When Secrets Get Committed: The role of code platforms in avoiding secret leakage

Ruining Yang[0009-0001-8169-6016], Billy Tsouvalas[0000-0002-8441-6489], and Nick Nikiforakis[0000-0002-9366-357X]

Stony Brook University, Stony Brook, NY 11794, USA
{ruiniyang, vtsouvalas, nick}@cs.stonybrook.edu

Abstract. Secret leakage refers to accidentally including sensitive tokens (such as API keys) in public software artifacts. Specific to code platforms (such as GitHub and GitLab), prior work has shown that developers commonly expose secrets in their open-source software, allowing fast-acting attackers to abuse them in order to steal data and resources. In this paper, we seek to understand the role of code platforms on curbing secret leakage, through a two-pronged analysis. First, we evaluate the ability of ten popular code platforms to detect and prevent secret leakage at the time of code publication. Next to detecting plaintext secrets, we also quantify whether these platforms are able to detect other popular secret formats, such as, Base64-encoded secrets and concatenated secrets. By deploying 300 repositories with real secrets, we find that just 20% of the evaluated platforms perform any type of secret detection, significantly underperforming popular secrets scanners.

Second, using the same controlled repositories, we perform an in-the-wild analysis of malicious scanners. Namely, over a period of one month, we observed 2,126 matched credential use attempts from 553 unique IP addresses, abusing secrets from 34 of our repositories.

Overall, our results highlight that hundreds of malicious bots are lying in wait looking for secrets that get accidentally published in open-source software. These results show that there is a real need, not just for better developer education, but also for the platforms themselves to engage in proactive detection of leaked secrets, before attackers do.

1 Introduction

Credential leakage (also known as secret leakage) in public code repositories has become a long standing problem. Developers often handle API tokens, cloud access keys, SSH private keys, and database connection strings in configuration files and CI/CD workflows. Once these files are pushed to public repositories, the accidentally-embedded credentials can now be discovered by platform scanners. Prior work and industry reports have shown that plaintext secret leakage is common in public repositories [21,9] with exposed credentials being quickly misused once publicly exposed [20].

Although developers remain primarily responsible for protecting their credentials, automated scanning by code platforms can be seen as a supplementary security tool to detect or block the exposure of sensitive information when code is submitted, without relying on every developer to configure local scanning tools. In this paper, we seek to answer the following two orthogonal questions: *i)* Do existing code platforms have any built-in protections that can proactively detect accidentally-embedded secrets? *ii)* In the absence of proactive detection, how fast are exposed secrets discovered by attackers?

To answer these questions, we construct 300 public code repositories across 10 code platforms (such as GitHub, GitLab, and Hugging Face). To quantify the platforms' abilities in detecting different types of secrets and formats, we employ 5 formats (e.g. plaintext, concatenated, and Base64) and use these formats across 6 types of credentials, including honeypot server credentials (such as, SSH and MySQL) as well as API keys for popular services (e.g. Amazon Web Services, Google Cloud Platform, and Microsoft Azure). By never reusing credentials across our public repositories, we can trace their usage back to the specific platform/repo combinations that were discovered by attackers. In this way, we are able to identify the types of secrets that attackers are able to detect en masse, as well as the code platforms that they focus on.

Our results show a wide divergence in platform and attacker abilities. On the platform front, out of the ten evaluated platforms, only two (GitHub and HuggingFace) perform any type of proactive secret scanning to protect and notify their users. Moreover, their proactive detection is limited to plaintext and Base64-encoded secrets. Contrastingly, we show that existing open-source tools that developers can use for secret scanning (such as, TruffleHog and Gitleaks) can detect significantly more types of exposed secrets.

By monitoring the usage of our intentionally leaked secrets from April 9 to May 11, 2026, we observed 2,126 use attempts from 553 unique IP addresses, tracing these back to 34 of our 300 repositories. Our results show that attackers currently focus their scanning effort on GitHub and GitLab and are only able to find plaintext and Base64-encoded secrets (matching the types of secrets that 2/10 coding platforms proactively discover). It is also worth noting that the time of discovery of different types of secrets varies across platforms and type of secret, with the fastest attacker finding and using our intentionally-leaked secret *7 seconds* after it was pushed on a code platform.

Overall, our study shows that there is substantial room for improvement in online code platforms. These platforms can assist developers by *i)* scanning code as it is pushed on the platform (80% of the platforms never warned us about any of our intentionally-leaked secrets) and *ii)* improving the rates of their secret detection, to at least match the performance of existing open-source secret scanning tools.

2 Background

2.1 Credential leakage in public repositories

Developers often handle credentials when writing application code. These credentials can appear in configuration files, environment variable templates, scripts, and infrastructure as code files. If these files are committed to a public repository, the credentials will exist not only in the contents of the current file, but also in commit history, forks, mirrors, and archives.

Prior work has shown that both the scale and use speed of these leaks are hard to ignore. Meli et al. found thousands of new secrets per day in public GitHub repositories [21]. GitGuardian also reports continued secret sprawl in public repositories [9]. For cloud credentials, Unit 42 reported malicious operations against exposed IAM keys within minutes [20]. Credential leakage in public repositories is therefore not only a sensitive string in source code. It can expose a usable identity or service entry point to external use.

2.2 Credential representation and scanning

Honeytokens are instrumented credentials that are intentionally exposed on public and private platforms. When they are used, researchers can count attempts by checking access logs on infrastructure they control. Prior honeypot and honey account studies show that this method can measure behavior after credential exposure [25,10,16,24].

Credentials make their way into repositories in several representations. For this work, the representation of secrets affects both defenders (can code platforms detect accidentally-leaked secrets?) and attackers (are attackers able to find leaked credentials, beyond the plaintext format?)

Plaintext is the baseline, where the credential appears in a directly usable form. Base64 is a reversible encoding. Kubernetes, for example, stores secret data as Base64-encoded values and explicitly warns that encoding is not encryption [8]. Split credentials are a reconstruction pattern: one value is divided into two parts and later joined by application code or deployment scripts.

Some developers choose to commit encrypted versions of their secrets in public repositories and then decrypt them on-the-fly using private (i.e. not publicly shared) keys/passwords. Given its popularity, in this paper we use Ansible Vault to cover this use case [3]. We examine two vault cases: one where the vault password remains outside the repository, and one where the vault password file and encrypted vault blob are committed to the same repository.

The scanning of code for accidentally leaked secrets can occur in two places. The first are local tools such as TruffleHog [7], Gitleaks [5], and Ggshield [4]. These tools scan local files and repositories for recognizable credential formats and high entropy strings, typically used by developers as part of their CI/CD pipelines. The second is code platforms, such as, GitHub Secret Scanning. Code platforms run such scanning after content is pushed and may notify repository owners or coordinate with providers to restrict credentials [2]. In this work, we

evaluate the ability of both local tools and code platforms to identify leaked secrets of different formats. The local tools give us a baseline to which we can compare the code platforms, since the platforms themselves could just run the exact same scanners whenever one of their users pushes new code on the platform.

2.3 Threat Model

In this paper, we assume external actors who are scanning and are able to read public repositories on the evaluated platforms. Such actors are able to run existing secret scanners or similar detection tools, inspect source code on the platforms, and connect to any service for which they recover a working credential. Rather than organization or enterprise accounts, our study focuses on individual user accounts, as they represent the most common deployment scenario.

3 Study design

In our study, we first define 6 credential types and the leakage formats used to expose them. For each credential type and format combination, we observe whether local detection tools report the credential and whether code platforms notify their users when they accidentally push code that includes credentials. We also build a controlled public repository experiment with 300 repositories across 10 code platforms, 6 credential types, and 5 leakage formats. Each repository contains a single unique credential. Observed authenticated use can therefore be attributed to a specific platform, credential type, and leakage format. These counts let us compare which credential and format combinations receive more observed authenticated use during the measurement window. Figure 1 shows our overall workflow.

3.1 Credential types and services

We selected 6 credential types to cover three access scenarios that commonly appear in public code: cloud control access, remote shell access, and database connections. The 6 types are Amazon Web Services (AWS) IAM access keys, Google Cloud Platform (GCP) service account credentials, Microsoft Azure service principal credentials, SSH private keys, MySQL connection strings, and MongoDB connection strings. These types also match credential categories reported in prior large-scale leakage measurements and industry reports [21,9].

For AWS, GCP, and Azure, we created dedicated identities in new, previously unused accounts and collected provider logs to record authenticated API events. AWS use is recorded through CloudTrail. GCP use is recorded through Cloud Audit Logs and Azure use is recorded through audit logs. These logs record the authenticated API call even when the requested operation is denied by the key's restrictive policy. We host our SSH, MySQL, and MongoDB honeypot services within a T-Pot deployment [6], which orchestrates Cowrie, Dionaea,

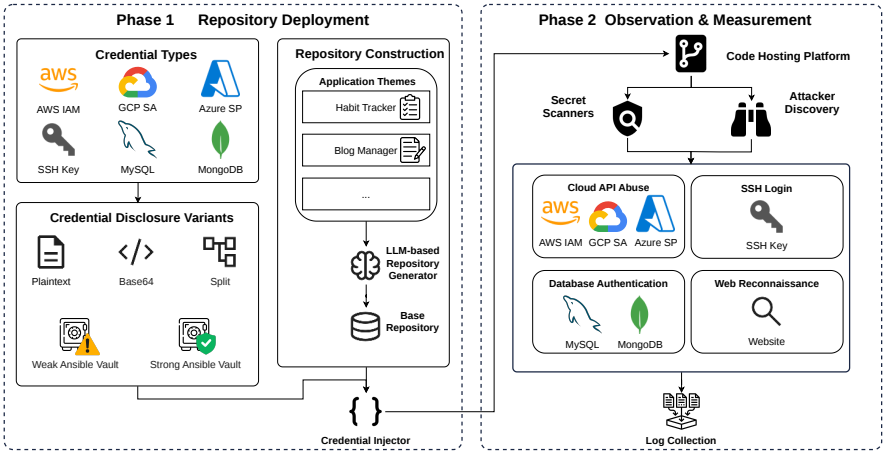


Fig. 1. Our analysis pipeline for intentionally leaking convincing secrets in code platforms and monitoring their resulting use.

and other tools under a unified system. For SSH, we deployed a Cowrie honeypot and created dedicated Ed25519 honeykeys. We disabled password login so ordinary password-bruteforcing attempts by bots are excluded from our experiments [27,23,11]. For MySQL and MongoDB, we ran Dionaea honeypots and configured them to accept only authentication attempts using the corresponding credentials placed in the repositories. All services record timestamps, credential identifiers, and hashed source IP addresses.

3.2 Leakage formats

For each credential type, we exposed one credential using 5 different leakage formats. We use these formats to compare how different representations affect platform detection as well as detection from attackers.

Plaintext. The credential is committed in a directly usable form. AWS keys appear as key value fields. GCP credentials appear as a service account JSON file. Azure credentials appear as service principal fields. SSH keys appear as PEM files. MySQL and MongoDB credentials appear as connection strings.

Base64. Each credential value is Base64-encoded. Recovering the credential requires identifying the complete encoded content and decoding it.

Concatenate. The credential value is split into two parts and stored in the same file. Each part uses part1/part2 or P1/P2 suffix to distinguish it. Recovering the credential requires concatenating the two parts in the correct order.

Ansible Vault – easy password. The credential is encrypted with Ansible Vault using a short common password, such as, 123456. The repository contains only the vault ciphertext.

Ansible Vault – hard password. The credential is encrypted with Ansible Vault using a high entropy password with more than 8 characters and a random

mix of digits and uppercase and lowercase letters. The password file and vault ciphertext are committed in the same repository.

3.3 Repository creation and platform deployment

As described in Sections 3.1 and 3.2, we study 6 credential types and 5 leakage formats. On each code platform, we create 30 repositories covering all combinations of the 6 credential types and 5 leakage formats. Each repository contains one unique credential. Credentials are not reused across repositories, platforms, credential types, or leakage formats. As such, we can later reconstruct where each of our abused secrets was extracted from.

We used an LLM to generate these repositories as small standalone Python projects. Each project has its unique name, file layout, and application theme that resemble ordinary code repositories, and contains several hundred lines of Python code on average. We verified that all 300 generated projects were executable and manually inspected a random sample of 20 repositories to confirm that their purposes matched the code logic. We do not add any repository descriptions, so they do not become another uncontrolled exposure path. Our goal was to produce code that has the appearance of legitimate open-source software, in order to avoid getting detected as potential honeypots by prospective attackers.

For code platform selection, we include mainstream platforms such as GitHub and GitLab as well as regional platforms and other public code platforms. For example, Gitcode and Gitee mainly target users in China, and Gitflic mainly targets users in Russia. We publish the repositories to GitHub, GitLab, Gitee, Hugging Face, Bitbucket, Gitcode, Codeberg, SourceForge, Gitflic, and Sourcehut. In total, we created 300 public repositories.

After initial deployment (April 9, 2026), we did not touch any repositories on any platforms for 18 days. Starting on April 28, 2026, we started pushing commits to produce continuous public activity signals for these repositories. Each repository on each platform is assigned one random commit time within each hour so that the repository remains active. These commits do not change the credential value or the leakage format. Our goal was to give the appearance of actively-developed projects as well as surface our repositories to attackers who only focus on the most recently-updated repositories.

3.4 Analysis

In this paper, we analyze the 300 repositories from three points of view. First, we use local open-source secret scanners to obtain a baseline of which secrets can be readily identified using readily-available, off-the-shelf code scanners. Second, we focus on proactive detection by the code platforms themselves, seeking to understand whether these platforms engage in secret leakage detection, in an effort to protect their own users against attacks. Last, we analyze the in-the-wild abuse of all intentionally leaked secrets across all services and platforms. Sections 4.1, 4.2, and 4.3 report these results in this order.

Local scanners. We first use local scanners to scan all 5 leakage formats for the 6 credential types. To cover different representations of the 6 credentials, we construct as many concrete representations as possible for each credential. For SSH, we test scanning results for both OpenSSH and EC keys. For MySQL, we test standard MySQL connection strings, such as `mysql://user:pass@host/db`, and JDBC format MySQL Java strings, such as `jdbc:mysql://`. For AWS, Azure, and GCP, we compare whether the variable name contains a keyword that local scanners may use, such as "api." The only exception is GCP plaintext. It is a complete JSON file, so we cannot manually add a specified keyword to a variable name when storing it. We opted to use three popular open-source scanners (TruffleHog, Gitleaks, and Ggshield) which we executed in isolated environments. We use only their default rule catalogs without further configuration.

Platform responses. A platform's response to leaked credentials is critical. Early identification of these credentials (ideally at push time), can reduce the usable exposure time after the credentials become public. We test whether different platforms block the push containing a credential, send an alert at the push, or only produce an alert after the push succeeds when we push repositories containing different credential types. After each repository is pushed to a platform, we monitor the repository through the platform UI and email notifications.

Authenticated use logs. All of our intentionally leaked secrets are real secrets that we generated for the purposes of this study. As such, we are able to record when attackers attempt to use them to perform an action at third-party API or get access to our deployed honeypots. A successful attempt is an authenticated cloud API call, an SSH login attempt using one of our honeykeys, or a database connection that authenticates with one of our credentials. These attempts are recorded in provider logs and at honeypot endpoints. Each credential carries an identifier that maps to exactly one platform, credential type, leakage format, and repository. Each successful attempt can therefore be attributed unambiguously to a specific repository. We also record the interval between repository deployment and the first successful attempt of that credential. For each of the six services, we issued the authentication from an external address using all credentials deployed in the repositories and they all appeared in the corresponding logs. All authentications were captured.

For SSH, MySQL, and MongoDB credentials, every repository includes the hostname of the corresponding honeypot service alongside the credential value, across all 5 leakage formats. Writing a hostname rather than a raw IP reduces the identifying signal created when repositories on different platforms share the same endpoint. We registered 10 domains and configured a separate subdomain for each (service, repository) combination, where each subdomain is a randomly generated string of letters and digits. Because these subdomains are never published outside their corresponding repositories, only a visitor that has read the repository content can know the hostname. In total, 150 hostnames (50 SSH, 50 MySQL, 50 MongoDB) are distributed across the 10 domains. We monitor access records for these hostnames to distinguish probes that only request web resources from attempts that actually use credentials to access the SSH and

database services. Requests for these web resources that do not complete authentication against any honeypot service are not counted as authenticated use attempts.

3.5 Ethical Considerations

Before any deployment of repositories or credentials, our study was reviewed and approved by our institution's IRB. Considering that our repositories are publicly available online and were not advertised, it is possible (although unlikely) that benign users interact with them. As such, we hashed the source IP addresses of requests towards our SSH, MySQL, and MongoDB instances, to ensure that no identifying information was collected. Moreover, since our repositories consist of otherwise innocuous scripts, there is no risk associated with benign users interacting with them. Last, in order to ensure that our deployed credentials are realistic, yet do not allow malicious users access to any funds, we employed real keys from reputable providers (AWS, GCP, Azure), which were restricted in terms of usage and did not allow for credit-consuming calls to complete.

4 Results

4.1 Credentials scanning tools

As described in Section 3.4, we evaluate the security scanning tools TruffleHog, Gitleaks, and Ggshield since these form our baseline for what code platforms should be able to readily do. Each scanner employs its default rules, without the addition of any custom patterns. In Table 1, we present the coverage of these tools across different credential formats.

The credential types were tested in multiple sub-variants: SSH keys in both OpenSSH and EC encodings, MySQL credentials as both standard and JDBC-format strings, and cloud credentials (AWS, Azure, and GCP) were evaluated both with and without a keyword, e.g., `API`, in the variable name.

Overall, Ggshield detected the most credentials with 26 of 55 (47.3%) cases covered, followed by Gitleaks with 24 of 55 (43.6%), and TruffleHog with 20 of 55 (36.4%). We highlight that none of the three scanning tools managed to detect any of the 22 Ansible Vault credential test cases.

Plaintext and Base64. The three scanning tools detected most of the plaintext and Base64 credential types. Regarding the Base64 format, TruffleHog and Gitleaks achieve this by decoding Base64 values before scanning, while Ggshield employs ad-hoc Base64 detectors (e.g., "Base64 AWS keys") to directly match the encoded string.

Database credentials present a different picture. Standard MySQL connection strings (e.g., `mysql://user:pass@host/db`) were caught only by Ggshield, JDBC-format MySQL strings and MongoDB credentials were detected by TruffleHog and Ggshield. The key differentiator across all three is whether a format-specific detector exists in the tool's default ruleset, e.g., Gitleaks does not include the JDBC scheme as a recognizable prefix.

Table 1. Credentials detection across tools and credential formats. Y = detected; N = undetected.

Credential	Variant	Format	TruffleHog	Gitleaks	Ggshield
SSH	OpenSSH	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	N
	EC	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	N
MySQL	Standard	Plaintext	N	N	Y
		Base64	N	N	N
		Concatenate	N	N	N
	JDBC	Plaintext	Y	N	Y
		Base64	Y	N	N
		Concatenate	N	N	Y
MongoDB		Plaintext	Y	N	Y
		Base64	Y	N	N
		Concatenate	N	N	N
AWS	w/ keyword	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	Y
	w/o keyword	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	Y
Azure	w/ keyword	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	Y
	w/o keyword	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	Y
GCP	w/ keyword	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	Y
	w/o keyword	Plaintext	Y	Y	Y
		Base64	Y	Y	Y
		Concatenate	N	Y	Y

Concatenate. TruffleHog failed to detect any concatenated credential, as its detectors require a structurally complete value to match. On the other hand, Gitleaks and Ggshield were partially successful, detecting concatenated AWS, Azure, and GCP credentials (where split halves retain enough entropy or structure to trigger their detectors). Beyond these cases, concatenate coverage was sparse, with MySQL and MongoDB credentials going largely undetected.

Ansible Vault. None of the three tools detected any credential encrypted with Ansible Vault, regardless of the password difficulty. Although every Ansible Vault ciphertext begins with the fixed prefix `$ANSIBLE_VAULT;1.1;AES256`, none of the three tools' default rulesets includes a detector for it, leaving all vault-encrypted credentials undetected.

Variable naming. We additionally evaluated whether the presence of a keyword in the variable name influences detection. Specifically, we evaluated each

Table 2. Platform profile.

Platform	Tranco Rank [1]	Notifications
GitHub	29	AWS plaintext, AWS Base64, GCP plaintext
SourceForge	224	—
GitLab	427	—
Hugging Face	1,831	AWS plaintext, AWS Base64
Bitbucket	2,129	—
Gitee	3,682	—
Codeberg	4,643	—
Sourcehut	30,813	—
Gitflic	33,823	—
Gitcode	51,196	—

credential type with and without the string `API` in the variable name. The keyword had no effect on detection outcomes across all tools and credential types. This confirms that the underlying detectors for AWS, Azure, and GCP credentials operate on credential structure/format or verification through the provider's API, rather than the context of the variable's name.

4.2 Credentials scanning on code platforms

In Table 2, we present the evaluated code platforms and the credential leakage notifications we received from each of them during the observation window. Among the 10 code platforms where we deployed credentials, only GitHub and Hugging Face sent us notifications. For the 8 remaining platforms, none of them enforced any restrictions or informed us of any post-publication credential leakage.

Specifically for GitHub and Hugging Face, we observe narrow coverage. GitHub notified us only for AWS plaintext, AWS Base64, and GCP plaintext, while Hugging Face notified us only for AWS plaintext and Base64. Overall, none of the platforms notified us regarding any database or SSH credential types, while no notifications were produced for the concatenate, or Ansible Vault formats.

Regarding the actions taken by the platforms that detected the credentials and notified us, GitHub enforced push protection at the `git push` stage for the AWS plaintext, Base64, and for the GCP plaintext credentials. This resulted in the commit being blocked until we selected a bypass reason on the platforms interface (e.g., choosing the `"I'll fix it later"` option). Once the block was bypassed, the push completed. At this stage, GitHub notified the corresponding cloud providers of the credential leakage. This was carried out via a secret scanning partner program between Github and popular providers (e.g., AWS, GCP).

Thus, AWS restricted API access to the affected keys, while GCP revoked the detected plaintext token outright, rendering it unusable.

In contrast to GitHub, Hugging Face only sent us email notifications. We observed no restriction or revocation to the credentials. We also note that the Hugging Face notifications arrived approximately one week after the repository publication, and the credentials remained usable and could produce authenticated use. Hugging Face email notifications for AWS plaintext and AWS Base64 appeared during our pilot deployment but did not recur during the main study deployment. The two deployments used different Hugging Face accounts, though in both cases the accounts were newly created and the AWS credentials were freshly generated. This indicates further latency or inconsistency regarding the notifications of credential leakage.

Table 3. Authenticated use by credential type and format.

	plaintext			Base64			concatenate	Ansible Vault easy	Ansible Vault hard
	hits	fastest	platform	hits	fastest	platform	hits	hits	hits
AWS	6/10	8min	GitHub	3/10	29min	GitLab	0/10	0/10	0/10
Azure	4/10	26min	GitLab	3/10	7s	GitLab	0/10	0/10	0/10
GCP	0/10	—	—	2/10	19.9d	GitLab	0/10	0/10	0/10
SSH	0/10	—	—	0/10	—	—	0/10	0/10	0/10
MySQL	5/10	27min	GitLab	4/10	25min	GitLab	0/10	0/10	0/10
MongoDB	4/10	24s	GitHub	3/10	27min	GitLab	0/10	0/10	0/10

According to GitHub's documentation [2], organization repositories under Team or Enterprise Cloud accounts with Secret Protection and non-provider patterns enabled may receive additional alerts. As discussed in our threat model of Section 2.3, we focus on individual user accounts, which represent the most popular deployment scenario and are often left unprotected by default.

4.3 Credential leakage

In this section, we present the authenticated requests made towards the deployed credentials over our one month deployment. Overall, we recorded 2,126 matched credential attempts from 553 unique IP addresses across 34 of the 300 deployed repositories and respective credentials.

Credential usage. In Table 3 we report the authenticated credential hits across repositories and credential formats. Moreover, we present the time interval between the repository publication and the first authenticated use of credentials. Matched attempts occurred only for plaintext and Base64 credentials, with AWS, Azure, MySQL, and MongoDB each receiving hits in both formats. GCP credential usage was limited to two Base64 repositories, while SSH, concatenate, and both Ansible Vault formats received no matched use.

We also analyzed requests to all 150 unique honeypot hostnames embedded in the SSH, MySQL, and MongoDB repositories. After excluding our validation traffic, only one hostname received external access: a single IP address generated seven HTTP requests over nine seconds to the hostname embedded in a Gitflic SSH concatenate repository, with no subsequent SSH authentication attempt. Because all subdomains are unique and randomly generated, these requests could only have come from a visitor who scanned the repository content.

Time from push to credential use. The fastest post-push credential usage was 7 seconds for an Azure Base64 credential on GitLab. Moreover, we observed very quick turnaround for a MongoDB plaintext credential on GitHub (24 seconds), and for an AWS plaintext credential on GitHub (8 minutes).

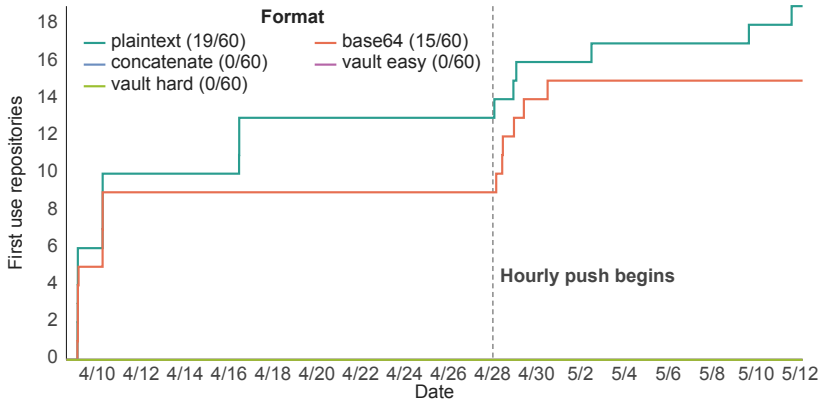


Fig. 2. First authenticated use by format.

The sub-minute first-use times suggest that automated scanners are continuously monitoring platform activity feeds and attempt credential use immediately upon discovery. This highlights the critical role of pre-push secret scanning, both in developer workflows and as a platform-enforced gate, to prevent credentials from ever reaching public repositories.

Credential use by format. Figure 2 shows the cumulative count of credentials receiving their first authenticated attempt over time, split by credential format. Plaintext credentials produced their first use in 19 of 60 (31.7%) repositories, and Base64 in 15 of 60 (25.0%), with plaintext consistently ahead throughout the window. Both curves accelerated after we began our hourly code pushes, as described in Section 3.3. Attackers made 1,039 attempts against plaintext credentials and 1,087 against Base64 ones, together accounting for all 2,126 matched attempts. The parallel growth of both curves confirms that Base64 encoding

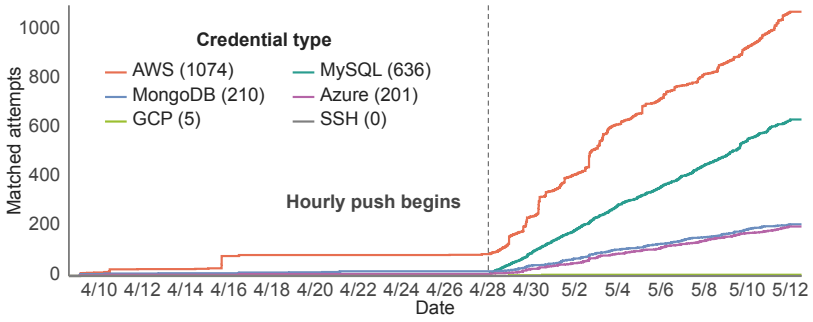


Fig. 3. Cumulative matched attempts by credential type.

provides no meaningful barrier against discovery, as attackers scan for and use encoded credentials at nearly the same rate as plaintext ones.

We provide a summary of the time intervals between push and credential use per platform in Table 5 of the Appendix.

Credential use volume. In Figure 3 we present the total credential attempts over time split by the credential type. AWS accounted for 1,074 attempts, MySQL for 636, MongoDB for 210, Azure for 201, and GCP for 5. SSH produced no matched credential use.

In terms of the volume of authenticated credential usage, we observe a different concentration pattern before and after the hourly push threshold. Before we started the automated hourly pushing to our repositories, for example, AWS attempts averaged 4.7/day (89 attempts over 19 days). After we began pushing hourly on April 28, AWS attempts rose to 70.4/day (985 attempts over 14 days).

In Table 4 we break down the 2,126 authenticated credential attempts by platform and credential type. Attempt volume is heavily concentrated on two platforms, namely GitHub and GitLab. These two platforms account for 1,940 attempts (91.3%). Regarding the credential type distribution among platforms whose credentials were used, we observe that GitLab attracted the bulk of AWS attempts with 664 of 1,074 (61.8%), whereas GitHub dominated MySQL with 548 of 636 (86.2%), Azure with 138 of 201 (68.7%), and MongoDB with 137 of 210 (65.2%) attempts. GCP accumulated only 5 attempts in total, while the SSH credentials were not used.

This concentration pattern likely reflects the scanning infrastructure that attackers use: tools and scripts are predominantly tuned to monitor the activity feeds of the most popular platforms, while different scanning pipelines target different credential types. Platforms with ranking lower in popularity or primarily focusing on regional audiences, appear to fall outside the reach/interest of most automated credential harvesters.

For completeness, in Table 6 of the Appendix, we present the distribution of source IP addresses that register hits on the deployed credentials in terms of ASNs and countries.

Table 4. Platform × credential attempts.

Platform	AWS	Azure	GCP	SSH	MySQL	MongoDB	Total
GitHub	261	138	2	0	548	137	1,086
GitLab	664	58	3	0	61	68	854
Hugging Face	142	3	0	0	5	0	150
Codeberg	0	0	0	0	20	0	20
Bitbucket	2	2	0	0	2	2	8
Gitcode	1	0	0	0	0	3	4
Gitflic	4	0	0	0	0	0	4
Gitee	0	0	0	0	0	0	0
SourceForge	0	0	0	0	0	0	0
Sourcehut	0	0	0	0	0	0	0
Total	1,074	201	5	0	636	210	2,126

5 Related work

Credential leakage. Prior work has primarily measured the prevalence of credential leakage in public repositories. Specifically, Meli et al. [21] measured secret leakage in public GitHub repositories and found thousands of newly exposed secrets per day, with many of them remaining accessible long after the initial commit. Feng et al. [19] combined password text and code context to identify plaintext passwords in over sixty thousand GitHub repositories. Last, Dahlmanns et al. [17] extended these measurement studies beyond source code platforms to include container images. The authors scanned 337,171 DockerHub images and over 8,076 private registries, finding secrets in 8.5% of the evaluated images. Furthermore, they confirmed 275,269 Internet-reachable hosts that could be authenticated to using the leaked credentials found in the container images. Recognizing the prevalence of credential leakage, in our study, we evaluate whether code platforms proactively detect credentials, and whether attackers find and use them in the wild.

Credential scanning. A parallel line of work evaluates tools for detecting secrets and credentials in code. Basak et al. [12] compared nine production-grade secret scanners and found that no single tool dominates across both precision and recall. Moreover, they attributed the detection inaccuracies to overly generic regular expressions and incomplete rulesets. Another work in 2025, proposed AssetHarvester [13], a static analysis tool that employs the position of secrets in regard to the asset they refer to in order to enhance the detection scheme. Last, Biringa and Kul [15] employed a machine learning approach, training BERT and GPT-2 classifiers on hard-coded credentials for credential detection. This approach yielded improved F1 scores, and established machine learning as a valid angle for credential detection. In our work, we extend this line of work, by employing acknowledged secret scanning tools (TruffleHog, Gitleaks, and Ggshield) against credentials of different formats (encoding, concatenation, etc.).

Honeypots. Honeypot research uses controlled decoy infrastructure to observe attacker behavior, and has been employed across a wide range of security research contexts, from network intrusion detection and malware analysis to studying the

tactics and tooling of adversaries at scale. Provos [25] introduced Honeyd, a virtual honeypot framework that simulates virtual hosts with configurable services at the network level. More recently, Munteanu et al. [23] reported on a three-year SSH honeynet covering 163 million attacker command execution sessions and documented growing attacker awareness of honeypot infrastructure. Song et al. [28] deployed 278 database honeypots spanning MySQL, MSSQL, PostgreSQL, Redis, Elasticsearch, and MongoDB. In the 58 medium and high interaction honeypots, they observed three categories: direct database manipulation, copying and deletion of targeted data, and use of the database to compromise underlying systems. We similarly deploy SSH and database honeypots to record authentication attempts. We use them as passive sensors of credential abuse and trace each attempt to a repository and credential format.

Honeytokens. Honeytokens are identifiable decoys whose use signals unauthorized access. In terms of honeytoken creation and generation, Bercovitch et al. [14] introduced HoneyGen. This automated system generates honeytokens across multiple types, establishing the principle that honeytokens must be realistic enough to be indistinguishable from genuine credentials. Reti et al. [26] extended this idea by using large language models for honeytoken generation, showing that LLM-generated honeytokens can match or exceed human-crafted ones in realism across a range of credential types. Regarding the distribution of honeytokens, Akiyama et al. [10] proposed HoneyCirculator, which distributes credential honeytokens and lures adversaries to a decoy server impersonating a compromised content management system. Examining the credential leak ecosystem, Onaolapo et al. [24] created 100 fake Gmail accounts and deliberately leaked their credentials online. They classified intruders in four categories; curious, gold diggers, spammers, and hijackers, while a key finding was that honey accounts are often accessed quickly after exposure. Extending the monitoring of compromise to full accounts, DeBlasio et al. [18] registered a unique honey email account at each of 2,300 third-party websites. As such, any subsequent unauthorized access to one of those accounts served as a passive tripwire revealing that the corresponding site was breached. Last, and closest to our approach, Mouw [22] directly measured the abuse of exposed secrets from public repositories, profiling how different secret types are discovered and used after being committed to code platforms. Our work extends this in three dimensions: we deploy credentials across ten code platforms, we systematically vary the credential format, covering formats beyond plaintext, and we evaluate six different credential types.

6 Discussion and Limitations

Overall, our results show that existing secret scanning tools (i.e. TruffleHog, Gitleaks, and Ggshield) are able to detect plaintext and Base64 credentials with reasonably-high accuracy. However, their coverage is rule-based and format-specific. Each tool detects credentials by matching against a hardcoded list of patterns tied to specific credential formats (e.g., AWS key prefix and JDBC connection strings), rather than any semantic understanding of what constitutes a

credential. This is most evident for Ansible Vault-encrypted files, where, despite every vault file beginning with the same fixed prefix, none of the tools includes a detector for it, leaving all vault-encrypted credentials undetected.

In terms of the credentials scanning taking place on code platforms, among the ten platforms evaluated, only GitHub and Hugging Face informed us of the possible credential leakage. The rest of the platforms (80%) produced no warnings, errors, or notifications to these credentials being made publicly available. Although GitHub and Hugging Face issued notifications, their coverage was narrow, limited to two or three credential types out of the six we tested. Last, we note that Hugging Face's alerts arrived approximately one week after the credential was leaked, which is well within the window for credential abuse. Beyond simply detecting, GitHub not only blocked the offending push but also notified the relevant cloud providers, triggering key restrictions by AWS and token revocation by GCP. In contrast, Hugging Face had no abuse-prevention mechanism other than the email notification. As such, we highlight that both the accurate detection of credentials at push time, as well as the timely notification significantly reduce the window in which an exposed credential can be abused. Even with GitHub's stronger response, individual developers remain outside its enterprise-tier protections. Because platform protections are incomplete, developers should run secret scanners before committing or pushing code and immediately revoke any credential that is accidentally exposed.

Last, our results regarding the authenticated use of leaked credentials offer three takeaways. First, Base64 offers no meaningful barrier after exposure, with attackers using Base64 credentials at nearly the same rate as plaintext (1,087 Base64 attempts vs. 1,039 plaintext attempts). Second, observed authenticated use was concentrated in AWS and database credentials; GCP received only 5 attempts across two Base64 credentials, while SSH received none. However, the GCP plaintext result should not be interpreted as evidence of attacker disinterest. GCP revoked the plaintext credential hosted on GitHub before it could be used, while the other nine plaintext credentials also received no matched use. Therefore, platform intervention explains only the GitHub case, not the aggregate 0/10 result. This case nevertheless shows that push time detection and immediate revocation can prevent downstream abuse. Third, scanning activity is platform-dependent: GitHub and GitLab alone accounted for 1,940 of 2,126 (91.3%) attempts, while other platforms saw little-to-no attacker traffic.

Limitations. Our measurements capture authenticated use of the credentials leaked in our repositories. However, this only represents a lower bound on attacker activity. As the hostname access on Gitflic demonstrates, a repository may be found without producing an authentication event. Our logs cannot determine whether the visitor failed to reconstruct the credential or reconstructed it but chose not to use it. Similarly, the absence of authenticated use for concatenate and Ansible Vault credentials does not mean these representations are unrecoverable; it only means we did not observe their use. Regarding the attribution of visits to unique source IP addresses, we note that IP-level attribution carries its

own uncertainty. A single attacker may operate across multiple IP addresses and a single address may represent shared infrastructure used by many independent actors. As such, our IP address counts reflect the distribution of access sources, rather than the number of distinct actors behind them.

7 Conclusion

Secret leakage in software is an important and largely unsolved problem. While the parties ultimately responsible are the developers pushing code, it is worthwhile to understand the role of code platforms in reducing its effects on the software ecosystem. In this paper, we investigated how code platforms respond to leaked credentials at publication time, what happens when these credentials remain exposed in public repositories, and we evaluated the detection coverage of secret scanning tools. We evaluated 6 credential types across 5 formats and deployed 300 public repositories with credentials attached to real cloud and honeypot services. Current platform protections were found to be limited or entirely absent: only two of the ten evaluated platforms (GitHub and Hugging Face) issued any notifications, and even those offered only partial coverage, flagging a narrow subset of credential types while leaving the rest undetected. In terms of the consequences of credential leakage, during our month-long deployment, we recorded 2,126 matched credential use attempts from 553 unique IP addresses involving credentials exposed in 34 of the 300 public repositories. With platforms providing limited or, in some cases, no proactive defenses against credential leakage, adversaries are free to find and exploit credentials at scale, while the users are left unprotected. While no single measure eliminates the risk of credential leakage, our secret scanning tool evaluation demonstrates that code platforms are falling short of even the baseline protection that freely available open-source scanners already provide.

Availability: To encourage further research regarding secret leakage, we will release our source code and measurement artifact upon publication.

Acknowledgments

We thank the reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-24-1-2193 and by the Army Research Office under grant W911NF-24-1-0051.

References

1. A research-oriented top sites ranking hardened against manipulation - Tranco — tranco-list.eu. `https://tranco-list.eu/`, [Accessed 10-05-2026]
2. About secret scanning - GitHub Docs — docs.github.com. `https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning`

3. Ansible Vault — Ansible Community Documentation — docs.ansible.com. https://docs.ansible.com/ansible/latest/vault_guide/vault.html, [Accessed 20-04-2026]
4. GitHub - GitGuardian/ggshield: Detect and validate 500+ types of hardcoded secrets with advanced checks. Use it as a pre-commit hook, GitHub Action, or CLI for proactive secret detection and security. — github.com. https://github.com/{G}it{G}uardian/ggshield, [Accessed 20-04-2026]
5. GitHub - gitleaks/gitleaks: Find secrets with Gitleaks — github.com. https://github.com/gitleaks/gitleaks
6. GitHub - telekom-security/tpotce: T-Pot - The All In One Multi Honeypot Platform — github.com. https://github.com/telekom-security/tpotce, [Accessed 27-05-2026]
7. GitHub - trufflesecurity/trufflehog: Find, verify, and analyze leaked credentials — github.com. https://github.com/trufflesecurity/trufflehog
8. Secrets — kubernetes.io. https://kubernetes.io/docs/concepts/configuration/secret, [Accessed 20-04-2026]
9. State of Secrets Sprawl Report 2023 — gitguardian.com. https://www.gitguardian.com/state-of-secrets-sprawl-report-2023
10. Akiyama, M., Yagi, T., Hariu, T., Kadobayashi, Y.: Honeycirculator: distributing credential honeytoken for introspection of web-based attack cycle. International Journal of Information Security **17**(2), 135–151 (2018)
11. Barron, T., Nikiforakis, N.: Picky Attackers: Quantifying the Role of System Properties on Intruder Behavior. In: Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC) (2017)
12. Basak, S.K., Cox, J., Reaves, B., Williams, L.: A comparative study of software secrets reporting by secret detection tools. In: International Symposium on Empirical Software Engineering and Measurement (ESEM) (2023)
13. Basak, S.K., English, K.V., Ogura, K., Kambara, V., Reaves, B., Williams, L.: Assetharvester: A static analysis tool for detecting secret-asset pairs in software artifacts. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 01–13. IEEE (2025)
14. Bercovitch, M., Renford, M., Hasson, L., Shabtai, A., Rokach, L., Elovici, Y.: Honeygen: An automated honeytokens generator. In: Proceedings of International Conference on Intelligence and Security Informatics (2011)
15. Biringa, C., Kul, G.: Detecting hard-coded credentials in software repositories via llms. Digital Threats: Research and Practice **6**(3), 1–16 (2025)
16. Bursztein, E., Benko, B., Margolis, D., Pietraszek, T., Archer, A., Aquino, A., Pitsillidis, A., Savage, S.: Handcrafted fraud and extortion: Manual account hijacking in the wild. In: Proceedings of the 2014 conference on internet measurement conference. pp. 347–358 (2014)
17. Dahlmanns, M., Sander, C., Decker, R., Wehrle, K.: Secrets revealed in container images: an internet-wide study on occurrence and impact. In: Proceedings of the ACM Asia Conference on Computer and Communications Security (2023)
18. DeBlasio, J., Savage, S., Voelker, G.M., Snoeren, A.C.: Tripwire: Inferring internet site compromise. In: Proceedings of the Internet Measurement Conference (2017)
19. Feng, R., Yan, Z., Peng, S., Zhang, Y.: Automated detection of password leakage from public github repositories. In: Proceedings of the 44th International Conference on Software Engineering. pp. 175–186 (2022)
20. Gamazo, W., Quist, N.: CloudKeys in the Air: Tracking Malicious Operations of Exposed IAM Keys. https://unit42.paloaltonetworks.com/malicious-operations-of-exposed-iam-keys-cryptojacking

21. Meli, M., McNiece, M.R., Reaves, B.: How bad can it git? characterizing secret leakage in public github repositories. In: NDSS (2019)
22. Mouw, M., Cox, M., Mijnen, F.: Profiling abuse of exposed secrets in public repositories. Ph.D. thesis, Ph. D. dissertation (2020)
23. Munteanu, C., Suriyanarayanan, Y.B., Smaragdakis, G., Feldmann, A., Fiebig, T.: Attacks come to those who wait: Long-term observations in an ssh honeynet. In: Proceedings of the ACM Internet Measurement Conference (2025)
24. Onaolapo, J., Mariconti, E., Stringhini, G.: What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In: Proceedings of the 2016 internet measurement conference. pp. 65–79 (2016)
25. Provos, N.: A virtual honeypot framework. In: USENIX Security Symposium. vol. 173, pp. 1–14 (2004)
26. Reti, D., Becker, N., Angeli, T., Chattopadhyay, A., Schneider, D., Vollmer, S., Schotten, H.D.: Act as a honeytoken generator! an investigation into honeytoken generation with large language models. In: Proceedings of the 11th ACM Workshop on Adaptive and Autonomous Cyber Defense. pp. 1–12 (2024)
27. Singh, S.K., Gautam, S., Cartier, C., Patil, S., Ricci, R.: Where the wild things are:{Brute-Force}{SSH} attacks in the wild and how to stop them. In: 21st USENIX Symposium on Networked Systems Design and Implementation (2024)
28. Song, Y., Smaragdakis, G., Griffioen, H.: Decoy databases: Analyzing attacks on public facing databases. In: Proceedings of the 2025 ACM Internet Measurement Conference. pp. 709–726 (2025)

A Supplementary tables

Table 5. Platform first use summary.

Platform	Fastest Hit	repos
GitLab	7s	9/30
GitHub	24s	9/30
Bitbucket	1.2d	8/30
Hugging Face	7.4d	4/30
Gitcode	7.4d	2/30
Gitflic	7.4d	1/30
Codeberg	30.6d	1/30
Gitee	—	0/30
SourceForge	—	0/30
Sourcehut	—	0/30

Table 6. Top source autonomous systems and source countries, ranked by unique IP addresses.

Organization	IP addresses	Attempts	Country	IP addresses	Attempts
AMAZON-AES	312 (56.4%)	409 (19.2%)	United States	480 (86.8%)	1,219 (57.3%)
GOOGLE-CLOUD-PLATFORM	138 (25.0%)	226 (10.6%)	Netherlands	18 (3.3%)	161 (7.6%)
NETROUTING-AS	9 (1.6%)	84 (4.0%)	United Kingdom	11 (2.0%)	195 (9.2%)
BLOOM-HOST	6 (1.1%)	175 (8.2%)	France	10 (1.8%)	97 (4.6%)
OVH	6 (1.1%)	24 (1.1%)	Germany	7 (1.3%)	23 (1.1%)
HETZNER-AS	6 (1.1%)	22 (1.0%)	Russia	6 (1.1%)	123 (5.8%)
CDNEXT	5 (0.9%)	10 (0.5%)	Singapore	3 (0.5%)	68 (3.2%)
LITESERVER	5 (0.9%)	11 (0.5%)	Poland	3 (0.5%)	51 (2.4%)
CLOUDFLARENET	4 (0.7%)	28 (1.3%)	Brazil	2 (0.4%)	23 (1.1%)
INFRALY-LLC	4 (0.7%)	6 (0.3%)	Romania	2 (0.4%)	2 (0.1%)
Others (42)	58 (10.5%)	1,131 (53.2%)	Others (10)	11 (2.0%)	164 (7.7%)
Total (52)	553	2,126	**Total (20)**	553	2,126

Table 7. IP address spread and repeated matched attempt concentration.

A. IP address spread

Group	IP addresses	Percentage
Total unique IP addresses	553	
Distinct services per IP address		
1 service	475	85.9%
2 services	53	9.6%
3 services	16	2.9%
4 services	9	1.6%
Distinct repositories per IP address		
1 repository	450	81.4%
2 repositories	66	11.9%
3 to 5 repositories	29	5.2%
≥6 repositories	8	1.4%

B. Top repeated IP addresses

IP address	Attempts
`cf7b161fee9f...`	297 (14.0%)
`211e7ae6a937...`	142 (6.7%)
`a8328af09527...`	125 (5.9%)
`128d980d38e8...`	109 (5.1%)
`eb9641fe43db...`	63 (3.0%)
`1a35a267a83b...`	62 (2.9%)
`8ec770e6a9ef...`	62 (2.9%)
`49d0633e945d...`	52 (2.4%)
`38b24592f39c...`	46 (2.2%)
`31423d599997...`	39 (1.8%)
Others (543 IP addresses)	1,129 (53.1%)
Total (553 IP addresses)	2,126