GitGuardian

Implementing Automated Secrets Detection for Application Security

Implementing Automated Secrets Detection for Application Security

How do we secure the new way of building software? Applications are no longer standalone monoliths, they now rely on thousands of building blocks: cloud infrastructure, databases, SaaS components such as Stripe, Slack, HubSpot... This is a significant shift in software development.

Dev & Ops teams from large organizations use thousands of secrets like API keys and other credentials in order to interconnect these components together. As a result, they now have access to more sensitive information than companies can keep track of.

The risk is that these secrets are now spreading everywhere. We call “secret sprawl” the unwanted distribution of secrets in all the systems developers use. Think about secrets hardcoded in centralized Version Control Systems, referred to in project management boards, shared through messaging systems, inside a Dropbox or within a Wiki. Secret sprawl is even more difficult to control with growing development teams, sometimes spread over multiple geographies. Not even taking into consideration that developers are under hard pressure due to a growing number of technologies to master and shortened release cycles.

In this whitepaper, we look at the implications of secret sprawl, and present solutions for Application Security to further secure the SDLC by implementing automated secrets detection in their DevOps pipeline.

What developers call a “secret” is anything that allows access to a system, often programmatically. API keys, private keys, database credentials, security certificates are perfect examples. Secrets are keys to the kingdom: they give access to cloud infrastructure, SaaS components, databases, internal portals or microservices...

Table of content

Understanding the benefits of mitigating secret sprawl

WHAT ARE THE THREATS ASSOCIATED WITH SECRET SPRAWL?

No company wants credit card numbers in plaintext in databases, PII in application logs, bank account credentials in a Google Doc. Secrets benefit from the same kind of protective measures.

As a general security principle, where feasible, data should remain safe even if it leaves the devices, systems, infrastructure or networks that are under organizations' control, or if they are compromised.

It is no surprise that credential stealing is a well-known adversary technique described in the MITRE ATT&CK framework.

MITRE ATT&CK T1081:
Credential Access /
Credentials in Files

MITRE | ATT&CK®

" Adversaries may search local file systems and remote file shares for files containing passwords. These can be files created by users to store their own credentials, shared credential stores for a group of individuals, configuration files containing passwords for a system or service, or source code/binary files containing embedded passwords. "

Source: https://attack.mitre.org/techniques/T1081/

Of course the term «passwords» must be taken in the broadest sense, and Application Security professionals prefer to talk about secrets. Secrets accessed by malicious threat actors can lead to information leakage and allow lateral movement or privilege escalation, as secrets very often lead to other secrets. Furthermore, once an attacker has the credentials to operate like a valid user, it is extremely difficult to detect the abuse and the threat can become persistent.

A FOCUS ON SECRETS IN SOURCE CODE: WHY ARE THEY SO BAD?

Surprisingly, secrets stored in source code is the current state of the world… although this is admittedly a bad thing.

- Source code is made to be duplicated and distributed, therefore lives in multiple places. Source code is a leaky asset and you never know where it is going to end up: it can be cloned to a compromised workstation or server, intentionally or accidentally published in whole or in part, uploaded to your website, released to a customer, pasted in Slack, end up in your package manager or mobile application…
- Additionally, it would just take one compromised developer account to compromise all the secrets they have access to.
- Hardcoded credentials make it very difficult to know what secrets a developer accessed, and almost impossible to roll keys after they leave.

SPOTLIGHT ON UBER
[1/2]

The Uber case is an interesting textbook case. We are leaving to the press the dramatic figures about the damage that hackers caused, because all security professionals know how serious credential theft can be. It is difficult however to find precise, reliable data about how hackers really operated, so we will focus on that instead. We're including below the link to the FTC report, which is to our knowledge one of the best sources of information on this case to date.

The first reported leak was due to a credential left in a public repository:

> **" First, on or about May 12, 2014, an intruder accessed Uber's Amazon S3 Datastore using an access key that was publicly posted and granted full administrative privileges to all data and documents stored within Uber's Amazon S3 Datastore."**

Source :
https://www.ftc.gov/system/files/documents/federal_register_notices/2018/04/152_3054_uber_revised_consent_analysis_pub_frn.pdf

SPOTLIGHT ON UBER
[2/2]

The second one was due to a credential exposed in a private repository that was compromised due to poor password hygiene and lack of MFA:

> “ **Second, between October 13, 2016 and November 15, 2016, intruders accessed Uber’s Amazon S3 Datastore using an AWS access key that was posted to a private GitHub repository. (…) Uber did not have a policy prohibiting engineers from reusing credentials, and did not require engineers to enable multi-factor authentication when accessing Uber’s GitHub repositories. The intruders who committed the 2016 breach said that they accessed Uber’s GitHub page using passwords that were previously exposed in other large data breaches, whereupon they discovered the AWS access key they used to access and download files from Uber’s Amazon S3 Datastore.** ”

Source:
https://www.ftc.gov/system/files/documents/federal_register_notices/2018/04/152_3054_uber_revised_consent_analysis_pub_frn.pdf

“ HOW BAD CAN IT GIT ? ”: the NCSU study that reports thousands of credentials leaked on public GitHub… Per day.

Here are some key takeaways:

- Independent study.
- Large scale study: millions of repositories and billions of files scanned, with over 200k credentials detected.
- Keys leaked at a rate of thousands per day.
- Conservative approach, targeting only 15 different types of API keys and 4 asymmetric private key types.

  > “ **Consequently, our work is not exhaustive but rather demonstrates a lower bound on the problem of secret leakage on GitHub. The full extent of the problem is likely much worse than we report.** ”

- Secrets are often leaked accidentally, not intentionally.
- High confidence that most of these secrets are indeed sensitive.
- Developer inexperience (measured as a small number of repos with few contributions on GitHub) is not strongly correlated with leakage.

Source:
https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04B-3_Meli_paper.pdf

GitLab's Security Trends analysis found that 18% of projects hosted on GitLab had identified leaked secrets.

This is still a lower bound, calculated using extremely simple detectors!

Source :
https://about.gitlab.com/blog/2020/04/02/security-trends-in-gitlab-hosted-projects/

Challenges associated with secret sprawl

1. THE GIT HISTORY MAKES IT MORE COMPLICATED THAN FIRST THOUGHT

Most vulnerabilities like cryptography weaknesses or SQL injection vulnerabilities only express themselves the moment the code is deployed. Exposed secrets are unlike these vulnerabilities, because any secret reaching version control system must be considered compromised and requires immediate attention. This is true even if the code is never deployed. Implementing secrets detection is not only about scanning the most actual version of your master branch before deployment. It is also about scanning through every single commit of your git history, covering every branch, even development or test ones.

Why do code reviews fail at secrets detection?

- Reviewers are only concerned with the difference between current and proposed states of the code, not with the entire history of the project. If a commit adds a secret and another one later deletes it, this has a zero net effect that is not of any interest to reviewers. But the vulnerability is there!
- Reviewers prefer to focus on errors that cannot be automatically detected, like design flaws. As a general principle, security automation should be implemented wherever it can, so that humans focus on where they bring the most value.

2. ENFORCING GOOD SECURITY PRACTICES AT THE ORGANIZATION LEVEL IS HARD

Difficulty increases with the size of the organization, number of repositories, number of development teams and their geographies, ...

Best practices to prevent secret sprawl include:

- Educating developers on why they must not hardcode secrets in code, ticketing systems, share them through messaging systems or in a Dropbox or a Wiki.
- Educating developers on how to safely store, share and retrieve secrets.
- Implementing automated secrets detection.

Of course, educating developers to not hardcode secrets in source code is a great starting point, but it is hardly scalable and still leaves too much space for human errors. Plus code reviews notably fail at detecting secrets, and take time and energy that would rather be spent on things where developers deliver the most value.

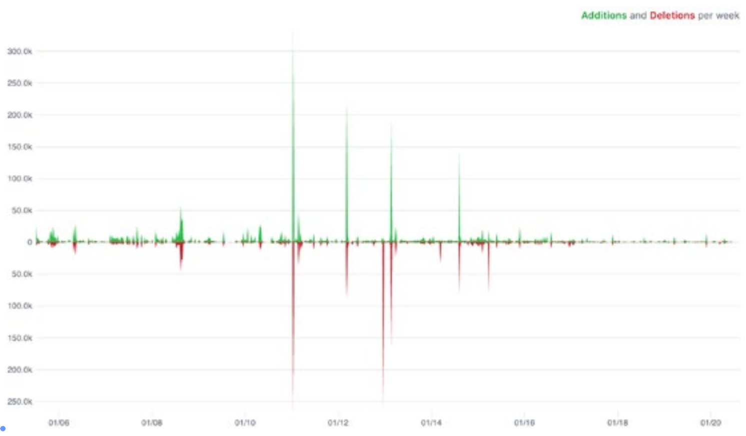
3. HOMEGROWN TOOLS AND SCRIPTS ARE HARD TO BUILD, MAINTAIN AND KEEP UP-TO-DATE

Some companies have built internal tools, often derived from Open Source. There are many Open Source tools that help you find leaked secrets, like truffleHog. Build vs Buy is an old dichotomy and you probably already have an opinion about it. An enterprise-grade solution is expected to provide precision, coverage and ease-of-use guarantees that come with tight integration into your workflows, without the burden of having to maintain it and keep it up-to-date.

A LOT OF SOURCE CODE LIVES IN THE HISTORY

Source :
https://github.com/django/django/graphs/code-frequency

This Django contributions graph is very common. There are as many additions than there are deletions! Deletions does not mean that the code cannot be accessed anymore. Deleted only means buried!

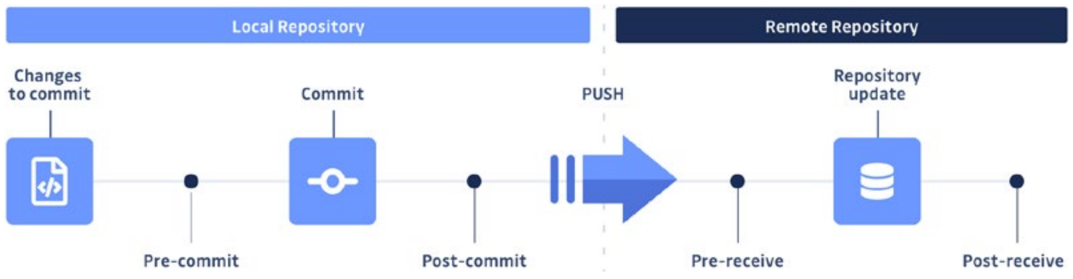


[Django contributions graph]

GitGuardian: automated secrets detection throughout the SDLC

Like SAST, DAST, dependency scanning or container scanning, secrets detection takes hard work and is an Application Security category in itself. But it's even more than that. Let us guide you through some of the key principles to automate secrets detection throughout your SDLC, and all the tools developers use (such as Slack, file sharing, ticketing systems).

WHERE IN THE SDLC TO IMPLEMENT AUTOMATED SECRETS DETECTION?



[Life of an incremental code revision (or "commit")]

The git protocol uses "hooks" to trigger certain actions at certain times in the software development process.

There are client-side hooks, that execute locally on developers' workstations, and server-side hooks, that execute on the centralized version control system.

Here are some general principles about fitting security into your DevOps pipeline:

- The earlier a security vulnerability is uncovered, the less costly it is to correct. Hardcoded secrets are no exceptions. If the secret is uncovered after the secret reaches centralized version control server-side, it must be considered compromised, which requires rotating (revoking and redistributing) the exposed credential. This operation can be complex and involve multiple stakeholders.

- People bend the rules, often in an effort to collaborate better together and do their job. Security must not be a blocker. It should allow flexibility and enable information to flow, yet enable visibility and control. On one hand, security measures will be bypassed, sometimes for the worst. But on the other hand, it is also good sometimes that the developer can take the responsibility to bypass them. Secrets detection is probabilistic: algorithms achieve a tradeoff between not raising false alerts and not missing keys. Which means that even the best algorithms can fail and need human judgement.

The previous principles advocate for the following:

- Client-side secrets detection early in the software development process is a nice to have: implement pre-commit or pre-push hooks when possible. The good thing with pre-commits is that the secret is never added to the local repository. This comes in handy since removing a secret from the git history can be very tricky, even client-side (server-side is even harder and requires to force push). Whereas the good thing with pre-push is that you've got an Internet connection there, allowing you to make API calls for example. This is not necessarily the case when committing.

- Server-side secrets detection is a must have: depending on the size of your organization, enforcing client-side secrets detection might not be an easy task, as this requires access to your developers' workstations. We've heard many times from Application Security professionals that this is not something they felt confident to do. In any case, keep in mind that client-side hooks can (and must, secret detection being probabilistic) be easy to bypass, hence the absolute necessity for server-side checks where the ultimate threat lies.

GitGuardian integrates natively with GitHub or GitLab (server-side) as a post-receive check.

You can also integrate GitGuardian anywhere in your SDLC using our API, which can be self-hosted on premise. For example, the API can be used to create a pre-push check or be integrated seamlessly in a CI pipeline.

HOW TO GET STARTED IMPLEMENTING SECRETS DETECTION?

With the nature of git comes a unique challenge. Most security vulnerabilities only express themselves in the actual version of the source code, once used in production. But old commits can contain valid secrets.

- First, scan existing code history (all commits from all branches in all projects) to start on a clean basis.
- Then continuously scan all incremental changes, every time a new commit is pushed to any branch of any project.

WHY IS IT HARD TO DETECT SECRETS?

Secrets detection is probabilistic: some secrets are easier to find than others. There is a tradeoff between low number of false alerts and low number of missed credentials.

Good secrets detection is a two-step process: harvest presumed credentials first, then get rid of your worst candidates. Each step can be achieved through a variety of methods, but it is really the subtle combination of all these methods that achieves the best performance!

STEP 1: HARVEST CANDIDATES

Method	**Pros**	**Cons**
Entropy: look for strings that appear random	• Good for penetration testing, open sourcing a project or bug bounties because it brings a lot of results. These results must be reviewed manually.	• Lots of false alerts (it is very frequent to see URLs, file paths, database IDs or other hashes with high entropy), which makes it impossible to use this method alone in an automated pipeline. • Some keys are inevitably missed because the entropy threshold to be applied depends on the charset used to generate the key and its length.
Regular expressions: match known, distinct patterns	• Low number of false alerts. • Known patterns make it easier to later check if the secret is valid or not or if this is an example or test key (see Step 2).	• Unknown key types will be missed • Credentials without a distinct pattern will be missed, which means lots of missed credentials! Think about passwords that can be virtually any string in many possible contexts, APIs that don't have a distinct format, ...

STEP 2: FILTER BAD CANDIDATES

Method	Pros	Cons
Look for known sensitive patterns in the context of the candidate. The idea is to aggregate weak signals. For example, a sensitive filename, combined with an assignment variable containing the word “key” in it, and the import of a Python wrapper for the Datadog API.	• Often allows to associate a presumed credential with a given service depending on the code surrounding it. This is helpful to validate the candidate by doing an API call, see next method!	• The notion of “context” is difficult to define (think of a large commit patch or file for example, or a variable declared in one location and used somewhere else in the repository).
Validate the candidate by doing an API call against the associated service.	• There can be no more doubt, your candidate is valid! Plus you can use the opportunity to gather information about permissions associated with the key and account owner. This information is useful for prioritization and remediation purposes.	• You need to know the associated service, or at least come up with a list of potential services. • Not all credentials can be easily checked programmatically. Think about OAuth strings, private keys, usernames and passwords, ... • Some services are not accessible from anywhere (like outside of a given private network), so the credential might be considered invalid despite still posing a threat.
Use a dictionary of anti-patterns to get rid of certain example or test keys. The presumed credential should not contain linguistic sequences of characters.	• Allows to filter certain credentials like those containing “EXAMPLE” or “TEST” or “XXXX” in them, or those found in test files or directories.	• There is no real con, this method is always good to implement, but won’t be able to filter all examples or test keys.

STEP 3: GITGUARDIAN’S SECRET SAUCE!

GITGUARDIAN’S SECRET SAUCE!

Which is not a secret anymore (as can be seen on our Twitter account!).

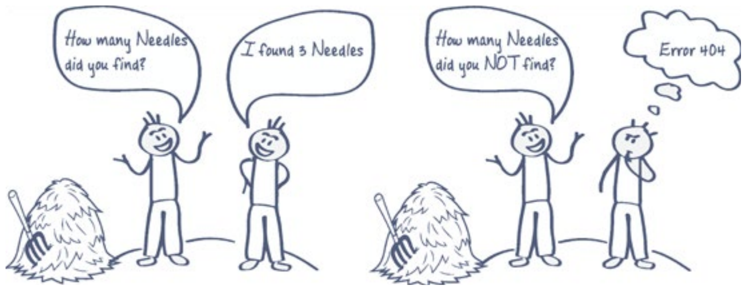
We’ve raised hundreds of thousands of alerts already, including pro bono alerts on public GitHub.

When raising alerts, we gather both implicit and explicit feedback:

- Explicit feedback when a developer or security team marks an alert as a false alert
- Implicit feedback when a developer takes down a public repository or deletes a public commit a few minutes after we sent an alert.

This feedback is then injected into our algorithms!

FINDING A SECRET IN SOURCE CODE IS LIKE FINDING A NEEDLE IN A HAYSTACK


How many Needles did you find?
I found 3 Needles
How many Needles did you NOT find?
Error 404

There are a lot more sticks than there are needles, and you don't know how many needles might be in the haystack. In the case of secrets detection, you don't even know what all the needles look like!

As a cybersecurity vendor, customers often ask us about the precision of our secrets detection algorithms. «What is the percentage of the secrets that you detect that are actual secrets?». This question is perfectly legitimate, especially in the context of security teams being overwhelmed with too many alerts.

Alarm fatigue is not the only pain. Considering the impact that a single undetected credential leak can have for an organization, we're also often asked: "How many secrets do you miss?".

Ideally, you want your detection system to achieve at the same time:

- A low number of false alerts raised, and
- A low number of secrets missed.

Balancing the equation to ensure that the algorithm captures as many secrets as possible without flagging too many false results is an intricate and extremely difficult challenge that takes a dedicated team.

WHAT ABOUT THE PROGRAMMING LANGUAGE THAT IS ANALYZED? [1/2]

This is the easy part of secrets detection, which, for the most part, is not language specific. Of course, there are some subtleties to take into account, like the way variables are assigned in any programming language. But there is no need to support all the different syntaxes in their greatest details. The same algorithms can be applied to any project, in any programming language.

WHAT ABOUT THE PROGRAMMING LANGUAGE THAT IS ANALYZED? [2/2]

A few other aspects to consider:

- When building algorithms for probabilistic scenarios, they will change over time. There is no perfect solution that can remain the same, trends will change, secrets will change, data will change, formats will change and therefore, your algorithm will need to change.
- You might want to be able to implement custom detectors, for example in order to detect API keys giving access to internal microservices specific to your company.

REMEDIATING EXPOSED SECRETS

Every time a secret is pushed to the git server, it must be considered compromised and revoked. In large organizations, remediating is often a shared responsibility between Development, Operations and Application Security teams. Revoking the secret might require special rights or approvals, some secrets might be harder to revoke than others, secrets need to be renewed and redistributed without impacting production systems and development work.

Apart from revoking the exposed secret, depending on your organization's policies, the git history might need a clean up, even if the secret is no longer active. This requires a 'git push --force', which comes with some risks as well since it might break ongoing changes derived from the working copy or cause irreparable data loss. This is a tradeoff, with no correct answer!

About GitGuardian

GitGuardian is a cybersecurity startup solving the issue of secrets sprawling within organizations, a widespread problem that leads to some secrets ending up in compromised places or in the public space. The company solves this issue by automating secrets detection for Application Security and Data Loss Prevention purposes. GitGuardian raised 12M$ in October 2019 and is backed by prominent investors including Scott Chacon, Co-Founder of GitHub, and Solomon Hykes, Founder of Docker. GitGuardian provides two tools aimed at securing two different perimeters.

The first product, GitGuardian Public Monitoring, scans all public GitHub, at scale, in real-time. The product links developers with their companies, and then monitors these developers, especially on their personal repositories, where 80% of the corporate leaks on GitHub occur. Companies often don't know that these repositories exist, don't have visibility on them, let alone the authority to enforce security measures there. The product comes in the form of a SaaS dashboard used by Incident Response, Threat Intelligence and Application Security teams to find leaked credentials, investigate and remediate quickly.

The second product, GitGuardian Internal Repositories Monitoring, scans corporate repositories, private or Open Source. The product is natively integrated with GitHub and GitLab. It includes an API as well to integrate anywhere in your SDLC and tools used by your developers. The product comes in the form of a dashboard used by Application Security teams to detect credentials and collaborate with developers to remediate quickly. Available in SaaS and On Prem.

HOW WE SELL CYBERSECURITY SOFTWARE AT GITGUARDIAN:
our Manifesto
[1/2]

Purchasing security software is hard and trust is an important factor. Our commitments:

- **We help first:** if you don't want to jump directly in a call with our sales reps, we are happy to share materials with you upfront. This way you can evaluate whether or not having a conversation with our reps is worth your time.

THIS IS HOW WE SELL CYBERSECURITY SOFTWARE AT GITGUARDIAN:
our Manifest
[2/2]

- **Consultative approach:** we will come up early in the sales process with a structured, straightforward questionnaire to help you evaluate your needs and requirements, weigh them so that you can compare us with your alternatives.
- **Radical transparency:** we are always keen on sharing the technical details of what we do with your technical teams. Even our secret sauce is, well, not a secret anymore!
- **Directness:** if we feel we are not a good fit for your needs, we will let you know early in the process, and suggest relevant alternatives.
- Products that are **easy to test:**
  - — For GitGuardian Public Monitoring: we've been monitoring the whole GitHub public activity and detecting secrets leaked there for over three years now. During the sales process, if you allow us to do so, we will show your security team the GitGuardian dashboard populated with actual data from your company's perimeter.
  - — For GitGuardian Private Monitoring: you will be given access to a free trial with unlimited features for you to test the product in real conditions before potentially buying.
- **Simple, predictable pricing.** You don't need a degree in maths to understand our quotation!

SOLOMON HYKES
Co-Founder of Docker

" Securing your systems starts with securing your software development process. GitGuardian understands this, and they have built a pragmatic solution to an acute security problem. Their credentials monitoring system is a must-have for any serious organization."

REFERENCES

algolia dashlane DATADOG GENESYS

MIRANTIS Qlik sendinblue SOCIETE GENERALE

WAVESTONE talend