GitHub | November 2021

Adopting and scaling GitHub Advanced Security in your company

GitHub Advanced Security (GHAS) is an application security solution that enables companies to approach security with a developer first mindset. There are multiple avenues that companies can follow to adopt any application security solution, but picking which one can be difficult. This whitepaper walks through an example rollout that larger enterprises and organisations can follow to adopt GHAS at scale.

Table of Contents

Abstract

Adopting GitHub Advanced Security (GHAS) can be approached in multiple ways and requires a strategic approach for success, especially in larger enterprises and organisations with thousands of repositories. The purpose of this whitepaper is to lay down a foundation for enterprises on how to adopt GHAS, but most importantly, scale it quickly and efficiently.

What We'll Cover in This White Paper

We will cover:

- The overarching strategy to adopting GHAS.
- The suggested path to enabling code scanning at scale.
- The suggested path to enabling secret scanning at scale.

Methodology

The methodology followed here is based on a successful customer implementation of GHAS, precisely, a larger company that adopted and scaled GHAS in months of enablement.

Audience

The audience of this whitepaper is people directly involved in rolling out GHAS in a large organisation/enterprise. Examples could be:

- People who are maintaining the centralised GitHub Instance.
- Security personnel who are helping drive the adoption of GHAS via policies and data verification.
- DevOps Engineers who are helping drive the adoption of GHAS via CI/CD and automation.

Acronyms

GHAS -> GitHub Advanced Security

SME -> Subject Matter Expert

Authors

Nick Liffen - GitHub Advanced Security Field Architecture Team

Executive Summary

This whitepaper covers the suggested path to adopt GitHub's application security solution at scale. Enabling GitHub Advanced Security across a large organisation can be broken down into three core phases:

Phase One - Strategic Enablement Alignment

Although it's appealing to rush into the implementation phase, take the time to align on how GHAS will be implemented in your enterprise. Additionally, think about what success could look like in the 3,6 and 9 months after adoption. This phase may only take a few days or a week, but it lays a solid foundation for the rest of the rollout.

Phase Two - Create Internal Documentation

Like the above phase, organisations tend to rush into the implementation phase, as that stage is perceived to provide the quickest time-to-value. However, without the proper documentation and asynchronous resources provided to aid developers, security engineers, etc., in consuming GHAS correctly, usually, the value gets lost in the rollout due to people not consuming GHAS in the correct way. Take the time to create internal documentation (such as training, how to remediate, where to go for questions, etc.), and then communicate this documentation (email, teams, slack, etc.) to the consumers of GHAS so once you rollout GHAS, teams and people know what to do.

Phase Three - Enable & Scale Code Scanning

GHAS is an ecosystem of multiple solutions; it's essential to start somewhere focused, not just with the rollout of GHAS. Typically, we see teams focus on code scanning, to begin with. Leverage the API's available and rollout code scanning by team and by language across your organisation automatically. This allows you to scale in an automated fashion and removes a lot of manual repeatable groundwork for developers and consumers of code scanning. Doing this will increase adoption.

Phase Four - Enable & Scale Secret Scanning

Finally, focus on the rollout of secret scanning. Secret scanning is a more straightforward tool to rollout, as it involves less configuration and touchpoints to enable. However, it's critical to have a strategy on how to handle results, new and old. Without a proper process in place, secret scanning can be challenging to manage efficiently. Phase four walks through how to focus your efforts productively on revoking the results of secret scanning. Specifically, focusing on the new secrets found after Secret Scanning has been enabled (in an automated way) to start with, then revoking the current secrets found after.

The above is a high-level overview of successful enablement; more detail is provided in this whitepaper.

On the other hand, different rollout strategies that have a higher probability of friction and failure are:

- **Overly Cautious, Phased Rollout**

  Try not to enable GHAS by solely following a phased, team-by-team rollout strategy. Doing this will slow down rollout time significantly and stop teams using GHAS who know what they are doing.

- **Big Bang Approach for Secret Scanning***

  Try not to click the enable all Secret Scanning button on day one. Although this seems tempting, this is a good way of taking up all active GHAS licences.

  * If you have purchased licences above your current active committer count, this approach is okay, but be wary that you will need to buy more licences if you grow in active committers after doing this.

- **Rushed Approach**

There isn’t a one size fits all approach, but doing everything at once will be pretty unsustainable unless you have a large team dedicated to the rollout. Create a plan, follow the plan.

Phase One - Strategic Enablement Alignment

Introduction

Before diving into the enablement of GHAS, it's essential to think about how GHAS should be rolled out across the enterprise. Below are high-level strategic pillars on how to enable GHAS for code scanning and secret scanning. Every company will have a slightly different approach to each section, so use this as a guide versus a line by line rollout plan.

Code Scanning Strategic Enablement

Code scanning is a great application security tool, but getting it rolled out across hundreds of repositories can be difficult, especially when done inefficiently. Below walks through a three-step strategic approach that gets organisations to adopt code scanning effectively.

1) Enable

To start with, focus on getting teams using code scanning. The more teams using GHAS, the more data you can use to drive tactical & specific remediation plans. Without the data, these conversations are difficult. Meaning phase one of the rollout should focus on leveraging API's, running internal office hours and enablement events.

The core focus is on getting as many teams using code scanning as possible. You obviously will mention the importance of remediation and encourage teams to remediate appropriately, but the focus is enabling and using versus fixing (for now). Below will go into detail on how to get this done.

2) Educate

Next, focus on education. Education is likely the most crucial part of the plan as it teaches developers what to do in different situations. Education will go over aspects such as:

- When a secret scanning alert is found, what do teams have to do?
    - Are there various processes for different types of secrets?
- When a code scanning alert is found, what do teams have to do?
    - What's the process that teams have to follow?

Educate teams on the *trust but verify* process. Ensure developers are empowered to maintain the security of their repository whilst also ensuring the security team are authorised to verify what the developers are doing is correct and in the best interest of security.

The above can be done as online sessions, Q&A's, etc. But before you go and start pushing remediation, run education sessions, so developers are up to speed and aware of what you are expecting and how to remediate.

3) Remediate

After teams have enabled code scanning, developers are familiar with internal remediation processes and handling different scenarios; it's time to focus on remediation. Realistically you are going to push a decentralised and possibly, centralised approach. Meaning let teams drive their remediation, but use Security Overview (or the tool of your choice) to verify teams are remediating correctly. Out of this, you will likely see about a 40% group either not remediating or marking vulnerabilities as won't fix. Use Security Overview to find these teams and have personalised conversations about why they aren't remediating. Link back to previous education sessions, and stress the importance.

Iterate on this model. You are now using Security Overview to verify and code scanning to trust.

Secret Scanning Strategic Enablement

With secret scanning, it's a little different; you have a few ways of doing this. The easiest way of doing this is just to click enable all at the organisation level. Doing this will turn it on everywhere, but this is a speedy way to 1) cause an internal fire when you discover hundreds of secrets and 2) take up most of your licences. So think a little more tactically when enabling GHAS Secret Scanning. A possible approach is:

1) Stop fueling the fire (e.g. stop developers committing secrets)

One of the main aspects you would like to get ahead of is stopping any new secrets from being committed. This will make you stable with your current secrets found and give you a place to reduce, not increase. To do this, think about what automation you can put into place. Use the webhooks provided to notify the right people when a secret is committed. Most likely someone in security. Once advised a secret has been revealed, follow up with developers early. The more developers are made aware that the security team gets informed instantly and follows up, the more likely developers will think twice about committing secrets. The early push on this will make a difference.

There is an example of consuming the GitHub Webhooks from Secret Scanning here: GitHub Secret Scanner Auto Remediator (GSSAR). This tool automatically revokes certain types of secrets found.

2) Reduce exposure of the most frequently committed secret types

Now the above is in place, it's good to start to revoke and remediate current secrets committed. Start with the top five most essential secrets (e.g. AWS, Azure, Custom Secret Pattern One, etc.). You can use the Security Overview in your organisation to filter by secret type and understand the number of secrets leaked under that type. Create a plan/process just for the five (or so) types of secrets. Each secret type will likely have similar processes but slightly differ when it comes to the actual remediation. (e.g. you may have to work with the internal AWS team to work out the scope of the secret and impact, and you may have to work with the internal Azure team to work out the same).

You can use the Secret Scanning Organisation API to pull the repositories in scope, and you can then use the repositories API to find out the admins on that repo on who you may need to follow up with.

3) Remediate the rest & Educate

Once you are comfortable with your most frequently committed secrets being remediated (in progress), plan for the other types of secrets to be revoked following similar processes as above, but with less urgent timelines. As long as Step 1 and Step 2 are going well (especially step 1), you should see the number of revoked secrets decreasing at a healthy rate.

As you focus on the final secret types, it’s essential to also focus on education. Every company has slightly different training portals. Try and assign training to people who have access to GitHub, which educates on the importance of why secrets shouldn’t be leaked whilst showing how to handle sensitive information appropriately. E.g. a vault management tool, process environments, etc.

Phase Two - Create Internal Documentation

Before sharing GHAS with any internal developers or turning it on for use, create documentation that defines processes that teams should follow. For example, when a secret gets committed or a code scanning alert is found. Even if the process simply asks the team to apply their best judgement, that is better than no documentation. Doing this will educate teams on what to do when (in the likely scenario) an alert is discovered. This will also prevent developers from getting bottlenecked when they have questions; if there were no documentation, they would come to you (the team rolling out GHAS), which would lead to frustrations as there is not enough autonomy.

Some organisations will have a developer portal or a custom knowledge management (like a SharePoint site or a wiki) site where this content will live. It doesn't matter where; it just needs to be put somewhere internally where developers typically go for documentation. Examples of what should live in this content (please link to public GitHub Docs whenever possible, no need to repeat):

- What is GHAS?
- How to enable Secret Scanning on a repository?
- How to enable Code Scanning on a repository?
- How should you use Code Scanning? Including:
    - How to use it within your CI choice (Actions, Jenkins, etc.).
    - If you have standard custom build processes, document how these may be configured using CodeQL.
- What to do when you find a Code Scanning Alert?
    - How to ignore false positives? Alerts found in tests?
    - Is there a rough Service Level Agreement (SLA) teams need to follow (note: we don't encourage SLA's for new vulnerabilities unless they are

    necessary. We encourage developers to fix vulnerabilities in the PR's whilst they write the code).

- What to do when you find a Secret Scanning Alert?
    - E.g. contact a member of the security team? Follow up with XYZ Team (XYZ Team being the team who may be maintaining the tool where the secret was committed).
    - You don't need to put heavy process here, but give a high-level overview of what may need to be done.

If you skip this step, your rollout won't go at the pace you hope. This step may slow the initial rollout by a week or two, but that time will be made up during the POC when developers don't need to come to your team for questions.

Phase Three - Enable & Scale Code Scanning

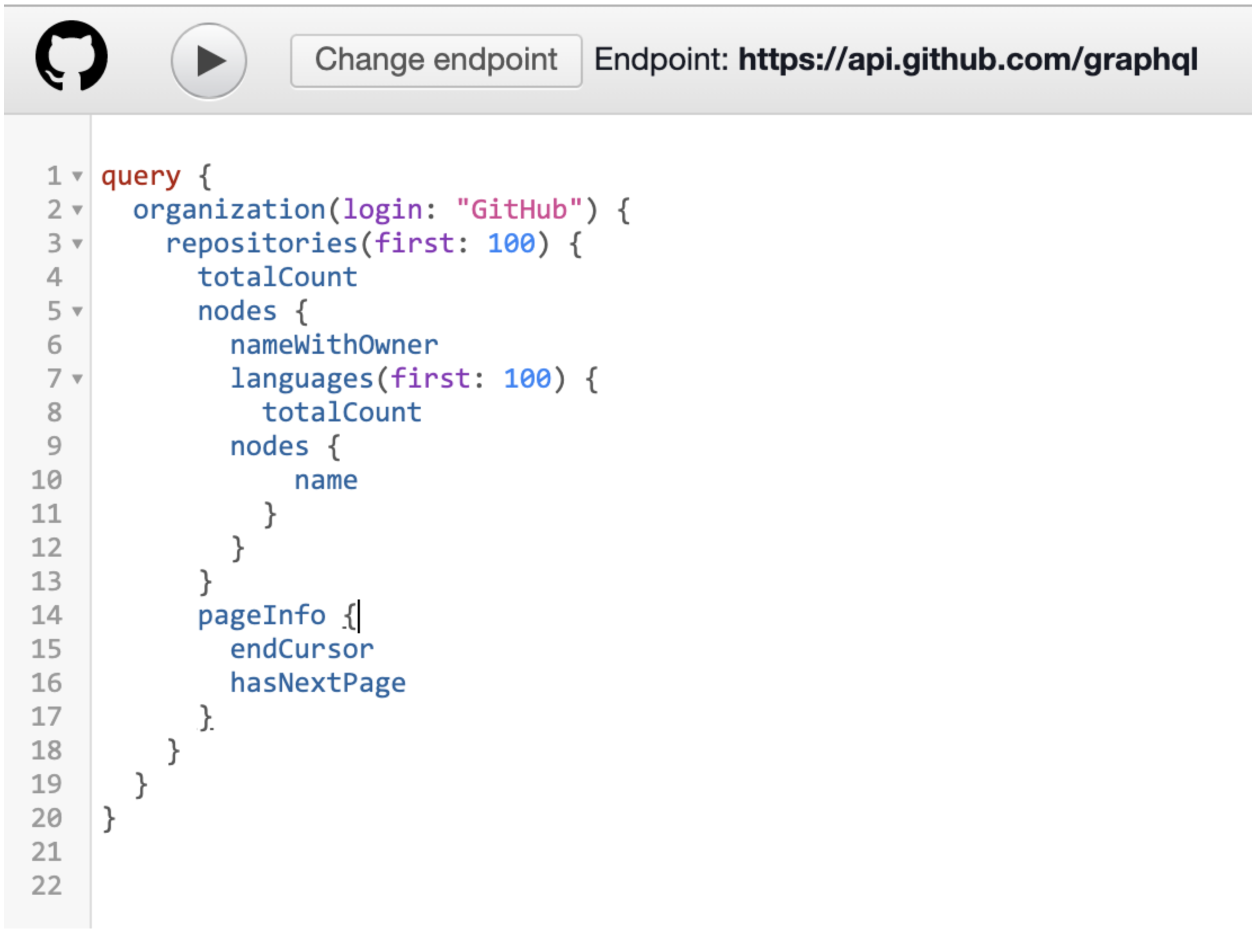
After the documentation stage is complete and published for people to see, the focus should shift onto enablement. This means getting as many people (teams) using code scanning as possible. The high-level process for enablement at scale looks like this:

- **Step One**: Communicate to all users, informing them that code scanning is enabled and can be leveraged today. Link to the documentation created in the previous stage, adding any timelines set out within the planning stage which teams need to follow.
- **Step Two:** Leverage automation to aid teams in getting up and running with code scanning. (This will be the significant work package of enablement).
- **Step Three:** Create internal knowledge around specific use cases (e.g. containers, typical Java build types, frameworks, etc.) and run office hours/lunch and learns to get teams with one-off build processes using code scanning.

Starting with a communication to all users letting them know code scanning is available will prompt teams who have heard of CodeQL and want to use a modern security tool to pick it up themselves and use it. This won't account for a very high percentage of adoption in the overall rollout, but this will empower developers and teams to feel like they can control security in their applications without waiting for a central team to get involved.

After the communication phase, bulk enablement via automation is an excellent approach to scale quickly. One of the best avenues is to break up mass enablement per language. This means enabling code scanning on all repositories that have the same language, creating a pull request for review that includes a sample codeql-anlaysis.yml file specific to those languages found in that repository.

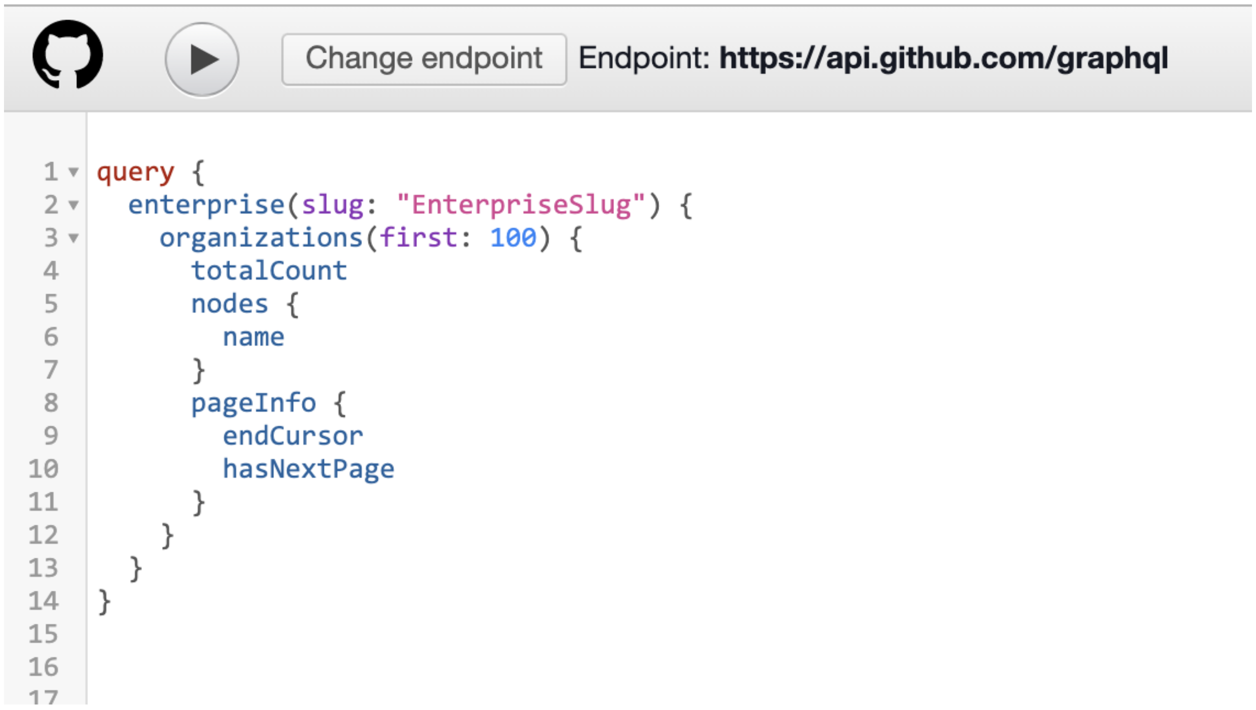
Firstly, collect the repositories by language at the organisation level. This will help identify which repositories use Java, JavaScript, Python, etc. A sample query could look like this:



```
query {
  organization(login: "GitHub") {
    repositories(first: 100) {
      totalCount
      nodes {
        nameWithOwner
        languages(first: 100) {
          totalCount
        nodes {
            name
          }
        }
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

Image One: Collecting all Repositories by an organisation, listing the languages.

Note: If you would like to do this across multiple organisations, for example, at the enterprise level, you can run the below query first, taking the organisation name and feeding it into the organisation variable above.



```
query {
  enterprise(slug: "EnterpriseSlug") {
    organizations(first: 100) {
      totalCount
      nodes {
        name
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

Image Two: Collecting all Organisations by an Enterprise

Take the data from the query in Image One and format it in a readable format which gives you something like this:

Language	**Number of Repos**	**Name of Repos**
JavaScript (TypeScript)	4,212	OrgName/RepoName OrgName/RepoName
Python	2,012	OrgName/RepoName OrgName/RepoName
Go	983	OrgName/RepoName OrgName/RepoName
Java	412	OrgName/RepoName OrgName/RepoName
Swift	111	OrgName/RepoName OrgName/RepoName
Kotlin	82	OrgName/RepoName OrgName/RepoName
C	12	OrgName/RepoName

		OrgName/RepoName

Table One: An example table showing repositories by a single language

Filter out the languages that are currently not supported by GHAS. (Make sure your script still pulls languages that GHAS doesn't support, as when language support is available, you have the script to pull the data).

> **Note:** A repository may have more than one language (monorepo, cloud-native, etc.), and if that's the case, you can create them in the same way as above:

Language(s)	**Number of Repos**	**Name of Repos**
JavaScript/Python/Go	16	OrgName/RepoName OrgName/RepoName
Rust/TypeScript/Python	12	OrgName/RepoName OrgName/RepoName

Table Two: An example table showing repositories by multiple languages

> If you do the above, filter out the languages that are not supported, don't filter out the repositories. You can still enable code scanning on repositories where one language isn't supported; it just will not scan that one language; it will still scan the others in the repositories where there is support.

Once you understand what repositories are tied to what language, it's time to enable GHAS code scanning across all these repositories, one language at a time. The step-by-step process for enabling GHAS should look like this:

1. Enable GitHub Advanced Security on the repository
2. Enable code scanning on the repository
3. Create a pull request into their default branch with an example codeql-analysis.yml file with a standard way of running CodeQL for that language.
4. Create an issue on the repository to explain why a pull request has been raised on their repository. Link to the previous communication sent to all users, but

explain a little about what the PR does, what next steps they have to take, their responsibilities, and how they should be using code scanning.

There is a publicly available tool that does steps 1-3. It's called the: ghas-enablement tool. Step 4 can be daisy-chained onto the end of the tool but doesn't come as standard because every company wants to give a slightly different message.

Note: It is essential to not just push into the repository's default branch. Doing this means you are not educating the developers on what to do and how to use code scanning. The pull request puts ownership on the development team to review and merge, giving them a sense of right and involvement in the process. The issue is also crucial; otherwise, the pull request gets created with no context.

Note: (Only for people using actions to control code scanning) If you do not use the ghas-enablement tool, keep in mind there is no API access to the .github/workflow directory. This means you can't create a script that runs without a git client underlying the automation. This is the current limitation of GitHub Actions. The proper workaround is to leverage bash scripting on a machine/container which has a git client. The git client can push/pull files into the .github/workflows directory where the Codeql analysis file should live.

Re-run the tool for every language where it makes sense. For example, JavaScript, TypeScript, Python, and Go likely have a similar build process and, therefore, a similar analysis file. These are good examples of where to start. The above can be done for languages such as Java/ C/C++, but more descriptive text will be needed for drafting into the issue due to the varied nature of how these languages build and compile.

Note: It is advised that you capture the pull request URLs created by automation, and every week checking and seeing which ones are getting closed out/not getting closed out. After a few weeks, it may be worth creating another issue if the PR is not closed or sending targeting internal emails to prompt action.

This leads to the next stage of enablement, which is creating internal SMEs (subject matter experts) and running office hours. The above automation will likely tackle a large percentage of your adoption, but this doesn't tackle one-off use cases where a specific build process or frameworks/libraries need feature flags to be enabled to work. A more personalised and hands-on approach is required here to push a high adoption, especially for Java/C/C++. At scale, you can't work one team at a time; with an enterprise of thousands of repositories, this will take too long. Instead, it's good to run weekly/bi-weekly office hours or lunch and learns.

Note: Office hours and lunch and learns are similar events. They are defined as a set time where people can come along and learn about a specific topic. In the context of this whitepaper, we'll demonstrate working with JSP and code scanning, or working with Java Spring and code scanning.

The value of doing this in an organisation is that teams can come to sessions that suit the topic you are discussing; and are relevant to them. Some sessions that may be relevant which have been run before are (but is entirely company-specific):

- Code scanning in a container
- Code scanning & Java Struts
- Code scanning & JSP

Most of the office hour sessions are focused on Java & C/C++ due to specific requirements around compilers and frameworks that may need a setup that isn't trivial.

Again, use the data you collected from the repositories by language task to create *targeted* office hours. You have a list of repositories that use Java, so you may generate office hours specific for Java and invite people who collaborate on Java repositories. The same for C/C++, and so on.

Conclusion

The above explains the approach organisations can take to enable code scanning. The advice would be to leverage automation as much as possible. GitHub offers extensive APIs that doesn't just cover code scanning itself but also covers APIs that support the implementation process (e.g. user, repository API). Finally, continue to communicate, iterate and adapt as new learnings arise throughout the rollout of code scanning in your organisation.

Phase Four - Enable & Scale Secret Scanning

GitHub's secret scanning capability is slightly different from code scanning since there is no specific configuration per programming language or per repository. This means enabling secret scanning at the organizational level *can* be easy. However, simply clicking: *Enable All* at the organisation level and ticking the option: *Automatically enable Secret Scanning for every new Repository* has some downstream effects that you should be aware of:

- **Licence consumption:** This will consume all your licences, even if no one is using code scanning. This isn't a problem if you do not plan on increasing the number of active developers you have in your organisation. However, if you see growth in the coming months in the number of developers, you will go over your licence limit and you may not be able to use GHAS on newly created repositories. This includes code scanning and secret scanning. So you should plan ahead.
- **Initial high volume of detected secrets:** If you are a large organisation, especially with a high percentage of outsourced developers, be prepared to see a high number of secrets found. Sometimes this comes as a shock/surprise to organisations, and then lots of alarms get raised. If you would like to turn it on across all repositories, be prepared to what to do when you see hundreds, if not thousands, of secrets, committed to your git history.

If you do enable all, the below approach is recommended:

1. **Focus on <u>new</u>ly committed secrets**

The first stage should be creating a process that handles any new credentials leaked from the secret scanner's enablement date. It is easy to focus on revoking the credentials already revealed and discovered by GitHub's secret scanning. However, while cleaning up these committed credentials, developers could continue to push new credentials accidentally. Meaning, in the overall bigger picture, your total secret count is staying around the same, not going down as you intended. This is why before focusing on revoking any current secrets, it is essential to stop the curve of new credentials being leaked. This will mean when you do get around to revoking current secrets, the total number decreases and does not stay the same/go up.

There are a few ways you could tackle newly committed credentials, but an example approach would be:

- **Step One - Notify:** Use the webhooks available to be notified of any new secret alerts. A webhook fires when a secret alert is either: created, resolved, or reopened. So for this use case, you can filter only the created alert type. You can then parse the payload, and integrate it into Slack, Teams, Splunk, Email, etc., anywhere which makes sense to you.
- **Step Two - Follow Up:** Create a high-level generic remediation process that is secret type agnostic. This can be simply reaching out to the developer who committed the secret and their technical lead on that project, letting them know the importance of not committing secrets to GitHub. Then, encouraging them to find where that token is being used, revoking it, and updating the token in locations where it is needed.
- **Step Three - Educate**: Create an internal training document assigned to the developer who committed the secret. Within this training document, you would go through the items discussed in Step Two, e.g. the importance of not committing secrets and the consequences. If the same person keeps on committing secrets, you could create an escalation process, but firstly use education to your advantage.

Repeat Step Two/Three for any new secrets leaked. This will encourage a shift in behaviour across developers, as they know this is being properly tracked, and there are possible consequences and repercussions.

> **Note:** You can automate Step Two. For large enterprises/organisations with hundreds of repositories, manually following up is unsustainable. You could piggyback onto the notification process defined in Step One. As part of the webhook, you get the repository and organisation object where the secret was

leaked. Using this information, you can contact the current maintainers on the repository and create a targeted automated email/message to these people, letting them know what has happened.

It is likely worth being open and transparent to your developers and users of GitHub before even starting Step One above, letting them know the process that has been defined and what happens if a secret is committed & pushed.

Note: For the more advanced organisations, who would like to do auto-remediation of certain types of secrets, there is an open-source initiative called GitHub Secret Scanner Auto Remediator. This is a solution you can deploy into your AWS, Azure, GCP environment and tailor to automatically revoke certain types of secrets based on what you define most critical. This is also an excellent way to react to new secrets being committed with a more automated approach.

2. **Remediate Previously Committed Secrets (But Start with the Most Critical Ones First)**

After you feel comfortable with the process created to monitor, notify and remediate newly published secrets, it's time to move on to doing the same for secrets discovered before GHAS was introduced. This is where it gets largely company-specific.

Every company has different types of secrets which are most important to them. For example, a company likely isn't worried about a Slack Incoming Webhook secret if they don't use Slack. However, suppose an AWS Access Key / Secret is leaked into a repository, and that company uses AWS heavily. In that case, that alert will need eyes on it quicker than other types of secrets. Additionally, a different company may not have an enterprise AWS account, but they have a GCP account to prioritise GCP over AWS. It's specific to an organisation.

What has been seen to work well is focusing on the top five most essential credential types for your company. There could be 10-20 (if not more) different secret types in the list of leaked secrets. It's untenable to think about remediating all of them all the same time. So, focus on firstly identifying what secrets you would like to remediate.

Once you know the secret types, you can do the following:

Step One:

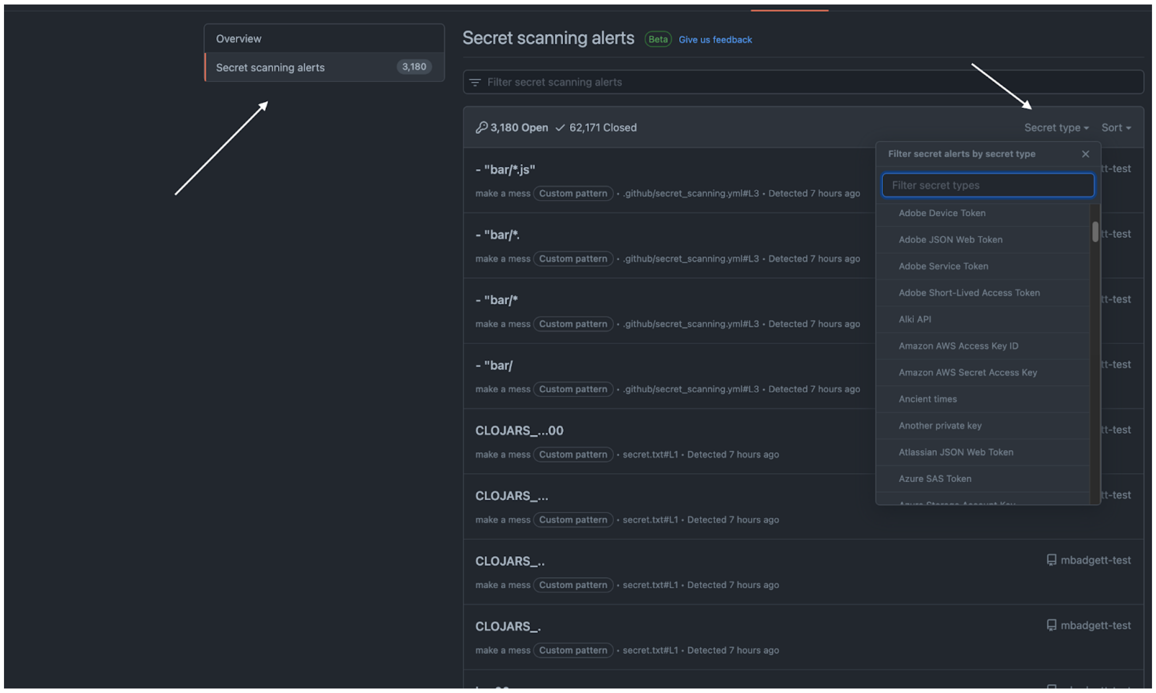
Define a process for how to remediate each *type* of secret. The actual procedure for each secret type is pretty different. For an AWS Key, you will have to work with the internal AWS team to work out what permissions the Key has if no one knows. For an Adobe Key, you are going to have to work with the Adobe team, etc. Get that process written down into a document (or web article in your knowledge portal).

> **Note:** When you create the process for revoking secrets, try and decentralize this onto the team maintaining the repository; not put the responsibility onto a central team to revoke. The reason? You want to teach and educate teams on the importance of not pushing credentials in the first place. If the developers know it won't be their responsibility to fix and have to go through the hassle of revoking secrets, there is a higher percentage they will not care *as much*. Going back to the principles of GHAS, the developers need to take ownership of security; they need to have the responsibility of fixing security issues, especially if they create them.

Step Two:

Now you have the process which teams need to follow for revoking credentials; the next step is finding what exposure looks like for that secret type and getting metadata associated with the leaked secret so you know who to communicate the process to.

Use GitHub Security Overview to collect this information. The screenshot below shows how to collect that information:



Overview
Secret scanning alerts 3,180

Secret scanning alerts Beta Give us feedback

Filter secret scanning alerts

3,180 Open ✓ 62,171 Closed — Secret type — Sort

- "bar/*.js"
make a mess Custom pattern · .github/secret_scanning.yml#L3 · Detected 7 hours ago

- "bar/*.
make a mess Custom pattern · .github/secret_scanning.yml#L3 · Detected 7 hours ago

- "bar/*
make a mess Custom pattern · .github/secret_scanning.yml#L3 · Detected 7 hours ago

- "bar/
make a mess Custom pattern · .github/secret_scanning.yml#L3 · Detected 7 hours ago

CLOJARS_...00
make a mess Custom pattern · secret.txt#L1 · Detected 7 hours ago

CLOJARS_...
make a mess Custom pattern · secret.txt#L1 · Detected 7 hours ago

CLOJARS_..
make a mess Custom pattern · secret.txt#L1 · Detected 7 hours ago — mbadgett-test

CLOJARS_.
make a mess Custom pattern · secret.txt#L1 · Detected 7 hours ago — mbadgett-test

Filter secret alerts by secret type

Filter secret types

Adobe Device Token
Adobe JSON Web Token
Adobe Service Token
Adobe Short-Lived Access Token
Alki API
Amazon AWS Access Key ID
Amazon AWS Secret Access Key
Ancient times
Another private key
Atlassian JSON Web Token
Azure SAS Token

Image Three: Screenshot which shows how to filter by secrets

Some information you are going to need to collect is:

- Organisation
- Repository
- Secret Type
- Secret Value
- Maintainers on Repository to contact

> **Note:** Only use the GUI if you have a few secrets leaked of that type. If you have hundreds use the API to collect information; it will be much quicker and repeatable if you need to rerun it.

Step Three:

After the information defined above is collected, create a targeted communication plan for the users maintaining the repositories in the scope of each secret type. This can be sent out via email/Teams/Slack, or even GitHub, by raising an issue on the repository; use the tool that most suits your enterprise. (Use APIs provided by these tools to send out the communications in an automated manner, it will help scale across multiple secret types.

3. **Continue to Revoke Different Secret Types - and Educate**

The final stage is focused on continuing where you left off in the previous stage, expanding the top five secret types to a more compressive list, with an additional focus on education. Continue to repeat Steps 1-3 above for the different secret types you continue to support.

> **Note:** Another reason why it's important to only start with the most critical five secret types is when you get to this stage, and you expand your coverage, you can take learnings from what went well/didn't go so well and apply them learnings to the upcoming secret types.

As you continue to build your remediation processes for other secret types, start to create proactive training material that can be shared with all current/new developers of GitHub in your organisation. So far, a lot of the focus has been *reactive* based on a secret being pushed. It is an excellent idea to shift to be *proactive* and encourage developers not to push credentials to GitHub. This can be achieved in multiple ways but even creating a one-two page document explaining the risks and reasons would be a great place to start.

--

> **Note:** The above walks through a high-level process focusing on the *enable all* approach at the organization level. However, the above principles can still be applied even if you take a more staggered approach, not the big-bang enablement approach. Some companies only enable secret scanning on repositories where code scanning is enabled, meaning both features of GHAS scale evenly. This makes the value story of GHAS a little more balanced, compared to secret scanning taking all licences on day one, without code scanning even being touched. If you take the approach of enabling secret scanning and code scanning simultaneously, you can still follow the new ->

current (critical)-> current & educate method. The main difference will be the volume of alerts you will get; it will be lower. The advantage of doing it this way is you can take more time rolling out your processes, and if you don't have the capabilities of automating a lot of steps, it can be a lot more manageable.

Conclusion

To conclude, there are multiple ways this document could and likely should be interpreted. The purpose is to share knowledge on an approach that has worked and does scale well, but each organisation may put their own flavour on implementing the core themes mentioned throughout.

The main points to take away from this whitepaper are:

- **Automation works.** Leverage the APIs and Webhooks available. If something isn't directly in the product but would help with the rollout of GHAS within your organisation, look and see how you can get data out of GitHub programmatically to aid your strategy.
- **Communication is key**. It's vital to be open and transparent to your users. Before introducing any new processes or automation, always inform the users that are affected. The best approach here is to create targeted communications. For example, if the change only affects one person who has leaked a secret, only inform that person. If a change only affects people writing Java, only tell people who write Java. This is an excellent way to create effective communications and stop people from ignoring messaging.
- **Take your time.** Rolling out GitHub Advanced Security isn't going to be a one-week rollout; it likely also isn't going to be a one-month rollout. Remember that GHAS isn't just a tool; it's aimed at shifting how security and developers work together to ensure security is truly a first-class citizen of software development.

We are excited you are on a journey with GitHub Advanced Security (GHAS). Let's continue to shift the way security and software are developed.

Appendix

Query One: Languages per Repository per Organisation

```
{
 organization(login: "GitHub") {
  repositories(first: 100) {
   totalCount
   nodes {
    nameWithOwner
    languages(first: 100) {
     totalCount
     nodes {
      name
     }
    }
   }
   pageInfo {
    endCursor
    hasNextPage
   }
  }
 }
}
```

Query Two: Organisations per Enterprise

```
{
 enterprise(slug: "GitHub") {
  organizations(first: 100) {
   totalCount
   nodes {
    name
   }
   pageInfo {
    endCursor
    hasNextPage
   }
  }
 }
}
```