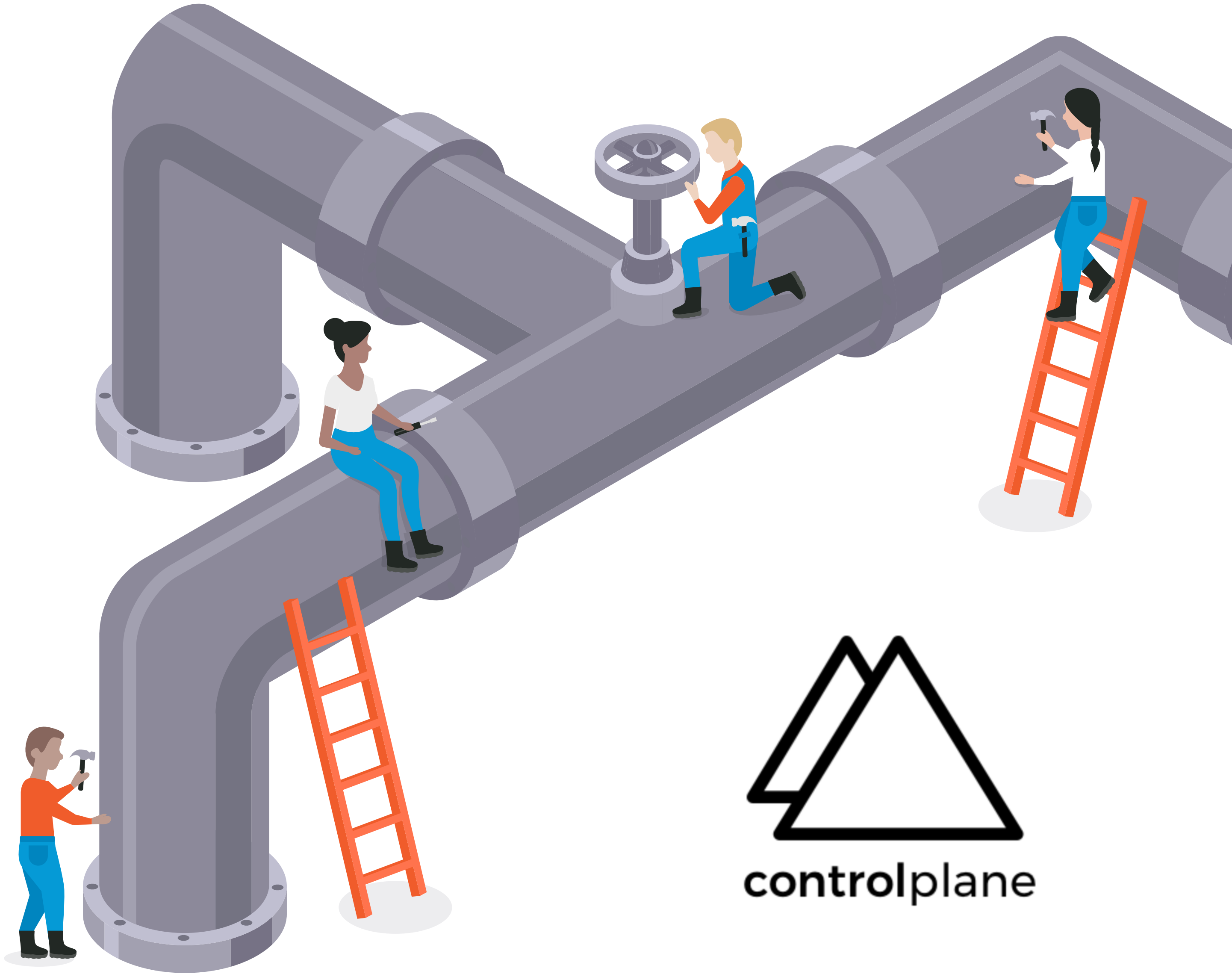
Hardening Git for GitOps



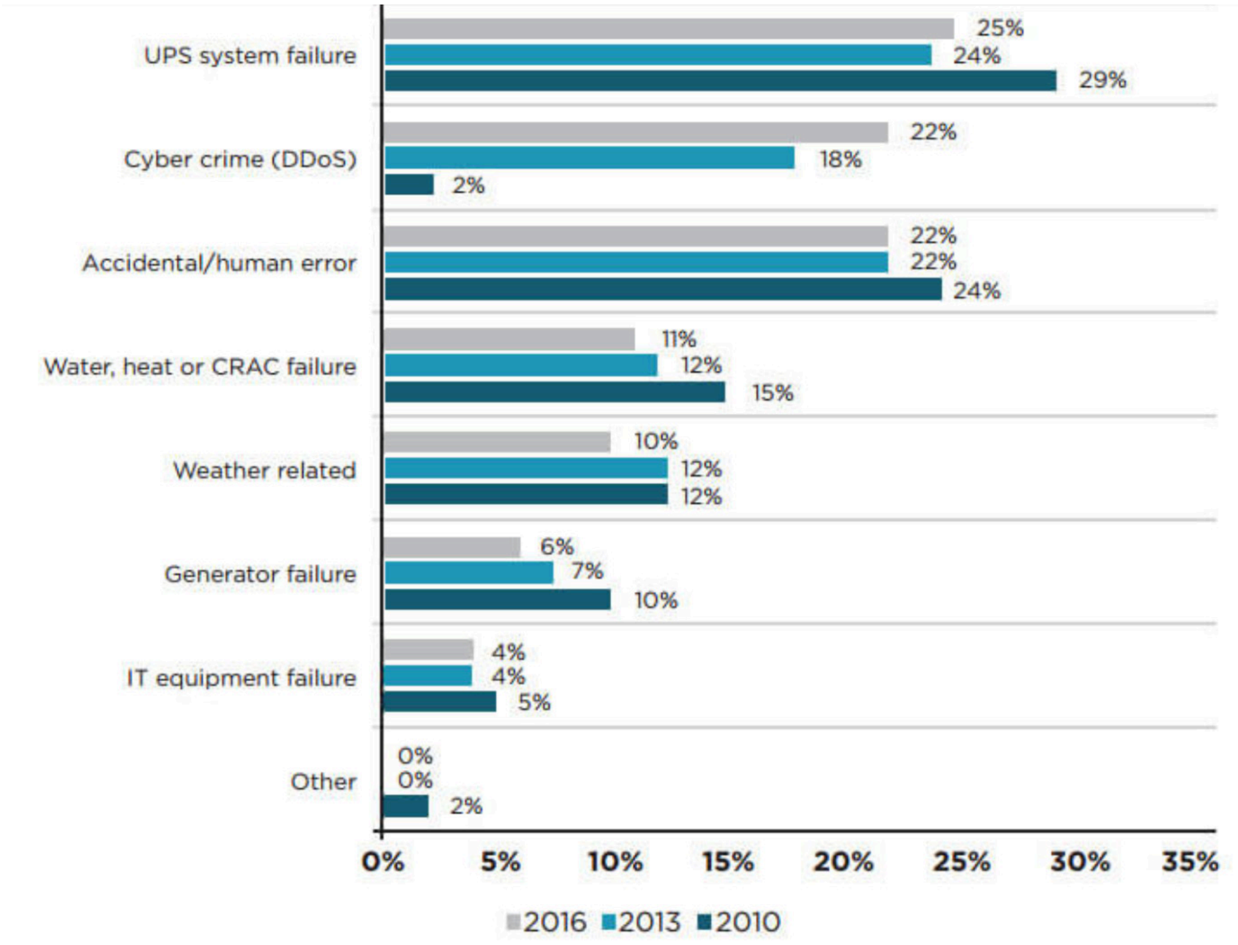
controlplane

weaveworks

INTRODUCTION TO GITOPS

Because cloud native systems like Kubernetes were designed from the beginning with the idea of software development and operations working together, operations tasks and how they get performed are integral components of cloud native architecture and design. Cloud Native systems use declarative constructs that describe how applications are composed, how they interact and how they are managed. This enables a significant increase in the operability, resiliency, agility and portability of modern software systems.

GitOps is an extension of **infrastructure as code** that can be applied to Kubernetes workloads. This means that the configuration of the application - and as much of the system as possible - is stored in **Git** that can be deployed automatically from Git and left untouched by manual operator intervention.



* Source: **https://www.ecmweb.com/power-quality/data-center-outage-costs-continue-rise**

A very human problem

Traditionally, the typical approach to human fallibility is to restrict deployment velocity with restrictive “change control” processes. This slows down the pipeline between developers and production, and presents an organizational and cultural barrier to the adoption of DevOps practices.

Change control is designed to provide a review step where a third party can verify proposed system changes. The reviewer may not have adequate understanding of the technical or complex system changes that often results in a needlessly slow process that provides little value.

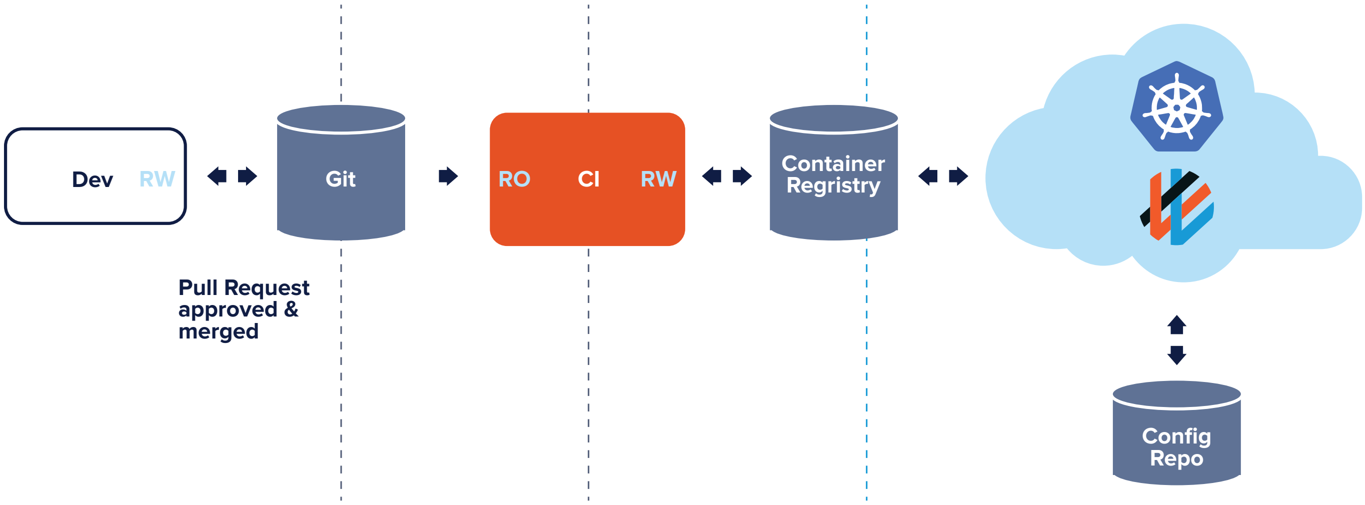
A change control note is not exhaustive and generally contains links to issues, which hopefully contain links to the actual source code changes that are being deployed. Those issues provide no guarantees that they will not be edited or deleted in the future. Given a change manager’s distance from the changes being made to application code, well-meaning change control processes often deteriorate into an audit trail for blame or, at best, risk ownership.

The GitOps Workflow

GitOps is an automation-driven process based on Pull Requests (or PRs). A PR is a machine and human-readable “change control note” that can be automatically applied to existing code in a Git repository. It contains the code to be changed, added, or removed, and some non-code text describing the change.

This does not prevent policy gateways from being used, and most real world GitOps deployments also include approval steps that are manual, for example to request a release manager or security team’s sign off.

GitOps Deployment Workflow



When a developer wants to ship changes to production they prepare a PR and submit it for review. At least one other team member (often two or more) reviews the changes for sanity, veracity, and unintended consequences - this brings the experience of multiple domain specialists together and increases the quality of the change.

If the change is deemed acceptable, then the PR is “merged”. This leaves a trail of users, timestamps, and exact character-by-character changes in Git, which can be reconciled against system state to audit deployments.

If the deployment is unsuccessful, or it succeeds but then fails unexpectedly, the previous configuration of the system can be restored automatically as the GitOps process saves the application’s entire deployment state in its history (application data, such as databases or queues, is explicitly not a part of this process). This also provides a safety net against accidental or malicious interference with the system via an API or console outside of the GitOps process.

These practices increase the stability of the system but do not make it infallible. For organizations that wish to defend themselves from malicious internal actors, or operate under high compliance requirements, there are some additional GitOps practices that can make things more secure.

The Cloud Native Delivery System

GitOps represents an operating model for application deployment, so it is natural for any security professional to ask “what are the threat vectors?”.

In a modern cloud native delivery system, the main moving parts are:

1) The runtime containers (“workloads”) and clusters (including dev and test clusters, but especially production)

2) Mechanisms to deploy or update workloads on a cluster, change their config, secrets, etc.

3) The build pipeline, assumed to be implemented using CI servers, GitHub Actions, or similar

4) Third party components (e.g. containers) and their repos (e.g. Docker Hub)

5) The Git repos used to store application, infrastructure, security, and compliance code, and hosted version control systems (VCS, e.g. GitHub or GitLab)

6) Identity and Access management for the entire system

In this white paper, we focus on the 4 most common threats and demonstrate how the use of Git may be secured using standard signing techniques and software configuration, as well as how other attacks on Git may be mitigated.

How Git Works*

Git operates as a **merkle tree**. This is a tree data structure that begins with the “Initial Commit” - the first entry into the empty Git data repository.

To add anything to a Git repository, changed files are compared with Git’s current state and “committed”. Committed files are bundled together, dated, and cryptographically hashed into the merkle tree. This process means the chain of commits can be trusted to be correct and unmodified.

> The hash function is SHA-1, which was considered broken for security purposes and deprecated by NIST in 2011. As Git’s original author Linus Torvalds pointed out, it does not matter to Git:
>
> SHA-1 as far as git is concerned, isn’t even a security feature. It’s purely a consistency check.

SHA-1 hashing can be manipulated in unintended ways. The **Shattered attack** pads data to collide two SHA-1 hashes, which could be used to overwrite existing trusted commits with malicious code. Git v2.13.0 moved to a hardened SHA-1 implementation which isn’t vulnerable to this attack.

We can consider the current hardened SHA-1 implementation as sufficiently collision-proof for our data consistency needs today. In case of future attacks Git’s hash function is gradually being **upgraded to SHA-256** (a non-trivial process).

Git’s implementation is trusted to be secure, however Git does not provide non-repudiation (that is, proof of authorship or prevention of history editing). On the contrary history manipulation is a core feature of Git: it was built for developers and is not entirely tamper-proof by default.

Because GitOps uses Git as its log format, additional controls must be used around Git to ensure its history is tamper proof. This enables audit of deployments (which should say exactly what was done and by whom) - the history should be provably accurate both when written (authorship and content) and if subsequently rewritten.

* For a deep dive see Ian Miell’s excellent **Learn Git the Hard Way**

ATTACKING GITOPS VIA GIT

This section outlines four of the most common threats in a cloud native delivery system and how to further harden the software supply chain by increasing the veracity and provenance of Git commits to GitOps repositories.

THREAT #1: GIT USERS CAN IMPERSONATE EACH OTHER

Git was not designed with strong identity guarantees. It is possible to author commits that appear to be committed by a different user. When those commits are pushed to a VCS via the CLI, SSH is used as the secure transport protocol, and authorization to the repository is gated by the SSH key on the user's account. However, this does not check that the identity of the user matches the commit message.

Git metadata such as author, email address, and date is editable and intentionally not validated by a remote repository upon receipt. By design, a user can push Git commits authored by users that do not match the email in the account that the SSH key used for authentication to the remote repository is registered to.

This prevents attribution when auditing GitOps deployment file changes. But with hosted services, a user must still authenticate to the platform before the commit is pushed so that there is a trail of identity to follow.

Mitigation: Enforce Strong Identity in VCS (GitHub/GitLab) with GPG Sign Commits

To ensure the identity of a user who's pushed one or more commits, GPG can be used to sign the commit message. With GPG, the author, committer, date information, and a hash of the file changes is included with every commit. This is supported natively by Git, as well as by GitHub and GitLab.

GPG signed commits proves that the user who authenticated with the service and pushed the commit (via SSH, username and password, etc) identified themselves correctly and is authorized by the remote Git server. It also ensures that they own the GPG keys linked with the user account. It still does not guarantee that the metadata is correct, but only that the user who signed the commit, cryptographically identified themselves, and trusted the authors of the commits they pushed.

As the remote Git server requires the user's public GPG keys to be uploaded, it is effectively a Public Key Infrastructure (PKI). Two factor identification (2FA) should be used on the hosted Version Control System's (VCS) web UI to protect the public keys in the users' accounts from stolen passwords and credential stuffing attacks.

* Examples of this can be seen in **these fraudulent commits** from Linus Torvalds. The ability to impersonate other users invalidates any audit guarantees GitOps could provide.

Physical GPG Keys increase security

Physical GPG keys (such as Yubikeys) can be used to increase the resistance of a user to device compromise. These keys work by moving or generating the user's private GPG key to (or on) the USB device. The device must be physically unlocked before the key can be used. This prevents loss of the user's device compromising their private key, and it also increases the resilience of a compromised device to key theft.

Since each commit is itself a cryptographic sum of all previous commits, and hashed together for veracity, the Merkle tree data structure that underpins Git, renders signing every commit technically unnecessary.

> Linus argues that the burden of verifying the signed code should be done when the code is merged into master - by an individual who signs off everything up to that point. "Signing each commit is totally stupid. It just means that you automate it, and you make the signature worthless" (**post**).

This approach means that the signer must understand all changes since the last signed Git commit—many of which may not have been authored by them. This not only makes the burden of merging and release tagging much greater, but it also increases the chance that an individual performing the signing of many commits will trust code that they do not fully understand.

Signatures do provide audit guarantees. Since every commit is signed, we know the identity of the committer not only via their local Git configuration (assuming it's untampered), but more authoritatively through the Public Key Infrastructure (PKI). SSH and GPG keys are distributed and controlled differently by users, pushing the attack vector closer to developers' workstations or towards identity compromise. This calls for a different set of controls, namely four eyes, and a segregation of duties.

Mitigation: Run GPG-Validating Code in CI

Once commits are cryptographically signed, the signatures should be validated.

Checking HEAD or merge commits would be sufficient if we were sure that commits to master were prevented. Checking the signature of a HEAD commit with **git verify-commit**:

```
$ git verify-commit HEAD

gpg: Signature made Fri 13 Jan 2019 12:42:17 GMT

gpg:                using RSA key
DEADB335F3BB2EC440A516D4C7B9EA6DD705D5F09

gpg: Good signature from “Andrew Martin <andy@control-plane.io>”
[ultimate]
```

The alternative - checking every commit - can require force push access if the history is polluted (as this will fail if any signatures are invalid).

This **check-commit-signature** script by **@isislovecruft** is an example of validating a repo’s signatures.

GitHub and GitLab can also **enforce signed commits**, but this does not guarantee the user’s key has not been revoked. As a best practice, a user should remove the key from their profile if they revoke it.

Keybase can also be used as public key infrastructure, but this does not have native integration into hosted VCS providers.

When merging PRs through the web UI, GitHub (although currently **not GitHub Enterprise**) will also add its own signed merge commit using **its own key** (for merge commits only - squash and rebase commits lose the GPG signatures of the commits. They compress as the matching commit hashes have changed.).

THREAT #2: MALICIOUS USER REWRITES HISTORY

Rewriting a common, shared history is useful for private forks and non-critical branches, but when a GitOps deployment branch is rewritten, this can erase valuable history. Overwriting a remote repository is called a "force push". To determine whether it is preferable to disable the force push feature, we should consider when it might be used.

When to use Force Push

A common use of a "force push" is to remove a commit containing plaintext credentials that have been mistakenly pushed to a shared repository. This could be perceived to reduce the attack window, but that implies that there is a security benefit in removing compromised credentials. Identifying whether the credentials have been viewed or pulled may be possible by monitoring the Git server access logs, but the time to search the centralized logs is similar to the speed of rotating the credentials for any other breach response.

Once credentials are known to be leaked they should be considered compromised and rotated. There is no benefit in rewriting the shared Git history to remove them. Removing a commit from Git does not immediately expunge it. Since Git's internal data storage isn't garbage collected until `git gc` is run, removed or orphaned commits may languish on file systems long after they have been removed from the repository.

Force push commits are still useful to a developer when cleaning up a personal branch's history before a pull request, and can be safely permitted on branches other than master or dev.

The danger for a GitOps workflow is to branches that trigger a deployment. If historical commits on those branches are rewritten, the audit trail is lost, and so they should be protected.

Mitigation: Prevent Force Pushes to Master Branch

The "protected branches" feature of hosted VCSs can be configured to prevent force pushes to a branch.

"Rebased" commits (those that rewrite history) cannot be prevented in Git itself, but both GitHub and GitLab provide force push protection for specified branches to prevent historical commits being rewritten. This can be automated with the GitHub Terraform Provider (for repositories under Terraform's control) or **pepper** (which sets all a GitHub organization's master branches to protected).

These tools should be run on a build server dedicated to security jobs, or a controlled task-running environment.

The administrators of Git are still able to disable force push prevention, but as this is rarely necessary should raise an alert. "Break Glass" procedures are encouraged to continue to commit changes via the standard pull request model instead of directly to the relevant repository, to take advantage of pipeline failsafe measures such as deployment, test, and verification checks.

Mitigation: Backup Git Repositories

If the Git server is hosted within an organization's infrastructure, ensure well-tested and frequent backups in case of malicious activity such as a history rewrite. Backups should be stored in a remote system with audit logging, and different administrators from those in charge of Git. This ensures that a malicious user cannot tamper with all copies of a repository.

THREAT #3: MALICIOUS USER REMOVES SECURITY FEATURES

A Git repo that has been tampered with renders the auditability of changes useless. In other words, if it cannot be guaranteed that the repo is in its original state, we cannot trust its contents.

Preventing force commits and enforcing GPG signature validation are options that an administrator of the hosted VCS can configure. However in the event that the administrator's account is compromised, further mitigation may require validating both the VCS's configuration, and the checks it performs, in duplicate, on a remote build server.

Mitigation: Configure Git Provider with Infrastructure as Code

Do not permit admins to manually change the hosted VCS configuration, and configure repositories and security permissions via the **Terraform providers** or its APIs.

When the configuration of the Git provider is held externally as code, unauthorized changes can be quickly detected and rolled back from a build server. This configuration-as-code approach confers the same benefits as GitOps, including changelogs, audit trails, and accountability, but applied to a remote system instead of an application configuration.

Mitigation: Monitor Git Provider's Audit Logs

Alerts should be raised on changes in the audit logs, and a build server job should continuously validate the state of the protected branches.

Depending on the size and structure of the organization, segregation of duties can be enforced between development teams in the hosted VCS and the team of administrators managing security settings on these services. Role Based Access Control (RBAC) permissions can be configured that enforce the least privilege. Alerts and audits in more evolved organizations can be handled by a dedicated Security Operations Center (SOC), or in smaller organization by a rotating group of administrators.

GitHub Enterprise supports log forwarding to **Logstash or Splunk**. Gitlab Enterprise's audit logs **do not currently support protected branch events.**

Mitigation: Verify Commits to Master

Git users should not be able to merge their own pull requests. This is a feature that is configurable on a per-branch basis with protected branches.

An administrator may temporarily allow commits to master that can insert a change and will run a validation on the build server to ensure that all commits to master also exist on a branch, originated from a merge from another branch, and are GPG signed with valid signatures from authorized members of the team.

This type of verification may be restricted to a short window of time before the source branch is deleted using git branch `--contains,` although `git name-rev` and `git log --ancestry-path --merges` both also provide historical context for a given commit.

This additional layer of validation defends against misconfiguration or accidents on the hosted VCS.

THREAT #4: OLD GIT CLIENT VERSIONS ARE INSECURE

Attacks on the Git protocol have been disclosed and fixed in previous versions. The fixes are backwards compatible, so that only newer clients are protected.

In **"On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re) introduces Software Vulnerabilities"** (Torres et al, 2016) a Metadata Manipulation Attack Taxonomy is proposed:

TELEPORT ATTACKS	
Branch Teleport Attack	A branch merge point is moved to point to a "WIP" or to a buggy code commit. This gets automerged on a developer's pull.
Tag Teleport Attack	A tag pointer is moved to a different place in the history and the wrong version is retrieved. For example to a previous version with known vulnerabilities.
ROLLBACK ATTACKS	
Branch Rollback Attack	Critical code is omitted.
Global Rollback Attack	Critical code is omitted.
Effort Duplication Attack	Coding effort is increased.
DELETION ATTACKS	
Branch Deletion Attack	A branch is missing.
Tag Deletion Attack	A tag is deleted.

Attacks on compromised repositories can rollback Git's internal pointers to serve vulnerable code versions. The Continuous Delivery "fix forward" mentality suggests that a GitOps repo should only ever be used at the tip of each branch, or HEAD depending on the repo configuration. If older revisions are returned, prior resource versions with known vulnerabilities may be transparently and inadvertently deployed.

This may be inconvenient when using a tag to define the last known good state of a repository during the deployment phase, so versions of Git that are not vulnerable to these attacks must be used.

Mitigation: Keep Software Versions Updated

Keep Git client versions updated.

Libraries that implement the Git protocol, but are not distributed by the Git project should be subject to particular scrutiny for the classes of attacks described above.

FURTHER PIPELINE HARDENING AND CONSIDERATIONS

An organization may want to restrict deployments to working hours, prevent deployments to production on a Friday afternoon, or enforce security teams to review certain changes. This can be applied to the Pull Request (to prevent the merge occurring) or to the build server itself.

Permitting only dedicated reviewers to merge Pull Requests can act as a change control or security review gate, with automatically generated release notes reflecting the application changes that are about to be deployed and that provide a clear picture of the changes.

Static analysis can also be performed on the contents of the Pull Requests themselves. Running tools such as **kubesec.io** (risk scores for Kubernetes resources) or **kubetest** (a Kubernetes resource unit test framework) as well as the YAML manifests also offers an insight into the contents of the changes being pushed. Changes that affect specific resource types (for example, `PodSecurityPolicies` or `NetworkPolicies`) can be tagged for review by the relevant teams (e.g. network and security engineers).

As part of a wider security strategy, tools like **Notary**, **Grafeas**, and **in-toto** prevent old Docker images from being deployed, as do Kubernetes cluster admission controllers such as **Kritis** and **Porteiris**.

Deploying and Operating Kubernetes in Production

Weaveworks and ControlPlane can provide developers and DevOps teams with the tools, skills and knowledge they need to successfully develop and operate modern and secure cloud native applications. Both of our teams are experts in Docker, Kubernetes, microservices and distributed systems.

We can help you architect, install, audit and secure Kubernetes clusters using cloud native technologies for cloud or on-prem. Weaveworks and ControlPlane can deliver use, administrator, and security training for both beginners and experienced professionals. We can also help with cloud native architecture consulting, design and compliance to get your Kubernetes platform enterprise ready.

**Contact us
for more details**