GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Network Monitoring and Threat Detection In-Depth (Security 503)"
at http://www.giac.org/registration/gcia

Defending Infrastructure as Code in GitHub Enterprise

GIAC (GCIA) Gold

Author: Dane Stuckey
Advisor: Johannes Ullrich

Accepted: 16 September 2019

Abstract

As infrastructure workloads have changed, cloud workflows have been adopted, and elastic provisioning and de-provisioning have become standard, manual processes. As a result, semi-automated infrastructure management workflows have proven insufficient. One of the most widely implemented solutions to these problems has been the adoption of declarative infrastructure as code, a philosophy and set of tools which use machine-readable files that declare the desired state of infrastructure. Unfortunately, infrastructure as code has introduced new attack surfaces and techniques that traditional network defense controls may not adequately cover or account for. This paper examines a common deployment of infrastructure as code via GitHub Enterprise and HashiCorp Terraform, explores an attack scenario, examines attacker tradecraft within the context of the MITRE ATT&CK framework, and makes recommendations for defensive controls and intrusion detection techniques.

© 2020 The SANS Institute, Author Retains Full Rights

© 2020 The SANS Institute Author retains full rights.

1. Introduction to Infrastructure as Code

Declarative infrastructure as code is a philosophy and set of tools which use machine-readable files to define the desired state of infrastructure. Typically, these machine-readable files are stored in a version control system (e.g. git) which can be collaboratively modified and reviewed by developer operations (DevOps) teams. Infrastructure as code allows organizations to perform infrastructure changes, reduce risk, and scale operations without relying on manual or semi-automated workflows.

A basic infrastructure as code deployment consists of a code repository and version control system to manage infrastructure as code definition files as well as an execution engine for implementing the infrastructure changes. Infrastructure as code typically relies on public or private cloud infrastructure (e.g. Amazon AWS, Microsoft Azure) versus bare-metal infrastructure. Using cloud infrastructure allows a DevOps team to quickly scale horizontally or vertically through code changes, obviating the need for additional hardware acquisition or datacenter expansion.

In the enterprise sector, GitHub Enterprise and HashiCorp Terraform are used in infrastructure as code workloads. A high-level diagram of this infrastructure as code workflow is shown in Figure 1 below.

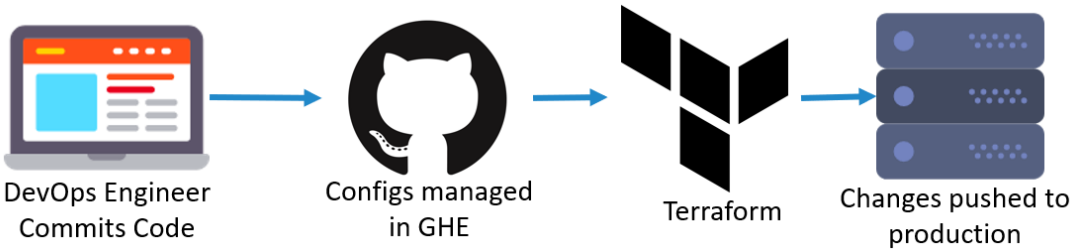


Figure 1 – IAC Deployment with GitHub Enterprise and Terraform

In this infrastructure as code deployment model, DevOps engineers make changes to infrastructure files and commit them to the code repository (GitHub Enterprise). A pull-request is opened by the engineer and merged to a master branch. Upon merging, the execution engine (Terraform) reads the infrastructure files and makes appropriate changes to production infrastructure.

Dane Stuckey

© 2020 The SANS Institute Author retains full rights.

Management of infrastructure through code introduces multiple new security challenges which must be considered by enterprise defenders. Legacy infrastructure deployment models have focused on tiers and silos of infrastructure to prevent complete compromise of an environment. A commonly cited example of an infrastructure tiering model is the Microsoft Active Directory Administrative Tiering Model (Microsoft, 2019) which is shown in Figure 2 below.

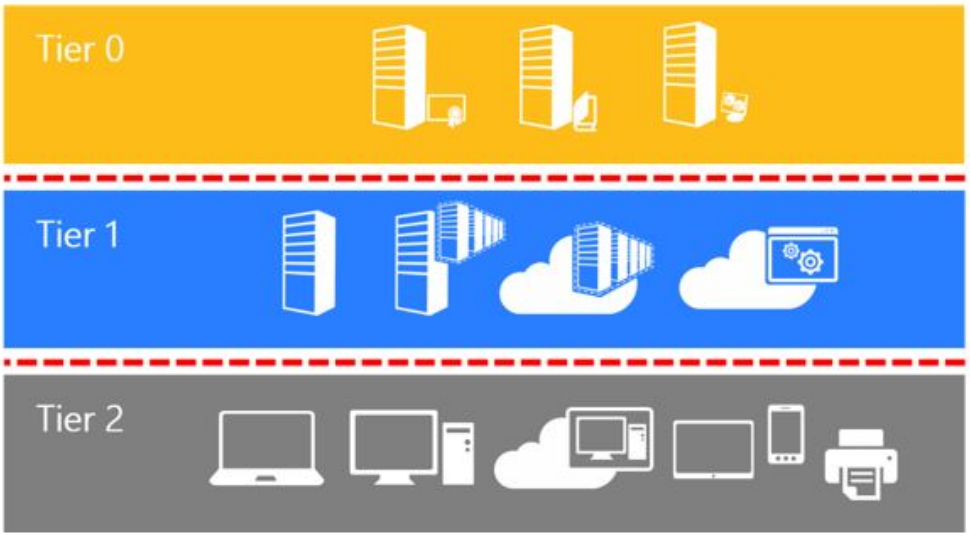


Figure 2 – Microsoft Tiering Model (Microsoft, 2019).

In the Microsoft Active Tiering Model, systems and management accounts are distinctly segregated to prevent movement between systems. In the event of a workstation compromise (Tier 2), an attacker would not have the appropriate rights or accesses to modify critical business systems (Tier 1) or management/identity systems (Tier 0). One of the most effective controls in this tiering model is the implementation of administrative control restrictions, which is shown in Figure 3 below.

© 2020 The SANS Institute Author retains full rights.

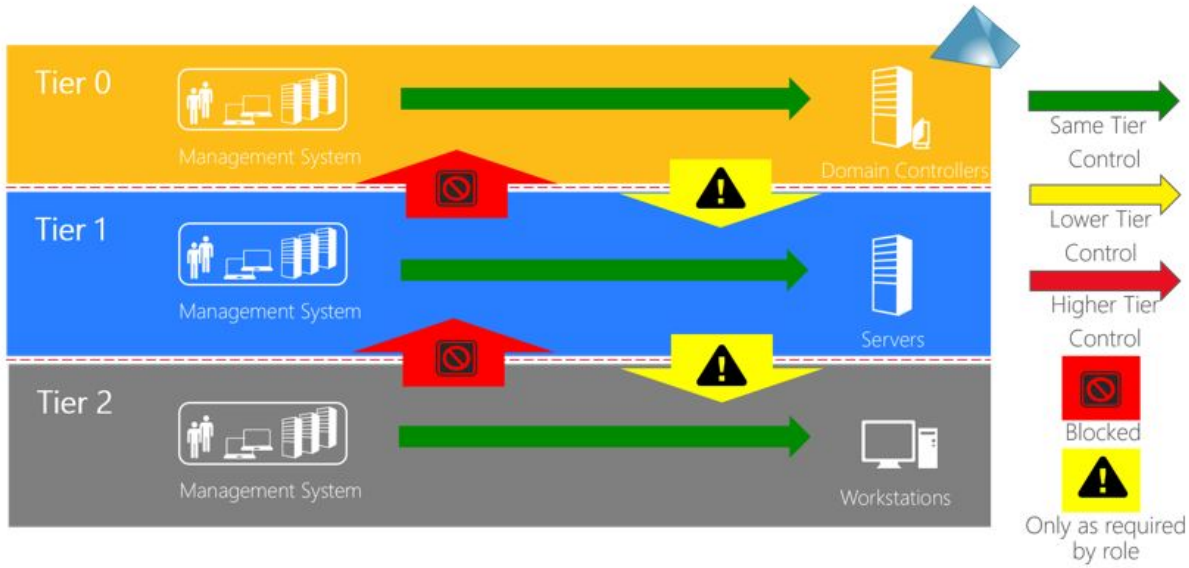


Figure 3 – Microsoft Control Restrictions (Microsoft, 2019).

Administrative controls restrictions in the Active Directory Tiering Model create explicit trust boundaries for each tier. A workstation administrator (Tier 2) must use a special account, device and independent management systems for performing their duties. There is no trust relationship between the administrator account or management systems from one tier and any other tier. While this imposes substantial friction for administration (e.g. multiple user accounts, multiple administrator devices, independent management systems), there are hard dividers between tiers which makes escalation incredibly difficult (Microsoft, 2019).

In the infrastructure as code world, this model typically breaks down completely. Whereas three independent management systems and administrator accounts were required to manage all tiers in the Microsoft model, infrastructure as code workflows typically manage assets of all tiers using a set of repositories stored in the same code repository which is accessible using a single account. In many organizations, management of virtual desktop infrastructure (Tier 2), business-critical systems (Tier 1), and domain controllers (Tier 0) may happen using the same user account, GitHub Enterprise code repository, and Terraform instance. This ultimately means that an attacker, if they can compromise a DevOps member's GitHub Enterprise account, can attack infrastructure across all three tiers without traditional exploitation or escalation techniques.

© 2020 The SANS Institute Author retains full rights.

2. GitHub Enterprise

Before delving into offensive and defensive techniques used in infrastructure as code, it is important to analyze the various components and primitives used in GitHub Enterprise. Many of these techniques require a nuanced and technical understanding of how GitHub Enterprise operates under the hood, typical workflow patterns for infrastructure as code, and oversights or misconfigurations that allow an adversary to successfully perform an offensive operation.

2.1. GitHub Enterprise Primitives

There are several primitives and concepts which apply to GitHub Enterprise and must be well-understood by network defenders. These primitives are outlined in Table 1 below. While many of these primitives are borrowed from the underlying git software version control system, they are contextualized for usage within a GitHub Enterprise instance (GitHub, 2019).

Name	Description
Organization	A shared space within GitHub Enterprise. An organization can have one or more repositories, have granular security and administrative settings, and have members invited to participate as members of a team.
Repository	The most basic element of code storage within GitHub Enterprise. A repository contains project information, files, code, and version history. A repository can be public or private and can have granular security and administrative settings applied to it.
Team	A grouping of individual users within GitHub Enterprise. Teams can be members of organizations and can have individual permissions and security settings applied to them.
Collaborator	An individual invited to collaborate on a repository. A collaborator can be given granular security permissions on a repository.
Commit	Changes to one or more files within a repository that are saved as a unique record. To change IOC in a repository, users will commit file modifications to a repository.
Branch	A branch represents a parallel copy of a repository that can be edited independently from other branches. When production changes are desired, branches will typically be merged into a single branch (usually master). This is usually performed using a pull request.

Table 1 – GitHub Enterprise Primitives.

2.2. Authentication and Authorization

GitHub Enterprise natively supports multiple authentication mechanisms (GitHub, 2019):

© 2020 The SANS Institute Author retains full rights.

- Integrated (built-in) authentication. (default)
- Central Authentication Service (CAS) – Single Sign-On
- Security Assertion Markup Language (SAML) – Single Sign-On
- Lightweight Directory Access Protocol (LDAP)

Users present outside of the selected identity provider may optionally be granted access using the integration (built-in) authentication provider. This can allow non-organizational members to be invited to the GitHub Enterprise instance as collaborators. Additionally, GitHub Enterprise can be configured to allow for unauthenticated read access for repositories. This configuration can be controlled by a GitHub Enterprise administrator.

GitHub Enterprise additionally supports optional or enforced multi-factor authentication for users. When using integrated (built-in) or LDAP authentication, GitHub Enterprise can act as a multi-factor authentication provider for new sessions. A GitHub Enterprise administrator can set site-wide multi-factor authentication enforcement to guarantee all users have enrolled. It is important to note that multi-factor authentication within GitHub Enterprise is only offered for built-in or LDAP authentications. External identity providers using CAS or SAML must enable multi-factor authentication outside of GitHub Enterprise.

Many non-web workflows will require the use of either basic authentication (HTTPS), OAuth or Personal Access Tokens (HTTPS with multi-factor authentication enabled) or SSH or deploy keys. The most common model relies on using SSH keys and the SSH transport protocol for working with a remote GitHub Enterprise server.

2.3. Auditing and Logging

Native audit logging is present in GitHub Enterprise and can be accessed on the server under /var/log/github/audit.log. The audit log file is rotated daily with seven days of retention by default (GitHub, 2019). These logs contain all pushes, pulls, and a variety of additional audited actions that have security significance. By default, GitHub Enterprise logs all push operations performed. This information includes the following:

- The user who initiated the push request.
- If the push was marked as a force push or not.
- The branch the push affected.
- The protocol used to push (e.g. SSH or TLS)
- The originating IP address of the request.

© 2020 The SANS Institute Author retains full rights.

Additionally, other security-relevant actions and events will be logged without further configuration. A summarized list of audited actions can be found in Appendix A.

For detection and investigation use cases, network defenders should capture and integrate these logs directly into their security information and event management (SIEM) system with a retention period greater than the industry dwell time for incident detection.

2.4. Repository Permissions

GitHub Enterprise has a granular role-based access control model which allows for delegation and assignment of rights for individuals, teams, and collaborators on a per-repository basis. GitHub Enterprise has the following access roles which may be applied to users, teams, deploy keys, or collaborators on a given repository (GitHub, 2019):

- **Read**: Allows read-only access to the repository.
- **Write**: Allows for writes, pull requests, status check creation, and other actions.
- **Admin**: Allows for writes, changes to repository security controls, protected branches, and other actions with strong security significance.

While many organizations may assign the admin role to their repository contributors or users for ease of administration and reduced friction, there are substantial security implications to doing so. Most notably, any security checks enforced on branches (e.g. branch protection, mandatory pull request review) may be disabled or overridden by a user with the admin role.

2.5. Pull Request Workflows

Pull requests are a common workflow for collaboratively merging changes to a repository in GitHub Enterprise. A pull request consists of a request to merge changes made on one branch of a repository (e.g. development) onto another branch (e.g. master). A pull request can allow for collaboration, discussion, commenting, human code review, and a variety of checks to occur before a merge is approved. Pull requests allow multiple users to independently develop, propose their changes, incite discussion, and handle any conflicts before affecting a production branch. Any user with write access to a repository may open pull requests with proposed changes. Additionally, protected branch rules may be used to require pull requests for modifications to specific branches. This process is visualized in Figure 4 below.

© 2020 The SANS Institute Author retains full rights.

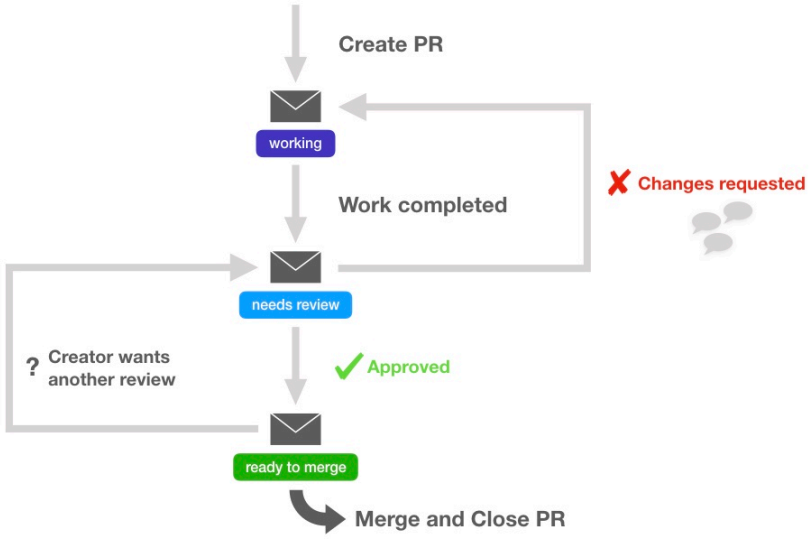


Figure 4 – Pull Request Workflow (Rose, 2019).

Pull requests may be used as both a quality and security control. Protected branches, which will be discussed in more detail in the next section, can require a pull request review before allowing a merge to complete. This can introduce a second human reviewer into the equation who performs a review of all changes made to the repository and must explicitly sign off on any new pull request. In the event a malicious actor attempted to introduce a malicious change into a repository, a pull request review may lead to detection through human scrutiny of the changes.

2.6. Protected Branches

Protected branches allow for the creation of security rules which can enforce certain workflows to occur based upon changes to one or more branches within a given repository. Protected branches can be configured by both the owner of and any user with the admin role within a repository. Figure 5 below shows the options available for protected branches on a given repository.

© 2020 The SANS Institute Author retains full rights.

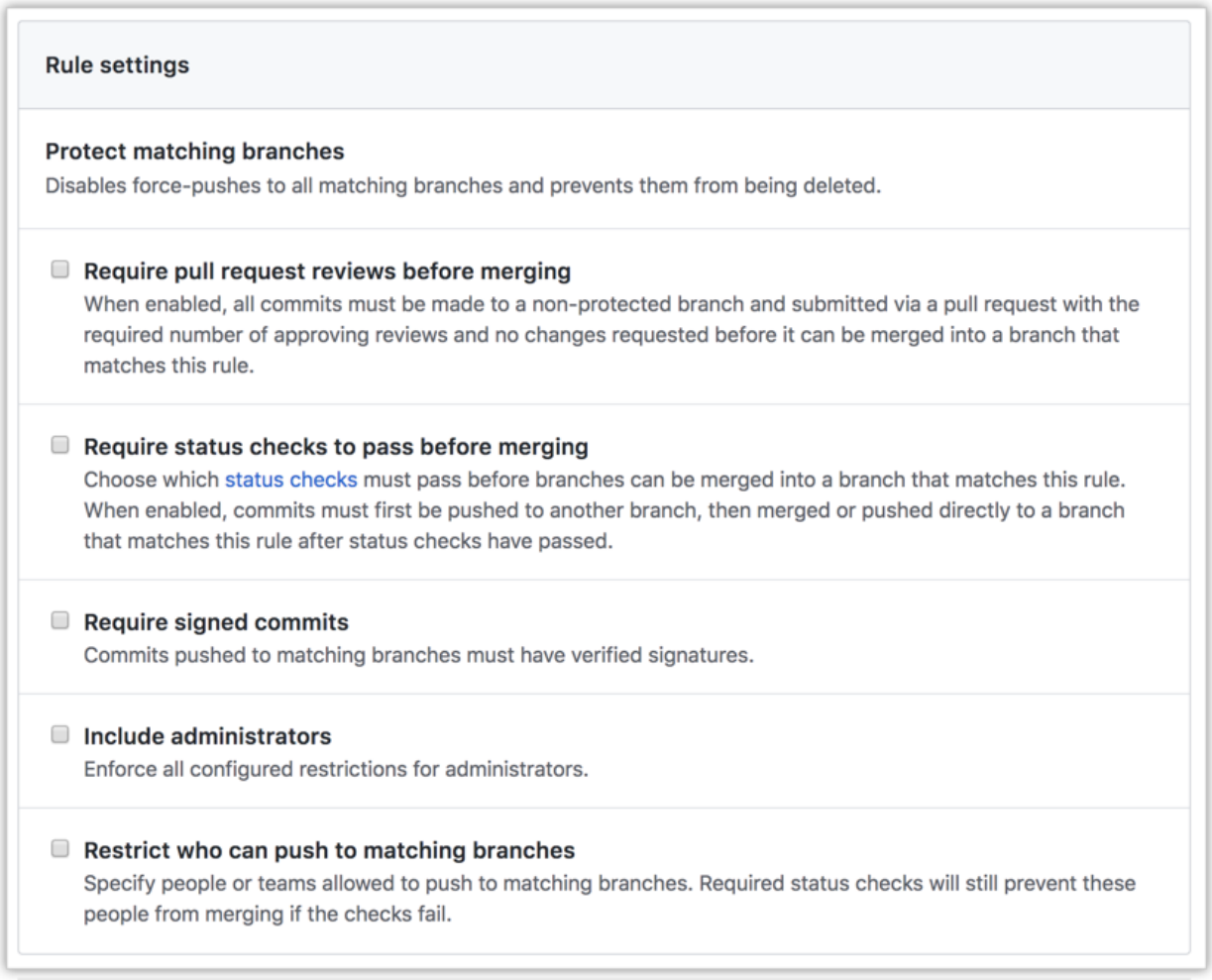


Figure 5 – Protected Branch Options (GitHub, 2019).

One benefit of marking branches as protected is that it affords protection against force pushes. A force push is a destructive action which will unconditionally overwrite the remote branch for repository with the state of the local branch. If there was a conflict between a user's local branch and the remote branch, they may be incentivized to perform a force push which could have disastrous consequences for the integrity of the repository. Protected branches also provide a secondary benefit of preventing accidental deletion of the branch.

2.7. Status Checks

Status checks are optional processes, typically based on integration with external systems, which will run against any push made to a repository. Status checks provide integration points for continuous integration systems to perform testing, validation, and security. While there

© 2020 The SANS Institute Author retains full rights.

are no status checks configured by default in GitHub Enterprise, they may be added by any user with write permissions to a repository. Additionally, successful status checks can be mandated before merging if configured as a protected branch rule.

There are two distinct types of status checks available in GitHub Enterprise: checks and statuses. Statuses are the simplest type of status check available. A status check allows for an external integration to view commit data and return a preconfigured state: error, failure, pending, or success. An example of a status integration would be an external continuous integration system (e.g. CircleCI) where the commit is run through a full build. When complete, CircleCI will return the overall exit code of the build (e.g. failure or success).

Checks are a more fully featured status check that does not rely on pre-configured build states and tightly integrates with GitHub Apps. For each commit made to a repository with checks enabled, a message is broadcast to all GitHub Apps configured on the repository. The GitHub apps receive the notification and, if applicable, run their code against the content of the commit. Checks allow for more granular linting, annotation, and integration within GitHub Enterprise than statuses. An example of a check would be an integration that performs linting of a programming language. With each commit, a python linter GitHub application receives a notification and validates that the submitted python code meets appropriate standards. If there is an issue detected, it will notify the author by annotating the incorrect line and offering a prompt to fix the issue. When a pull request is submitted, the configured status checks are run. Figure 6 below shows two status checks which both passed:

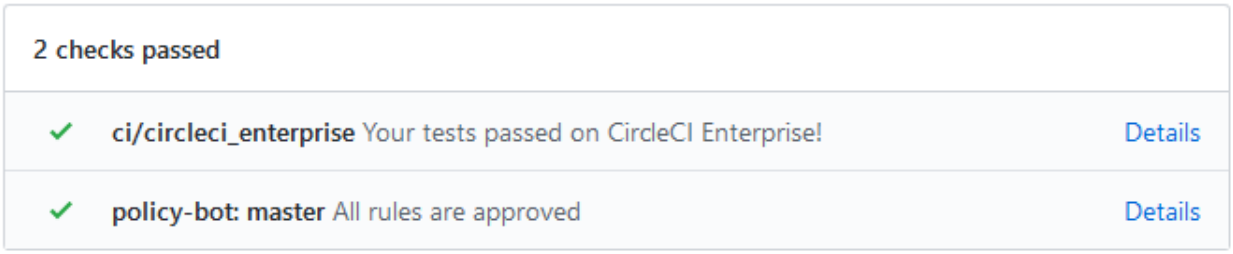


Figure 6 – Successful Status Checks.

While status checks may provide security controls for a given repository, there are some important limitations which must be acknowledged. As noted in the GitHub Enterprise documentation, “Anyone with write permissions to a repository can set the state for any status

© 2020 The SANS Institute Author retains full rights.

check in the repository" (GitHub, 2019). In a hypothetical example, a repository may be configured with a status check which looks for malicious strings or content within a commit. If there are malicious strings or content, the status check returns failed, otherwise, the check passes. If a malicious actor commits an offending string that fails the status check, they can write a successful message via the status API to unblock their pull request. This has substantial implications if an organization relies on status checks for security controls.

2.8. Pre-Receive Hooks

Pre-receive hooks are a rudimentary form of status checks that run a script locally on the GitHub Enterprise server and return a Boolean status for a given commit: accepted or rejected. Pre-receive hooks can be used to perform testing, validation, and security.

An example of a pre-receive hook is a script which looks for regular expression patterns in the content of a commit. If a number is observed in a commit which matches the regular expression for a U.S. social security number, the pre-receive hook script exits with an error code of 1, passing a rejected message back to the GitHub Enterprise server. As a result, the commit is rejected by the server.

Unlike status checks, pre-receive hooks do not allow for arbitrary status updates by users and only accept output from the script running locally on the host. This provides significantly stronger security protections for the checks on protected branches at the cost of performance considerations for the server. As each pre-receive hook script runs locally on the GitHub Enterprise server, there are significant stability and performance risks to consider.

3. Attack Scenario

This paper examines a common deployment of infrastructure as code via GitHub Enterprise and HashiCorp Terraform, explores an attack scenario, examines attacker tradecraft within the context of the MITRE ATT&CK framework, and makes recommendations for defensive controls and intrusion detection techniques.

For the attack scenario that will be examined, GitHub Enterprise v2.18.1 will run on a virtualized Linux server. Primary authentication occurs via a security assertion markup language (SAML) single sign-on (SSO) provider with secondary authentication using the Duo Security software. A sensitive repository, known as goldmine, manages Azure-based infrastructure and

© 2020 The SANS Institute Author retains full rights.

stores Terraform configuration files. Commits to the master branch of goldmine automatically apply to production infrastructure via Terraform automation. To protect against malicious commits to master, the master branch does not allow direct pushes, and additionally requires a pull request from another branch with a mandatory review by an employee. Users interact with GitHub Enterprise via the web GUI and through deploy and SSH keys. Figure 7 below shows the typical workflow for making changes to the goldmine-managed infrastructure by a user.

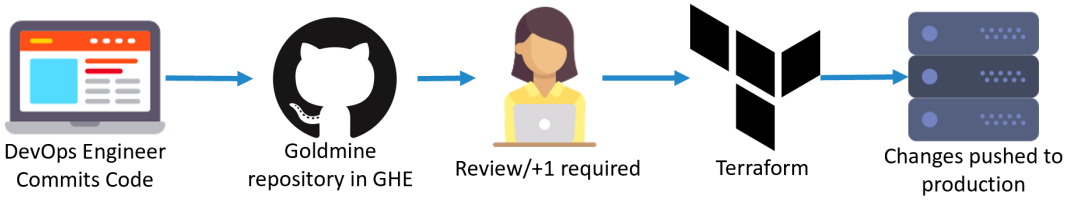


Figure 7 – Workflow for Goldmine Infrastructure Changes.

In this scenario, there are several weak points that the adversary may attempt to exploit. Firstly, management of the sensitive tier-1 infrastructure uses a normal, unprivileged user account. Secondly, the goldmine repository uses some security controls, including role-based access control (RBAC) and mandatory pull request reviews. However, the GitHub Enterprise administrators have not enabled several security-critical features and configurations which an adversary may exploit. Lastly, any infrastructure changes committed to master are automatically applied by Terraform. These security considerations are shown below in Figure 8.

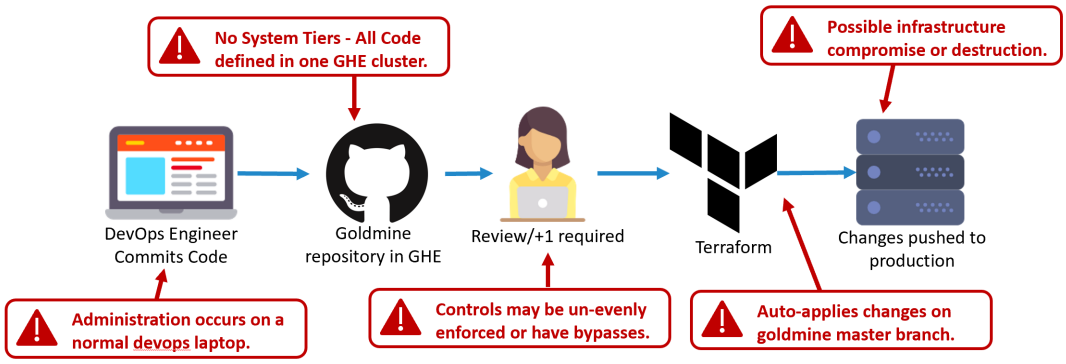


Figure 8 – Security Considerations for Goldmine Infrastructure Workflows.

© 2020 The SANS Institute Author retains full rights.

4. Offensive and Defensive Techniques

The offensive techniques used within the GitHub Enterprise instance were then mapped against the MITRE ATT&CK Framework. The ATT&CK Framework is "a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations" (MITRE, 2019) and is a standard framework used by network defenders. While discrete techniques for GitHub Enterprise do not exist within the ATT&CK Framework at the time of this publication, best-fitting categories were selected Each offensive technique identified was annotated with both a high-level technique in addition to a lower-level, more specific technique.

4.1.1. Third Party Software – IAC Repository Compromise (T1072)

Description: A malicious actor, with access to a repository containing infrastructure as code configuration can perform malicious modifications. This could result in unauthorized arbitrary code execution across the infrastructure managed within the GitHub Enterprise instance. This technique is highly dependent on the security controls configured for the repository and the nature of the infrastructure as code deployment. In the most rudimentary scenarios, an adversary may introduce a malicious package as part of a packer or Amazon Machine Image (AMI) build. More advanced techniques may include modifying legitimate scripts to load executable code, using backdoor user accounts, or performing malicious actions directly against the infrastructure (e.g. destruction).

Proactive Hardening: Due to the size, scope, and complexity of this technique, proactive hardening steps are outlined in Section 5 (Additional Defensive Recommendations).

Detection Strategies: Due to the size, scope, and complexity of this technique, detection strategies are outlined in each of the other MITRE ATT&CK techniques.

4.1.2. Account Manipulation – User Personal Access Token (T1098)

Description: A malicious actor, with access to an interactive web session for a user on GitHub Enterprise, can generate a long-lived personal access token. This token can be used in place of a password for access over HTTPS or the API, bypassing multi-factor authentication on subsequent connections. This token can be scoped to have near-full control over repositories, organizational settings, GPG keys, and other security-critical controls. Figure 9 below shows the personal access token generation page.

© 2020 The SANS Institute Author retains full rights.

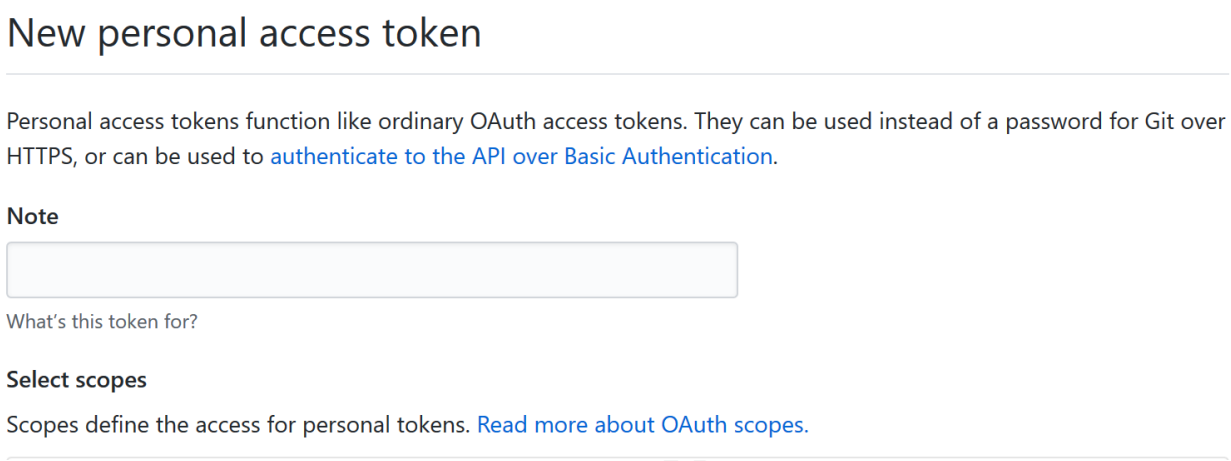


Figure 9 – Personal Access Token Generation.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as an `authentication` event with type `oauth_access.create`. Develop and implement alerting for creation of tokens for sensitive or privileged accounts.
Native Alerting	GitHub will natively alert users with an e-mail notification when a new token has been created. This will include the name and scope. The user may visit GitHub Enterprise to revoke any unusual tokens.

4.1.3. Account Manipulation – User SSH Key (T1098)

Description: A malicious actor, with access to an interactive web session for a user on GitHub Enterprise, can associate an SSH public key with the user. This token can be used in place of a password for access over SSH, bypassing multi-factor authentication on subsequent connections. This key provides access to all public and private repositories to the user, with the equivalent permissions granted to the user. Figure 10 below shows the SSH key association page.

© 2020 The SANS Institute Author retains full rights.

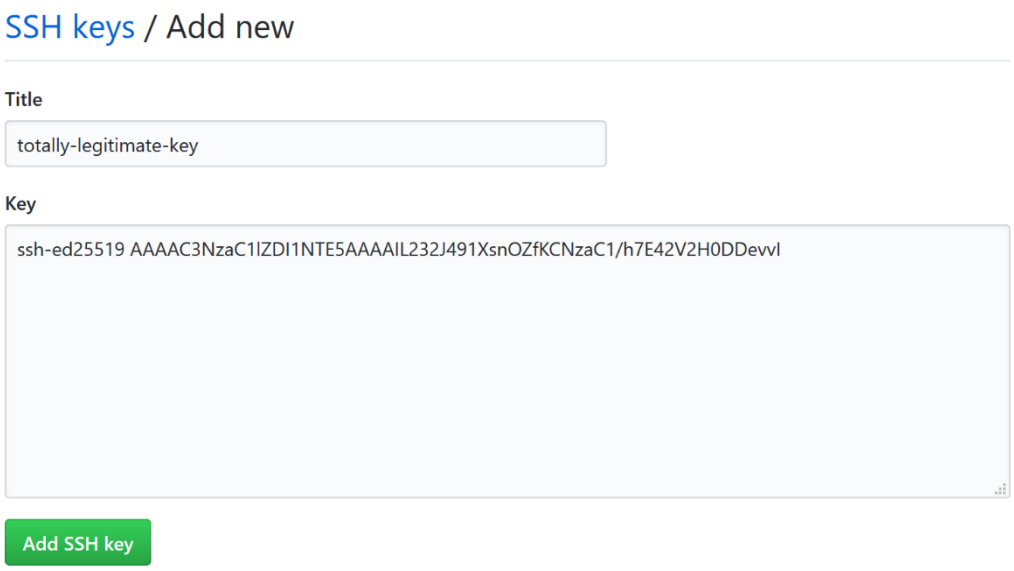


Figure 10 – Adding an SSH Key.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as an `authentication` event with type `public_key.create` or `public_key.update`. Develop and implement alerting for creation of new public keys for sensitive or privileged accounts.
Native Alerting	GitHub will natively alert users with an e-mail notification when a new SSH key has been associated. This will include the key name and fingerprint. The user may visit GitHub Enterprise to delete any key associations.

4.1.4. Account Manipulation – Repository Deploy Key (T1098)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, can create a deploy key association for a repository they have administrative rights over. This allows an SSH key to have read-only or read-write access to the repository. This key can be used in place of a password for access over SSH, bypassing multi-factor authentication on subsequent connections. Figure 11 below shows the deploy key creation page.

© 2020 The SANS Institute Author retains full rights.

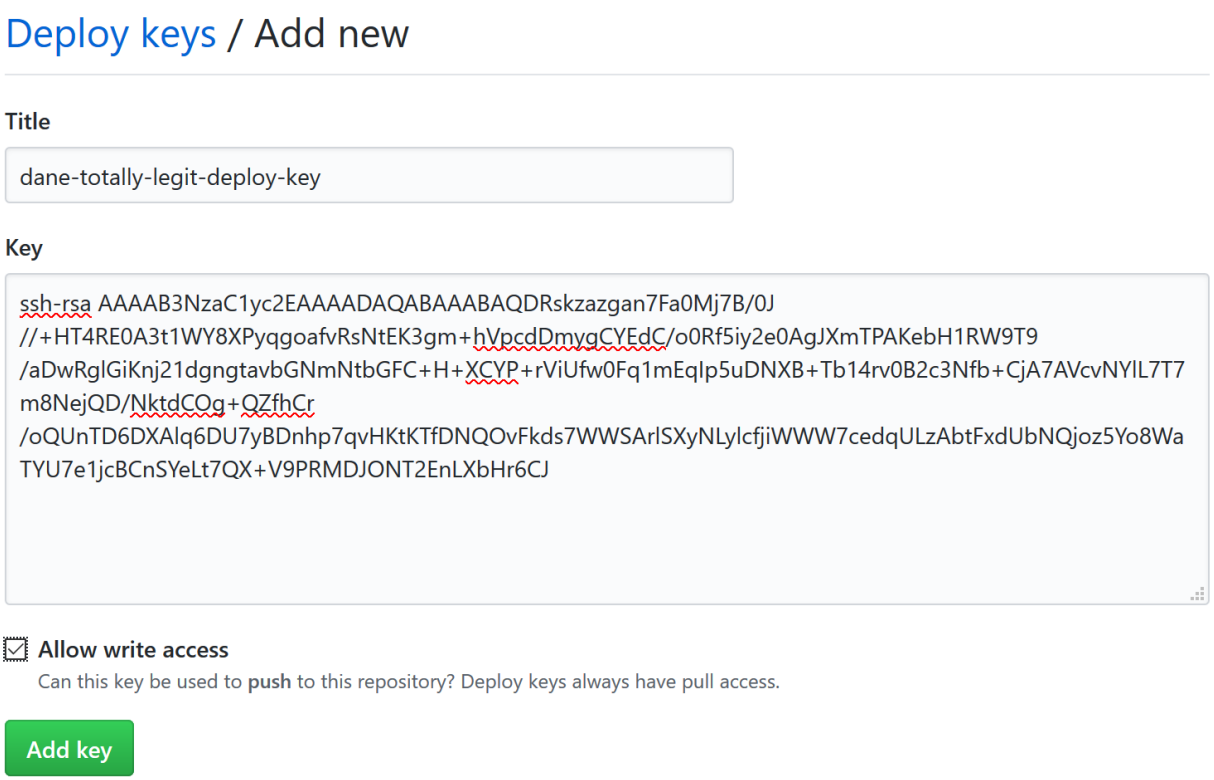


Figure 11 – Adding a Repository Deploy Key.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as an `authentication` event with type `public_key.create` or `public_key.update`. Develop and implement alerting for creation of deploy keys for sensitive or privileged repositories.
Native Alerting	GitHub will natively alert users with an e-mail notification when a new SSH deploy key has been associated. This will include the key name and fingerprint. Any repository administrator can visit GitHub Enterprise to delete any key associations.

4.1.5. Account Manipulation – External Collaborator Invite (T1098)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, and administrative rights over a repository can invite an external collaborator to a repository. The collaborator may be granted read, read-write, or admin rights to the repository.

© 2020 The SANS Institute Author retains full rights.

The collaborator may be another compromised GitHub user account or, depending on the authentication configuration, an account outside the scope of the organization. Figure 12 below shows an invitation to an external collaborator.

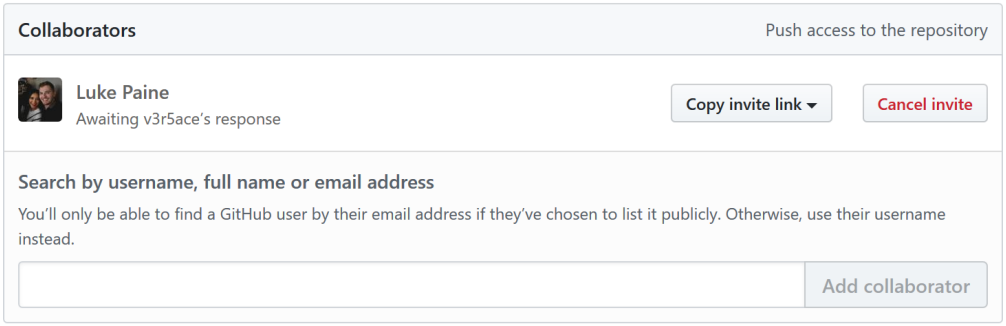


Figure 12 – Inviting an External Collaborator.

Proactive Hardening: To mitigate this technique, perform the following:

ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Disable the "Allow members to invite outside collaborators to repositories for this organization" feature. While this will not break this technique entirely, it will only allow organizational administrators to invite third party accounts.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as a `repository` event with type `repo.add_member`. Develop and implement alerting for invitations of external collaborators the organization.

4.1.6. Account Manipulation – Repository Privacy (T1098)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, and administrative rights over a repository can change the visibility from private to public. The following command-line snippet uses a personal access token to modify a repository's visibility to public.

© 2020 The SANS Institute Author retains full rights.

```
curl -H "Authorization: token TOKEN" --request PATCH -d '{"name": "goldmine",
"private": "true"}' https://github.local/api/v3/repos/mercurial-mining/goldmine
```

Proactive Hardening: To mitigate this technique, perform the following:

ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Enable the “private repositories” permissions at the organizational level. This will prevent non-owner users from being able to create public repositories. Users with the owner permission will still be able to create or make public repositories.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as a `repository` event with type `repo.access`. The event will be of type PATCH and notes a change to the visibility of the repository. Develop and implement alerting for visibility changes to sensitive repositories, visibility changes from unusual tools (e.g. curl).

© 2020 The SANS Institute Author retains full rights.

4.1.7. Account Manipulation – Organization Default Permissions (T1098)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, and with owner rights over an organization, can change the default permissions for all repositories in an organization. Figure 13 below shows the available options.

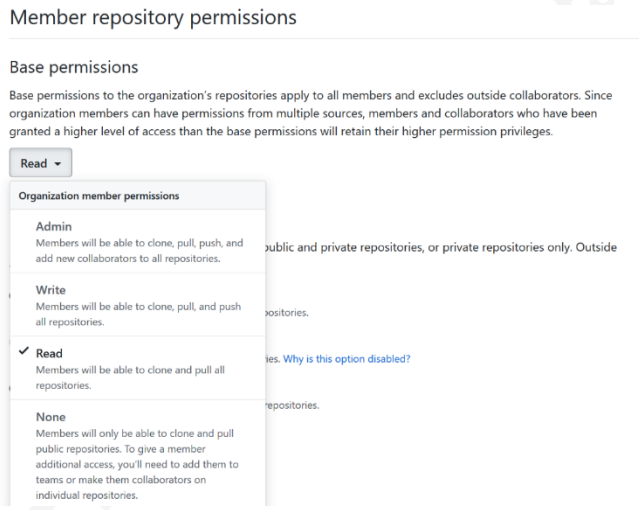


Figure 13 – Default Permission Settings.

Users in a repository are granted permissions of either admin, write, read (default), or none. This permission applies dynamically to all repositories contained within the organization.

The following command-line snippet uses a personal access token to modify the default permissions.

```
curl -H "Authorization: token TOKEN" --request PATCH -d
'{"default_repository_permission":"admin"}'
https://github.local/api/v3/orgs/mercurial-mining
```

Proactive Hardening: To mitigate this technique, perform the following:

ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Configure all organizations to have a "default_repository_permission" of write, read, or none.

© 2020 The SANS Institute Author retains full rights.

	Programmatically revert any default permissions set to admin.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as a `organization` event with type `org.update_default_repository_permission`. The HTTP request will be a PATCH and notes a change to the repository permissions. This event will include old and new values. Develop and implement alerting for default permission changes to sensitive repositories, permission changes from unusual tools (e.g. curl).

4.1.8. Disabling Security Tools – Branch Protection (T1089)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, and with administrative rights over a repository, can disable branch protection. Branch protection is used as a security mechanism by requiring pull request review or other checks to prevent merging of malicious code to the master branch.

There are two ways an adversary can bypass branch protection. The first is by abusing the default configuration setting where branch protection restrictions can be unilaterally bypassed by a user with administrative rights on the repository. Figure 14 below shows the configuration option in the default (unchecked) state:

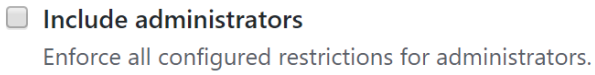


Figure 14 – Default Branch Protection Option.

© 2020 The SANS Institute Author retains full rights.

In this scenario, the malicious user simply needs to open a pull request or commit to the master branch and override the default settings. Figure 15 below shows that a user with administrative rights, and the default branch protection options, can still bypass mandatory pull request reviews on the repository:

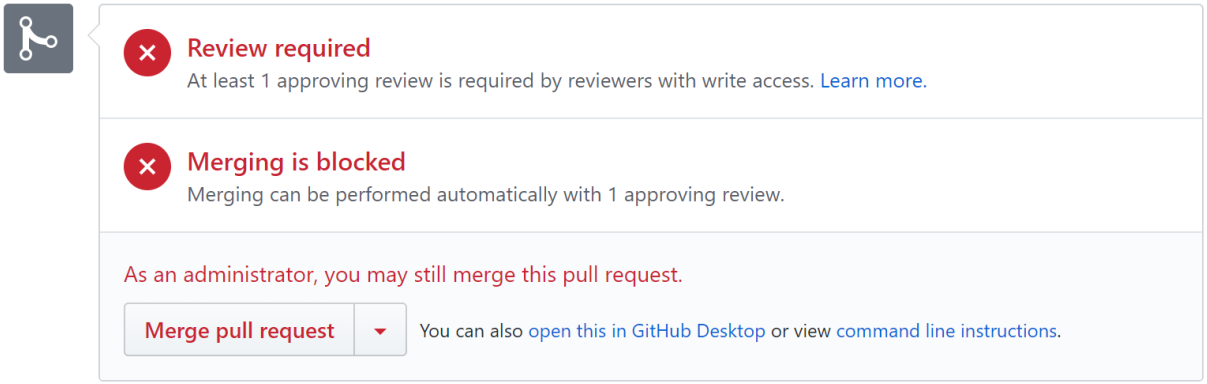


Figure 15 – Admin Force Override.

The second bypass strategy requires a user with administrator rights over the repository. Even if the "Include administrators" option is checked, branch protections can be modified and disabled at the repository level. In this example, an adversary can simply disable branch protection, make a malicious commit or pull request, and re-enable branch protection. As there is no default alerting on disabling or enabling branch protection, this activity may go entirely unnoticed.

Proactive Hardening: To mitigate this technique, perform the following:

ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Minimize the number of users with administrative rights over repositories. If users need self-service access to manage sensitive repositories, use an alternate administrator account and only assign read/write permissions to their normal account.

Detection Strategies: Perform the following to detect this technique:

Name	Description

© 2020 The SANS Institute Author retains full rights.

GitHub Log Monitoring	GitHub will log this technique as a `protected branches` event with type `protected_branch.destroy or protected_branch.policy_override.` Develop and implement alerting for destruction or modification to branch protection on sensitive repositories. Branch protection policy overrides should be a high-fidelity indicator of malicious activity.

4.1.9. Disabling Security Tools – Status Checks (T1089)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, and administrative rights over a repository can disable status checks. Status checks are used as a security mechanism for tests which must be passed prior to merging to the master branch.

There are two ways an adversary can bypass status checks. The first is by performing an arbitrary POST action to mark the check as complete. Any user with write access to a repository is able to forge successful status checks against the GitHub Enterprise server. An example JSON payload is shown below:

```
{
  "state": "success",
  "target_url": "https://localhost/build/status",
  "description": "This is totally okay, don't worry!",
  "context": "default"
}
```

The status JSON payload is posted to the status API endpoint referencing the SHA hash of the commit in question:

```
POST /repos/:owner/:repo/statuses/:sha
```

Once the POST has completed, the status check will reflect the state of the JSON payload (success) and, if there are no other compensating controls, allow for merging.

The second bypass strategy requires a user with administrator rights over the repository. In this example, an adversary can simply disable status checks, make a malicious commit or pull

© 2020 The SANS Institute Author retains full rights.

request, and re-enable status checks. As there is no default alerting on disabling or enabling status checks, this activity may go entirely unnoticed.

Proactive Hardening: To mitigate this technique, perform the following:

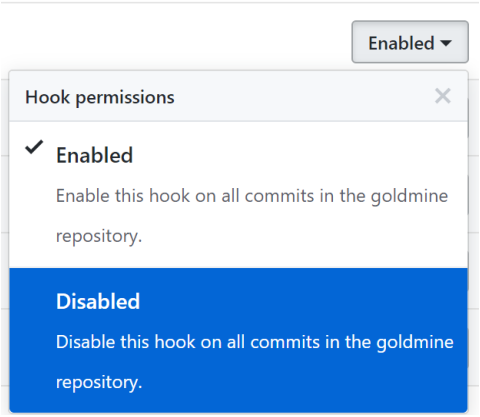
ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Minimize the number of users with administrative rights over repositories. If users need self-service access to manage sensitive repositories, use an alternate administrator account and only assign read/write permissions to their normal account.

Detection Strategies: There are no suitable detection strategies to note.

4.1.10. Disabling Security Tools – Pre-Receive Hooks (T1089)

Description: A malicious actor, with access to an interactive web session, or the GitHub Enterprise API, and administrative rights over a repository can disable pre-receive hooks. Pre-receive hooks are used as a security mechanism by running a server-side script on the GitHub Enterprise server which must be passed prior to merging to the master branch.

Since pre-receive hooks operate as a script on the GitHub Enterprise server, the only effective security bypass is to disable the pre-receive hook on the repository. This requires using a user account with administrative rights on the repository. Figure 16 below shows the GUI option for disabling a pre-receive hook:



© 2020 The SANS Institute Author retains full rights.

Figure 16 – Disabling a Pre-Receive Hook.

As there is no default alerting on disabling or enabling pre-receive hooks, this activity may go entirely unnoticed.

Proactive Hardening: To mitigate this technique, perform the following:

ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Minimize the number of users with administrative rights over repositories. If users need self-service access to manage sensitive repositories, use an alternate administrator account and only assign read/write permissions to their normal account.
NOMAP	GitHub Enterprise Configuration	Install and enforce pre-receive hooks at the organization level. Pre-receive hooks can be configurable, enabled, or disabled at the org layer which is inherited by all child repositories. Opting in all repositories to security pre-receive hooks substantially increases the cost of attack.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will log this technique as a `protected branches` event with type `pre_receive_hook.enforcement`. Develop and implement alerting for destruction or modification to pre-receive hooks on sensitive repositories.

4.1.11. Valid Accounts – SSH Key Theft (T1078)

Description: A malicious actor, with access a workstation with a legitimate GitHub Enterprise user, may discover and exfiltrate valid deployment or user SSH keys. GitHub Enterprise allows users to perform authentication using SSH keys for interacting with repositories.

Proactive Hardening:

Name	Description
SSH Key Passphrases	Train users to implement passphrases on SSH keys. This increases the cost of success for attackers who will need to deploy keylogging or other input capture attacks to use the stolen SSH key.

© 2020 The SANS Institute Author retains full rights.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	Monitor GitHub audit logs for concurrent SSH activity from multiple source IP addresses.

4.1.12. Account Discovery – User Permissions (T0007)

Description: A malicious actor with access to the GitHub Enterprise API may perform enumeration and discovery of permissions for a compromised user. This discovery technique involves the use of several API endpoints to determine organizational, team, and repository permissions for a given user account. Table 2 below notes the API endpoints used and their purpose in this technique:

Example API Command	Description
curl -H "Authorization: token TOKEN" https://github.local/api/v3/user	Discover information about the owner of the personal access token.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/user/teams	Discover information about the teams the token owner is a member of.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/user/orgs	Discover information about the orgs the token owner is a member of.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/user/repos	Discover information about the repos the token owner is associated with:

Table 2 – API Endpoints Used in Technique.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	Monitor web server logs for GETs to these API endpoints. A cluster of GET requests for user-specific permissions could be indicative of possible account takeover activity. Additionally, develop alerts around unusual access patterns, user agent strings, or connectivity to the GitHub API from atypical network locations.

© 2020 The SANS Institute Author retains full rights.

4.1.1. Account Discovery – Organization Enumeration (T0007)

Description: A malicious actor with access to the GitHub Enterprise API may perform enumeration and discovery of users, teams, organizations, and repositories This discovery technique involves the use of several API endpoints to retrieve information on users, teams, and organizations across the GitHub Enterprise installation. Table 3 below notes the API endpoints used and their purpose in this technique:

Example API Command	Description
curl -H "Authorization: token TOKEN" https://github.local/api/v3/users	Discover all user accounts on the GitHub Enterprise server.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/organizations	Discover all the visible organizations on the GitHub Enterprise server.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/orgs/:org	Discover the default permission for repositories in the organization.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/orgs/:org/members	Discover the members of an organization. This can identify if members are organizational administrators.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/teams/:team_id/members	Discover the teams present in an organization.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/teams/:team_id/members	Discover the members of a team.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/orgs/:org/outside-collaborators	Discover external collaborators of an organization.

Table 3 – API Endpoints Used in Technique.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	Monitor web server logs for GETs to these API endpoints. A cluster of GET requests for these endpoints could be indicative of possible account takeover

© 2020 The SANS Institute Author retains full rights.

and reconnaissance activity. Additionally, develop alerts around unusual access patterns, user agent strings, or connectivity to the GitHub API from atypical network locations.

4.1.1. Account Discovery – Repository Enumeration (T0007)

Description: A malicious actor with access to the GitHub Enterprise API may perform enumeration and discovery of repositories and their configuration. This discovery technique involves usage of several API endpoints to retrieve information on repositories, branches, contributors, and permissions and rights across the GitHub Enterprise installation. Table 4 below notes the API endpoints used and their purpose in this technique:

Example API Command	Description
curl -H "Authorization: token TOKEN" https://github.local/api/v3/orgs/:org/repos	Discover all repositories present in an organization.
curl -H "Authorization: token TOKEN" --request https://github.local/api/v3/repositories	Discover all public repositories present in the GitHub Enterprise instance. If using a site administrator token, this will include private repositories.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/user/repos	Discover all repositories the user has access to. This includes permissions read, write, and admin across repositories in the GitHub Enterprise instance.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo	Discover basic information about a repository, including default branch, default permissions, and other information.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/collaborators	Discover all collaborators, teams, and their permissions on a repository. Additionally, identify external collaborators.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/contributors	Discover all users who have contributed code historically in a repository.

© 2020 The SANS Institute Author retains full rights.

curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/branches	Discover all branches for a repository. This includes whether protection is enabled or enforced on a branch.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/pulls	Discover all pull requests (open or closed) for a repository. This includes information on how they were merged, reviewers, and other security-relevant information.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/commits	Discover all commits on a repository. Includes author, commiter, verification, and other relevant information.

Table 4 – API Endpoints Used in Technique.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	Monitor web server logs for GETs to these API endpoints. A cluster of GET requests for these endpoints could be indicative of possible account takeover and reconnaissance activity. Recursive enumeration of information on repositories, especially across multiple repositories, may be a high-fidelity alert. Additionally, develop alerts around unusual access patterns, user agent strings, or connectivity to the GitHub API from atypical network locations.

4.1.2. Account Discovery – Security Controls Enumeration (T0007)

Description: A malicious actor with access to the GitHub Enterprise API may enumerate and discovery native security controls. This discovery technique involves usage of several API endpoints to retrieve information on pre-receive hooks, branch protection, and other security controls on the GitHub Enterprise installation. Table 5 below notes the API endpoints used and their purpose in this technique:

Example API Command	Description
curl -H "Authorization: token TOKEN" -H "Accept: application/vnd.github.eye-	Discover all pre-receive hooks configured and enforced on an organization.

© 2020 The SANS Institute Author retains full rights.

scream-preview" https://github.local /api/v3/orgs/:org/pre-receive-hooks	
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/pulls/:number/reviews	Discover if pull request reviews are required for merging, if pull requests have been merged without a review, and other PR-specific security controls.
curl -H "Authorization: token TOKEN" -H "Accept: application/vnd.github.luke-cage-preview+json" https://github.local/api/v3/repos/:owner/:repo/branches/:branch/protection	Discover the branch protection configured for a specific branch. This will indicate if required status checks are enabled, if pull request reviews are enabled, if enforcement is required for administrators, and other branch protection information.
curl -H "Authorization: token TOKEN" https://github.local/api/v3/repos/:owner/:repo/commits	Discover if signed commits are required or commonplace in the repository.

Table 5 – API Endpoints Used in Technique.

Proactive Hardening: There are no hardening recommendations noted for this technique.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	Monitor web server logs for GETs to these API endpoints. It should be relatively unusual for any user to perform enumeration of many of these API endpoints. GET requests to branch protection or pre-receive hooks should be high-fidelity detections given how rarely users need to interact with, or modify, these protective controls. Additionally, a cluster of GET requests for pull request and commit API endpoints across multiple repositories could be indicative of possible account takeover and reconnaissance activity.

4.1.3. Data from Information Repositories – Malicious App (T1020)

Description: A malicious actor, with access to a malicious GitHub App or OAuth App installed within GitHub Enterprise, may be able to steal, collect, and exfiltrate sensitive information.

Proactive Hardening: To mitigate this technique, perform the following:

© 2020 The SANS Institute Author retains full rights.

ID	Name	Description
NOMAP	GitHub Enterprise Configuration	Limit the administrators present on repositories and organizations to the minimum required. If possible, use a tiered administrator model to only allow administrator access from separate accounts. Additional security configuration and hardening are specified in the Organization and Repository Security section under Additional Defensive Recommendations.

Detection Strategies: Perform the following to detect this technique:

Name	Description
GitHub Log Monitoring	GitHub will only log installation and modification to applications. Active data collection or exfiltration may not be detected in GitHub audit logs.

5. Additional Defensive Recommendations

Network defenders are recommended to evaluate and implement the following defensive recommendations.

5.1. GitHub Enterprise Server Security

The security of infrastructure as code in this scenario rests on the security of the GitHub Enterprise server. Compromise of the GitHub Enterprise server, without other compensating controls, would result in a catastrophic scenario for the organization.

Enterprises deploying GitHub Enterprise should follow the security best practices outlined in the deployment guide provided by GitHub.

The GitHub Enterprise server should be deployed in an isolated network with minimal network exposure, firewall enforcement, and centralized logging. Firewalls gate interactive access to administrator web URIs, management ports, and other sensitive services allowing origination only from dedicated bastion hosts or administrative subnets. Egress from the GitHub Enterprise server should be strictly controlled with a whitelist of domain and IP with which it can communicate.

© 2020 The SANS Institute Author retains full rights.

If possible, an endpoint detection and response (EDR) tool (e.g. osquery) deployed in full-auditing mode on the GitHub Enterprise servers will provide intrusion detection telemetry for the host. All users and administrators must use multi-factor authentication. GitHub Enterprise site administrators must use a secondary account purely for administration of the GitHub Enterprise application and other similar tier systems. Network defenders should create alerting and detection strategies for interactive administrator logins to the GitHub Enterprise server, modification of site administrator accounts, and other events related to site administrators.

5.2. GitHub Enterprise Repository Security Tiers

Network defenders should perform their own risk assessment of sensitive repositories stored in their GitHub Enterprise account and implement security controls commiserate with risk. A reference Repository Security Tiering Model with additional information is available in Appendix B.

Discretionary access controls applied to organizations and repositories within GitHub Enterprise leads to uneven protections, security bypasses, and opportunities for attacker exploitation. Security-conscious organizations should focus defensive efforts on building a standard for repository and organization security controls and applying them uniformly using automation. All repositories and organizations are periodically re-evaluated for compliance against this standard and deviations generate reports or alerts for network defenders to investigate.

Most employees using GitHub Enterprise do not need administrator rights or access over repositories or organizations. Where possible, companies should use automation to manage memberships of organizations, teams, and repositories against identity provider (e.g. Active Directory) groups. Removing administrator access to repositories and organizations dramatically reduces the number of security bypasses and attack techniques possible within GitHub Enterprise. If users need to self-service changes to their repositories or organizations, create a secondary account purely for administration of these.

All private and sensitive repositories must belong to an organization that is not commingled with external, non-sensitive or public repositories. All members of the internal organization must

© 2020 The SANS Institute Author retains full rights.

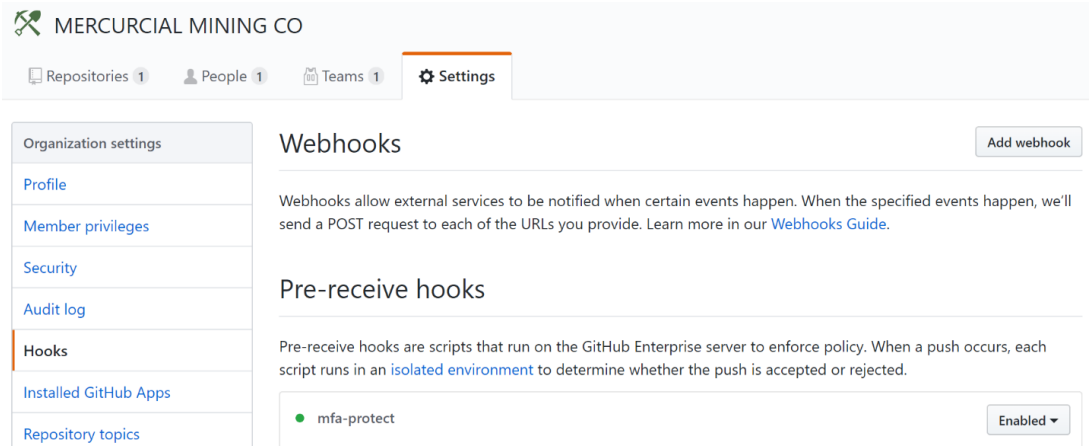
have multi-factor authentication mandatorily enforced. This reduces the likelihood of misconfiguration, account takeover, abuse of connected applications (e.g. CI/CD pipelines), and other exploitation scenarios.

All organizations must have the default repository permission set to either none, read, or write. No standard user may have admin rights over a repository or organization. All default branches (e.g. master) are protected and reject force pushes. Sensitive repositories require pull request reviews, pre-receive hooks, or status checks as security controls. All protected branches should have enabled the "include administrators" flag to prevent administrators from bypassing security controls. Authorized GitHub and OAuth Applications are audited and alerts fire for installation of unknown or new applications.

5.3. Duo-Bot: Enabling MFA on Pull Requests

Duo-Bot is an open-source GitHub Enterprise security app developed by Palantir Technologies which uses pre-receive hooks to perform multi-factor authentication challenges on commits. Any repository configured with Duo-Bot will automatically deny any attempts to alter the default branch (master) via git push or pull request until they have successfully performed a multi-factor authentication challenge via Duo. This guarantees that any commit or pull request to master has undergone an additional challenge by the person making the change.

Duo-Bot functions as a pre-receive hook running on the GitHub Enterprise server. This, unlike status checks, makes the server more resilient to attack as a malicious actor cannot simply POST to the check API to make the check succeed. Rather, an attacker would either need to successfully social engineer a user into approving the multi-factor authentication attempt on their behalf, disable pre-receive checks within their target repository or organization, or find a novel



Dane Stuckey

© 2020 The SANS Institute Author retains full rights.

bypass. Duo-Bot can be installed on a per-repository or organizational basis. Figure 17 below shows that Duo-Bot has been installed and enforced for all repositories owned by the organization.

Figure 17 – Duo-Bot Enforced for all Repositories.

An example workflow is an attempt to push a commit directly to master. As master is the default branch, and Duo-Bot is enforced for the repository, this workflow should fail until the user has successfully passed their multi-factor authentication check. An example of this workflow is highlighted in Figure 18 below:

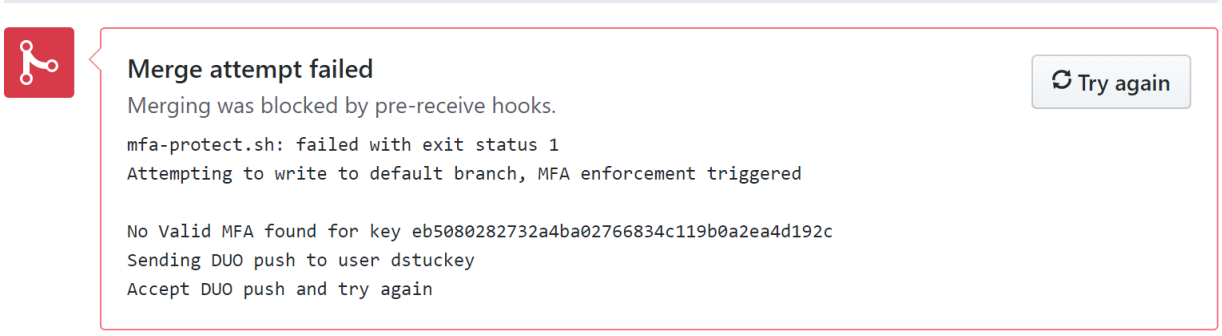


Figure 18 – Merge Blocked by Duo-Bot.

The git push has been rejected and the user has been issued a Duo prompt on their mobile device. They validate that this is a legitimate operation and therefore approve it. They then successfully re-run the git push to master for the repository. This is shown in Figure 19 and a console snippet below:

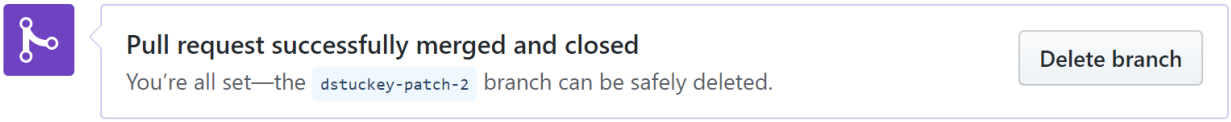


Figure 19 – Merge Allowed by Duo-Bot.

```
remote: Attempting to write to default branch, MFA enforcement triggered
remote: Record created at 16 Sep 19 00:13 UTC for user dstuckey is accepted and valid
```


© 2020 The SANS Institute Author retains full rights.

```
remote: Valid MFA acceptance found from dstuckey
24af9df..d03b1ce  master -> master
```

It is highly recommended that network defenders consider deploying mandatory security controls like Duo-Bot on critical repositories in their environment. Adding multi-factor authentication challenges for manipulation of infrastructure as code repositories significantly raises the difficulty of success for attackers.

5.4. Code Signing

Enabling signed commits dramatically increases the likelihood that a given commit originated from the author (or someone with access to the author's GPG/SMIME key). If a malicious actor attempted to change a signed commit before a remote push (e.g. local access to the repository on the workstation), the signature would not successfully validate, and the push would be rejected.

Signed commits protect against a successful compromise of the GitHub Enterprise server itself. If a repository requires all commits to be signed and validated, an actor with access to the underlying repository on the GitHub Enterprise server does not have the author's GPG/SMIME key. As such, they may be able to cause substantial harm, including deleting all data, but they are unable to forge a commit which can pass signature validation. Terraform validation of GPG/SMIME signature prior to apply infrastructure changes reduces likelihood of compromise of production infrastructure. Figure 20 below shows a successfully validated commit using the signed commit feature.

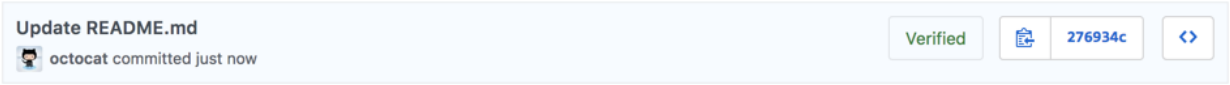


Figure 20 – Signed Commit Validation (GitHub, 2019).

Network defenders should use hardware-backed security keys (e.g. Yubikeys) for highly sensitive repositories.

© 2020 The SANS Institute Author retains full rights.

6. Conclusion

The ease of adopting infrastructure as code, combined with a lack of defensive tooling and controls, creates substantial risk for organizations. As noted in the results of the attack scenario, traditional security controls prove ineffective at identifying or mitigating adversary activity. It is likely that few organizations specifically harden or monitor GitHub Enterprise for abuse, leaving substantial opportunity for attackers.

As of the time of publication, there is no large corpus of attacker tradecraft or associated incidents related to infrastructure as code compromise. While this may be due to a lack of defensive telemetry and incident detection, or a lack of maturity in offensive operations, abuse of these tools will likely become more commonplace in the future. Network defenders should perform threat modeling of their GitHub Enterprise organizations, repositories, and users, and adopt an "assume breach" mentality. Authentication-based security should be deemed insufficient and mandatory security controls should be configured for commits and changes to sensitive infrastructure as code repositories. Security telemetry and logs from GitHub Enterprise should be ingested into a Security Incident and Event Management (SEIM) tool, and associated alerting and detection strategies should be implemented.

If appropriately managed and configured, GitHub Enterprise can provide robust defenses against these attack techniques, provide valuable security telemetry, and generate opportunities for incident detection.

© 2020 The SANS Institute Author Retains Full Rights

© 2020 The SANS Institute Author retains full rights.

7. Appendix A: Logging Categories

Log Category	Example Actions Logged
Authentication	Authentication-related events including creation and destruction of OAuth tokens, SSH keys, deploy keys, and multi-factor authentication configuration. Interactive login attempts will be located under the Users audit log category.
Hooks	Creation, association, destruction, and events related to GitHub Enterprise hooks.
Configuration	Changes to anonymous git access and repository creation controls.
Issues and Pull Requests	Creation, updates, comments, and destruction of issues or pull requests.
Organizations	Events related to organization changes, including deletions and transformations.
Protected Branches	Enabling, disabling, changes, enforcement, and other protected branch-related events.
Repositories	Changes to privacy, state (e.g. creation, destruction, archive, transfer), configuration, and anonymous access for repositories.
Site Admin Tools	Actions performed by GitHub site administrators including user impersonation, repository unlocking, disabling, and archiving, and other highly privileged actions.
Teams	Creation, modification, and destruction of teams.
Users	Authentication events, profile modifications, user synchronization jobs, credential modifications, login events, and other user-specific events.

© 2020 The SANS Institute Author retains full rights.

8. Appendix B: Repository Security Tiering Model

Level 1: No Protections

- Repository is managed manually.
- Access control lists are discretionary.
- Users have elevated access to repository.
- Pull request reviews may be required.

Level 2: Policy-bot

- Policy-bot is implemented to enforce reviews.
- Sensitive changes require two person integrity.

Level 3: Mandatory Repository Controls

- Repository membership and settings are managed via code.
- Access control lists are mandatory and enforced via code.

Level 4: Repository Administrator Tiering

- No users have admin rights on the repository.
- Tier-1 or bot accounts are required to perform admin functions.

Level 5: Multi-Factor Auth on Merge

- Duo-bot is deployed for pull requests to master branch.

Level 6: Code Signing

- Code signing is required for commits.
- Code signing keys are stored on physical hardware.

Level 7: Physical Separation

- Multiple GitHub Enterprise instances for each tier.
- No tier violations between GHE instances.

© 2020 The SANS Institute Author retains full rights.

9. Appendix C: Repository Security Checklist

Security checklist: Network defenders should complete the security task described in each of these topics for sensitive repositories and organizations.

- ☐ Perform a risk assessment for the repository and assign a security tier based on risk.
- ☐ Audit the repository and organization for external collaborators. Never invite untrusted collaborators to an organization containing sensitive repositories.
- ☐ Enable branch protections for sensitive branches.
- ☐ Ensure the repository is marked private.
- ☐ Audit and remove unnecessary installed GitHub Apps and OAuth Apps.
- ☐ Audit and remove unnecessary deploy keys.
- ☐ Ensure developers are using SSH keys protecting with strong passphrases.
- ☐ Ensure the repository has logs ingested in a SIEM.
- ☐ Implement alerting and detection strategies for repository in SIEM.
- ☐ Apply tier-specific security controls as needed:
    - ☐ Require reviews and two-person integrity for changes.
    - ☐ Implement policy-bot for content-based reviews.
    - ☐ Automate organization and repository membership and permissions.
    - ☐ Remove user admin rights on organization and repository.
    - ☐ Require multi-factor authentication (duo-bot) for sensitive merges.
    - ☐ Require hardware-backed code signing for commits.
    - ☐ Move sensitive repositories to a tier specific GHE instance.

Dane Stuckey

© 2020 The SANS Institute Author retains full rights.

10. Appendix D: Policy-bot

Policy-bot is a GitHub App which enforces approval policies on pull requests (Palantir, 2019). Policy-bot is open source (https://github.com/palantir/policy-bot) and provides complex approval features on a per-repository basis:

- Require reviews from specific users, organizations, or teams
- Apply rules based on the files, authors, or branches involved in a pull request
- Combine multiple approval rules with and and or conditions
- Automatically approve pull requests that meet specific conditions

As noted in Appendix B: Repository Security Tiering Model, network defenders should evaluate policy-bot to protect critical sensitive repositories. While policy-bot can provide increased security and reduce friction for merging to critical repositories, it relies on required status checks and is subject to potential bypasses highlighted in 4.1.9 (Disabling Security Tools – Status Checks). It is recommended that network defenders layer additional security controls and implement alerting and detection strategies for policy-bot bypasses.

© 2020 The SANS Institute, Author Retains Full Rights

© 2020 The SANS Institute Author retains full rights.

11. Acknowledgments

This research would not have been possible without standing on the shoulders of giants. I would also like to thank Will Schroder (@harmj0y) and Lee Christensen (@tifkin) from SpecterOps for their inspiration and initial research on this topic. I would also like to thank Elliot Graebert, Billy Keyes, and Alex Lake from Palantir for their tireless efforts improving infrastructure as code security and open-sourcing multiple defensive tools.

© 2020 The SANS Institute, Author Retains Full Rights

Dane Stuckey

© 2020 The SANS Institute Author retains full rights.

References

Center for Internet Security. (2019, 09 08). *CIS Benchmarks*. Retrieved from CIS Security: https://www.cisecurity.org/cis-benchmarks/

Defense Information Systems Agency (DISA). (2019, 09 08). *Security Technical Implementation Guides (STIGs)*. Retrieved from DoD Cyber Exchange Public: https://public.cyber.mil/stigs/

Exablue. (2019, 09 07). *GitHub Enterprise Remote Code Execution*. Retrieved from Exablue Blog: https://www.exablue.de/blog/2017-03-15-github-enterprise-remote-code-execution.html

GitHub. (2019, 09 08). *About commit signature verification*. Retrieved from GitHub Enterprise: https://help.github.com/en/enterprise/2.18/user/articles/about-commit-signature-verification

GitHub. (2019, 09 08). *About GitHub Applications*. Retrieved from GitHub Developer Documentation: https://developer.github.com/apps/about-apps/

GitHub. (2019, 09 08). *About pre-receive hooks*. Retrieved from GitHub Enterprise: https://help.github.com/en/enterprise/2.18/admin/developer-workflow/about-pre-receive-hooks

GitHub. (2019, 09 08). *About Protected Branches*. Retrieved from GitHub Enterprise: https://help.github.com/en/enterprise/2.18/user/articles/about-protected-branches

GitHub. (2019, 09 08). *About status checks*. Retrieved from GitHub Enterprise: https://help.github.com/en/enterprise/2.18/user/articles/about-status-checks

GitHub. (2019, 09 06). *Audited Actions*. Retrieved from GitHub: https://help.github.com/en/enterprise/2.18/admin/installation/audited-actions

GitHub. (2019, 09 08). *Authenticating Users For Your GitHub Enterprise Server Instance*. Retrieved from GitHub Enterprise Documentation: https://help.github.com/en/enterprise/2.18/admin/user-management/authenticating-users-for-your-github-enterprise-server-instance

Dane Stuckey

© 2020 The SANS Institute, Author Retains Full Rights

© 2020 The SANS Institute Author retains full rights.

GitHub. (2019, 09 06). *GitHub Audit Logging*. Retrieved from GitHub: https://help.github.com/en/enterprise/2.18/admin/installation/audit-logging

GitHub. (2019, 08 17). *GitHub Enterprise*. Retrieved from GitHub: https://github.com/enterprise

GitHub. (2019, 09 06). *GitHub Glossary*. Retrieved from Github: https://help.github.com/en/enterprise/2.18/user/articles/github-glossary

GitHub. (2019, 09 08). *Statuses*. Retrieved from GitHub Rest API Documentation: https://developer.github.com/v3/repos/statuses/

HashiCorp. (2019, 08 17). *Terraform*. Retrieved from Terraform: https://www.hashicorp.com/products/terraform/

Kakavas, I. (2019, 09 07). *The road to your codebase is paved with forged assertions*. Retrieved from Economy of Mechanism: http://www.economyofmechanism.com/github-saml

Krebs, B. (2019, 09 07). *Google Security Keys Neutralized Employee Phishing*. Retrieved from Krebs on Security: https://krebsonsecurity.com/2018/07/google-security-keys-neutralized-employee-phishing/

Microsoft. (2019, 08 17). *Securing Privileged Access Reference Material*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/windows-server/identity/securing-privileged-access/securing-privileged-access-reference-material

MITRE. (2019, 09 07). *Exploit Public-Facing Application (T1190)*. Retrieved from MITRE ATT&CK: https://attack.mitre.org/techniques/T1190/

MITRE. (2019, 09 07). *Initial Access (TA0001)*. Retrieved from Initial Access: https://attack.mitre.org/tactics/TA0001

MITRE. (2019, 09 06). *MITRE ATT&CK Enterprise Tactics*. Retrieved from MITRE ATT&CK: https://attack.mitre.org/tactics/enterprise/

Palantir. (2019, 08 17). *Bulldozer - Pull Request Auto-Merge Bot*. Retrieved from GitHub: https://github.com/palantir/bulldozer

Palantir. (2019, 08 17). *Duo-Bot - Force Duo challenges on GitHub actions*. Retrieved from GitHub: https://github.com/palantir/duo-bot

Dane Stuckey

© 2020 The SANS Institute Author retains full rights.

Palantir. (2019, 09 06). *Inside DevOps: Staying Compliant, Staying Productive*. Retrieved from Palantir Tech Blog: https://medium.com/palantir/inside-devops-staying-compliant-staying-productive-242b9a972cd1

Palantir. (2019, 08 17). *Policy-Bot - Enforce approval policies on GitHub pull requests*. Retrieved from GitHub: https://github.com/palantir/policy-bot

Rose, M. (2019, 08 09). *Life of a Pull Request*. Retrieved from Coderose: https://www.coderose.io/life-of-a-pull-request/

Tsai, O. (2019, 09 07). *GitHub Enterprise SQL Injection*. Retrieved from Orange: http://blog.orange.tw/2017/01/bug-bounty-github-enterprise-sql-injection.html

Tsai, O. (2019, 09 07). *How I Chained 4 vulnerabilities on GitHub Enterprise, From SSRF Execution Chain to RCE!* Retrieved from Orange: http://blog.orange.tw/2017/07/how-i-chained-4-vulnerabilities-on.html

Dane Stuckey

© 2020 The SANS Institute, Author Retains Full Rights

© 2020 The SANS Institute Author retains full rights.