams.

By adopting a proactive approach to vulnerability management, organizations can reduce the risk of cyberattacks, data breaches, and other security incidents. We highlighted some real-world breaches that could have been avoided with the implementation of vulnerability scanning, thus guaranteeing adherence to industry regulations and standards and fostering trust among customers and stakeholders

Evolving vulnerability management for cloud-native architectures is a critical step toward ensuring the security and reliability of cloud-based systems. As the cloud continues to play an increasingly important role in modern business, organizations must prioritize their vulnerability management strategies to stay ahead of potential threats and protect their sensitive data and assets.

The "Sysdig 2024 Cloud Native Security and Usage Report" provides valuable insights into the latest threats and trends in cloud-native environments, and can help you gain a deeper understanding of the evolving landscape and how to take proactive steps to secure your environment. Don't wait until it's too late – stay ahead of the game with Sysdig.

See how Sysdig helps you secure every second.

Take the next step.

REQUEST A DEMO →

sysdig

E-BOOK

EBK-007 REV. B 3/24

About Sysdig

In the cloud, every second counts. Attacks move at warp speed, and security teams must protect the business without slowing it down. Sysdig stops cloud attacks in real time, instantly detecting changes in risk with runtime insights and open source Falco. Sysdig correlates signals across cloud workloads, identities, and services to uncover hidden attack paths and prioritize real risk. From prevention to defense, Sysdig helps enterprises focus on what matters: innovation.

Sysdig. Secure Every Second.