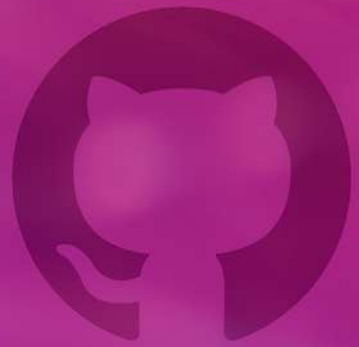
CHECK POINT™

Security Best Practices for GitHub



GitHub

GitHub is an integral part of a developer's workflow. No matter what organization or development team you go to, GitHub is ever-present in some form. It is used by over 65 million developers, 3 million organizations and hosts over 200 million repositories. Due to its sheer size and in comparison to other market offerings, it is the largest source code host in the world.

GitHub's market dominance is propelled by its ease of use and is extensively supported, especially by major cloud providers such as AWS, Microsoft Azure and Google Cloud. A range of users and their level of expertise use GitHub — from hobbyists to large enterprise organizations.

But what is security like on GitHub?

GitHub provides a myriad of tools and repository settings to prevent data breaches and leaks. The root cause of a security issue is often human oversight or lack of knowledge. According to a study published in 2019, after a comprehensive scan of public GitHub repositories, a total of 575,456 instances of sensitive data such as API keys, private keys, OAuth IDs, AWS access key ID and various access tokens were discovered on the platform. The primary risks of these exposures include but are not limited to monetary loss, privacy breaches, compromised data integrity, and different levels of abuse.

All this is preventable if certain practices and steps are followed. **This eBook will explore 21 GitHub security practices that can increase the robustness of your repositories and help implement a security-first approach for your development teams.**

Why you need to step up your GitHub security practices

Digital security is something that all software development teams know they should implement — but it is often the last thing they do, if not at all. It's not hard for sloppy practices and routines to end up costing your infrastructure and data integrity.

GitHub is synonymous with code version control and application development process flows. While other competitors such as Azure Pipelines and AWS CodeCommit, they do not have the same market hold, community backing and prevalence as GitHub.

According to a study from North Carolina State University, a six-month continuous scan of over a million GitHub accounts revealed that text strings containing usernames, passwords, API tokens, database snapshots, cryptographic keys, and configuration files were publicly accessed through GitHub. This included over 212,000 Google API keys, over 26,000 AWS Access Keys, and a combined total of over 28,000 social media access tokens.

What's more worrying is that there were 542 Stripe Standard API keys publicly available on GitHub. This is enough to give a malicious user direct access to a monetary-based account with real customer details and the ability to create fraudulent charges at the expense of the business's reputation and data integrity.

Security best practices every GitHub user needs to know

1 Never store credentials and sensitive data on GitHub

GitHub's purpose is to host code repositories. Beyond the permissions you set on your account, there is no other method of security that will ensure that your secret keys, private credentials, and sensitive data remain within a controlled and secured environment.

A git code commit maintains a history of what has been added and removed, making your sensitive data permanent on a branch. The issue of a potential data or infrastructure breach can increase exponentially when branches are merged and forked.

The easiest method to mitigate this risk is not to store credentials and sensitive data in the code before committing to a branch. However, mistakes can happen. **To programmatically prevent this, tools like git-secrets can be employed in your CI and CD pipelines.** This prevents code with sensitive data committed to it from reaching GitHub by breaking the build process. Another good method is to use secret and identity management tools such as Vault and Keycloak.

2 Disable forking

Forking is a git technique that lets a developer create a copy of a repository without implicating the original code. While forking is great for experimentation and sandboxing, it can also lead to an inability to keep track of where your sensitive data and private credentials end up.

A repository may originally be private but a fork can quickly expose everything into the public space. The risk is increased exponentially with each fork that occurs, creating a tree-like chain of security breaches through the exposed sensitive data.

To prevent this from occurring, disable the ability to fork **a repository to help mitigate the risk of sensitive data does make it into your code.** This can be achieved by clicking on the 'Settings' option of your organization, navigating to 'Member privileges', and under 'Repository forking', unchecking the option to fork will disable the ability to fork private repositories.

3 Disable visibility changes

Sometimes, a developer has more permissions and power than required by the scope of their role. It is easy for a developer without
a security-first centric mind to accidentally change the visibility of a repository.

The more people who have access to this ability to change the visibility, the higher the potential points of failure if sensitive data is present in your code repository. To prevent this from occurring, you can set the ability to change a repository's visibility to the organization owners only, or allow admin privileged members the power.

To implement this, **navigate to 'Member privileges' under your 'Organization settings' and uncheck the permission for all members of the organization to set the visibility under 'Repository visibility change'.**

4 Validate your GitHub applications

When it comes to security, you are as strong as your weakest link. Modern and distributed teams tend to be made up of external and third party teams. **Validating your GitHub applications involves keeping track of your third-party developers and their accessibility levels.** This also means that you revoke their access as soon as they leave the organization, or are no longer working on the code.

Different levels of accessibility should also be linked to their roles and involvement in the project. For example, a code auditor only needs the ability to pull code but no need to create commits. Pull and merge requests should only occur after a series of checks and code validation has been made by another with the right seniority and authority.

5 Enforce 2-factor authentication

While 2FA should not be considered and end-to-end method of protecting from data breaches, two-factor authentication (2FA) is now the industry standard for account security. It should also be your organization's standard security requirement to prevent code leakages through insecure accounts.

2FA adds an extra layer of security when logging into GitHub and can be enforced at the organizational level through your organization's settings. To do this, navigate to your organization list, select 'Organization security' and under 'Authentication' select 'Require two-factor authentication for everyone'.

When you click 'Save', you may be prompted with details concerning individuals who do not have 2FA activated. They will be removed from the organization and can only be added back in once 2FA has been implemented on their account. You can view members who have been removed in your organization's audit logs.

6 Implement SSO (GitHub enterprise only)

SAML single sign-on (SSO) is a feature that is available to GitHub Enterprise only. With this feature, organizations on GitHub can control accessibility by explicitly giving access permissions to specific resources such as individual repositories, pull requests, and issues raised. This allows the organization to segment accessibility to different parts of the code push, pull, and review process.

SAML SSO also lets you set up approved identity providers. This means that you can restrict users to using only the organization's accounts to sign in, rather than use privately owned GitHub accounts. This can mitigate potential security issues that may occur if accessibility was granted to a GitHub account.

7 Limit access to allowed IP addresses

For large organizations, keeping track of everyone can be hard and time-consuming. Whilst it is not impossible to keep on top of everything, sometimes slip-ups happen, and accounts that should not have access still retain their permissions. The risk is increased if the team member leaves the organization.

A quick and simple way to prevent unwanted accessibility is to limit access through IP addresses. This means that only members who are on-premise, or those who have access to the company's maintained static IP remote networks will have entrance into the organization's repositories and associated code work.

To limit, manage and whitelist IP addresses, navigate to your organization and select 'Organization security'. Here, you can configure your list of specific IP addresses or ranges in CIDR notation.

8 Tightly manage external contributor permissions

External contributors can join your team at any point. Your organization may be trying to speed a project along through outsourcing, or bring in external expertise to help fill a team gap. The churn rate for external members can vary, depending on the organization's needs.

The higher the turnover of external contributors, the higher the security risks. By tightly managing external collaborators and contributors, you reduce the number of redundant users and their accessibility to your code repositories. One way to manage external collaborators is to centralize access and permission granting abilities to an admin. Doing so can also reduce your long-term cost of access due to GitHub's 'per-user' pricing.

9 Revoke permissions in a timely manner

It is good practice to implement a security policy that outlines what happens to accessibility when a team member leaves the organization or is no longer required for the project. This includes time to revoke accessibility for different types of accounts.

Sometimes, a team member may still need access to the code but isn't required to contribute. **Revoking write permissions or switching them over as a maintainer role may be more suitable for the task.** This methodology follows the principle of least privilege, which is granting the permissions required to perform a specific task. Doing so will ensure that everyone who has access to the code only does so within the limitations of their repository contribution scope.

10 Require commit signing

Commit signing is a process of cryptographically signing the code merge for verification and traceability. This is important for code audit trails because it is not hard to pretend to be someone else in a commit. All it takes is a malicious user to change their username and email address in git config and push an exploitive code merge.

You can set up your Git to sign commits through GPG and configure your commits with a private key in your git config. Once this is done, you can add your GPG key to Gitbub. Now when commits are made, a 'verified' badge shows up next to the commit.

11 Enforce code review before commit

Enforcing code reviews can prevent malicious code from getting merged into the branch officially. Code review is also a good practice for detecting code smell, which can lead to future vulnerabilities and long-term security risk issues.

GitHub has a pull request facility that lets authorized team members discuss and review potential changes before it gets merged into the base branch. When a pull request is made, assignees can be attached to pull requests to notify them of the pending review required.

12 Add a Security.md file

A security.md file is the security policy for your repository. The purpose of this file is to officially document processes and procedures relating to security.

These processes and procedures include vulnerability reporting, confidentiality requirements, encryption standards, token accessibility, usage of email addresses, HTTPS requirements, usage of cloud, CDNs, backups, authentication requirements, and maintenance of data integrity procedures.

The security.md file can act as a valuable guide for developers and provide a centralized space where security expectations have been made explicit for the organization.

13 Rotate SSH keys and Personal Access Tokens

SSH key rotation can be used as a periodic purge of potentially breached access keys. It is good practice to have an expiry date on all SSH keys and personal access tokens in your security requirements policy.

It should be noted that while SSH key rotation can be automated through GitHub's API, changing personal access tokens is a manual process and can only be done by the user. For manual removal of SSH keys on GitHub, navigate to your 'Settings', and under 'SSH and GPG keys' you will find a list of all your currently issued access keys.

14 Audit all code uploaded to GitHub

It is easy to add external code repositories as part of your application building process. In addition to this, legacy code from other parts of your organization's software may also make it into another repository.

The issue of importing legacy code is that what used to be secure may no longer be the case. Setting a requirement to audit all code that gets uploaded to GitHub can be time-consuming, but also beneficial to the long-term health of your application and software architectural integrity.

15 Review your Github audit logs for suspicious activity

GitHub has an audit log facility that lets an organization's admin quickly review actions performed by other members of the team. **The details of who did what can help flag suspicious activities and create a quick trace profile based on the user's action, country-based location of the action, and the date and time of the occurrence.**

These three pieces of information can help an administrator detect anomalies and pinpoint their origin quickly.

16 Enable alerts for vulnerable dependencies

As software projects grow in size, so do their dependencies. Vulnerable dependencies — especially third-party dependencies, which are external to your organization — are at most risk due to their status and lack of general control over how a package or module gets updated. For small projects, it is not too hard to keep track of them. However, as projects grow larger, it is easy for these dependencies to get lost.

Vulnerable dependencies can come in many forms — from versioning to new discoveries of security risks overtime. GitHub has a feature that detects vulnerable dependencies in public repositories. To enable alerts, you can do this through the 'Security & analysis' option in your organization's settings.

17 Employ automated secret scanning at pre-commit

Many are under the impression that if the source code is private, hardcoding credentials should also remain safe. However, **it is considered bad practice because private repositories do not offer the same level of protection and encrypted vaults**, nor do they offer the same degree of control over accessibility rotation. It is also not hard to duplicate and distribute source code.

In a continuous integration and continuous delivery cycle, speed is a key to shipping code. This can lead to accidental commits of sensitive data. Automated secret scanning can reduce the risk of such credentials from accidentally getting exposed.

18 Clear your GitHub history

GitHub is fantastic at keeping logs of every change ever committed. However, this can pose a problem if sensitive data made it into your code repository. Cleaning your GitHub history is a two-step process. The first is to invalidate any tokens and keys that made it into the code. The second step is to use the git filter-branch command to purge and rewrite your repository's history.

Changing a commit further upstream is a major deal because it impacts all subsequent commits that have already been made. **It is good practice to merge and close all pull requests before running through your GitHub history. This is to prevent anomalies and unexpected bugs from occurring.**

19 Enable git branch protection

An accidental branch deletion or git squash merge can result in lost data or create breaches in the code through the introduction of vulnerabilities. Branch protection is a GitHub feature that allows you to protect specific git branches from unauthorized modifications. The purpose of this feature is to ensure that collaborators do not make permanent changes to a branch through processes such as deletion and forced pushes.

Other branch protection methods include requiring signed commits to ensure authenticity, traceability, and pull requests to prevent unauthorized code merges.

20 Add sensitive files to .gitignore

As projects grow in size and complexities, so does the sensitive data required to make your local machine work. These files tend to be unique and sit on the deployed server, away from any public-facing interfaces — making it hard to access.

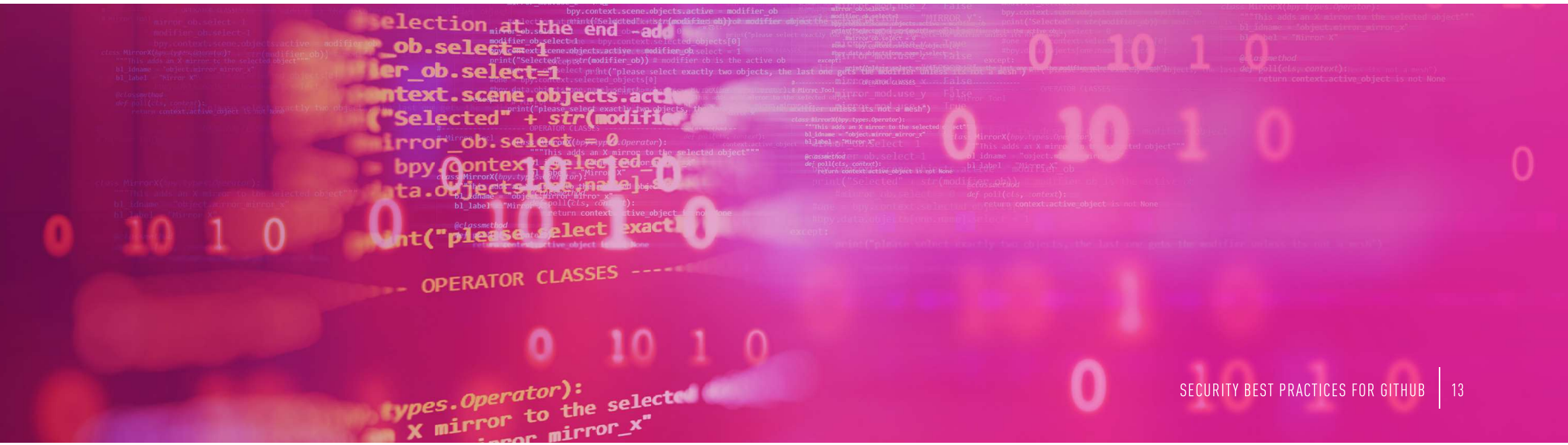
When in development mode and localhost, accessibility to these tokens and keys is required for software progression. **gitignore will ensure that your sensitive data does not get accidentally merged** and pushed to your GitHub repositories.

21 Employ a “secrets vault” service

As your project grows, so do the number of ‘secrets’ such as cryptographic keys, tokens, passwords, certificates, and API keys **rather than placing your trust on GitHub, you can externalize it into a ‘secrets vault’ service.**

A vault is a tool for securing highly sensitive data whilst providing a unified access interface. In addition to this, vaults provide tighter access controls and audit trails that let an admin easily detect vulnerabilities and breaches.

However, vaults tend to neglect the “last mile” of sensitive data delivery.



How CloudGuard Spectral can help you

CloudGuard Spectral can help secure your digital assets by equipping your teams with an automated security-first tool and platform. We know that DevOps live in the command line — so we've designed our tools to be compatible with your current workflows.

All you have to do is add Spectral to the CI as a build step. We then help scan your codebase using our array of detectors for risks such as sensitive data and credentials. If there is a problem with security, Spectral can automate a failed build and alert you to the issues. Our suite of services includes full reporting and tracing source files and positions for risk assessment and mitigation.

When your codebase is secured by design, it reduces the potential modes of malicious entrances. Our tools and platform is built in Rust — a language that excels at performance, safety, and safe concurrency. Secure your codebase today and ensure that your code is clear from potential security breaches and leaks. For more information, spectralops.io

Worldwide Headquarters
5 Ha'Solelim Street, Tel Aviv 67897, Israel | Tel: 972-3-753-4555 | Fax: 972-3-624-1100 | Email: info@checkpoint.com

U.S. Headquarters
959 Skyway Road, Suite 300, San Carlos, CA 94070 | Tel: 800-429-4391; 650-628-2000 | Fax: 650-654-4233

www.checkpoint.com