Approaches for Securing Application Development Environments and Artifacts

Published 20 March 2018 - ID G00343645 - 61 min read

By Analysts Michael Isbitski

Supporting Key Initiative is Application and Data Security

Technical professionals responsible for security of development resources need to secure heterogeneous environments and technology stacks, as well as the secrets needed to instantiate them. This research covers approaches for securing common tools and artifacts in modern application development.

Overview

Key Findings

- Development teams employ version control and repositories for maintaining code and artifacts. However, security teams sometimes lack visibility into these systems or don't fully leverage the inherent security capabilities.
- Exposure of source code in hosted repositories and secrets within that source code are pain points for organizations and a frequent root cause of incidents or breaches.
- Virtualization, containerization and cloud technology have made it easier for development teams to provision assets. However, these assets are rarely given the same level of due diligence regarding hardening and monitoring as in production, making them prime targets for exploitation.
- Organizations that make use of externally sourced code must contend with issues of potential malicious code injection. Integrity of source code may be limited to basic supplier attestation.

Recommendations

Technical professionals focused on application security should:

- Standardize on one version control system (VCS), like Git, and ensure all teams have appropriate access to code repositories. Leverage integrity checking, access control and integrations with other tooling such as IAM and SIEM.
- Use secrets management to provide a mechanism for storing secrets and sensitive data outside of source code and infrastructure build scripts.
- Leverage hardened baselines and infrastructure as code to create images and build scripts. Maintain them in VCS for development and operations teams to instantiate from. And audit these assets at build time as well as for changes during runtime.
- Require programmatic analysis of externally sourced code to uncover potentially malicious code. This should be integrated with either internal VCS or supplier VCS, based on where code is committed.

Analysis

Application development teams require a range of disparate tooling to perform their work. Tools and machine resources may exist on-premises or in cloud providers, and they may be physical systems, virtual machines or containers. In addition, numerous artifacts are created as output during these efforts, including source code, infrastructure as code (IaC) and test data. Technical professionals responsible for security of development resources need to protect these heterogeneous environments, multiple technology stacks and development artifacts.

Gartner clients raise several concerns when it comes to securing development processes and environments.

The abuse cases include:

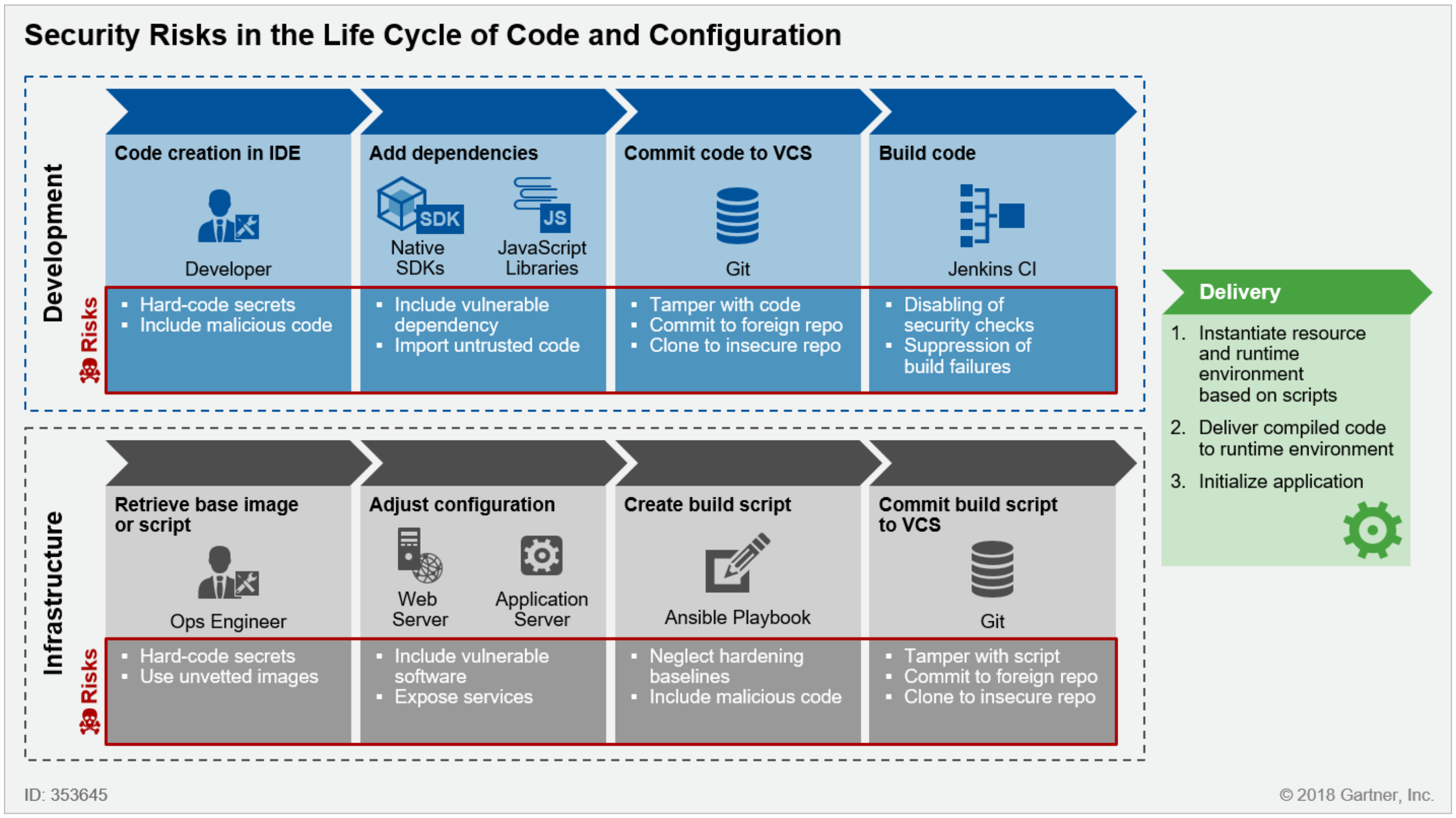
- Exposure of secrets in source code and infrastructure as code
- Risk of source code theft or unintentional leaks
- Deliberate creation of malicious code or logic bombs in source code
- Unintentional inclusion of vulnerable or untrusted code

This research covers approaches for securing the code, tools and artifacts that accompany modern application development. These approaches partially overlap with security approaches for DevSecOps, as discussed in "Structuring Application Security Practices and Tools to Support DevOps and DevSecOps." (https://www.gartner.com/document/code/337322?ref=grbody&refval=3869263) Not included in this assessment are secure coding and architecture practices. Validating code with application security testing (AST) is also not included. These topics are covered in "A Guidance Framework for Establishing and Maturing an Application Security Program" (https://www.gartner.com/document/code/315333?ref=grbody&refval=3869263) and "How to Integrate Application Security Testing Into a Software Development Life Cycle," (https://www.gartner.com/document/code/317333?ref=grbody&refval=3869263) respectively.

Security of development environments also requires structured approaches to data security, network security and security monitoring. However, differentiation from what is done for production assets is minimal if not duplicated. As such, they are covered briefly with links to relevant research for further insight.

Figure 1 depicts application development and application infrastructure operations activities. At a high level, it shows the creation and building of the code. In the case of the latter, it specifically shows the creation of assets for application code to run on and the deployment of application code to those instances. The two sets of activities become more tightly intertwined in DevOps practices, as well as adoption of cloud and container technologies. Artifacts from the activities include plaintext source code, application configuration, binary objects, build scripts, virtual machine images and container images. Throughout this slice of the life cycle, potential risks emerge, which are also called out in Figure 1.

Figure 1. Security Risks in the Life Cycle of Code and Configuration



Source: Gartner (March 2018)

During the processes of source code development, significant risks emerge, such as:

- Hard-coding of secrets
- Inclusion of malicious or untrustworthy code
- Inclusion of vulnerable dependencies
- Committing code to unprotected locations

Similarly, with the creation of IaC that is used to instantiate application infrastructure, risks emerge, such as:

- Hard-coding of secrets
- Use of unvetted images
- Exposure of services
- Script tampering

Table 1 highlights the mitigating capabilities and processes that can be leveraged to secure development environments and artifacts. These elements of development exist as code and configuration, software and tools, and developer and nonproduction environments.

Table 1: Capabilities and Processes to Secure Development Environments and Artifacts

Mitigating Capability or Process ↓	Code and Configuration ↓	Software and Tools ↓	Developer and Nonproduction Environments ↓

Mitigating Capability or Process ↓	Code and Configuration ↓	Software and Tools ↓	Developer and Nonproduction Environments ↓
Version control systems and repositories	Control code revisions, time-stamping, commit signing and integrity checking	Provide versioning and storage for binaries, build scripts and IaC	Provide versioning and storage for images used to instantiate development instances
Secrets management	Generate and allocate secrets, store secrets outside of source code	Generate and allocate secrets, store secrets outside of build scripts and IaC	Store and protect secrets used to bootstrap development instances
Static application security testing	Scan code for embedded secrets, malicious code or logic bombs	N/A	Uncover where sensitive data might be generated and stored and whether it is sufficiently protected
Software composition analysis and open-source software (OSS) governance	Scan and control dependency import from public sources	N/A	N/A
Data masking	N/A	N/A	Use static data masking for sensitive data such as PII, PHI, and NPI in nonproduction environments
Hardening	N/A	Define standards for security of integrated development environments (IDE) and client and server-side applications	Define standards for security of OS, virtual machine or container
Continuous configuration automation	N/A	Audit against baselines and enforce compliance, support patching processes, enable uniformity of application infrastructure	Instantiate development instances based on IaC
Virtualization and containerization	N/A	Use VMs, containers and service virtualization to provision development assets, or use virtual desktop infrastructure (VDI)/desktop as a service (DaaS) for highly sensitive development	Isolate development instances within virtual machine and container networks

Source: Gartner (March 2018)

For instance, when protecting software and tools, secrets management provides a mechanism that should be used to generate, store and allocate secrets. This includes authentication credentials and API keys that should be kept outside of build scripts and IaC. The sections that follow describe these security approaches in more depth.

Code and Configuration

Protecting against the variety of abuse cases against source code — such as exposure, theft, tampering and malicious code injection — encompasses the plaintext source code that development teams create, as well as application configuration. The source code itself is a form of intellectual property for the organization, often containing proprietary functionality or algorithms. Application configuration may also include highly sensitive data such as passwords, tokens or encryption keys — broadly categorized as "secrets" — that provide access to systems throughout application infrastructure. These numerous artifacts end up being stored in multiple locations, including desktop IDEs and VCS repositories, and they need to be protected appropriately. The current standard for most development organizations with VCS is to use distributed models, which increases the challenge of centralizing and controlling code. And with the creation and adoption of SaaS services for development, the landscape of where these artifacts and supporting tools live expands further.

Structure Your Version Control and Repositories

Analysis of VCS product selection, centralized-vs.-distributed version control, monolithic-vs.-multi repository, and schemes for branching or merging are out of scope for this research. For additional technical detail on VCS, see the Background on Version Control Systems section.

From the security perspective, it is more a matter of selecting and standardizing on one or a small number of VCS tools and associated code maintenance processes. This can help avoid splintered development efforts and simplify both the software development life cycle (SDLC) and secure SDLC (S-SDLC). Gartner recommends Git-based VCS for most organizations because it is well-established and well-supported.

The variety of VCS hosting and consumption models provides a great deal of flexibility for organizations when determining how they will provide VCS functionality, but it also raises a number of security concerns. These concerns are often focused on controlling whether internal teams are able to publish internal company intellectual property to public repositories, or whether internal teams are permitted to fetch source from public repositories that may not be well-vetted. The former introduces risks of sensitive data leakage of proprietary source and even potential breach if secrets are embedded within source. The latter introduces risk of vulnerable or malicious code making its way into an organization's codebases and application infrastructure. Additionally, where and what type of VCS is used can impact availability or ability to integrate other necessary functions like identity and access management (IAM), centralized logging and auditing, and security monitoring. You should leverage all of these capabilities in enterprise implementations of VCS.

Secure the VCS and Artifacts

Due to the wide range of VCS options in the underlying mechanism or the repackaged vendor solutions, it is difficult to cover all possible permutations and settings an organization might need to secure its VCS. For additional technical detail on some inherent VCS security issues, see the Inherent Security Issues With VCS and Repositories section. Table 2 details some of the main issues to look for and related best practices.

Table 2: Sample VCS Security Best Practices

Practice ↓	Description ↓
Restrict public access	With the use of VCS SaaS, private will almost always be the desired state. This should be monitored and restricted accordingly.
Enforce authentication and authorization	Enforce AuthN/AuthZ to the VCS and projects. Integrate with pre-existing SSO or IAM solutions to keep identity centralized, and leverage two-factor authentication (2FA) for higher levels of assurance if supported.
Enforce logging and auditing	Enable VCS logging and auditing features. Review logs periodically, particularly for projects with the most sensitive code. Integrate VCS logging with security information and event management (SIEM) tools, if supported.
Protect data in transit by disabling unsecure protocols	Disable unsecure protocols, and force the use of HTTPS and Secure Shell (SSH).

Practice ↓	Description ↓
Employ commit signing	Enforce use of commit signing (e.g., GNU Privacy Guard [GPG]) if supported by the VCS.
Utilize branch protection	Leverage branch protection features to protect a particular branch from pushes or deletions. This isn't available in Git stand-alone but is available in many of the commercial variants such as GitHub, GitLab and Microsoft Team Foundation Server (TFS).
Monitor for unprotected cloned repositories	■ Monitor for the execution of clone functions through VCS logging or other monitoring tooling such as endpoint protection (EPP) or SIEM. Cloud access security brokers (CASBs) can also be useful in cases of VCS SaaS. ■ Monitor for creation and exposure of the .git folder (or .hg with Mercurial) through EPP or SIEM. ■ Block Git requests to external, untrustworthy repositories, or whitelist access to only vetted repositories using protective controls like CASBs (for VCS SaaS/PaaS) or secure web gateways (SWGs).
Prevent exposure of cloned repository directory	Block access to the .git folder (or .hg with Mercurial) of any externally exposed assets and/or internal assets if deemed sensitive enough. This can be done through web server configuration, keeping repository folders at a higher-level parent directory than the web-hosted directory and disabling directory browsing.
Protect data at rest in the repository by employing repository encryption as needed	Encrypt the repository if necessary, such as when mandated by compliance, but ensure that key management is handled adequately. Because cloning is a feature of Git repositories, and it is trivial to duplicate code in a VCS for any user that has access (or even via manual cut and paste), encrypting all source in all possible locations may be difficult, if not impossible.
Follow patch management process	Monitor for and regularly apply patches to your instances of VCS. This requires a combination of native VCS logging for known instances and other tooling like CASBs and SIEM to identify unknown instances. Like any software, bugs and vulnerabilities are uncovered over time, evidenced by published Common Vulnerabilities and Exposures (CVE) for Git and Apache Subversion (SVN). Vulnerabilities may exist within the server-side VCS code, the underlying protocols, or client-side VCS components like desktop interfaces and IDE plug-ins.

Source: Gartner (March 2018)

Address the Problem of Unsecure Secrets Storage

Creation and unsecure storage of secret data typically starts in development. Unfortunately, this sometimes continues on into production deployment. Embedded secrets may surface later as part of regular security testing or security monitoring. In some cases, it may also become a root cause for a security incident or data breach. It is rarely sufficient in almost all operating environments to rely on the principle that secret data is protected within the confines of your organization's data center or local servers. Additional technical information on secrets management capabilities can be found in the Emergence of Secrets Management section. And methods that can be used to discover or block embedded secrets are covered in the Discovering Hard-Coded Secrets section.

Some common scenarios that emphasize the need for secrets management include:

- Most organizations, and certainly development teams, are at least experimenting with cloud-based deployments. Cloud assets require credentials, cryptographic key material and/or secret data to instantiate.
- Transport Layer Security (TLS) is employed to protect data in transit and to enable HTTPS. Part of enabling TLS involves creation and installation of a certificate key pair. Storing the private key becomes problematic and, if not done securely, can compromise encrypted communications.

- Application infrastructure components typically need to communicate with other system elements, such as a DBMS or other middleware, to provide full functionality. This requires preconfigured service account credentials, certificates or API keys to enable interoperability. Storing this secret data securely is a challenge.
- Encryption keys are often needed to facilitate encryption of sensitive data at rest, often seen in mobile application development. Embedding key material in source code for mobile apps that are distributed to the public is a recipe for disaster because binaries are readily decompiled or disassembled.

Secrets can be fetched manually by users, but commonly, programmatic access is necessary. This is critical in DevOps practices, where continuous integration (CI) and continuous delivery (CD) processes are more automated. Human involvement to "key in a password" is an impossibility or creates too much latency in build processes. Historically, application teams have sometimes tried to conceal this type of secret data in environment variables. However, such a practice is not a replacement for proper secrets management, and attackers will often harvest secret data stored in environment variables.

Programs must be able to securely fetch the secret data they need to compile, deploy and run a given application or sets of APIs on a target asset. This can bury some of the underlying problem, and you will hear the term "turtles all the way down" to describe the regression dilemma that arises. That is, to get access to the secret contained in the secret vault, you must first provide some type of authentication secret, such as a credential or authentication token. If you are trying to do that programmatically and at scale, such as in application bootstrapping, it is difficult to accomplish without embedding a secret and defeating the security of secrets management.

Select a Secrets Management Capability

Selecting a secrets management solution to address the problem of embedded secrets in development code and configuration is dependent on the application architecture within your organization. Container-based solutions will only be useful in container environments. Docker or Kubernetes deployments may only constitute a small percentage of your development and production environments across the enterprise. Continuous configuration automation (CCA) options will only be useful if you've embraced IaC concepts and processes, and they don't cover some of the other use cases where secrets must be generated and allocated dynamically. Those tools are best-suited for facilitating secrets sharing in infrastructure build scripts. Cloud service provider (CSP) implementations such as Amazon Web Services Key Management Service (KMS) in tandem with Parameter Store or Amazon Simple Storage Service (S3) may make sense for complete AWS deployments, but not hybrid implementations.

It can become a choice between robustness of security feature set versus usability and automation within cloud and container environments. Instantiation of secrets management services that requires secure authentication, or subsequent authentication into the secrets management vault when provisioning an asset, may be problematic in cases such as fully automated container provisioning, which is common in CD within DevOps. If your organization and use cases can support manual intervention, such as where an admin provides credentials manually as part of secrets management server startup or app provisioning, this will be less of an issue. Otherwise, this introduces the problem of having to hard-code certificates or credentials in code or configuration to facilitate automation of secure authentication into secrets management services. This is no more secure than storing secrets "in the clear" in source code or configuration, or stashing data in unprotected repositories. It is only shifting the type and location of secrets being stored.

Dedicated secrets management solutions like HashiCorp Vault provide the largest range of functionality, such as semi-automation or full automation of secrets rotation, dynamic secret generation, split-key cryptography, and hardware security module (HSM) support. They are more platform-agnostic, providing flexibility in those cases where application infrastructure is very diverse within the organization. Dedicated solutions can help avoid lock-in with a specific cloud provider, although it does bind you to the secrets management vendor and their tools. Case studies and analysis on cloud-native secrets management, like that of AWS KMS, versus dedicated secrets management can be found with Netflix (https://www.usenix.org/sites/default/files/conference/protected-files/enigma_haken_slides.pdf) , Segment (https://docs.google.com/presentation/d/1ipP2eB9pW5j3WDvzCGz9Wy4MBoK2SMvOTZtPdf5kSNs/edit#slide=id.g1fa55b980e_1_257) and Threat Stack (https://www.threatstack.com/blog/cloud-security-best-practices-finding-securing-managing-secrets-part-2/) . Any tool you select to provide secrets management becomes a potential point of failure in application availability. As a result, vendor maturity, product maturity and service availability are all critical factors in selection.

Control the Use of Unvetted Components and Public Repositories

Though use of open-source components and freely available source is of great benefit to development, the reality remains that such source contained within public repositories is unvetted. Components obtained from internet sources should be validated to ensure they meet your organization's requirements and are free from known vulnerabilities or CVEs. These components end up in codebases during development or are pulled in dynamically during instantiation and runtime. If unchecked, vulnerable dependencies carry over to production. This is largely the realm of software composition analysis (SCA) and open-source software governance (OSSG) from vendors such as Flexera, JFrog, Sonatype, Snyk, Synopsys (Black Duck or Protecode) and WhiteSource. The topics of software composition analysis and open-source software governance are covered in "Structuring Application Security Practices and Tools to Support DevOps and DevSecOps." (https://www.gartner.com/document/code/337322?ref=grbody&refval=3869263) Such tooling is also critical in securing your IaC in development, which can carry over to production.

Gartner recommends employing such tooling to address the challenge of vetting dependencies, particularly in development efforts because components may change numerous times until production release. It is difficult to institute manual review or governance processes because the rate of change can be quite high. Waiting for approval for a given component to pass compliance requirements is a guarantee that application releases will be negatively impacted, especially those being actively developed, built and released. Additionally, components and dependencies have a life cycle of their own, which may exhibit vulnerabilities over time. Without the aid of SCA and OSSG, it becomes difficult to manage dependencies at scale and to identify known vulnerabilities in published components.

SCA excels at identifying component use in application codebases, binaries and container images for known vulnerabilities and license constraints. OSSG takes the analysis a step further and provides greater visibility and control throughout the development life cycle. In the context of securing development and controlling use of components, OSSG can help:

- Advise developers on vulnerable components before they are imported into the IDE, a code repository or a binary repository
- Limit the use of unvetted and/or known vulnerable components, libraries, frameworks and container images, based on customizable policy
- Restrict access to public repositories and package managers as determined by the organization
- Restrict components that may be safe for use in development but not production, or vice versa, that may result from OSS license constraints
- Protect from deliberate malicious code injection via external dependencies by ensuring that no one can include components without the proper validation

It is possible such controls could be circumvented by determined developers, such as in the case of connecting via an external network and/or a cutting and pasting source code. However, these systems can still institute control if they are integrated as part of a structured CI/CD build pipeline. Upon detection of unvetted components, they can fail builds and alert development, operations and/or security staff.

Closely monitoring dependencies and use of package managers also serves as a form of technical control in supply chain hygiene. As an example, if a developer outside your organization changes or removes an open-source dependency that your codebase relies on, any future builds will break unless you've kept a local copy and modified your source accordingly to point to it. This was an unfortunate reality in 2016 for any node application that relied on the "left-pad" package in npm. Countless application builds failed after the author pulled the package over a legal dispute. [1]

Similarly, if a developer deliberately injects malicious code into dependencies, it may not be caught immediately and could end up in your code. Numerous examples exist where vulnerabilities in OSS components have been uncovered over time. Such is the case with Apache Struts or OpenSSL, both of which have suffered from numerous, nondeliberate vulnerabilities. Some complex weaknesses or vulnerabilities are not uncovered with automated code scanning, and engineers can still miss issues in manual code review, especially when accounting for varying experience and skill sets.

CASBs can be leveraged to monitor and/or block access to certain SaaS repositories like GitHub, but the range of support for such repositories in API-integrated modes can be a limiting factor. While a given vendor may support common repositories like GitHub, support may be lacking for other services like GitLab or Visual Studio Team Services. SWGs can also be leveraged to block access to public repositories wholesale. However, strict enforcement may exacerbate the issue described earlier of development teams circumventing controls and accessing repositories from external or unprotected networks and retrieving code. CASBs are covered in detail in "How to Secure Cloud Applications Using Cloud Access Security Brokers" (https://www.gartner.com/document/code/323355?ref=grbody&refval=3869263) and "Cloud Access Security Brokers: A Comparison of Platform Solutions." (https://www.gartner.com/document/code/317345?ref=grbody&refval=3869263) SWGs are covered in "Using Secure Web Gateway Technologies to Protect Users and Endpoints." (https://www.gartner.com/document/code/337219?ref=grbody&refval=3869263)

Software and Tools

In addition to source code and configuration, securing development also includes tools that development teams require in order to create and deploy code. This includes the IDEs and the machine resources that code is built and deployed to in order to advance through the later stages of the SDLC, like quality assurance (QA), user acceptance testing (UAT) and system integration testing (SIT). In a typical enterprise, these are separate and sometimes (network) isolated environments. Development teams need to be able to deploy the code they create to application infrastructure in order to validate that application code works as expected.

Define Development Assets With Infrastructure as Code

For modern development and technology stacks, adapt infrastructure-as-code practices for creation of development servers and application runtime environments. For additional technical detail on IaC practices, see the Background on Infrastructure as Code section. The topic of IaC is also covered in "Using DevOps Tools for Infrastructure Automation." (https://www.gartner.com/document/code/326084?ref=grbody&refval=3869263)

As with application source, leverage secrets management to avoid hard-coding credentials, encryption keys and private key portions of certificates in server build scripts and images. Secrets should be allocated dynamically as part of instantiation to avoid potential exposure within IaC.

Also, like your application source code, IaC should also be maintained with proper versioning and an enterprise repository. Lines can start to blur between VCS and binary repositories, particularly with respect to the various object types being stored in a repository. The broad delineations include plaintext source, IaC (build scripts), compiled binaries/components and images. However, core functionality of the binary repository remains the same as that of VCS, which includes access control, auditing, integrity checks and signing. Repository managers such as JFrog Artifactory and Sonatype Nexus have also emerged as universal repositories. They can serve not only as a binary repository for a wide variety of object types, but also as connection points or proxies to other repositories, package managers and registries (in the case of containers).

In the absence of IaC practices, rely on other traditional methods such as checklists for server builds, server imaging, software installation, manual configuration and patching. However, these can be error-prone, time-consuming and difficult to manage effectively in a large enterprise. This is especially true for development environments that are more ephemeral and exhibit high rates of change. Inconsistencies and misconfigurations between development, test and production environments quickly become apparent, which can impact functionality of code, testing results and efficacy of security controls. Organizations, particularly for development resources, should begin exploring or expanding IaC practices.

Secure and Audit Development IaC

Like production assets, build scripts and images for development assets must be secured initially based on hardening baselines and then audited over time. This is especially true for development assets where new components may be pulled in during runtime. Many of the CCA vendors provide prehardened scripts to start with that can serve as baselines for OS and applications. These scripts may include services or configurations that you don't want in your network, and they should be vetted and customized accordingly. Additionally, public container registries like Docker Hub provide the latest builds of Docker images for a given software package. Like CCA builds scripts, container images may contain services or configurations you don't want in your environment — development, production or otherwise. Restricting access to public repositories such as Docker Hub may be necessary to control the use of unvetted container images in development environments.

In the case of Docker Hub, official repositories such as NGINX, Apache httpd and Ubuntu are scanned with Docker's own version of SCA, which can help expose known vulnerable components. [2] Docker also supports SCA as part of Docker Enterprise Edition instances, but it is discontinuing SCA for private repositories in Docker Hub in March 2018. [3]

Many of the SCA and OSSG vendors have support for scanning container images, if not container registries, directly. Container security vendors such as Aqua Security, Layered Insight, NeuVector, StackRox and Twistlock provide SCA for container images prior to container instantiation as well as during container runtime. CoreOS Clair (https://github.com/coreos/clair) also exists as an open-source SCA alternative to the commercial off-the-shelf (COTS) solutions providing container image scanning.

Processes and tooling for auditing IaC is covered in "Structuring Application Security Practices and Tools to Support DevOps and DevSecOps." (https://www.gartner.com/document/code/337322?ref=grbody&refval=3869263) The mechanisms used to audit IaC in development environments or elsewhere include:

- Open-source auditing tools such as Chef Inspec, Serverspec and Testinfra
- Commercial auditing tools such as Chef Compliance (or within Chef Automate, a COTS CCA)
- Vulnerability assessment (VA) and security configuration assessment tools; see "A Comparison of Vulnerability and Security Configuration Assessment Solutions" (https://www.gartner.com/document/code/322810?ref=grbody&refval=3869263)
- Cloud workload protection platforms (CWPP), specifically those tools providing configuration management (CM), cloud infrastructure security posture assessment (CISPA) and/or VA functionality

Secure Developer Tools

Barring special-purpose tools, such as software development kits (SDKs) or key generation tools, the predominant tool for developers is the IDE. Compromised IDEs (or plug-ins that integrate with them) can pose a risk of introducing malware into compiled source. This was demonstrated in 2015 with the XcodeGhost malware that targeted the Xcode IDE used for macOS and iOS app development. Any apps compiled with an infected version of the IDE were trojanized, which was over 4,000 according to FireEye analysis. [4] This impacted not just the apps distributed in public app stores, but also installations of the Xcode IDE within organizations. It's a modern example of the attack Ken Thompson described in "Reflections on Trusting Trust," where a compiler could be compromised in order to trojanize the application code it compiles. [5]

The risk of infected IDEs is a far-reaching concern. It is certainly impacting to ISVs that are creating and selling software. But it is also a concern for any organization that acquires code from outsourced and/or offshored resources. Infecting the IDE enables stealthy code injection and propagation. This includes a variety of malicious code, including back doors, logic bombs and surveillance mechanisms. Cloud IDEs such as AWS Cloud9 are mostly immune to this type of attack, barring an exploit against the CSP itself, a man in the middle (MITM) attack or attacks targeting installed extensions. However, they create separate complexity by having no client software to integrate security tooling into. Webhooks can be leveraged to integrate security if supported. Otherwise, organizations must focus scanning efforts on commits within the VCS or during the build process.

The best defense against using altered tools like IDEs is ensuring that only signed binaries are used. Modern OSs enforce code-signing checks as part of their security model. You should install and run only those executables that are from trusted sources with valid code signatures that match the expected authoring party. Also, verify that file hashes match that of the original installation source.

Detecting and blocking execution of infected binaries and malware is the realm of other protective technologies like endpoint protection (EPP). EPP is covered in "The Evolving Effectiveness of Endpoint Protection Solutions" (https://www.gartner.com/document/code/323224?ref=grbody&refval=3869263) and "Evaluation Criteria for Endpoint Protection Platforms." (https://www.gartner.com/document/code/346995?ref=grbody&refval=3869263) Note that EPP should generally not be configured to monitor directories that contain build output. This has been known to slow down or negatively impact CI/CD build processes, and such directories should be set as exclusions in EPP. You can also leverage SWG to block connections to or downloads from untrusted or malicious sources.

Virtualize and Containerize to Improve Security

Virtual machines and containers, and clusters of them (such as with Kubernetes), are often used to spin up development assets as needed. They are a form of IaC, provide portability and can exist on-premises or in the cloud. Though there is more upfront complexity in provisioning them correctly, use of virtualization and containerization provides greater ability to control assets.

Virtualization also comes in other forms that can be beneficial to security. Outside of virtual machines and containers, two other approaches used in creation of development environments include:

- **Service virtualization:** Leverage application runtime virtualization, such as the Java Virtual Machine (JVM) or similar virtualized runtime environment. This is useful for mimicking web application, web API and mobile application communications for the purposes of mocking, prototyping, development and testing. These tools — from vendors such as CA Technologies, Micro Focus, IBM, Parasoft and SmartBear — virtualize messages and communications between applications and/or APIs. The topic of service virtualization is covered in "A Guidance Framework for Testing Web APIs." (https://www.gartner.com/document/code/320726?ref=grbody&refval=3869263)
- **Desktop virtualization:** Leverage hypervisors and virtualization to provide a virtual "desktop" through a web browser or mobile application, which is especially useful for instantiating isolated development environments. Virtual desktop infrastructure is the on-premises implementation of this and includes Citrix XenDesktop and VMware Horizon. Desktop as a service is the cloud-hosted variation and includes Amazon WorkSpaces, Citrix Cloud and VMware Horizon DaaS. Desktop virtualization is covered in "Choosing Between VDI and DaaS for Your Virtual Desktop Solution." (https://www.gartner.com/document/code/334191?ref=grbody&refval=3869263)

In the case of service virtualization, there is no asset to secure. Communications are fully emulated. In some cases, vendors also provide nonfunctional security testing capability in addition to the expected functional testing features. With desktop virtualization, the development environment is fully virtualized and can be further isolated, reducing some the headaches of maintaining and securing development assets.

Secure Related SDLC Systems

Other assets and servers in the development ecosystem must also be secured. This can include systems such as application development life cycle management (ADLM), CI/CD and bug tracking. The range of possible vendors and products is large. At a high level, securing these related systems includes:

- Enforce authentication and authorization into these SDLC systems. Not all developers need to see functional requirements, test results or code snippets for all application projects, and especially so for nonfunctional security-related items.
- Enforce use of HTTPS when web interfaces are available to ensure data is protected in transit.
- Ensure that CI/CD servers are appropriately hardened and that access control is enforced:
  - Limit the ability to modify build tasks and workflow, especially security-related tasks like SCA and AST on established applications.
  - Monitor for suppression of build failures, especially those where security checks fail.
- SDLC systems may also contain detailed vulnerability information or sensitive data such as PII, PHI or primary account number (PAN). This may require additional data protection measures and access controls.

Handle Test Data in Development Environments

Security of development environment data, or test data, is an extension of data security practices. Data protection techniques and technology vary depending on where the data is transmitted, used or stored. Approaches include anonymization, data masking (static and dynamic), encryption and tokenization. Also, regulations such as General Data Protection Regulation (GDPR), Health Insurance Portability and Accountability Act (HIPAA), and Payment Card Industry Data Security Standard (PCI DSS) can impact what data is deemed sensitive and for specific subsets of individuals. Related data security research includes:

- "Consuming DBaaS Securely: Comparing Options for Securing On-Premises and Cloud Databases" (https://www.gartner.com/document/code/328293?ref=grbody&refval=3869263)
- "Understanding the Implications of GDPR for the Technical Professional" (https://www.gartner.com/document/code/350321?ref=grbody&refval=3869263)

- "Protecting PII and PHI With Data Masking, Format-Preserving Encryption and Tokenization" (https://www.gartner.com/document/code/343738?ref=grbody&refval=3869263)

Going into enforcement in May 2018, GDPR mentions application development as being potentially in scope when processing or storing data on European Union citizens, particularly if using "real" data as test data. Generally, static data masking is seen as an adequate control for sensitive data such as PII, PHI, and NPI in nonproduction environments. Production data usually calls for real-time data protection techniques like dynamic data masking or tokenization. And regulatory compliance may necessitate other access controls, data protection techniques or use of encryption.

Take a Measured Approach to Development Network Isolation

Isolating or connecting production and nonproduction environments becomes a balance of high security versus development agility. Regulation or corporate policy may require you to perform a level of network isolation for development environments. This is typically accomplished via network segmentation or microsegmentation when dealing with virtualized environments. It may also include other network security controls like network access control (NAC). Related research on zoning, segmentation and microsegmentation includes "Decision Point for Postmodern Security Zones" (https://www.gartner.com/document/code/337220?ref=grbody&refval=3869263) and "Comparing Products for Microsegmentation in Virtualized Data Centers." (https://www.gartner.com/document/code/321031?ref=grbody&refval=3869263) IaaS providers like AWS and Azure provide their own microsegmentation mechanisms. Respectively, this includes Amazon VPCs and Azure VNets, which are covered in "Best Practices for Amazon VPC and Azure VNet." (https://www.gartner.com/document/code/337223?ref=grbody&refval=3869263)

Gartner clients sometimes struggle to find the appropriate balance of openness and restriction in development environments. The decisions are often a factor of development history, IT and network security culture, and other compounding factors, like outsourcing or offshoring of development. Some important observations include:

- Not having a cloud and container strategy will quickly become a pain point. VPCs, IaC, VMs and containers are ideal technology fits for provisioning of development environments and pave the way for DevOps (or DevSecOps). Development teams are likely already embracing these technologies, potentially as unsanctioned or unmonitored activity.

- Taking an overly restrictive stance will result in splintered development environments. It is a low-cost, easy effort for an application team to procure AWS, Azure or Google Cloud Platform (GCP) subscriptions. Taking an overly aggressive stance in locking down development environments will make it difficult, if not impossible, for developers to perform work. This can inadvertently promote a "cloud exodus." It can also severely impact application releases and the ability for the organization to carry out its business if it is very technology- and development-focused.

- Being too open or liberal may result in the utilization of too many cloud IaaS or PaaS services. Side effects of this problem may include too many administrative GUIs, web APIs, command line interfaces (CLIs), templates and uniqueness to manage all environments effectively. This drives the need for solutions like those provided by CASB and CWPP vendors.

- As development teams embrace container technology, visibility and control of these environments by security teams can erode. Container networks are not addressable in the same way as virtual machine networks and physical networks.

Monitor Development Tools and Workstations

Monitoring of development tools and workstations is mostly the realm of SIEM and user and entity behavior analytics (UEBA), covered in "A Comparison of UEBA Technologies and Solutions" (https://www.gartner.com/document/code/317414?ref=grbody&refval=3869263) and "SIEM Technology Assessment." (https://www.gartner.com/document/code/325571?ref=grbody&refval=3869263) Variations from what is done in production are minimal, though the types of events you may be looking for can vary. This research has touched on them throughout, including various use cases such as accessing SDLC systems, committing code to VCS, cloning code repositories, and generating or storing secrets.

This will predominantly be accomplished with SIEM. High-maturity organizations dealing with advanced use (or abuse) cases may also look to leverage:

- Endpoint data loss prevention (DLP) to discover and/or block deliberate source code theft or accidental leakage

- UEBA to monitor developer behavior within a code repository if the UEBA supports direct integration
- UEBA to monitor behaviors of development teams and to detect possible insider threat. This is more about identifying anomalous behavior from the normal developer baseline as opposed to deep analysis of what a malicious developer might be creating in code. This may help identify potential abuse cases such as data exfiltration or source theft, and to a lesser extent, someone inserting known malicious code. Examples of behavior analysis in development include:
  - How or when do developers modify code? Is it a typical pattern for the specific developer?
  - Where is a developer storing or retrieving code? Is a developer leveraging the established enterprise VCS?
  - Is a developer accessing the VCS from the usual network or geographic locations? Is a developer fetching source and pushing it to illegitimate or untrusted locations?
  - Has a developer been browsing illegitimate sites or communicating with known malicious parties?

Strengths

Practitioners should take note of the following strengths associated with securing development environments:

- Tooling to help secure development environments and artifacts is plentiful. There are ample open-source, freemium or COTS tools for VCS, binary repositories, secrets management, CCA, IaC auditing and developer IDEs. You don't need to procure expensive products to secure your development environments and tools.
- IaC capabilities make it possible to automate provisioning, auditing, reconfiguration and/or redeployment of application infrastructure over time. Application infrastructure used in development can be quickly torn down and redeployed with patches and configuration changes as an alternative to traditional patch or system management. Script reuse across environments can also alleviate some of the issue of variance between application environments. Virtualization and containerization have commoditized provisioning of development environments without the need for expensive hardware, excluding some special cases like on-premises VDI.
- Secrets management has emerged to provide capabilities for generating and storing secrets dynamically. These capabilities reduce the likelihood that someone will hard-code a secret in source code, configuration or build scripts, presuming this person has been provisioned and trained properly on the use of such systems.
- Much of the tooling useful for securing development environments is available in cloud-hosted variants. This further removes requirements of bare-metal servers and eases integration with other tooling via web APIs and webhooks.

Weaknesses

Practitioners should take note of the following weaknesses associated with securing development environments:

- The democratization of development has resulted in splintered development tooling and an increasingly large attack surface. There is no shortage of tooling, languages, frameworks, libraries and open-source components. This creates additional burden on security teams that are tasked with securing development environments. It also often results in unmanaged, vulnerable resources.
- Security tools are limited in their ability to detect malicious code and logic bombs. In most cases, tools such as static AST (SAST) will detect potential cases based on other suspicious functionality, which often requires further manual review. Writing custom rules for SAST engines can increase effectiveness. However, it requires deeper understanding of code structure and styles used by the organization. It also requires advanced subject matter expertise with the given SAST tool to create the necessary customization.
- Effective secrets management, particularly as part of application infrastructure bootstrapping, is a difficult practice. Though secrets management capabilities exist now to protect secrets, the solutions carry high complexity and a familiarity with cryptography. Securely authenticating into a secrets management vault is a relatively easy task when handled manually, but doing it in an automated, programmatic fashion at scale is another matter entirely. The temptation to embed a secret to securely authenticate to a secrets vault may present itself, and the "turtles all the way down" regression dilemma arises.

- Detection of embedded secrets is not an exact science given variation in what strings or numbers represent. Tooling to uncover embedded and exposed secrets is also scattered. Adequate detection can require a combination of repository scanning, SAST, special-purpose tools and CSP-provided tools.

Guidance

Gartner Welcomes Your Feedback

We strive to continuously improve the quality and relevance of our research. If you would like to provide feedback on this research, please visit Gartner GTP Paper Feedback (http://surveys.gartner.com/s/gtppaperfeedback) to fill out a short survey. Your valuable input will help us deliver better content and service in the future.

Standardize Development Tooling and Secure Repositories

It is rare that SDLC systems will be entirely under the control of application security teams. In most cases, installation and maintenance of the many SDLC systems is handled directly by development teams or development operations teams. Ideally, application security teams should be involved with procurement and implementation of SDLC systems like VCS. Application security practitioners should also look to partner with these owning teams to ensure that security is accounted for in their implementation, operation and maintenance.

Standardize on VCS along with the type and number of repositories. Typically, this would be a Git-based VCS. Operations teams should have access into these systems, particularly as part of DevOps practices where IaC, not just application source code, might be stored in the VCS. VCSs are a control point for code, as well as an integration point for security tooling and auditing. Security teams also require access in order to integrate capabilities that can help protect application source code, application configuration, secret data and infrastructure as code. Leverage the native security features of VCS, such as hashing, signing, access control and reporting.

If development teams within the organization are working with multiple languages, technology stacks and object types, different IDEs and repository instances may be required to support the complete range of development activities. Consequently, installing and maintaining multiple VCS instances can create additional burden on development, operations and security teams. Simply put, if your source code and configuration exists in too many locations — scattered on-premises and in the cloud — those are additional locations that you need to identify, maintain and secure. More than likely, it can result in operational or security gaps, which can lead to data leakage or source code theft. Find a balance on the minimum number of VCSs that are required to empower development teams; security teams must also help protect and monitor these systems.

Monitor and Control Use of Public Cloud Repositories

Public cloud repositories may be useful for experimental development or rapid prototyping, but it creates difficulties for an organization that is attempting to control code sprawl and all development artifacts. Cloud-based repositories must also support integration with other security and nonsecurity tooling via webhooks, but the reverse may not be true for other capabilities in your toolchain. VCS SaaS providers may also offer private hosting, which provides a higher level of control and reduced exposure for the organization. Prefer the use of private repositories and on-premises instances of repositories where possible. Use other tooling, such as CASBs and SWGs, to detect and/or block the use of unsanctioned repositories.

Leverage Secrets Management Tools

Secrets management is critical for dynamically generating, accessing and storing secrets without embedding the sensitive information in code, configuration or build scripts. Secrets management tooling provides high security vault mechanisms where secret data is encrypted, where authentication and authorization is enforced, and where access to secrets is properly audited. Leverage a combination of repository scanning, SAST, special-purpose secrets-finding tools and CSP-provided tools to uncover where embedded secrets may be inadvertently embedded and to avoid exposure.

Combine Techniques to Detect Malicious Code Creation or Inclusion

Detecting creation or injection of malicious code such as backdoors or logic bombs is a highly complex problem. Malicious code creation or injection may be deliberate in some cases, but more frequently, it is a result of accidental inclusion. No security vendor can guarantee perfect accuracy, and even high-maturity organizations struggle to address the problem. It is also not a concern for all organizations unless they are highly regulated and are processing/storing/transmitting highly sensitive data and/or outsourcing a significant portion of development work. Detection requires a mixture of multiple tools and processes, including AST during code commits and integrated with VCS, dependency analysis and governance, malware analysis, monitoring of developer behavior, and manual code review.

Embrace Infrastructure as Code for Development Assets

Use IaC concepts and CCA tooling to create both images and build scripts (aka playbooks and recipes). Many pre-existing scripts exist that are based on established, hardened baselines and benchmarks. These scripts should be maintained in VCS and repositories for development and operations teams to instantiate from. Additionally, you should audit any provisioned development assets over time for changes during runtime. This accounts for scenarios such as installation of a dependency after the application infrastructure is instantiated or discovery of a new vulnerability in a component used in a development instance that was only scanned at build time.

Have a Strategy in Place for Development Environments, and Avoid Splintered Efforts

Avoid splintered efforts with use of cloud assets such as IaaS and PaaS. Development teams are likely evaluating or deploying cloud technologies today because it reduces overhead with delivery of application code and application infrastructure. Being overly restrictive with development environments can result in development teams setting up virtual or container networks within your own network, or worse, separate networks hosted within CSPs as unknown subscriptions. The latter is another form of "shadow IT," which can create security gaps in development and potential exposure of secret material, sensitive data or company intellectual property such as source code.

Standardize on one or a small number of CSPs and native capabilities they provide. For multicloud environments, prefer universal or platform-agnostic tooling for portability and to reduce dependency on a given CSP.

The Details

Background on Version Control Systems

VCSs, sometimes also referred to as source control management (SCM), are designed to maintain versions of objects when they are checked in by development, infrastructure and security teams. They're fundamental to continuous integration and continuous delivery, and they enable a range of security and nonsecurity processes and tool integration. They should be configured to enforce authentication and authorization, natively or through integration with third-party IAM, when accessing the repository to push or pull code. Additionally, they provide integrity checks and audit trails by hashing and time-stamping when data is committed to the repository. An object can be application code and associated configuration as well as compiled artifacts or application infrastructure itself, such as in the case of images or server build scripts.

A VCS in modern application development is typically one of two types:

- Distributed VCS (DVCS) — Most commonly Git, but sometimes an alternative like Mercurial, depending on organizational preference or history of development process
- Centralized VCS (CVCS) — Most commonly Apache Subversion or Microsoft Team Foundation Server, though centralized version control presents scalability issues in larger development teams or where code change is frequent

These VCSs exist as open-source software, providing version control as well as repository server functionality. Repositories come in a variety of flavors, with additional features as well as different functionality to better serve compiled dependencies or source. A nonexhaustive list of some of the variations and vendors:

- Commercial Git-based VCS, including Atlassian Bitbucket and Microsoft TFS 2017 (using the default Git DVCS as opposed to CVCS)
- Artifact or binary repositories to host larger or binary objects beyond just plaintext source, which become beneficial for storing and versioning artifacts like compiled dependencies and container images (These include repository managers like Sonatype Nexus and JFrog Artifactory and also container registries like Docker Trusted Registry.)
- VCS traditionally focused on source control with extensions to support larger objects or binary types, such as Git LFS and Perforce

- VCS SaaS, which may be public or private (GitHub and GitLab fall into this category, where many open-source software project teams publish code to public repositories on GitHub for use by the development community. GitHub repositories may also be private though, which can be hosted as a SaaS by GitHub, hosted in an IaaS — such as AWS, Azure or Google Cloud Platform — or deployed within your organization's network and servers. Microsoft offers TFS as a VCS SaaS in the form of Visual Studio Team Services, and Atlassian Bitbucket can also be consumed as VCS SaaS in the form of Bitbucket Cloud. Not to be outdone, AWS offers a VCS SaaS in the form of CodeCommit.)

Inherent Security Issues With VCS and Repositories

Git is not without its own inherent security issues, one of which includes the issue of Git repository cloning. [6, 7] Git cloning effectively duplicates an entire repository, including history, metadata and source code contained within. It is an inherent function of Git, and the cloned repository can be stored anywhere. Any user who has access to the repository can clone it. This is by design for Git and other DVCSs, allowing any repository to become the master copy in the event of repository loss. This is unlike CVCSs, which rely on availability of the central repository. This resiliency of a DVCS like Git becomes a potential downside, though, with respect to security.

Although it is possible to enforce authentication, authorization and encryption on a given repository, those controls don't necessarily follow the cloned copy. This scenario is touched on within the context of AWS CodeCommit in "An Assessment of AWS Development Tools for Continuous Integration." (https://www.gartner.com/document/code/318912?ref=grbody&refval=3869263) Other OSS tools exist, such as git-crypt (https://github.com/AGWA/git-crypt) to selectively encrypt data being stored in a Git repository based on administrator-defined file patterns and git-remote-gcrypt (https://spwhitton.name/tech/code/git-remote-gcrypt/) , which can be used to encrypt an entire repository. However, these tools can create compatibility issues with other CI/CD systems, and they add complexity of key management to decrypt data within the repository.

Teams may inadvertently or deliberately store such sensitive data to simplify and automate version control. Encryption of a repository carries the same risks of general data encryption, including potential data loss.

If your organization is encrypting code repositories entirely or in part, any loss of associated encryption keys guarantees that all data contained within the repository is lost.

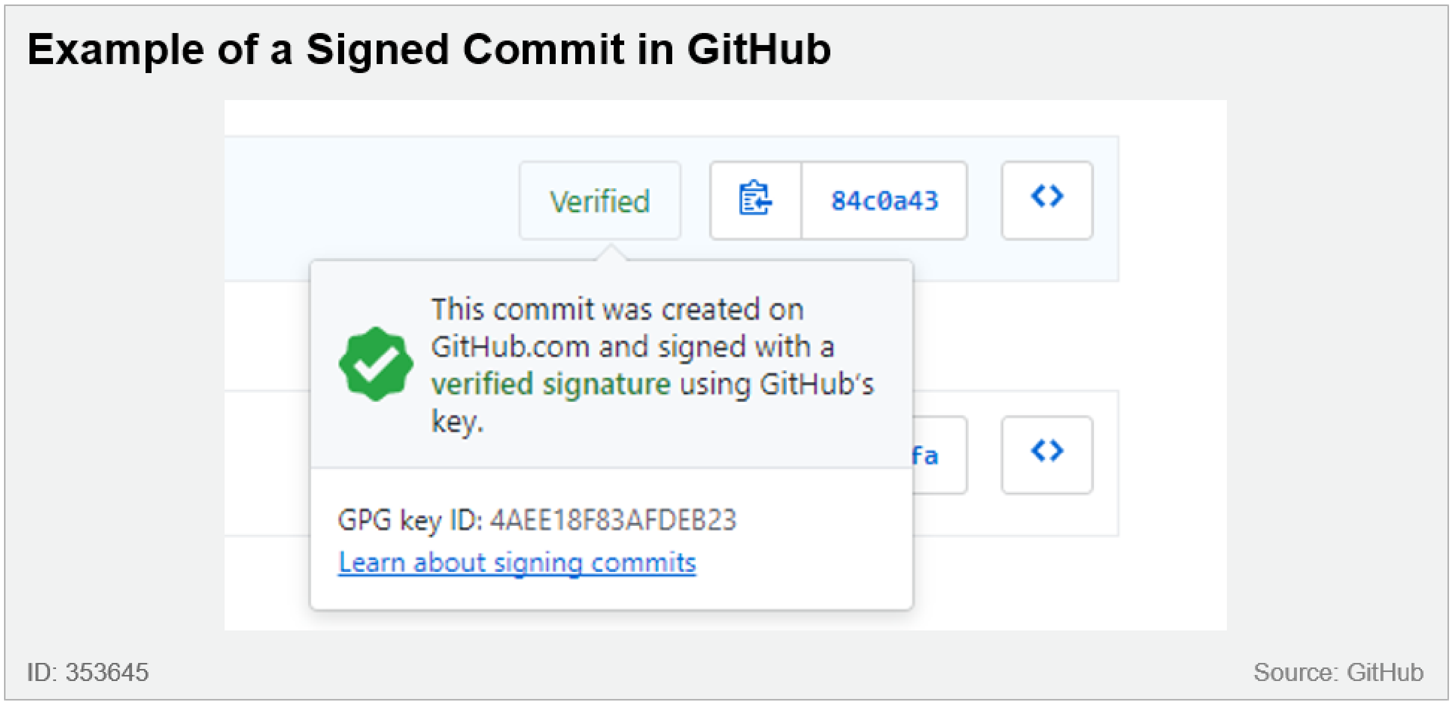
Similar issues of data exposure can arise with other VCSs. Of particular concern is where a repository may inadvertently be placed on an internet-facing host. There is ample security research on how to locate and harvest data from code repositories. [8, 9] Some related scanning tools include DVCS-Pillage (https://github.com/evilpacket/DVCS-Pillage) , dvcs-ripper (https://github.com/kost/dvcs-ripper) and GitTools (https://github.com/internetwache/GitTools) . It can be useful for your teams to leverage these tools to scan for potentially exposed repositories that attackers might search for and harvest.

Integrity and Authenticity Requires Both Hashing and Signing

VCSs typically provide a hashing mechanism of objects to enforce integrity, commonly Secure Hash Algorithm 1 (SHA-1) and Message-Digest Algorithm 5 (MD5). The hash or checksum validations help with verifying if the source is what is expected and hasn't been tampered with in transit or at rest. Tampering could be through misconfigured access controls at the source repository, some exploit that makes it possible to modify data unexpectedly, or modification of data in transit. Use of HTTPS makes this more difficult for attackers, barring a man-in-the-middle style attack, and by ensuring proper authentication and authorization with a repository, you can raise the bar further. However, hashes and checksums still don't provide authenticity that comes from signing. Also, older hashing algorithms like SHA-1 and MD5 have been proven weak, and development work is underway in some cases to update the VCS to support more robust algorithms like SHA-256 (SHA-2) or SHA-3 in Git and Mercurial. [10, 11]

VCSs, at least the mature ones, also often offer signing mechanisms to provide further authenticity and accountability of committed code. Enablement of the feature by default is not always a given, and users may have to manually enable the capability or enforce use of the necessary extensions to provide signing. GPG is the most commonly seen mechanism for enabling signed commits, such as with git gpg (https://git-scm.com/book/id/v2/Git-Tools-Signing-Your-Work) and Mercurial gpg (https://www.mercurial-scm.org/wiki/GpgExtension) . However, the usual key management headaches still apply, including key generation, key expiration and key revocation. Administrators and users will need to overcome the usual learning curve of asymmetric encryption because commit signing is just another variation of digital signatures. Signing is available within GitHub (https://help.github.com/articles/signing-commits-with-gpg/) and GitLab (https://docs.gitlab.com/ee/user/project/repository/gpg_signed_commits/index.html) , and leveraging the capability may be simpler in comparison to bare installations of Git or Mercurial. An example of a signed commit in GitHub can be seen in Figure 2.

Figure 2. Example of a Signed Commit in GitHub


Example of a Signed Commit in GitHub

This commit was created on GitHub.com and signed with a verified signature using GitHub's key.

GPG key ID: 4AEE18F83AFDEB23

ID: 353645

Source: GitHub

Source: GitHub

Signing capability may not be provided natively with package managers, which is the case for some common ones such as Bower, Composer, npm, NuGet and Yarn. Yarn itself is technically a client or proxy to npm, and while it is touted as secure by the creators, only integrity checking is used. Yarn currently lacks proper signing, [12] and signing is currently in active development for NuGet. [13] Though a given VCS or package manager may support signatures, it may not be supported for all object types.

In some cases, there are also underlying issues with the implementation of Git [14] and Mercurial. [15] Attacks against hashing algorithms (that is, creating collisions) have been well-known for some time, but in the context of VCSs or file creation, it can be difficult to carry out a practical attack. Centrum Wiskunde & Informatica (CWI) Amsterdam and Google proved the possibility of a collision attack in the "SHAttered" research published in February 2017 (https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html) . The research team leveraged the compute power of the Google cloud infrastructure to perform enough computations to create a SHA-1 collision.

Unfortunately, this leaves organizations with a VCS that may be considered potentially "weak" in the area of hashing by strict security standards. Although it is important to be aware of these inherent weaknesses, the combination of signing via GPG and hashing with SHA-1 should be sufficient for all but the most risk averse organizations. Being active open-source projects, Git or Mercurial will likely see improved crypto mechanisms over time, depending on user demand or security incidents. This is especially true for Git, where there are number of established vendors repurposing it as commercial offerings.

Decentralization of VCS

Development teams typically work within an integrated development environment, such as Eclipse or Microsoft Visual Studio. They commit code to a VCS as part of ongoing development work, which may be a local repository copy in the case of DVCSs. VCS centralization can present some challenges with DVCSs, which is decentralized by design and provides users the ability to clone repositories where they have access. This is truer of Git stand-alone as opposed to the commercial variants, where vendors add features to maintain a level of centralization.

Potential VCS SaaS Impacts

Cloud-hosted repositories can also be impacted by issues that affect the SaaS or IaaS provider itself. An example of this occurred from September 2016 through February 2017, where it was discovered that a buffer overflow vulnerability existed within Cloudflare's network of reverse proxies, potentially exposing HTTP traffic. [16] If HTTP traffic included sensitive data, which could have included data like VCS credentials or proprietary source code, this data was potentially exposed. The Cloudflare vulnerability was a rare event and not specific to the security of development resources. However, the potential risk may be a deciding factor for some organizations in deciding whether to host sensitive data — like proprietary source code — internally or in cloud repositories.

Emergence of Secrets Management

Secrets management has emerged as a capability to enable secure creation, storage and retrieval of secrets. The term "secret" is used broadly to refer to a variety of data types, including:

- User accounts and passwords
- Machine (or service) accounts and passwords
- One-time passwords
- API keys
- Authentication tokens
- Private key portions of certificates used in Secure Sockets Layer (SSL)/TLS encryption
- Key material for symmetric encryption

Secrets management can be highly useful in development activities, specifically in the area of avoiding the embedding or hard-coding of secret data in source code, build scripts, infrastructure as code and so on. At a high level, secrets management aims to provide the following functionality around secrets:

- Enforce authentication and authorization for users or machines requesting secrets
- Audit secrets access, including reads, writes, rotations and revocations
- Encrypt the secret data at rest, possibly also supporting the use of hardware security modules or OS-provided secure key store for increased security

Variations on Secrets Management Capabilities

Secrets management is often included within privileged access management (PAM) and application-to-application password management (AAPM) tools, but the capability also exists as dedicated tooling. Secrets management functionality is found in the following tools:

- Dedicated COTS options such as BeyondTrust PowerBroker Password Safe, CyberArk Conjur Enterprise, HashiCorp Vault Enterprise and Thycotic Secret Server
- Dedicated OSS options such as Square Keywhiz (https://square.github.io/keywhiz/) or the OSS versions of CyberArk Conjur and HashiCorp Vault
- CSP implementations such as AWS Key Management Service (KMS) with Parameter Store or AWS KMS with Amazon Simple Storage Service (S3)
- Container orchestration engine implementations such as Docker Swarm or Kubernetes
- CSP implementations of container orchestration engines such as Amazon Elastic Container Service (ECS), Amazon ECS for Kubernetes (EKS), Azure Container Service (AKS), or Google Cloud Platform Kubernetes Engine
- Application orchestration engine implementations such as HashiCorp Nomad (via integration with Vault)

- Continuous configuration automation and configuration management (CM) implementations such as ansible-vault (http://docs.ansible.com/ansible/2.4/ansible-vault.html) , Chef Data Bag (https://docs.chef.io/data_bags.html) and Puppet Hiera (https://puppet.com/docs/puppet/5.3/hiera_intro.html) .

In this context of securing development environments and artifacts, we are less concerned with managing access to privileged credentials or administrative sessions, the traditional definition of PAM. Rather, we are seeking to avoid hard-coding of secret data needed to instantiate and run applications.

Securely Authenticating Into a Secrets Vault at Scale

Secrets management handle the dilemma of automating authentication to the vault differently based on their pedigree and dependence on CSP services. In AWS scenarios, it's heavily dependent on AWS KMS, which was designed for management of cryptography keys used in data encryption. However, it is sometimes employed in tandem with other services to create a general-purpose secrets management solution. Credstash (https://github.com/fugue/credstash) is one example, which repurposes AWS KMS and Amazon DynamoDB to facilitate storage and exchange of credentials. AWS KMS is used to generate an encryption key and encrypt the credential. It then uses DynamoDB to store the wrapped encryption key and encrypted credential. This type of implementation can also be used within node.js applications (https://www.npmjs.com/package/credstash) or terraform scripts (https://www.fpcomplete.com/blog/2017/08/credstash) . Note, however, that credstash was created prior to the existence of dedicated secrets management tools like Vault.

Credstash became popular in some circles for ease of use and simplification of the complex matter of secrets management. However, it raises the question of whether to deploy a dedicated secrets management solution. In the case of terraform, HashiCorp's own Vault might offer more functionality and flexibility as opposed to repurposing CSP functionality. There are also some security issues in the design of credstash, the primary of which is its dependence on the security boundary of AWS IAM roles. Access to the encryption key needed to unwrap a given secret is based on the boundary of the provisioned machine instance. Any user on that instance will have access to fetch items from the DynamoDB store and decrypt credentials for that machine instance. Variations on the credstash approach exist, and all of them leverage AWS KMS and share the same downside. Sneaker (https://github.com/codahale/sneaker) is another OSS alternative to credstash that works in a similar fashion. Many leverage AWS Parameter Store (https://docs.aws.amazon.com/kms/latest/developerguide/services-parameter-store.html) within EC2 Systems Manager (Simple Systems Manager/SSM) and use of secure string parameters, which still ultimately rely on AWS KMS. Examples include AWS CodeDeploy (https://aws.amazon.com/blogs/mt/use-parameter-store-to-securely-access-secrets-and-config-data-in-aws-codedeploy/) integration and Amazon ECS (https://aws.amazon.com/blogs/compute/managing-secrets-for-amazon-ecs-applications-using-parameter-store-and-iam-roles-for-tasks/) integration.

Not to be outdone in the cloud-native secrets management space, Google also provides secrets management functionality in Google Cloud Platform (https://cloud.google.com/kms/docs/secret-management) , which makes use of Google Cloud KMS and Google Cloud Storage in a similar way. Microsoft provides similar capability with Azure Key Vault (https://docs.microsoft.com/en-us/azure/key-vault/key-vault-get-started) , and also offers a secrets management tool targeted toward development in the form of Secret Manager or user secrets (https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?tabs=visual-studio) for .NET Core projects. However, Microsoft does not provide any form of a trusted store or encryption for secrets in that solution. Microsoft's tool is designed mostly as a mechanism to shift embedded secrets out of source code to a separate configuration store. As a result, and for most use cases, it is best to avoid it for the time being or rely on Azure Key Vault for enterprise-class secrets management.

Discovering Hard-Coded Secrets

There exists a range of tools to unearth embedded secrets. It's likely that finding secrets is easier than adequately protecting them in the first place, demonstrated by the high number of recent breaches related to repository exposures. In many cases, secrets or source code were stored in unprotected public repositories, which can be easily harvested by attackers.

There are also cases where private internet-hosted repositories are compromised via leaked credentials, which highlights the need for dedicated secrets management to keep secrets out of code and data repositories. In November 2017, Skyhigh coined the attack "GhostWriter" (https://www.skyhighnetworks.com/cloud-security-blog/skyhigh-discovers-ghostwriter-a-pervasive-aws-s3-man-in-the-middle-exposure/) to describe a type of MITM attack against unprotected S3 buckets. If code or secrets that originate from development activities are stashed in unprotected S3 buckets, they can be not only read, but also stealthily modified. Realistically, this would be true for any type of repository that is not adequately secured, particularly a cloud-hosted one. Catchy name aside, it highlights the potential attack of unauthorized exposure of secrets and/or modification of code. This includes things like injection of malicious code or vulnerable dependencies.

Though there is a variety of options for secrets management, setup and operation can be complex. It requires deep understanding of the hybrid environments that most organizations operate. Machine instances, which may be ephemeral, are being spun up at scale across cloud providers and environment types. There is a high probability that an unknowing developer or infrastructure engineer will stash code, configuration, scripts and, more importantly, secrets in unprotected locations to facilitate application delivery and operation. Identifying deliberate or unintentional embedding of secrets early in the SDLC and prior to code delivery becomes critical for most enterprises.

Techniques and tooling you can use to uncover secrets are spread across the following:

- **Dedicated secrets scanning tools:** An emerging category of tooling (often OSS) designed to scan source code and repositories for inadvertently embedded secrets. Some examples include:
  - git-secrets (https://github.com/awslabs/git-secrets) : Created and open-sourced by AWS Labs; enables blocking a commit of sensitive or secret data based on regular expression (regex) match.
  - Repo Supervisor (https://auth0.engineering/detecting-secrets-in-source-code-bd63b0fe4921) : Created and open-sourced by Auth0; can scan code during Git pull requests or scan directories via a CLI. As part of the product roadmap, functionality may be expanded to include remediation countermeasures leveraging AWS KMS and credstash.
  - Truffle Hog (https://github.com/dxa4481/truffleHog) : Searches through Git repositories to uncover secrets. This can be done through regex patterns or by searching for high entropy values that may indicate the presence of a token, key and so on.
- **CSP implementations of secrets scanning tools:** Another emerging category of tooling offered by CSPs to scan source code directly or repositories for secrets. These are limited in availability to specific regions or still in preview. Examples include:
  - Amazon Macie (https://aws.amazon.com/macie/) : Scans Amazon S3 repositories for the purpose of data classification and to uncover potentially sensitive data, including secrets as well as PII. The capability also employs behavior analysis to determine how data is being stored or accessed to help uncover other potential threats. The tool is not limited to just S3 scanning for sensitive data. It also currently supports analysis of CloudTrail data with support for other AWS data stores being part of the product roadmap.
  - Azure Credential Scanner (aka CredScan) (https://azure.microsoft.com/en-us/blog/managing-azure-secrets-on-github-repositories/) : Launched by Microsoft in November 2017 and focused on discovery of Azure related secrets. The service is operated automatically by Microsoft and currently only supports scanning of GitHub repositories. In the event a secret is discovered, Microsoft notifies the Azure subscription owner. Microsoft also provides a variation of the engine in the form of "Continuous Delivery Tools for Visual Studio," (https://marketplace.visualstudio.com/items?itemName=VSIDEDevOpsMSFT.ContinuousDeliveryToolsforVisualStudio) which is a client-side extension for Visual Studio 2017 that can also check for Azure secrets.
- **Static AST:** Analyzes original source code (or compiled binaries, depending on the vendor), supporting detection of embedded secrets in source or potentially malicious code, such as a logic bomb. For secrets discovery, this may sometimes be limited to basic cases where function names or variables include "username," "password" or "key" or other string-based pattern matches. This needs to happen before code is committed and built; otherwise, a given secret becomes part of the compiled application and would need to be revoked. The topic of SAST is covered in "How to Integrate Application Security Testing Into a Software Development Life Cycle." (https://www.gartner.com/document/code/317333?ref=grbody&refval=3869263)
- **Cloud workload protection platforms:** Discover unprotected cloud repositories like Amazon S3 buckets. This specifically includes the CWPP vendors that provide cloud infrastructure security posture assessments such as Alert Logic, CloudCheckr, Dome9 and Evident.io. Note that these tools may be more focused on identification of such repositories and misconfiguration as opposed to analyzing the data within them to uncover secrets.

- **CASBs and secure web gateways:** Discover, protect and/or block use of SaaS services like Amazon S3, where they may be used to store source code that could contain secrets. This includes unsanctioned use where you don't want any users leveraging SaaS repositories, as well as sanctioned use, where you may allow use of SaaS repositories but want to ensure sensitive data isn't being leaked or is encrypted at rest. CASBs and SWGs can monitor for such activity and block the activity entirely based on defined policy. CASB vendors include Skyhigh, Bitglass, CipherCloud, Microsoft, Netskope and Symantec and are covered in detail in "Cloud Access Security Brokers: A Comparison of Platform Solutions." (https://www.gartner.com/document/code/317345?ref=grbody&refval=3869263)

- **VCS:** May offer capability to block pushing of secrets to the code repository based on commit hooks or file pattern matches. Git offers Git Hooks (https://git-scm.com/book/gr/v2/Customizing-Git-Git-Hooks) , which enable execution of custom scripts client- or server-side, and gitignore (https://git-scm.com/docs/gitignore) to ignore files based on file pattern matches. In a similar vein, GitLab provides the ability to define push rules to help prevent secrets ending up in the repository where they might be exposed. However, even though file patterns are customizable via regex, it is currently limited to file pattern matching and lacks content inspection to determine if a file contains secrets within.

If you opt to use cloud-hosted repositories like Amazon S3 or GitHub, it is imperative that you lock it down appropriately. Ideally, secrets should also be stored separately from source code and configuration. Both services have become a common source of leaked data. This may be in part due to broad adoption of the service and high volumes of users increasing the likelihood of attacks. However, the bigger contributing factor is likely complexity, especially when dealing with high volumes of distributed source code and IaC. AWS, GitHub and similar cloud services provide a range of controls and monitoring capabilities. Unfortunately, organizations overlook how internal teams might be storing data in cloud repositories, or they struggle with leveraging tools to mitigate exposure of source code and secrets.

Background on Infrastructure as Code

IaC enables an organization to create a server build once and reproduce instances elsewhere in high volumes. It also enables packaging of application code with the application infrastructure, which paves the way for other practices such as containerization. IaC definitions are often in XML or YAML format, and build scripts generally use Python or Ruby. Serving as a form of "golden image," IaC helps provide uniformity across the different application environments, including development, QA, UAT and production. The umbrella of IaC includes build scripts and images for instantiation of application infrastructure in bare-metal, virtual machine or container environments. And these environments may be on-premises, externally hosted or cloud-hosted. For most organizations, application environments are a mixture of all these resource types and hosting methods. IaC practices provide flexibility when instantiating assets for any type of environment, especially useful for development assets. It also provides inherent ability to more effectively audit those assets. CCA is the tooling used to facilitate IaC practices.

AWS and Microsoft provide CCA capabilities to support IaC practices via AWS CloudFormation and Azure Resource Manager, respectively. Dedicated CCA tools include Chef, Puppet, Red Hat (Ansible), and SaltStack, which remove the IaaS dependency and support deployment of assets anywhere. Universal options like HashiCorp Terraform abstract this even further, providing ability to deploy complete application infrastructures that are spread across environments and machine resources. Terraform can also be used to fold existing build scripts from disparate CCA tools into unified infrastructure profiles. Though Terraform is useful for automating application infrastructure deployment, without appropriate guardrails, it can introduce unnecessary complexity for provisioning of individual development environments. For that reason, HashiCorp also maintains Vagrant with a specific focus on development environment provisioning.

Distinguishing Vagrant From Terraform

Vagrant and Terraform are both actively developed projects from HashiCorp. The primary design focus with Vagrant is building and managing development environments, typically as virtualized instances on an individual developer workstation. Terraform is primarily designed for building and managing application infrastructure, with an emphasis on IaaS and PaaS providers like AWS, Azure and Google. Though Terraform can theoretically be used to provision and manage sets of virtual machines in development environments, it is better-suited for multicloud and hybrid cloud deployments. [17]

Vagrant provides some features that are useful for individual development environments that Terraform does not. Specifically, this includes synchronized folders, HTTP tunneling and automated network configuration, more akin to something you would find with VMware Workstation Pro or VMware Workstation Player. A Vagrant IaC artifact is in the form of a Vagrant file, a Ruby-based script that describes the machine configuration or multimachine environment. It can make use of a variety of virtual machine and container providers, including VirtualBox, Hyper-V, Docker and VMware. The Vagrant Share (https://www.vagrantup.com/docs/share/) capability within the Vagrant engine leverages ngrok to provide remote sharing via HTTP (aka webhooks), SSH or direct TCP communication with Vagrant Connect (https://www.vagrantup.com/docs/share/connect.html) . Note that some features, such as enforcing HTTPS, require a paid subscription with ngrok.

Vagrant provides some native security controls such as SSH key encryption and TLS enforcement on non-HTTP connections. However, Vagrant Share carries some inherent weaknesses that HashiCorp readily discloses. [18, 19] These shared connections become additional attack vectors for your development environments, so they need to be disabled, restricted or monitored appropriately.

Evidence

[1] "How One Developer Just Broke Node, Babel and Thousands of Projects in 11 Lines of JavaScript," (http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/) The Register.

[2] "Docker Security Scanning," (https://docs.docker.com/docker-cloud/builds/image-scan/#opt-in-to-docker-security-scanning) Docker Docs.

[3] "Docker Cloud/Hub Security Scanning Deprecation FAQs," (http://success.docker.com/article/scanning_deprecation_faqs) Docker.

[4] "Protecting Our Customers from XcodeGhost," (https://www.fireeye.com/blog/executive-perspective/2015/09/protecting_our_custo.html) FireEye.

[5] "Reflections on Trusting Trust," (https://dl.acm.org/citation.cfm?doid=358198.358210) ACM.

[6] "git clone," (https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) Atlassian.

[7] "git-clone," (https://git-scm.com/docs/git-clone) Git.

[8] "Pillaging DVCS Repos," (https://www.defcon.org/images/defcon-19/dc-19-presentations/Baldwin/DEFCON-19-Baldwin-DVCS-WP.pdf) DEF CON.

[9] "Don't publicly expose .git or how we downloaded your website's sourcecode — An analysis of Alexa's 1M," (https://en.internetwache.org/dont-publicly-expose-git-or-how-we-downloaded-your-websites-sourcecode-an-analysis-of-alexas-1m-28-07-2015/) Internetwache.

[10] "git/Documentation/technical/hash-function-transition.txt," (https://github.com/git/git/blob/752414ae4310cd304f5e31649aaab2dcf307057c/Documentation/technical/hash-function-transition.txt) GitHub.

[11] "SHA1 and Mercurial Security," (https://www.mercurial-scm.org/wiki/mpm/SHA1) Mercurial.

[12] "Yarn Is Micro Secure," (https://snyk.io/blog/yarn-is-micro-secure/) Snyk.

[13] "Package Signatures Technical Details," (https://github.com/NuGet/Home/wiki/Package-Signatures-Technical-Details) GitHub.

[14] "Notes Towards Detached Signatures in Git," (https://bitbucket.org/ojacobson/grimoire.ca/src/75a219a061b60bb32948b8a2b71c8ccf1dc19a62/wiki/git/detached-sigs.md) Atlassian.

[15] "Commit Signing Plan," (https://www.mercurial-scm.org/wiki/CommitSigningPlan) Mercurial.

[16] "Incident Report on Memory Leak Caused by Cloudflare Parser Bug," (https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/) Cloudflare.

[17] "Vagrant vs. Terraform," (https://www.vagrantup.com/intro/vs/terraform.html) HashiCorp.

[18] "Security," (https://www.vagrantup.com/docs/share/security.html) HashiCorp.

[19] "Vagrant Connect," (https://www.vagrantup.com/docs/share/connect.html) HashiCorp.

Recommended by the Author

Structuring Application Security Practices and Tools to Support DevOps and DevSecOps (https://www.gartner.com/document/3830131?ref=ddrec&refval=3869263)

The Evolving Effectiveness of Endpoint Protection Solutions (https://www.gartner.com/document/3744126?ref=ddrec&refval=3869263)

A Comparison of UEBA Technologies and Solutions (https://www.gartner.com/document/3645381?ref=ddrec&refval=3869263)

SIEM Technology Assessment (https://www.gartner.com/document/3810941?ref=ddrec&refval=3869263)

Decision Point for Postmodern Security Zones (https://www.gartner.com/document/3798767?ref=ddrec&refval=3869263)

Using DevOps Tools for Infrastructure Automation (https://www.gartner.com/document/3745722?ref=ddrec&refval=3869263)

How to Plan and Architect a DLP Program (https://www.gartner.com/document/3738021?ref=ddrec&refval=3869263)

Assessing Terraform for Provisioning Cloud Infrastructure (https://www.gartner.com/document/3823221?ref=ddrec&refval=3869263)

Recommended For You

Structuring Application Security Practices and Tools to Support DevOps and DevSecOps (https://www.gartner.com/document/3830131?ref=ddrec&refval=3869263)

Container Security — From Image Analysis to Network Segmentation, Options Are Maturing (https://www.gartner.com/document/3888664?ref=ddrec&refval=3869263)

A Guidance Framework for Establishing and Maturing an Application Security Program (https://www.gartner.com/document/3893568?ref=ddrec&refval=3869263)