Control for Secret Leakage in an Age of Cloud

Liang Huang, and Chandler C. Vaughn

Abstract—**In this paper, we explore public source code repositories on GitHub, with the aim of understanding the extent that internal corporate passwords and Cloud-related credentials are leaking out into the public domain through the regular use of these source code repositories. We utilize publicly accessible information, infrastructure and tools to search for and collect candidate source code files that may inadvertently hold secrets. We then analyze these source code files for secret credential and login patterns with a targeted list of popular subsystems and API credential patterns, and then analyze and summarize the results. Finally, we present a recommended proxy system that can safeguard identity of workloads and applications, as well as keep centralized control and secrecy over secrets. The hope being that a system such as the one we propose can both enable an IT department with the necessary security policy and controls for secrets, while enabling a distributed set of development resources a path to building applications without risking exposure of credentials in source code.**

Index Terms — **Computer Security; Data Security; GitHub; Information Security; Open Source; Source Code Repository; Security Management; Secret Management; Software Engineering**

I. Introduction

We have seen tremendous growth, productivity, and progress over the past several decades in terms of software development, tooling, automation, and software delivery times. Along with this evolution, many organizations have embraced new ways of developing software, along with modern day tooling to help accelerate the processes. Additionally, we have seen companies, both small and large, embrace an Open Source movement where portions of a company's source code are released to create public projects [1]. GitHub, a well-known public and private source versioning and repository, has grown along with these trends. This has happened so much so, that knowing how to use GitHub is often viewed as a requirement in software development job interviews [2].

However, we also live in a Cloud and Internet related environment where everything is connected, and subsequently everything is vulnerable to attack. We have also seen that with accelerating speed of development, more mistakes can be made that potentially expose companies to unknown risks and attack vectors [2]. One primary risk is that so called "empowered" individuals within an organization accidently leak critical security information to the public, without anyone knowing. Often secrets are either stored directly in the text of source code or stored in configuration files. The latter gives some broad protection if the directory is secured, but the overarching potential for secret leakage poses several questions:

- How is key rotation handled in the company as employees leave the company?
- If there is a security breach of secret leakage, how long does it take to mitigate the risk?
- Is there an ability to audit who has accessed a secret, and when it was accessed?
- Is there a methodology and structure to implement different levels of secrets that limit authorization to only what is required for an application, service, or employee?
- Is the responsibility of keeping secrets secret in the hands of each individual employee where it is difficult to control, or is it controlled and verified centrally?

We posit that many open source projects within GitHub, run by companies, pose an unnecessary risk to organizations. This is likely due to the fact that many do not have good code scanning in place to highlight the potential for exposure of security credentials, network topologies, and internal systems. Through an eagerness to deliver, and what might be considered an aggressive approach to releasing software quickly, we hypothesize that many company's internal processes fail to catch leaks of secrets and credentials to public forums. Once leaked, even for short time, secrets should be considered compromised. Further, code repositories such as GitHub are specifically designed to track changes. Once a commit is made, removing a credential or secret from the history of the repository is both cumbersome to perform, and difficult to ensure.

We intend to analyze, through a structured approach, the extent to which credentials are exposed by many of these well-intending companies and individual developers. We do this by mining GitHub for potential files that have secrets or credentials stored within them. GitHub is unfortunately limited in flexibility for regular expression searching through its Application Programming Interface (API). However, it does have a relatively robust Advanced Search through its web

This work was supported in part by the Southern Methodist University Data Science Program. L. Huang is with Southern Methodist University, Dallas, TX 75275 USA. (e-mail: lhuang@ smu.edu). C. Vaughn, is also with Southern Methodist University, Dallas, TX 75275 USA. (e-mail: cvaughn@smu.edu).

interface. Unfortunately, the Advanced Search is unsuited to the task of harvesting potential secrets over time and would be difficult to automate.

Thus, we designed a two-stage approach to mining by creating a candidate file harvester that searches through GitHub repeatedly searching for defined keyword signatures. These files are downloaded and timestamped. Through the course of several weeks, the harvester searches and deduplicates any files found. A second secret search phase is then performed on the full text source code files via local regular expression search.

This analysis is meant to be an observational study, in an effort to verify and potentially size the overall problem as it exists at the time of writing. As such, we do not infer causal claims and confine the analysis to overall observation of the current state of security posture. We intend simply to provide the reader with an understanding of what is being leaked, and how easy it may be to locate secrets in GitHub.

We conclude this analysis by explaining our overall methodology and results, in an effort to provide a basic framework for companies for protecting secrets. This framework will help inform a company so that they can put in policy and processes to protect themselves. In addition, we provide an overall test implementation that helps mitigate the overall potential of secret leaks, by describing an architecture of secret management and workload identification. This reference architecture should be deemed as one layer of risk prevention in an overall program to secure corporate assets.

II. Related work

Launched in 2008 by Tom Preston-Werner, Chris Wanstrath, P.J. Hyett, and Scott Chacon, GitHub today represents one of the largest code repositories in the world. After 10 years of growth, it was finally acquired by Microsoft for $7.5 Billion. [3]. The repositories, and their public nature, have served as fertile ground for researchers and the open source movement overall [2].

Even with its visibility and size, though, the security of leaked secrets and potentially private information has largely been an afterthought. The acceleration of Cloud and virtualization and containerization technologies have helped promulgate the issue of leaked credentials and has made control specifically difficult for IT departments leveraging GitHub [4]. There has been research and analysis into "left over secrets" pushed to GitHub, into containers, and into virtual machine configurations, as well as analysis on vulnerabilities [5], [6].

To date, however, very few researchers have done thorough analysis on how broad of a problem there is with leaked credentials on GitHub. The two most notable and relevant studies are by Sinha et al. [7], and Meli et al. [8]. These two studies are some of the only peer-reviewed studies on this topic, with the latter being the most comprehensive on both view, length, and analysis. The Meli study, performed from October 31, 2017 to April 20, 2018, involved 6 months of data collection, and yielded 4,394,476 million candidate files from 681,784 repositories. While such a study is beyond the scope of this paper, we view their work as an exemplary review for guidance on methodology [8].

Additionally, it is worth noting that toolsets exist for analysis repositories for potentially leaked secrets. Tools such as truffleHog [10], GitSecrets [11], detect-secrets [12], and GitRob [13] were originally born out of the idea that secrets potentially show up in public repositories. In early 2014, Amazon Web Services (AWS) realized that their credentials to their cloud services were being leaked into public domain. As a result, AWSLabs created GitSecrets as an integrated scanner for AWS access credentials for git. Sadly though, this is a point solution that largely addresses AWS-related leakages. There are ways to add additional signatures to GitSecrets, which makes it more multipurpose, but the solution overall relies on the diligence and care of the end developer [12].

API keys and passwords can engender a high degree of entropy, for obvious reasons. This is, in part, why many of these tools include features for detecting high entropy strings; TruffleHog and GitRob both employ such features [10], [13]. Other approaches include using regular expressions to identify fixed structured strings, on a set of known targets. Both approaches suffer from false positives and require careful attention to tuning for yielding proper results with real credentials.

As noted by Meli et al. [8], we come to a similar conclusion that currently no robust system exists for monitoring and mining GitHub for leaked credentials on a large and consistent scale. This is an inconvenient truth and is further evidence that the onus for control of these secrets lies in the end users and companies that employ them. For this study, we focus on developing our own file harvester based on a predefined set of 85 keywords and utilize 34 specific credential signatures to identify secrets. In this way, we limit the scope of the problem and focus specifically on a subset of high-value target secrets for Cloud provider infrastructure and application subsystems.

III. Methodology & Approach

GitHub has expansive capability to its API, however where it seriously lacks is in regular expression search. Unfortunately, the API is limited to simple keyword searching [14]. To accommodate more sophisticated searching and elimination of potential false positives, we take the aforementioned two-phase approach.

Phase One: The first phase casts a wide net searching for keywords in files and is built into a "harvester" application to loop through a list of carefully chosen keywords (see Appendix: Table II). The API call specifically searches for a keyword and tells GitHub to order by the index in descending order. In this way, we capture newly committed files first, with each pass of the loop on a keyword. GitHub limits the return results to 1000 files, only indexes files smaller than 384k for search and imposes significant rate limits on successive API calls [14]. Due to the latter limitation, the harvester was crafted with delays in line with rate limitation guidelines [22], [23], [24]. After file search results are gathered, the harvester performs checks to validate whether the file is already in the harvester file repository, and if it is not represented, the harvester downloads and timestamps the file with a datetime. This process makes sure the process does not double count files that show up repeatedly in the search and allows us to understand the current state of secret leakage as of today. (see Fig. 1)

We are, of course, only scratching the surface of the problem as its scoped to recently committed files and we do not explore long-aged files or the commit history; both could yield significant secret leakage results. This first phase of candidate file capturing began June 12, 2019 and ran until July 14, 2019, for a total of 25 days. It is important to note that we could have easily let this process continue, however were limited by time to aggregate results.

Over 25 days of running the harvester, and through the overall candidate file search process, we managed to capture 131,491 unique files that potentially could contained secrets. This represents files from 22,700 repositories on GitHub, and roughly 4.2 gigabytes of candidate source code and log files.

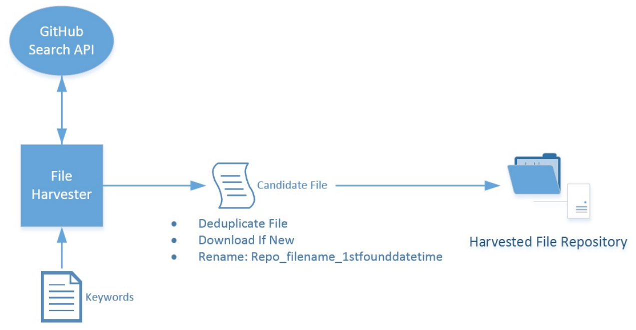


Figure 1: Phase One: Candidate File Harvesting

Phase Two: Next, we implemented a parser to hunt out specific fixed structure secrets from the source code for popular platforms, along with a select few high-value targets such as asymmetric private keys. Core to this process is a carefully crafted group of regular expressions (see Appendix: Table III). This group of regular expressions was assembled to seek out specific, known, fixed patterns of API keys and text for the providers and strings in question, and represents a limited list of expressions both specifically created for this project, and borrowed from tools such as truffleHog and existing research [8], [10], [11], [13].

The Secret Parser (see Fig. 2) performed file scanning for each of the files and regular expression patterns. Based on what it found, we aggregated the results into an analysis friendly format which included the Signature key, a Signature:File key, and the found secrets within the file. After aggregation was performed it was rudimentary to view and analyze located secrets.

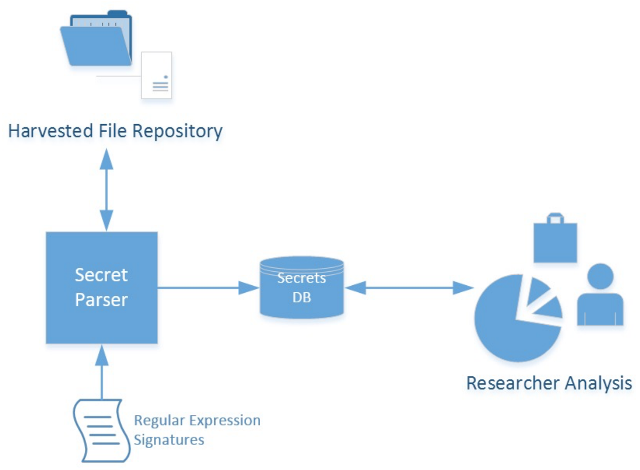


Figure 2: Phase Two: Secret Extraction and Analysis

It is important to note here that as the project team members performed this harvesting process on publicly available data, and at no time were any of the credentials or secrets utilized to test for potentially unauthorized access to real world systems in the wild. Strict control over the collected data was maintained during this project to protect any secrets that were located.

IV. Results

In aggregating the secrets found, we can see that out of 131,491 files, we found 27,847 potential secrets. As many of these secrets were repeated, deduplicating the results shows 10,388 distinct secrets located (see Table I).

Table I: Secrets Found by Signature

Signatures	Secrets	Distinct Secrets
Google API Key	12081	4010
RSA private key	3406	18
Generic Secret	2848	1834
Google OAuth ID	2540	1163
Generic API Key	2207	1399
AWS API Key	937	505
Password in URL	925	673
SSH (EC) private key	536	8
Google (GCP) Service-account	522	2
Twitter Oauth	514	188
SSH (OPENSSH) private key	311	9
GitHub	191	122
PGP private key block	155	8
SSH (DSA) private key	149	5
Facebook Oauth	147	122

Google OAuth Access Token	140	115
Google Oauth	79	76
Slack Webhook	54	46
Twitter Access Token	18	18
Twilio API Key	18	12
MailGun API Key	13	13
Stripe Standard API Key	12	7
Square OAuth Secret	11	11
Square Access Token	7	7
Amazon MWS Auth Token	7	4
PayPal Braintree Access Token	6	4
MailChimp API Key	5	5
Facebook Access Token	4	4
Slack Token	3	3
Heroku API Key	1	1
Grand Total	**27847**	**10388**

The bounty holds some interesting results in the fact that the predominant secrets leaked relate to Google API keys (38% of the distinct secrets found), with generic API keys, generic passwords in URL strings and AWS keys also ranking highly.

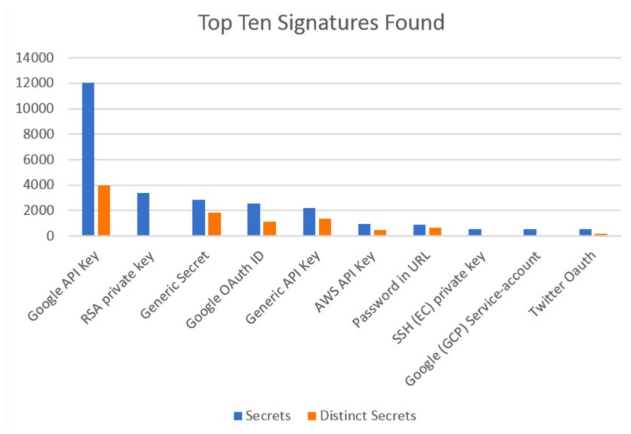


Figure 3: Top 10 Signatures Found

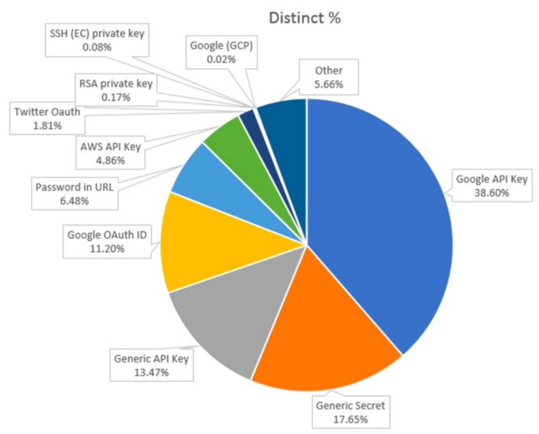


Figure 4: Distinct Secret Signature Type Percentage

While RSA private keys ranked highly on an absolute count basis (3406), we found them to be relatively minor on a distinct basis (18), which reveals that these keys are likely repeated throughout many files by a relative few users (see Fig. 3 and Fig 4).

The harvesting process yields significant proof that this problem is still pervasive in the market and open source community. And while this study is limited to users of GitHub, there is reason to believe that this problem goes beyond just the tool of GitHub itself, and that we would likely find similar results if studying other public source code repositories. With this in mind, we turn our attention to how to solve the underlying problem, and how to inject a level of control into development processes and security architectures so that secrets remain secret with high guarantee.

V. Mitigations - Keeping Secrets, Secret

As we have shown, the problem of secret leakage exists significantly in GitHub repositories. We make no assessment on the "blast radius" of some of these secret leaks but rather make the assertion that any leakage of secrets is potentially a breach of underlying security protocols and policy. Further, any leakage, even if corrected quickly, represents a potential for full breach since there is no way of knowing who has seen the secret or who has captured it for future use.

A. Motivation

In evaluating this problem there have been many suggested solutions. Much of the focus historically has been on one of monitoring for leakage, as opposed to proactive prevention [11], [15]. There have, however, been efforts around prevention. One such suggestion is to utilize Git plugins to scan source code prior to committing it to a repository [8], [12]. We find this recommendation useful, but insufficient. The core issue is related to secret management, and while a plugin solution is beneficial, it still relies on front-line employees for its effective implementation and regimented use and provides IT and Engineering departments little control and visibility into the secret management processes. There is little control over policy as a result.

When we talk about secret management, we are really scoping the discussion to managing a diverse set of sensitive credentials and managing it such that a diverse and potentially distributed set of individual developers and users can leverage those sensitive credentials to build and operate applications. Examples of these types of credentials include user passwords, API access or security tokens, database credentials, and of course cryptographic keys or certificates. These secrets end up living in plain text source code, CI/CD pipelines, Chef, Ansible, Puppet, or Terraform files, as well as source code repositories like GitHub. For an IT department to maintain proper control over these assets, it leads us to one unifying conclusion to work from which is that these secrets must be centralized if they are to be controlled.

With this in mind, we can outline a basic framework for what we would want a potential secret management system to do for us, and the capability we would want it to provide. We would potentially want a system that provides:

- A simple way to execute key or password rotation
- An audit trail of who could potentially access a secret, and when they actually do access a secret
- Minimization to secret sprawl and leakage
- Encryption at rest and in transit of sensitive data
- Access control
- Strong guarantees for identity, access control, audit logging, and non-repudiation

Further, this system should be articulated to expect the modern-day infrastructure state of application development. Applications are now a combination of both in-house development assets, along with various Software-as-a-Service (SAAS) related application platforms. In this way, any system must expect the distributed and diverse nature of potential secrets.

There are several ways to solve this problem once we scope the answer to centralization. The issue is both one of identity, and similarly of verification of access. And, while identity can be handled in a variety of ways, a Kerberos backed system or another system that can leverage X.509 based certificates, is the most appropriate place to begin.

B. Proposed Model

As a way to achieve the aforementioned goals, we propose a model system which leverages digital signatures and identity services, a centralized secret store, anonymous functions, and a credential proxy. And while this architecture (see Fig. 5) creates a high-value target with its secret store, we argue that it does so no more than would a regular identity provider would. Due to the amount of control it provides, any risk of breach is outweighed by the fact that strict security boundaries and controls can be placed on the system to create a secure environment for secret management and secret usage. Similarly, the proposed architecture allows for the auditing and non-repudiation required to isolate breach-related issues much quicker than would otherwise be possible

Core to this system architecture is the fact that workloads and applications maintain a sense of encrypted, credential-less access to high value systems, while having this access based on identity and access levels. All communication is performed with Secure Socket Layer and Transport Layer Security (SSL/TLS), and identity and access controls are dictated centrally by the identity provider and intermediated by the credential proxy. As shown in the architecture diagram, the model fully encapsulates authentication, public and private key management, secret credential management for the systems being utilized, API proxying, and any lambda expressions required to intermediate systems that have no direct API in which to interact.

.

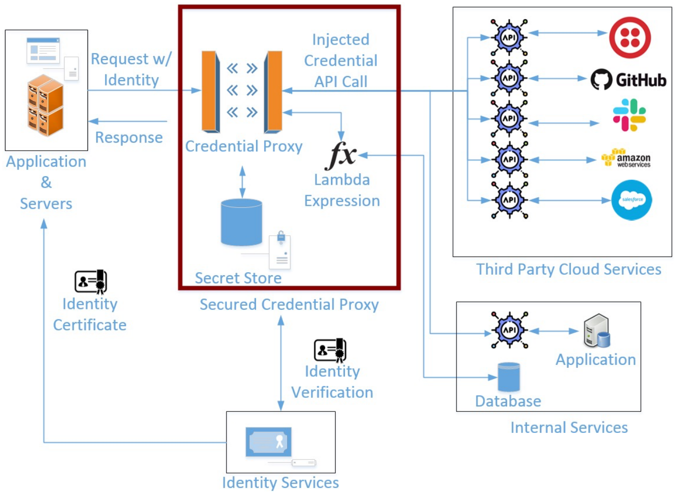


Figure 5: Centralized Secret Store w/ Credential Proxy Architecture

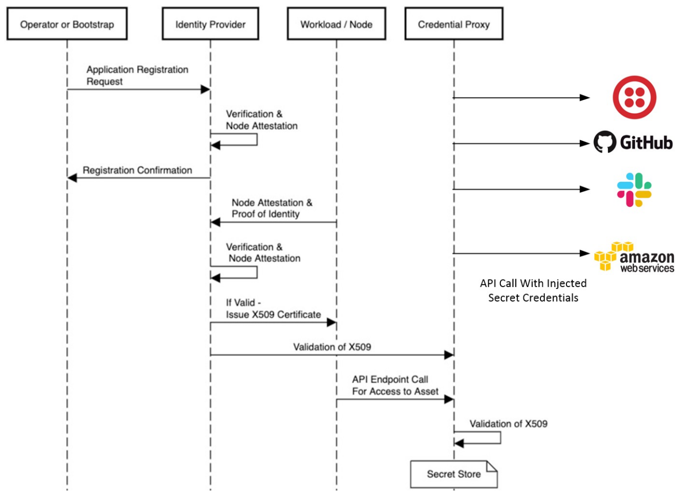
To properly understand why we believe this model shields secrets and provides a solution for secret leakage, we can describe the purpose of each component. There are five primary actors in this model. This includes:

1. *Calling Applications* needing to interact with a system,
2. *Third Party Cloud Services*, which typically are integrated into via OAUTH or API key authentication,
3. *Internal Services*, which can be comprised of other databases, subsystems, or other applications internal to corporate infrastructure. These systems may or may not have API-level access associated to them. Many of these systems, such as the case with database systems, are client-server based and require connection handling at the application level when they have no overlaying API,
4. A *Secured Credential Proxy*. As diagramed this proxy includes secret management, identity verification and authentication enforcement, proxy and caching activities for API passthrough calls to APIs, and lambda expression handling and connection reuse for systems that require client-server connections, and;
5. *Identity Services* which provides identity registration and governance, certificate management and distribution, and key management.

Noted in item 3 above, we leverage the concept of lambda expressions as a vehicle handle connectivity to client-server systems. This serves a few purposes, 1) it makes functional encapsulation of disparate connectivity methods straight forward, 2) it allows our system to use these functions as an efficient, event-driven byproduct of proxy invocation, 3) it maintains our ability to reuse connections if appropriate. This is desirable since setting up new connections is an expensive procedure on a relative per query basis due to buffer allocations [16], and 4) the overall ephemeral nature of these proxy invocations strengthens our security posture and allows for relative simplistic procedures for credential rotations.

Sequencing of how our model system functions is shown below (see Fig. 6) and as follows:

1. A workload or application is registered with the identity provider and receives an X.509 certificate of identity if approved. This is done either by a bootstrap process, or an Operator. This populates the identity registry.
2. A workload, node, or application authenticates to the identity provider either utilizing a cryptographic key pair or in the case of specific infrastructure providers like Amazon Web Services (AWS) an Instance Identification Document (IID) or something similar. [17] Once authenticated, the workload asks, 'who am I?'
3. Based on access controls, a X.509 certificate of identity is then forwarded not only to the workload, but also to the proxy system so that it can do validation. It is important to note that the proxy can make a request to the identity provider in real time to validate a X.509 certificate if required.
4. If a workload needs access to resources that potentially need sensitive secrets, the workload sends a request to the proxy, for a specific API endpoint of the proxy, along with the X.509 certificate proving its identity.
5. The proxy receives the request, validates the certificate of the requestor, and then if valid, locates the necessary credential secret, and makes an external request to the requested resource with secrets injected into the request.
6. Assuming success, the proxy returns the payload back to the calling workload.


Operator or Bootstrap
Identity Provider
Workload / Node
Credential Proxy
Application Registration Request
Verification & Node Attestation
Registration Confirmation
Node Attestation & Proof of Identity
Verification & Node Attestation
If Valid - Issue X509 Certificate
Validation of X509
API Endpoint Call For Access to Asset
Validation of X509
Secret Store
GitHub
amazon webservices
API Call With Injected Secret Credentials
Figure 6: Sequencing for Centralized Secret Store w/ Credential Proxy

C. Test Implementation & Features of Our Model

To provide a test implementation of these ideas, we constructed the general tenants of this system into AWS infrastructure, accessing the GitHub API with a single secret key. This key is entirely shielded from view for a potential developer that might want to utilize it and is fully controlled by an Administrator. The system provides to a potential developer the basic constructs available to develop against, while maintaining sensitive information completely outside of any development process. This is due to the fact that credential and secret access in this model is fully based on the identity of where the potential developer code is deployed. We leverage AWS Instance Identification Documents (IIDs) to ensure identity verification. IIDs represent a JSON file that describes the virtual machine and includes a signature and PKCS7 signature. This information can be used to verify accuracy, origin, and authenticity of the workload. PKCS7 signatures are verified against an appropriate AWS public certificate for the AWS Region [17].

In our test implementation, secrets are never in application developer generated source code, and therefore will never be committed to GitHub by a developer. Additionally, secrets can be rotated at will without impacting production or deployed code. We view this as a high benefit since it creates a moving target for a potential attacker, in addition to fast mitigation for a potential secret leak, and a good avenue for security hygiene through secret rotation when an employee leaving the company. The latter case, where an employee leaves, being an issue only if the employee in question had Administrative access to the secret in the first place.

We can thus assert that our test system provides:

- A simple way to rotate the API key without impacting production code,
- Ample logging of access to sensitive resources,
- A complete seal on potential secret leakage due to the use of public source code repositories,
- Full encryption of secrets in transit, provided by SSL/TLS, and;
- Strong guarantees for identity, authentication, access control, audit logging, and non-repudiation for the requesting node or workload.

This is in addition to the technical benefits of having the potential to reuse connections for client-server connectivity, caching API call results in the centralized API proxy, and the potential benefits concerning the ability to centrally scale out application and credential proxy resources, thereby minimizing undue burden to IT personnel.

Using a single EC2 Linux instance, the AWS API Gateway configured to proxy requests and inject a secret key, an appropriate Identity Access Management (IAM) role to allow for node attestation, identity, and access control, along with a simple example program written in python, we construct the implementation below (see Fig. 7) [18]. The secret key in this case is a GitHub API access key, which is injected into the HTTP Authorization header at run time. This injection is fully concealed from the calling application. The proxy is configured for passthrough content handling, and so that the

HTTP headers are mapped with the following parameters, where <GitHub API Secret Key> is replaced with a valid GitHub API key:

- Accept-Encoding = 'application/json'
- Authorization = 'token <GitHub API Secret Key>'

All communication channels operate strictly over TLS version 1.2 [19].

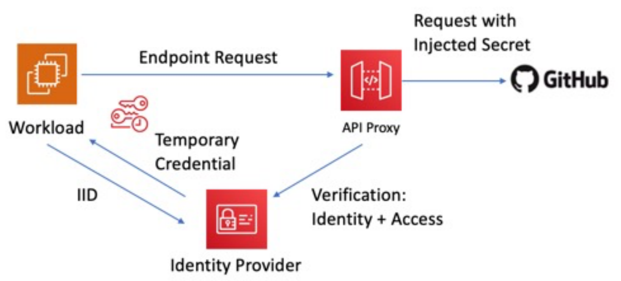


Figure 7: AWS Test Implementation

D. Limitations & Future Research

Security for this model hinges on identity for the workload, and the security controls for the overall application infrastructure. The obvious point can be made that if the server that houses the application is breached, an attacker could potentially gain privileged access to sensitive resources by impersonating a registered application identity from that server. This is, of course, the worst-case scenario that could occur, and the counter argument is still compelling. The attacker would have to first breach security boundaries for the company, and then breach server security to gain access to the server. And once this is complete, the attacker would be able to gain access to resources through the credential proxy. However, at no time would an attacker, in this case, be able to gain access to actual secrets. As with any security model, there must be layers. Any comprehensive security program would include firewalls, intrusion detection, and an intrusion prevention mechanism. And, if there is a potential breach, an IT organization should be able to isolate applications and workloads in questions, and quickly rotate secrets, to mitigate overall risk and threat surface areas.

Additionally, it is important to recognize that this architecture creates a throttle point for applications being proxied. Care must be given to architecting the proxy for proper scale, and for a path to scale up as needed. We do not explore this in this paper; however, this is an area where valuable new research and development could be performed.

Source code for the implementation is provided for this implementation and represents the potential integration path for a distributed development staff. Importantly, this implementation pattern can be extended and enhanced to include additional types of SaaS integrations and credentials. For example, one could replace the AWS identity provider with an in-house Kerberos implementation, and utilize the emerging standards for Secure Production Identity Framework for Everyone (SPIFFE) and their SPIFFE Verifiable Identity Document (SVID) format which leverages X.509 extensively [20]. Doing so would maintain the general reference architecture and preserve the trust and policy frameworks. This would represent another recommended next step with this research and design.

VI. Conclusion

Secret leakage is potential problem that affects almost every company that utilizes modern day cloud infrastructure, and it is especially problematic for companies that utilize public repositories. The predominance of online source code repositories like GitHub only serves to bring the problem to the forefront as a challenge. Through our harvesting techniques we have shown that its relatively simple to data mine secrets from GitHub. Even with GitHub's API rate limitations and controls we were able to harvest, on average, more than 400 distinct secrets from GitHub repositories every day during our 25-day window. This problem, unfortunately, is fundamental to the secrets management processes that companies, and developers utilize. Due to the very nature of distributed and wide-scale collaborative activities for development, we must conceive of a new systematic process that stems the root of the problem. In our estimation, the root of the problem is either 1) that a company uses public repositories at all, or 2) that developers are inherently untrustworthy with secrets. Seeing that removing the option of utilizing public repositories is de minimis for some companies, prevention of secret leakage then requires an entirely new architectural pattern like the proposed.

The solution is to centralize secret management, and place strict policy and extensive control over secrets. This includes practices like regular rotation of secrets, audit logging, and identity and access verification. Trust and identity must be strict cornerstones to this model. By managing the environment differently for identity, trust, and secret policies, one can centralize the management processes and force a distributed workforce to comply with it. Our test implementation is one such model, and as we have designed and shown we believe that this could be easily extended into a fully operationally ready system relatively quickly.

There are several emerging standards that still need to be explored to handle workload and application identity. We recommend future research in the area of secret management through the use of identification frameworks such as SVID and SPIFFE. The capabilities and extensibility of these frameworks have the potential to be utilized for generic secret management, in addition to being equally usable with more enterprise grade solutions.

APPENDIX

Table II: GitHub API Search Keywords

objectrocket.com:	ACCESS_SECRET=
mongodb.net:	ACCESS_TOKEN=
access_token	ACCOUNT_SID=
access_secret	AWS-KEY=
api_key	AWS-SECRETS=
client_secret	AWS.config.accessKeyId=
consumer_secret	AWS.config.secretAccessKey=
customer_secret	AWSACCESSKEYID=
user_secret	AWSSECRETKEY=
secret_key	AWS_ACCESS=
-----BEGIN RSA PRIVATE KEY-----	AWS_ACCESS_KEY=
-----BEGIN EC PRIVATE KEY-----	AWS_ACCESS_KEY_ID=
-----BEGIN PRIVATE KEY-----	AWS_CF_DIST_ID=
-----BEGIN PGP PRIVATE KEY BLOCK-----	AWS_DEFAULT
-----BEGIN OPENSSH PRIVATE KEY-----	AWS_DEFAULT_REGION=
Amzn	AWS_S3_BUCKET=
EAACEdEose0cBA	AWS_SECRET=
ya29	AWS_SECRET_ACCESS_KEY=
access_token	AWS_SECRET_KEY=
https://hooks.slack.com	AWS_SES_ACCESS_KEY_ID=
rk_live_	AWS_SES_SECRET_ACCESS_KEY=
sq0atp	SENDGRID=
sq0csp	SENDGRID_API_KEY=
AKIA	SENDGRID_FROM_ADDRESS=
AKIAJ	SENDGRID_KEY=
AKIAI	SENDGRID_PASSWORD=
aws_access_key_id	SENDGRID_USER=
aws_secret_access_key	SENDGRID_USERNAME=
AIza	TWILIO_ACCOUNT_ID=
mongodb://admin	TWILIO_ACCOUNT_SID=
awsAccessKeyId	TWILIO_API_KEY=
awsSecretAccessKey	TWILIO_API_SECRET=
Mongourl	TWILIO_CHAT_ACCOUNT_API_SERVICE=
mongoUrl	TWILIO_CONFIGURATION_SID=
postgresql://	TWILIO_SID=
psycopg2.connect(	TWILIO_TOKEN=
.apps.googleusercontent.com	TWILO=
--username=	TWITTER=
ACCESSKEY=	TWITTEROAUTHACCESSSECRET=
ACCESSKEYID=	TWITTEROAUTHACCESSTOKEN=
ACCESS_KEY=	TWITTER_CONSUMER_KEY=
ACCESS_KEY_ID=	TWITTER_CONSUMER_SECRET=
ACCESS_KEY_SECRET=	

Table III: Regular Expression Signatures

Slack Token:	(xox[p|b|o|a]-[0-9]{12}-[0-9]{12}-[0-9]{12}-[a-z0-9]{32}),
RSA private key:	-----BEGIN RSA PRIVATE KEY-----,
SSH (OPENSSH) private key:	-----BEGIN OPENSSH PRIVATE KEY-----,
SSH (DSA) private key:	-----BEGIN DSA PRIVATE KEY-----,
SSH (EC) private key:	-----BEGIN EC PRIVATE KEY-----,
PGP private key block:	-----BEGIN PGP PRIVATE KEY BLOCK-----,
Facebook Access Token:	EAACEdEose0cBA[0-9A-Za-z]+,
Facebook Oauth:	[f|F][a|A][c|C][e|E][b|B][o|O][o|O][k|K].*['|\][0-9a-f]{32}['|\"]",
Twitter Oauth:	[t|T][w|W][i|I][t|T][t|T][e|E][r|R].*['|\][0-9a-zA-Z]{35,44}['|\"]",
Generic Secret:	[s|S][e|E][c|C][r|R][e|E][t|T].*['|\][0-9a-zA-Z]{32,45}['|\"]",
Generic API Key:	9]\ 5 5y,
GitHub:	[g|G][i|I][t|T][h|H][u|U][b|B].*['|\][0-9a-zA-Z]{35,40}['|\"]",
Google Oauth:	(\client_secret\":\"[a-zA-Z0-9-_]{24}\")",
AWS API Key:	AKIA[0-9A-Z]{16},
Heroku API Key:	[h|H][e|E][r|R][o|O][k|K][u|U].*[0-9A-F]{8}-[0-9A-F]{4}-[0-9A-F]{4}-[0-9A-F]{4}-[0-9A-F]{12},
Slack Webhook:	https://hooks.slack.com/services/T[a-zA-Z0-9_]{8}/B[a-zA-Z0-9_]{8}/[a-zA-Z0-9_]{24},
Twilio API Key:	SK[a-z0-9]{32},
Twitter Access Token:	[0-9a-zA-Z]{45},
Password in URL:	[a-zA-Z]{3,10}://[^/\\s:@]{3,20}:[^/\\s:@]{3,20}@.{1,100}[\'\\s]",
Mongodb URI:	^(mongodb:(?:\/{2})?)((\w+?):(\w+?)@|:?@?),
RDP URI:	^(rdp:(?:\/{2})?)((\w+?):(\w+?)@|:?@?),
Twitter Access Token:	[t|T][w|W][i|I][t|T][t|T][e|E][r|R].*[1-9][0-9]+-[0-9a-zA-Z]{40},
Facebook Access Token:	EAACEdEose0cBA[0-9A-Za-z]+,
Google API Key:	AIza[0-9A-Za-z\\-_]{35},
Google (GCP) Service-account:	\type\":\"service_account\"",
Google OAuth ID:	[0-9]+-[0-9A-Za-z_]{32}\.apps\.googleusercontent\.com,
Stripe Standard API Key:	sk_live_[0-9a-zA-Z]{24},
Stripe Restricted API Key:	rk_live_[0-9a-zA-Z]{24},
Square Access Token:	sq0atp-[0-9A-Za-z\-_]{22},
Square OAuth Secret:	sq0csp-[0-9A-Za-z\-_]{43},
PayPal Braintree Access Token:	access_token\$production\$[0-9a-z]{16}\$[0-9a-f]{32},
Amazon MWS Auth Token:	amzn\.mws\.[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12},
Twilio API Key:	SK[0-9a-fA-F]{32},
MailGun API Key:	key-[0-9a-zA-Z]{32},
MailChimp API Key:	[0-9a-f]{32}-us[0-9]{1,2}

Acknowledgment

C. Vaughn would like to expressly thank Dr. Shaibal Chakrabarty, Southern Methodist University, for his research guidance with this work.

References

[1] Shahrivar, S., Elahi, S., Hassanzadeh, A., & Montazer, G. (2018). A business model for commercial open source software: A systematic literature review. Information and Software Technology, 103, 202-214.

[2] Kalliamvakou, E., Damian, D., Blincoe, K., Singer, L., & German, D. M. (2015, May). Open source-style collaborative development practices in

commercial projects using GitHub. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 574-585). IEEE Press.

[3] GitHub. (2019, July 12). Retrieved July 16, 2019, from https://en.wikipedia.org/wiki/GitHub

[4] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing Security of Virtual Machine Images in a Cloud Environment," CCSW, 2009.

[5] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A Security Analysis of Amazon's Elastic Compute Cloud Service," SAC, 2012.

[6] R. Shu, X. Gu, and W. Enck, "A Study of Security Vulnerabilities on Docker Hub," CODASPY, 2017.

[7] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and Mitigating Secret-Key Leaks in Source Code Repositories," Mining Software Repositories, 2015.

[8] Meli, M., Mcniece, M. R., & Reaves, B. (2019). How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. Proceedings 2019 Network and Distributed System Security Symposium. doi:10.14722/ndss.2019.23418

[10] TruffleHog. [Online]. Available: https://github.com/dxa4481/truffleHog

[11] GitSecrets [Online]. Available: https://github.com/awslabs/git-secrets

[12] /@cdavis_. (2019, February 06). Discovering keys and secrets - CryptoMove Blog: Moving Target Data Protection. Retrieved July 16, 2019, from https://blog.cryptomove.com/discovering-keys-and-secrets-855f6e857a37

[13] GitRob. [Online]. https://github.com/michenriksen/gitrob

[14] (2019) GitHub API Search. [Online] Available: https://developer.github.com/v3/search/

[15] D. Bourke. (2017, Oct.) Breach Detection at Scale. [Online]. Available: https://developer.atlassian.com/blog/2017/10/project-spacecrab-breach-detection/

[16] DuBois, P., Hinze, S., & Pedersen, C. (n.d.). MySQL 5.0 certification study guide. Indianapolis, IN: MySQL Press.

[17] Instance Identity Documents. (n.d.). Retrieved August 2, 2019, from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-identity-documents.html

[18] Vaughnch. (2019, July 22). Vaughnch/security-proxy. Retrieved July 22, 2019, from https://github.com/vaughnch/security-proxy

[19] Choose a Minimum TLS Version for a Custom Domain in API Gateway. (n.d.). Retrieved August 2, 2019, from https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-custom-domain-tls-version.html

[20] SPIFFE Components. (n.d.). Retrieved July 22, 2019, from https://spiffe.io/spiffe/

[21] M. Soto, F. Thung, C. P. Wong, C. L. Goues, and D. Lo, "A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions," in 2016 IEEE/ACM 13th Working Conference

[22] GitHub. (2018, Oct.) About Token Scanning. [Online]. Available: https://help.github.com/articles/about-token-scanning/

[23] (2019) GitHub API Rate Limiting Documentation. [Online]. Available: https://developer.github.com/v3/#rate-limiting

[24] (2019) GitHub Content API Documentation. [Online]. Available: https://developer.github.com/v3/repos/contents/#get-contents

[25] Latest Ransomware Attack Targets GitHub Security Vulnerability. (2019, May 14). Retrieved July 27, 2019, from http://blog.klocwork.com/open-source/github-ransomware-attack-source-code/