Detecting and Mitigating Secret-Key Leaks in Source Code Repositories

Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, Senthil Mani
IBM Research
{vibha.sinha, diptsaha, pdhoolia, ropadhye, sentmani}@in.ibm.com

***Abstract*—Several news articles in the past year highlighted incidents in which malicious users stole API keys embedded in files hosted on public source code repositories such as GitHub and BitBucket in order to drive their own work-loads for free. While some service providers such as Amazon have started taking steps to actively discover such developer carelessness by scouting public repositories and suspending leaked API keys, there is little support for tackling the problem from the code sharing platforms themselves.**

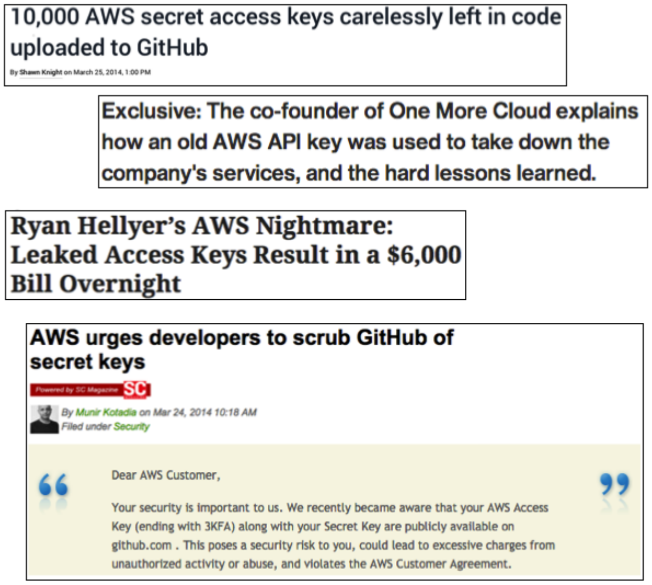
In this paper, we discuss practical solutions to detecting, preventing and fixing API key leaks. We first outline a handful of methods for detecting API keys embedded within source code, and evaluate their effectiveness using a sample set of projects from GitHub. Second, we enumerate the mechanisms which could be used by developers to prevent or fix key leaks in code repositories manually. Finally, we outline a possible solution that combines these techniques to provide tool support for protecting against key leaks in version control systems.

I. Introduction

Many web and mobile based applications interact with external services hosted by providers such as Facebook, Google and Amazon through Web APIs. The mechanism for authentication between the application and the service is often through an API key or a pair of an API client identifier and a secret key.

Since these services are intended to be invoked by server-side components of applications, the keys themselves would ideally lay dormant only in server-side program memory and thus be inaccessible for users of the application. However, many application developers choose to host their application source code publicly on repositories such as GitHub and BitBucket to incorporate contributions from the open-source community. In this scenario, if the developer keys are embedded within source code, they can be easily stolen by a malicious user who can authenticate themselves as the developer and misuse the services for their own profit. In fact, several news articles [1], [2], [3], [4] reported exactly this attack as malicious users began stealing leaked Amazon AWS keys from GitHub projects to run intensive compute jobs, which were billed to the victims – in some cases the bills ran up to several thousand US dollars. Figure 1 contains snippets from these news articles.

Key leaks on version control systems such as Git repositories are compounded by the fact that merely deleting an accidental leak does not plug it completely, since the keys will exist in the project's change history. Systems such as Git provide non-trivial mechanisms to re-write history [5], which are none-the-less destructive since they require explicit action from other members of the project who are working on clones of the repository.



10,000 AWS secret access keys carelessly left in code uploaded to GitHub

By Shawn Knight on March 25, 2014, 1:00 PM

Exclusive: The co-founder of One More Cloud explains how an old AWS API key was used to take down the company's services, and the hard lessons learned.

Ryan Hellyer's AWS Nightmare: Leaked Access Keys Result in a $6,000 Bill Overnight

AWS urges developers to scrub GitHub of secret keys

Powered by SC Magazine SC

By Munir Kotadia on Mar 24, 2014 10:18 AM
Filed under Security

Dear AWS Customer,

Your security is important to us. We recently became aware that your AWS Access Key (ending with 3KFA) along with your Secret Key are publicly available on github.com . This poses a security risk to you, could lead to excessive charges from unauthorized activity or abuse, and violates the AWS Customer Agreement.

Fig. 1. Snippets from news articles that reported AWS key leaks. Simply searching for "AWS key leaks" on the Web reveals many more.

In this paper, we first discuss different approaches that attackers may use to discover leaked API keys within source code, such as simple pattern-based search, heuristics-driven filtering and source-based program slicing. We then discuss the available tool support for developers to prevent or repair key leaks within their own projects manually. Finally, we combine these aspects to outline a possible solution for proactive tool-based protection of key leaks within source control repositories.

Section II outlines key-leak detection techniques, while Section III outlines mitigation techniques. We present related work in Section IV and conclude in Section V.

II. Key Leak Detection Techniques

Before discussing ways to mitigate accidental leakage of API keys on code repositories, it is worthwhile to survey some methods for discovering API keys from such repositories. We shall evaluate the effectiveness of these methods using a sample set of projects derived from GitHub.

```
String ACCESS_KEY_ID
  = "AKIA2E0A8F3B244C9986";
String SECRET_KEY
  = "7CE556A3BC234CC1FF9E8A5C324C0BB70AA21B6D";

AWSCredentials creds =
  new BasicAWSCredentials(ACCESS_KEY_ID, SECRET_KEY);

AmazonSimpleDBClient client
  = new AmazonSimpleDBClient(creds);
```

Fig. 2. Sample Java code snippet containing (fictional) embedded Amazon AWS keys. The underlined constructor call is the slicing criteria for data-flow based key leak detection.

```
OAuthClientRequest request = OAuthClientRequest
  .tokenProvider(OAuthProviderType.FACEBOOK)
  .setGrantType(GrantType.AUTHORIZATION_CODE)
  .setClientId("950513172001321")
  .setClientSecret("3b2e464637e5159024254dd78aadb17a")
  .setRedirectURI("http://localhost:8080/facebooklogin")
  .setCode(code)
  .buildQueryMessage();
```

Fig. 3. Sample Java code snippet containing (fictional) embedded Facebook OAuth keys. The underlined method call is the slicing criteria for data-flow based key leak detection.

A. *Sample Selection using Keyword Search*

Let us consider the point of view of an attacker who wishes to steal keys or passwords leaked on GitHub. The first and most naive approach is to simply use GitHub's in-built search function to look for keywords or file-names that may indicate the presence of such keys. In fact, GitHub reportedly suspended its search function temporarily [6] in 2013 after several SSH keys were stolen by simply looking for files named `id_rsa` or searching for the string `---BEGIN RSA PRIVATE KEY---` which is typically present at the head of RSA private key files generated by `ssh-keygen`.

For other types of confidential keys, there is not necessarily a file format but simply a pattern of characters. Table I enumerates patterns for API keys of a handful of API providers, as described by the authors of [7]. However, GitHub does not allow searching of regular expressions in code, and thus the naive approach to search for such patterns is to create a clone of every repository – essentially a mirror of GitHub – and then search their contents for such patterns. This is in no means a trivial task, since GitHub is known to host several million repositories. While projects such as GHTorrent [8] have attempted to provide a queryable mirror of millions of GitHub repositories, it would be far more efficient for an attacker to first reduce the sample set to projects that are likely to contain an embedded API key by searching for references to client APIs which consume such keys.

For example, if an attacker is looking for applications that make use of Amazon's AWS, they can search for calls to Amazon's client API. Figure 2 shows a sample Java code snippet using one of Amazon's client APIs – `BasicAWSCredentials` – that simply takes the client and secret keys as input strings rather than picking them up from a secure key store or other configuration file. Similarly, Figure 3 shows an equivalent client operation for the Facebook API that takes OAuth keys as strings. Since OAuth is a generic standard, there are multiple library implementations – the example shows usage of the Apache Oltu[1] library.

TABLE I
REGULAR EXPRESSIONS FOR API KEYS OF VARIOUS SERVICE PROVIDERS [7].

Service Provider	Client ID	Secret Key
Amazon AWS	`AKIA[0-9A-Z]{16}`	`[0-9a-zA-Z/+]{40}`
Bitly	`[0-9a-zA-Z_]{5,31}`	`R_[0-9a-f]{32}`
Facebook	`[0-9]{13,17}`	`[0-9a-f]{32}`
Flickr	`[0-9a-f]{32}`	`[0-9a-f]{16}`
Foursquare	`[0-9A-Z]{48}`	`[0-9A-Z]{48}`
LinkedIn	`[0-9a-z]{12}`	`[0-9a-zA-Z]{16}`
Twitter	`[0-9a-zA-Z]{18,25}`	`[0-9a-zA-Z]{35,44}`

For the purposes of our evaluation, we assume the role of an attacker who is looking to steal Amazon AWS keys and hence our sample set consists of 84 projects that were returned in one page of results when searching GitHub with the string "BasicAWSCredentials".

B. *Pattern-based Search*

Since our data-set contained projects that are likely candidates for leaking Amazon AWS credentials, we first performed a full-text search for strings that match the pattern for Amazon credentials as listed in Table I. Of the 84 projects, 51 projects had a match with a total of 2,457 instances. Clearly, a simple pattern search results in a large number of false positives.

Remembering that our data-set consisted of projects that make use of Amazon's Java-based Client API, we next restricted our search to only string literals in Java source files within these repositories. We made use of the Eclipse JDT [9] parser to generate abstract syntax trees (ASTs) from Java source files and performed a pattern search on string literal nodes. Using this technique, our results reduced to 30 instances in 14 projects matching the Client ID pattern, and 43 instances in 17 projects matching the Secret Key pattern. We find more strings matching the key pattern than the Client ID pattern since the Client ID pattern is restricted to those strings starting with an explicit letter sequence `AKIA`. Hence, in general this technique is still limited to returning false positives.

C. *Heuristics-driven Filtering*

In [7], the authors use additional criteria to filter false positives, such as by looking at instances where a match for the Client ID and a match for the Secret Key appear within 5 lines of each other, presumably because they are hard-coded within the client API call or in a constants file. This is a useful heuristic, which in our case returns 30 instances of ID-key pairs in 14 projects that we manually verified were actually leaked credentials.

While this approach is generally precise, it is liable to miss instances of leaked keys where the credentials are not close together. There was also one instance of an ID and key pair

[1] https://oltu.apache.org

in which the ID did not start with the prefix AKIA. Since the chosen-pattern for the ID itself was insufficient, its secret key, though it matched the pattern, was not returned by the proximity heuristic.

A different approach to reduce false positives in strings that match the pattern of the API key is to try and guess whether they are auto-generated or hand-written. We noticed that many false positives were actually human-readable strings such as "SomeLabelTextInMyAppWhichIsAPerfectMatch", or placeholders such as "00000000...". To remove such results, we applied a standard password strength estimator [10], which penalizes strings with repeating characters or those that contain dictionary words while accepting strings that have a high amount of apparent randomness (entropy). This technique returned 34 instances of Secret Key matches in our data-set, with 100% recall and precision of 91%. The false positives that remained were auto-generated key-like strings that were used for different purposes, such as identifiers of serialized objects.

D. Source-based Program Slicing

A heavy-weight approach to finding exactly those strings that flow in-to the Client API calls is to employ program slicing. There are several existing techniques to precisely trace the value of a string even if it subject to operations such as concatenation and substring-extraction [11], [12], [13]. However, these approaches are an overkill for our purpose since we do not expect hardcoded secret keys to undergo such transformations.

We therefore evaluate a lightweight source-based flow-sensitive program slicing algorithm that was developed in-house. This algorithm builds a system dependence graph similar to the classic HRB slicing [14], but is context-insensitive in that we do not match call-return points precisely.

We ran the slicer on our data-set of 84 projects with the slicing criteria being the constructor call of BasicAWSCredentials, as shown in Figure 2. The analysis returned 40 instances of non-null and non-empty strings. We further applied the password strength filter as above to prune placeholder strings such as "SOME_KEY" and we were left with 26 instances. This technique gave 100% precision and 84% recall at the cost of some computational effort. The apparent loss of recall was because not all strings flowed into the client API calls. For example, in one project we found multiple API key pairs all defined in one file, but only one of them was being used by the application (and was detected by this method).

These results are summarized in Table II, which additionally includes results for a similar evaluation on Facebook OAuth keys, where the pattern is as listed in Table I and the slicing criteria is the call to setClientSecret, an example of which is shown in Figure 3.

III. Key Leak Mitigation Techniques

We now discuss some tools that could be useful for developers to protect themselves against accidental key leaks.

TABLE II
Results of evaluating different key detection techniques for two types of API key usages.

	Amazon AWS			Facebook		
# Projects	84			30		
True leaks	31			16		
Filters	**#**	**Pr**	**Rc**	**#**	**Pr**	**Rc**
P	2,457	1%	100%	40	40%	100%
P + J	43	72%	100%	29	55%	100%
P + J + U	30	100%	97%	16	87.5%	87.5%
P + J + E	34	91%	100%	22	73%	100%
P + J + D + E	26	100%	84%	13	100%	81.25%

Metrics: **Pr** = Precision, **Rc** = Recall;
Filters: **P** = Pattern match on all Java files, **J** = Pattern match on Java string literals, **U** = User-ID + Secret-Key proximity, **E** = Entropy (password strength estimation), **D** = Data-flow analysis (program slicing)

A. Preventing Leaks Explicitly

Best practice suggests that applications that use secret keys should not embed them within source code, but instead read them from a file that is not tracked by their version control repository. In Git, for example, one can specify a list of files that should be ignored in a file named .gitignore. Often a template file with dummy key values is checked-in so that project members can re-create the key file in their individual work-spaces by simply copying the template and replacing the placeholder values with the real secret keys.

One issue with this approach is that project members must find alternative ways of sharing secret keys amongst themselves. This issue becomes notable if the application is using multiple keys or if the project team changes frequently.

An alternate solution is to allow the file with API keys to be checked-in to the shared repository, but in an encrypted fashion. Tools such as git-crypt [15] allow transparent encryption and decryption of files within Git repositories. A user who creates and shares a sensitive file can specify a list of PGP users who are allowed to access that file. The tool encrypts the file with each privileged user's public key and stores multiple encrypted copies in the code repository. When another project member performs a check-out, the tool will find the right copy corresponding to that PGP user and decrypt with their private key accordingly.

Of course, with all such solutions, the burden of creating the black-list of files falls on the user. Many a time, users fail to take such measures and accidentally commit files with sensitive information. Since version control systems are designed to keep a history of every change intact, it is not trivial to undo such an accidental leak.

B. Remedying Leaked Keys

Most version control systems such as SVN and Git do not store each revision in it's entirety, but simply as a *diff* since the previous version. Thus, in theory it should be possible to remedy an accidental leak by removing the sensitive information at the point it was committed, and allowing the remaining commits to remain intact.

Unfortunately it is not really so easy to do this. For example, Git identifies commits by a SHA1 hash, which is a function of

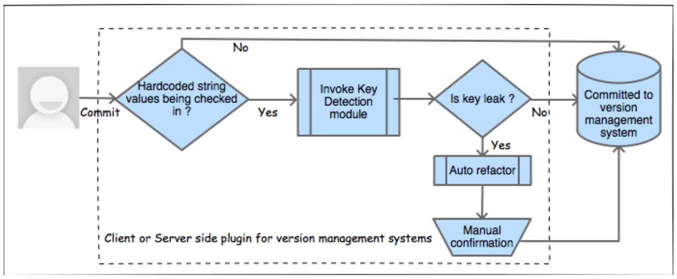

No
Hardcoded string values being checked in ?
Commit
Yes
Invoke Key Detection module
Is key leak ?
No
Yes
Auto refactor
Manual confirmation
Committed to version management system
Client or Server side plugin for version management systems

Fig. 4. Flow-chart of a pro-active key leak detection and mitigation solution implemented as a hook or plug-in on top of a version-control system.

the contents of the commit. If the contents change, so does the hash, and hence the contents of the next commit also change (since the pointer to its parent now has a different value), and this snowballs all the way up to the head of the branch. Fortunately, Git allows a way to perform forceful rewriting of history using a tool called `git filter-branch` [16]. This tool allows a user to run a command (such as deletion of a file or replacement of a string within a file) at any given point in the project's history, and re-constructs the entire branch with the new hashes. The downside of this action is that the user who performed this purge is now out-of-sync with other project members, since they are working with a different base commit. Hence, project members who have concurrently committed new changes to the original branch have to manually re-base their changes to the filtered branch, which can result in peculiar inconsistencies.

In any case, removing such a key leak from history is not a complete solution, since any sensitive information that has been shared publicly for even an instant can be considered to be compromised. Users should ideally have their leaked keys or passwords changed with their service providers to prevent mis-use by a malicious user who may have read the contents in the interval before the project's history was re-written. This tool is thus mainly useful for the removal of other types of sensitive data, such as personal identification information.

C. Preventing Leaks Automatically

Given that it is possible to detect key leaks with high precision using simple heuristics, it should be possible for enhancing version control systems by pro-actively warning a user who may be accidentally leaking a secret key.

We propose a simple solution, that combines the methods of key leak detection and mitigation discussed in this paper, implemented as a plug-in to a version control system that is fired when a user checks-in new code. The flow-chart for such a system is shown in Figure 4. Whenever a new file is added to the repository or an existing file has been changed, a key leak detection module is launched. If a potential key leak is detected, the system warns the user about it and suggests possible auto-refactoring using one of the previously discussed mitigation techniques.

Our proof-of-concept implementation is in the form a *pre-commit hook* for Git, designed for Java projects. The hook applies the pattern detection and entropy-based filtering techniques on lines added or modified in a commit. For source files, we restrict analysis to string literals only. If potential key leaks are detected, users can choose to (1) replace the key string with dummy values, (2) add the file to `.gitignore` (non-source files only) or (3) ignore the warning, in case it was a false positive. If a warning is ignored, the containing file is not scanned in the future for key leaks. We do not currently support transparent encryption or history re-writing.

Of course, no tool can guarantee to catch *all* instances of accidental leakage of sensitive information, and hence developers still need to be smart about taking necessary steps to protect themselves. However, as the precision and recall numbers in Table II show, it is possible to catch a large fraction of such accidental leaks and in these cases tool assistance could save developers from incurring significant losses.

IV. Related Work

Both pattern based [17] and data-flow based [18], [19] techniques have been applied for detecting deliberate leakage of user credentials by malicious software. Similar techniques [7] have also been applied for detecting accidental leakage of credentials in innocuous client-side software. We have addressed another facet, where the software itself may be intended to deployed server-side, but the leakage of credentials occurs through its source code if the application is hosted on open-source code sharing repositories.

Service providers have been urging developers to take measures to prevent such credential leakage [1], but there is no tool support from the version control systems themselves to protect the user against accidental key leakage to the best of our knowledge.

V. Conclusion

Increasingly developers and organizations are warming up to the idea of sharing their code and/or code examples on public code management platforms such as GitHub. Even within enterprises, instead of keep code in dedicated version management systems, projects maintain their code in shared systems.

Accidental leakage of secret keys in such code repositories is a real problem and attacks that mine such platforms to steal API keys have already occurred, causing thousands of dollars in losses for users and service providers alike.

While tools exist to allow developers to host projects requiring API keys on shared code repositories by using measures such as ignore-lists and transparent encryption, accidents are still likely to occur as long as it is the sole responsibility of the developer to safeguard themselves by using these tools.

We argue that more support is required from the version control systems or their plug-ins to assist developers in protecting themselves against such accidents. We have described different ways to detect the presence of API keys or other credentials in source code and presented a solution for tool-based pro-active mitigation of accidental key leaks in code repositories.

References

[1] http://www.itnews.com.au/News/375785, aws-urges-developers-to-scrub-github-of-secret-keys.aspx.

[2] http://www.net-security.org/secworld.php?id=16566.

[3] http://searchcloudsecurity.techtarget.com/news/2240224543/Old-AWS-API-key-led-to-search-providers-cloud-security-breach.

[4] https://securosis.com/blog/my-500-cloud-security-screwup.

[5] https://help.github.com/articles/remove-sensitive-data.

[6] http://it.slashdot.org/story/13/01/25/132203/github-kills-search-after-hundreds-of-private-keys-exposed.

[7] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14, 2014, pp. 221–233.

[8] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, 2013, pp. 233–236.

[9] "Eclipse JDT Core," http://eclipse.org/jdt/core.

[10] "A realistic password strength estimator," https://github.com/dropbox/zxcvbn.

[11] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *SAS'03*.

[12] T. Tateishi, M. Pistoia, and O. Tripp, "Path- and index-sensitive string analysis based on monadic second-order logic," in *ISSTA '11*.

[13] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *ESEC/FSE 2013*.

[14] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*

[15] "Transparent file encryption in git," https://github.com/AGWA/git-crypt.

[16] https://www.kernel.org/pub/software/scm/git/docs/git-filter-branch.html.

[17] M. Roesch, "Snort - lightweight intrusion detection for networks," in *LISA '99*.

[18] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets." in *SOAP '14*.

[19] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *TRUST'11*.