GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

Interested in learning more?

Check out the list of upcoming events offering
"Hacker Tools, Techniques, and Incident Handling (Security 504)"
at http://www.giac.org/registration/gcih

Finding Secrets in Source Code the DevOps Way

Author: Phillip Marlow, phillip@marlow1.com
Advisor: Dr. Johannes Ullrich

Abstract

Secrets, such as private keys or API tokens, are regularly leaked by developers in source code repositories. In 2016, researchers found over 1500 Slack API tokens in public GitHub repositories belonging to major companies (Detectify Labs, 2016). Moreover, a single leak can lead to widespread effects in dependent projects (JS Foundation, 2018) or direct monetary costs (Mogull, 2014). Existing tools for detecting these leaks are designed for either prevention or detection during full penetration-test-style scans. This paper presents a way to reduce detection time by integrating incremental secrets scanning into a continuous integration pipeline.

1. Introduction

Storing sensitive credentials in source code is a common problem. DevOps environments are particularly susceptible to this since they encourage automating the full build and deployment process for software and these credentials are needed by the continuous deployment pipeline to fetch artifacts, deploy results, or many other uses. As Dr. Eric Cole says, "Prevention is ideal, but detection is a must" (SANS, 2019). Without knowing when a credential is compromised, no one can do anything to mitigate the risk the exposure of the credential presents.

Researchers found in 2016 over 1500 Slack API tokens that had been leaked in public GitHub repositories belonging to major companies (Detectify Labs, 2016). These tokens might provide access to exfiltrate internal communications or mine for other shared credentials, such as server or database access. Leaks such as these can have widespread effects beyond the individual service to which the leaked credential applied. For example, the compromise of a single credential from an unrelated breach led to arbitrary code execution on any system that used the ESLint dependency at the time (JS Foundation, 2018). ESLint reached about 3.5 million installs during the week of the incident. In another case, one analyst unintentionally leaked an AWS access key on github.com, which led to direct monetary costs due to fraudulent use of AWS services (Mogull, 2014). Because of the potentially devastating impact of a compromised credential, it is important that development and operations teams identify these compromised credentials early so that they can revoke the credentials before anyone can use them for malicious purposes.

Several tools exist to detect credentials exposed in public source code repositories. A useful tool for penetration testers is truffleHog (Ayrey, 2018), which is designed to scan the entire history of a source code repository to look for potentially-leaked secrets. This approach is thorough, but slow and computationally wasteful in a DevOps pipeline because the tool scans large sections of history it had already analyzed on repeated invocations. Two tools designed for use by developers, git-secrets (Dowling, 2018) and git-hound (Gabrielse, 2017), use git hooks to run before every commit or push to ensure a user never publishes a secret in source control. However, because non-administrator users cannot install git hooks on sites such as gitlab.com, it is not possible

Phillip Marlow, phillip@marlow1.com

to use tools such as these on the server-side where their use is enforceable (GitLab, 2019). On the other hand, Yelp's detect-secrets project (Loo, 2018) integrates well into a DevOps pipeline. However, it requires a persistent server (Yelp, 2019) to perform scans and store metadata which increases the cost of use. Small or budget-constrained projects often use sites such as GitLab or GitHub and may not be able to afford the increased cost of that additional persistent server.

While existing tools fill important roles in the DevOps ecosystem, there remains a need for an inexpensive way to scan for and alert on compromised credentials. Development teams would also benefit from a technique to enforce this scanning on the server-side during continuous integration. This paper presents the strengths and weaknesses of a newly-developed tool, ci_secrets, which fulfills these needs. In addition, this paper will present a recommendation on when using this tool is appropriate and opportunities for future work in scanning for compromised credentials.

2. ci_secrets

2.1. Description

ci_secrets is a simple-to-use tool to scan for credentials in a git repository as part of a DevOps continuous integration (CI) pipeline. Because it is intended for use in this fashion, it is completely self-contained and does not require any persistent storage or other tools. Its output is designed to be easily parsed within a CI script so that appropriate action can be taken such as failing the pipeline and sending alerts.

In order to limit duplicate scanning of the same source code changes, ci_secrets requires a parameter specifying how far back in the commit history the current changeset extends. When ci_secrets is already in use for a project, the commit that the parameter identifies should be the last commit that the tool scanned. The limit of any given scan is set via a command line argument referencing the ID of the latest-scanned commit. ci_secrets uses this method because the command line argument is an easy way to provide this information to the tool from the CI pipeline. Although reusing an existing CI environment variable would require less configuration by the user, it would also limit the types of CI systems in which ci_secrets could be used to a small, predefined list. By using

Phillip Marlow, phillip@marlow1.com

a command line argument, it is easy for a project to insert ci_secrets into a pipeline run by any CI service that supports Python.

ci_secrets checks all file changes in each commit for secrets, starting from the most recent commit and going back until it finds the latest-scanned commit. These individual changes are scanned using the same plugin system used in Yelp's detect-secrets project (Yelp, 2018). This system is beneficial because it allows any developer to create new plugins for scanning for credentials by conforming to the existing API. It also provides several open source plugins that are available for use out-of-the-box, including detection plugins for common credentials such as AWS keys, private key files, and Slack API tokens.

2.2. Use in Continuous Integration

ci_secrets addresses two common source control styles. The first, simplest style is single branch development, where each commit is added one after another on a single branch within the source code repository. In this situation, ci_secrets scans each commit one at a time back to the latest-scanned commit. The second style is one which relies on branching and merge, or pull, requests. In this style, ci_secrets scans each branch as if the single branch style applies. Then, when a user creates a merge or pull request, the common ancestor of the branches is determined; ci_secrets checks each commit individually until it reaches that common ancestor. It is possible to use either of these styles regardless of the choice of source control or continuous integration tool. However, there are some minor differences in how to set up and configure the scan based on the data provided by the tool.

2.2.1. Use in GitHub/Travis CI

Travis CI provides the `TRAVIS_COMMIT_RANGE` environment variable to determine which commits are included within the push or pull request (Travis CI, 2019). ci_secrets determines the latest-scanned commit from the first commit in the range provided as `TRAVIS_COMMIT_RANGE` environment variable's value. If it is the first commit on a new branch, this variable is empty. Because ci_secrets requires a value for the last scanned commit, a user can pass the flag value of 0000000000000000000000000000000000000000 when `TRAVIS_COMMIT_RANGE` is empty.

Phillip Marlow, phillip@marlow1.com

When creating a pull request using GitHub and scanning with Travis CI, a merge commit is first created and then passed to the CI system. The completed merge commit presents a challenge because ci_secrets cannot determine if this merge commit is a result of a pull request, or if the previous commit just happened to be a merge commit. Therefore, the caller needs to specify whether or not the pull request contains a merge commit. ci_secrets provides the `--includesMergeCommit` flag for this purpose, and it should be specified when scanning pull requests from GitHub in Travis CI. Figure 1 shows an example of how to specify this.

```
script:
    - export COMMIT_RANGE=${TRAVIS_COMMIT_RANGE:-"0000000000000000000000000000000000000000"}
    - export LAST_COMMIT=${COMMIT_RANGE%%.*}
    - ci_secrets --since $LAST_COMMIT --includesMergeCommit --log INFO
```

Figure 1: Script for running ci_secrets in Travis CI.

2.2.2. Use in GitLab CI

The correct way to provide the latest-scanned commit when using GitLab CI depends on whether a scan will cover a single branch or if the scan is part of a merge request. GitLab CI provides an easy way to distinguish which jobs should be run for merge requests using the `only` and `except` keywords. A job configured to scan a merge request should be configured with `only: merge_requests` to run only when there is an open merge request for that branch. Figure 2: Script for running ci_secrets in GitLab CI for merge requests.Figure 2 shows an example of this configuration. The `CI_MERGE_REQUEST_TARGET_BRANCH_NAME` environment variable is only available when there is an open merge request, so configuring the distinction between merge requests and non-merge requests is important. This variable provides the name of the target branch and is used by ci_secrets to determine a common ancestor with the current branch. This common ancestor is used as the last scanned commit to limit the scope of the scan.

```
script:
  - ci_secrets --since origin/$CI_MERGE_REQUEST_TARGET_BRANCH_NAME --log INFO
only:
  - merge_requests
```

Figure 2: Script for running ci_secrets in GitLab CI for merge requests.

Phillip Marlow, phillip@marlow1.com

Alternatively, when scanning within a single branch, users should configure the ci_secrets job with `except: merge_requests` to skip running when within the context of the merge request. The job will still run when a merge request is open, but it will run as a separate pipeline. In this case, scans should be limited in scope to only changes that have been made since the most recently-scanned ancestor using the `CI_COMMIT_BEFORE_SHA` environment variable. This variable takes into consideration whether a single commit or multiple commits have been pushed to the repository simultaneously. Figure 3 shows an example configuration for running ci_secrets for merge requests.

```
script:
  - ci_secrets --since $CI_COMMIT_BEFORE_SHA --log INFO
except:
  - merge_requests
```

Figure 3: Script for running ci_secrets in GitLab CI for non-merge requests.

By specifying different configurations for merge requests and ordinary updates, users can better specify the range of commits that ci_secrets should scan. While `CI_COMMIT_BEFORE_SHA` can specify this sufficiently for incremental updates, it does not cover an entire merge request. Because of this, the result from ci_secrets may give a false impression that a leak has already been resolved when used as part of a merge request. Section 3 covers this situation in more detail.

3. Analysis

Three test scenarios were created to evaluate the performance of ci_secrets in detecting leaked credentials. The first scenario is a simple commit of a credential which the user then pushes to the source control server. The second scenario simulates a user pushing several commits at the same time to the source control server, and the leaked credential is in the middle of the published commits. In the third scenario, a user has incorrectly removed a leaked credential. Specifically, a credential is committed but later removed in a separate commit. Finally, this paper also considers the additional complexity resulting from using a branching source control method and merge or pull requests.

Phillip Marlow, phillip@marlow1.com

In each test scenario, the user adds Amazon Web Services (AWS) secret and access keys to the repository. The keys added are stored in a credentials file which is commonly used to configure the AWS CLI tool with access to one or more accounts. In order to prevent the compromise of a real AWS account during the test, the keys used for testing are the example keys published by Amazon which have the same format as real keys but do not provide access to an account. The AWSKeyDetector plugin can detect these keys when ci_secrets scans the commit containing them.

3.1. Test Scenario 1

In the first test scenario, where a secret is simply added and pushed to source control, ci_secrets computes the changes between the new commit and the previous one and then uses the configured plugins to scan those changes for secrets. When it finds a secret, it increments a count of found secrets and prints a summary of each secret to the log. Based on finding this secret, it exits with a failure code so that the CI job fails. When this happens, the CI system sends notifications so that recipients can further investigate and remediate the finding. Figure 4 shows this failure in build #51745431.

When a developer adds and publishes subsequent commits, those scans are independent of the existence of any secrets in earlier commits. This independence is by design so that only new changes can be scanned to speed up the continuous integration process. However, this also means that the build status of the project can return to "passed" without resolving the underlying credential leak. Figure 4 shows this case where build #51746328 passed. This situation includes cases where a user improperly removes credentials by adding a new commit to remove them. Because of this, it is important to investigate and address every failure from the ci_secrets scan as it occurs so that no leaked credentials can fall through the cracks.
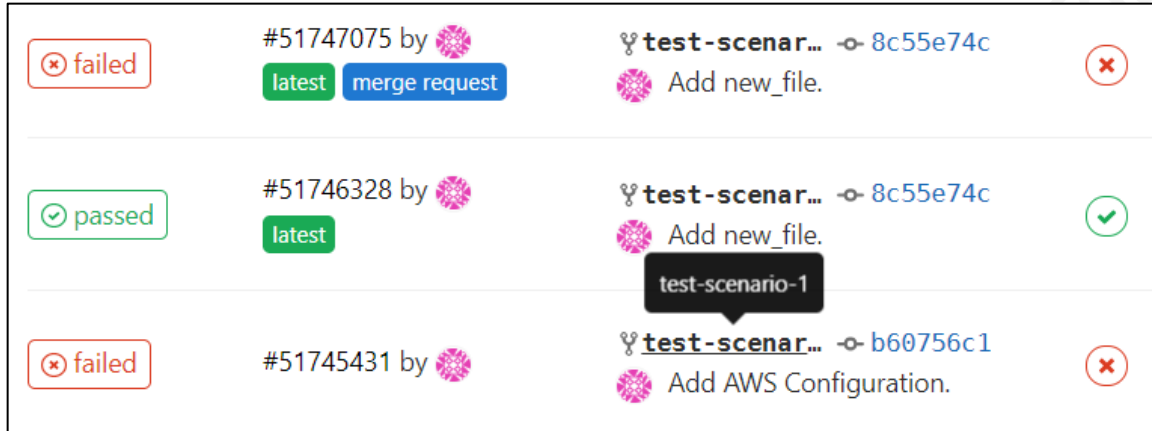
Phillip Marlow, phillip@marlow1.com

Figure 4: Test Scenario 1 in GitLab CI.

The risk described above can be somewhat mitigated when all development is done using a branching strategy. Although ci_secrets scans each commit independently, the entire branch is scanned during the merge or pull request process, providing a second opportunity to catch the missing leak. So, in this test scenario, if subsequent commits do not contain secrets which cause the build to fail, such as in commit 8c55e74c in Figure 4, the build resulting from the merge or pull request would still fail because it will identify secrets contained in any of the new commits. Figure 4 shows this when build #51747075, labeled as a merge request, fails. However, while this is a useful mitigation, it should be considered a backup and not a primary method of identifying when credentials need to be revoked and rotated. If ci_secrets only scans for leaked secrets when a user opens a pull or merge request, that configuration may result in the scan happening a considerable time after the leak initially occurred. Scanning during both incremental development and pull or merge requests ensures both a quick time to detect leaks as well as thoroughness in coverage of published source code.

3.2. Test Scenario 2

The second test scenario contains a secret credential which is part of a commit that is pushed to the source control server simultaneously with other commits containing no secrets. ci_secrets then scans the changes between each commit in sequence until it reaches the last-scanned commit. In this case, when a commit does not contain a secret, ci_secrets continues scanning with the next commit. When it does find a secret within a commit, it performs as if that commit had been the only commit to be scanned. Specifically, ci_secrets increments a count of found secrets and logs a summary of each

Phillip Marlow, phillip@marlow1.com

secret found. However, if there are additional commits that ci_secrets has not yet scanned, it does not exit with a failure code until it scans all commits. By doing this, ci_secrets ensures that it will log and notify on all secrets in the repository, limiting the extent of any additional manual investigation. Figure 5 shows the red X indicating that a build covering commits c9ef7e4 through 7f7b433 failed due to credentials in commit 9c41c1a.
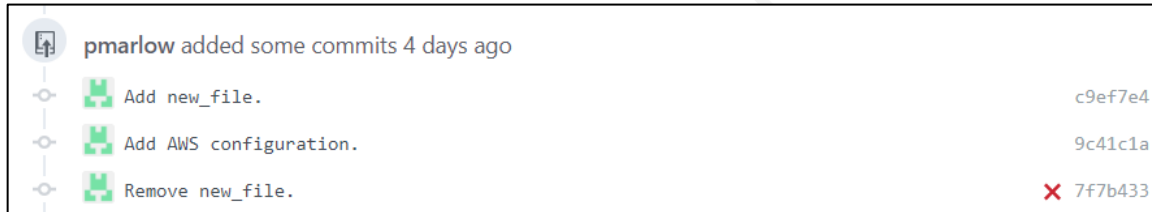

Figure 5: Test Scenario 2 in a GitHub pull request.

This scenario presents the same risks associated with subsequent batches of commits which are published simultaneously, as in the first scenario above. Specifically, because each set of commits is scanned independently from previously published sets, it is possible that the build will return to a successful status when a user publishes a subsequent batch of commits which does not contain any new secrets. The same mitigation using merge or pull requests is possible, and the same concern about the duration in which a leaked secret remains available but unresolved is still valid. Figure 6 shows this situation where a pull request is failing despite the last commit on the branch passing.

Figure 6: Travis CI status for Test Scenario 2 in a GitHub pull request.

By checking all commits on a branch during a pull or merge request, the second test scenario and the first are identical for any case where a branch contains more than one commit. This scenario happens because commits do not store any metadata about

Phillip Marlow, phillip@marlow1.com

when they are published, so it is impossible to tell from source control alone how many commits a user published to the server at one time.

3.3. Test Scenario 3

For the third test scenario, a secret credential is added in one commit and then improperly removed in a subsequent commit. If these commits are pushed independently to the source control server, they are also scanned independently by ci_secrets. This situation is one realization of the risk outlined in the first test scenario. The build resulting from the commit that added a secret would be scanned and fail, while the subsequent build resulting from the commit that removed the secret would be scanned and pass. If instead the commits containing the addition and subsequent removal of the secret are published simultaneously, the CI system only runs a single build and therefore a single scan. That scan will identify the commit which leaked the secret, fail the build, and cause the CI system to send an appropriate notification. Figure 7 shows this situation where commit 0678825f adds the credentials, commit 1c080eb3 removes them, and the build is run and fails after that. Either way, ci_secrets correctly identifies the leaked secret by performing a commit-by-commit scan through new changes rather than only considering the net changes. Leaked credentials are detected even when they are never immediately available from an instantaneous snapshot of the published code but exist in the change history.
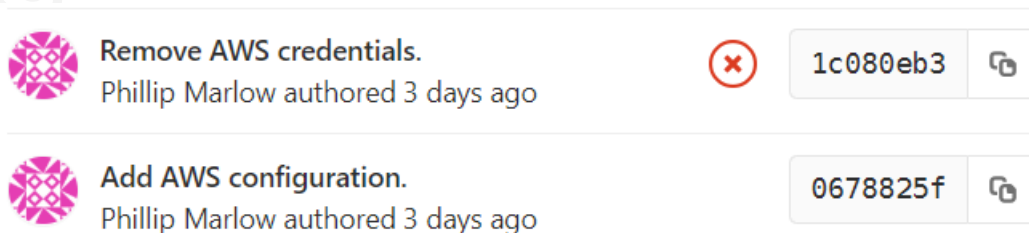

Figure 7: Test Scenario 3 in GitLab.

3.4. Complexity with Branching

Although the sections above already address branching, there are some additional complexities which result from trying to scan for secrets in a branching repository. These issues arise from difficulties in identifying the source from which a branch was created, in trying to run a ci_secrets scan after a merge commit, and in efficiently identifying all the commits ci_secrets still needs to scan on a given branch.

Phillip Marlow, phillip@marlow1.com

3.4.1. Identifying the Branch Source

Git does not store metadata on the source of branches when a user creates a branch. Practically, this results in no way for ci_secrets to determine the correct common ancestor to use as the latest-scanned commit to limit the scope of its scan. Three cases need to be considered regarding the state of a branch when trying to limit the scope of a scan: a new branch, a branch under active development, and a branch ready to be merged.

When the branch is at its end and ready to be merged, not knowing the source of the branch becomes irrelevant, because we know the target branch for the merge. Often this will be the source branch, but it does not have to be. Either way, a common ancestor can be found based on the target branch to provide a limit to the scope of the scan. When the target is the source branch, ci_secrets scans every commit since the original branching occurred and duplication of scanning is limited to any scans that have occurred during the development of the branch. If the target is another branch, it is likely that some duplication of scanning will result, but this will still be an improvement over re-scanning the entire repository.

A branch under active development is the simplest case. It can stand on its own without any regard to whether or not new commits are occurring on the main or master branch, or if new commits are occurring on any other branch. Users apply these meanings on top of the source control model, as they are not inherent to the model itself. Practically, this means that ci_secrets can compare each new set of commits to the previously-scanned set on the branch and there is no concern about an unknown source of the branch.

Finally, the most difficult case is the beginning of a new branch. Because a user can publish more than one commit at a time and there is no metadata tracking the source of a branch, it is not possible to determine the appropriate common ancestor that ci_secrets will use as the latest-scanned commit. In testing, this results in GitLab CI providing a fake commit value of 0000000000000000000000000000000000000000 in the variable CI_COMMIT_BEFORE_SHA, while the value of TRAVIS_COMMIT_RANGE in Travis CI is empty. ci_secrets, not knowing the appropriate commit at which to stop its scan, prints a warning to the log. Additionally, it assumes that the most common case is that a single commit creates the new branch and therefore only scans a single commit

Phillip Marlow, phillip@marlow1.com

back into the branch history. The result is a risk that a credential leaked in an earlier commit to the new branch and ci_secrets will not find it. By scanning the entire branch before the merge, this risk is partially mitigated since ci_secrets will find secrets at that time. However, the leaked credential may go undetected for a significant amount of time before that occurs. Also, if the branch is abandoned or deleted, ci_secrets may never properly identify the leak.

3.4.2. Scanning after a Merge Commit

When using GitLab CI, the context for scanning a merge request maintains the source and target branches of the merge as separate branches. Since the source branch is checked out and the name of the target branch is in the `CI_MERGE_REQUEST_TARGET_BRANCH_NAME` environment variable, it is easy for ci_secrets to compare these to find a common ancestor. However, when using GitHub and Travis CI, a merge commit is first created and then passed to the CI system. This commit presents a challenge because ci_secrets cannot determine if this merge commit is a result of a pull request or if the previous commit just happened to be a merge commit. Therefore, the caller needs to specify whether or not the pull request contains a merge commit. ci_secrets provides the `--includesMergeCommit` flag for this purpose.

If the configuration omits the `--includesMergeCommit` flag, ci_secrets identifies the parent of the merge commit as the common ancestor of the merged branches and does not continue to scan the rest of the branch. When the configuration includes the flag, the scan continues back along the source branch until it once again reaches a commit whose parent is the common ancestor. This configuration option allows ci_secrets to correctly scan the source branch but puts the onus on the user to configure the `--includesMergeCommit` flag correctly for their environment. If the caller does not set the flag correctly, cases such as test scenario three where a credential is added and incorrectly removed may not emerge during the pull request. There are two methods to mitigate this risk. First, a user should always configure the tool to scan each set of commits when the CI server receives them. By doing this, a user will identify the leak earlier and without ever encountering the situation dealing with the merge commit. Second, a user should only need to configure this setting once — at the time that a project starts using ci_secrets. At that time, concerns about security and correctly configuring the

Phillip Marlow, phillip@marlow1.com

new tool will be at their highest. After that, the configuration will remain until someone actively modifies it.

3.4.3. Branches within Branches

Because there are no special branches within Git, it is possible to create a new branch starting from any other branch. Therefore, it is possible to have branches which are multiple levels "deep" that ci_secrets needs to scan. If all these branches are pushed to the CI server and scanned as developers create and merge them, ci_secrets treats these branches as it would any other, and, as a result, will correctly identify leaked secrets. However, if a sub-branch is developed and merged locally without being pushed to the CI server and scanned, ci_secrets will not identify all the commits from that sub-branch when scanning the branch into which it merged. Although in the author's experience, this is an uncommon situation, it still needs to be considered as part of the overall project risk. This risk is especially important to consider if the user is concerned about a potential insider threat especially since this scenario provides a known way to exfiltrate credentials without detection.

3.5. Notifications

The identification of secrets is useless unless the user is aware that a leak has happened. Therefore, it is important that appropriate notifications are sent to users when ci_secrets has detected a leak. The main method for this notification is failing the build in the CI pipeline. This failure is highly visible to developers and appears prominently in several locations regardless of the choice of source control and CI system. In GitLab CI, it appears at the top of the repository view when viewing a branch and within the merge request. It also appears throughout the various views in the CI/CD section. GitHub displays this prominently on the pull request overview and within the individual pull requests. GitHub also shows the status of any builds from commits included in the pull request's history.

By default, when a build fails, both GitLab CI and Travis CI will send email notifications to the users who published the changes that broke the build. The CI system also sends these notifications when ci_secrets detects a leak. By default, GitLab provides the log text in the body of the email, showing the results of the scan as seen in **Error!**

Phillip Marlow, phillip@marlow1.com

Reference source not found.. This format makes it immediately clear to the user that the failure was because of leaked credentials.

Figure 8: Email notification from GitLab CI showing the detection of an AWS access key.

In contrast, Travis CI provides only a link, as seen in **Error! Reference source not found.**, but the full log text is available after clicking through to the Travis CI website. The contents of the email do not immediately inform the user that the build

Phillip Marlow, phillip@marlow1.com

failed because of leaked credentials and might, on its own, mean that the leak is not resolved in a timely fashion.

Figure 9: Email notification from Travis CI showing the failure of a build when a secret is found.

Because these failures are critical to security, more contributors than just the user who caused the leak may need to be made aware of the failure. For example, a security team or the project owner may also need to receive a notification when a leak occurs. Both GitLab CI and Travis CI provide several options for specifying additional notifications. For additional email notifications, GitLab CI provides the Pipelines emails integration, and Travis CI provides the `notifications:  email:` configuration option. Similar integration and configuration options exist for other notification mechanisms such as Slack and custom webhooks. By default, these notifications may be too noisy since they may trigger for reasons other than security, but they can be filtered by looking for the ci_secrets job name.

4. Conclusion

ci_secrets successfully identifies and alerts on several common cases of leaked secrets in a DevOps pipeline. This tool provides an option for developers and incident handlers to quickly identify leaked credentials so they can investigate and revoke them before anyone can use the credentials maliciously. However, ci_secrets is not a silver

Phillip Marlow, phillip@marlow1.com

bullet, and users should understand its limitations when deciding to include this tool as part of a secure development program.

As ci_secrets is a new tool, there remain many opportunities to study and improve on both ci_secrets and the context in which it runs. These may include studying how ci_secrets behaves in other development methodologies using Git, such as when a project uses re-basing as a merge strategy. One opportunity for further development is to extend ci_secrets to handle source control systems other than Git, such as Subversion or Mercurial. There also may be opportunities to improve how ci_secrets handles the branching issues identified in the analysis section of this paper. One possible approach for future development in improving the handling of branches could explore how continuous integration systems can be modified to provide additional information to tools such as ci_secrets during scanning. Another approach might consider how Git could be enhanced to provide more metadata about the history of branches.

In conclusion, ci_secrets provides a robust and flexible solution to the problem of detecting sensitive credentials stored in source code in a DevOps environment. The potential for complete automation of the build and deployment process that DevOps environments offer is tremendously valuable for developers yet presents unique challenges from a security perspective. A lightweight tool, which does not require a persistent server, will enable many users in common use cases to mitigate this potentially severe and expensive risk. The ci_secrets tool works well to address the core vulnerability of exposed credentials in source code and provides developers and incident handlers with a straightforward method to better secure their software development process.

Phillip Marlow, phillip@marlow1.com

References

Ayrey, D. (2018, December 6). *truffleHog*. Retrieved from GitHub:

    https://github.com/dxa4481/truffleHog

Detectify Labs. (2016, April 28). *Slack bot token leakage exposing business critical*

    information. Retrieved from Detectify Labs:

    https://labs.detectify.com/2016/04/28/slack-bot-token-leakage-exposing-business-

    critical-information/

Dowling, M. (2018, October 24). *git-secrets*. Retrieved from GitHub:

    https://github.com/awslabs/git-secrets

Gabrielse, Z. (2017, January 8). *git-hound*. Retrieved from GitHub:

    https://github.com/ezekg/git-hound

GitLab. (2019, January 19). *Custom Git Hooks*. Retrieved from GitLab Docs:

    https://docs.gitlab.com/ee/administration/custom_hooks.html

JS Foundation. (2018, July 12). *Postmortem for Malicious Packages*. Retrieved from

    ESLint: https://eslint.org/blog/2018/07/postmortem-for-malicious-package-

    publishes

Loo, A. (2018, June 11). *Yelp's Secret Detector: Preventing Secrets in Source Code*.

    Retrieved from Yelp Engineering:

    https://engineeringblog.yelp.com/2018/06/yelps-secret-detector.html

Mogull, R. (2014, January 7). *My $500 Cloud Security Screwup*. Retrieved from

    Securosis: https://securosis.com/blog/my-500-cloud-security-screwup

SANS. (2019, January 19). *Advanced Security Essentials - Enterprise Defender*.

    Retrieved from SANS: https://www.sans.org/course/advanced-security-essentials-

    enterprise-defender

Phillip Marlow, phillip@marlow1.com

Travis CI. (2019, February 25). *Environment Variables*. Retrieved from Travis CI:

https://docs.travis-ci.com/user/environment-variables/

Yelp. (2018, December 28). *detect-secrets*. Retrieved from GitHub:

https://github.com/Yelp/detect-secrets

Yelp. (2019, January 9). *detect-secrets-server*. Retrieved from GitHub:

https://github.com/Yelp/detect-secrets-server

Phillip Marlow, phillip@marlow1.com