# Security Assessment of slf4j on behalf of Open Source Technology Improvement Fund

## TABLE OF CONTENTS

## EXECUTIVE SUMMARY

### Include Security (IncludeSec)

IncludeSec brings together some of the best information security talent from around the world. The team is composed of security experts in every aspect of consumer and enterprise technology, from low-level hardware and operating systems to the latest cutting-edge web and mobile applications. More information about the company can be found at www.IncludeSecurity.com.

### Assessment Objectives

The objective of this assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. Recommendations were provided for remediation steps which slf4j could implement to secure its applications and systems.

### Scope and Methodology

Include Security performed a security assessment of slf4j's Security Review. The assessment team performed a 10 day effort spanning from April 25th – May 6th, 2022, using a Standard Grey Box assessment methodology which included a detailed review of all the components described in a manner consistent with the original Statement of Work (SOW).

### Findings Overview

IncludeSec identified 3 categories of findings. There were 0 deemed to be "Critical-Risk," 0 deemed to be "High-Risk," 0 deemed to be "Medium-Risk," and 1 deemed to be "Low-Risk," which pose some tangible security risk. Additionally, 2 "Informational" level findings were identified that do not immediately pose a security risk.

IncludeSec encourages slf4j to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

### Next Steps

IncludeSec advises slf4j to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used by as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist slf4j in improving their SDLC in future engagements by providing security assessments of additional products. For inquiries or assistance scheduling remediation tests, please contact us at remediation@includesecurity.com.

## RISK CATEGORIZATIONS

At the conclusion of the assessment, Include Security categorized findings into five levels of perceived security risk: Critical, High, Medium, Low, or Informational. **The risk categorizations below are guidelines that IncludeSec understands reflect best practices in the security industry and may differ from a client's internal perceived risk. Additionally, all risk is viewed as "location agnostic" as if the system in question was deployed on the Internet. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. Any discrepancies between assigned risk and internal perceived risk are addressed during the course of remediation testing.**

**Critical-Risk** findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

**High-Risk** findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

**Medium-Risk** findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

**Low-Risk** findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration, and outdated patches or policies.

**Informational** findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application. Any informational findings for which the assessment team perceived a direct security risk, were also reported in the spirit of full disclosure but were considered to be out of scope of the engagement.

The findings represented in this report are listed by a risk rated short name (e.g., C1, H2, M3, L4, and I5) and finding title. Each finding may include if applicable: Title, Description, Impact, Reproduction (evidence necessary to reproduce findings), Recommended Remediation, and References.

# INTRODUCTION

## Project Scoping

On April 25th, 2022, the assessment team began analyzing the slf4j (Simple Logging Façade for Java) application. The following areas were of key focus during the assessment:

- **Manual Code Review** – Assessing code using a combination of static analysis, dynamic analysis, and manual review.
- **Threat Modeling** – Assessing potential threats, attacks, and mitigations.
- **Review of Supply Chain Security Against SLSA** – Assessing the current state of the supply chain security in the **slf4j** project against the SLSA model v0.1.
- **Evaluation of Continuous Integration Pipeline and Automated Security Testing** – Assessing the current configuration of automated security testing for the project.

## Testing Methodology

A dedicated instance of the slf4j application was provided. Testing of the application involved both dynamic and static application testing. Dynamic testing involved interacting with slf4j testing instance that was provided. Static testing was performed by manual source code review of each in-scope repository.

## Threat Modeling

### Application Decomposition

The framework is lightweight and was found to present a limited attack surface by itself. Since it is a popular framework integrated into many downstream projects, the assessment focused primarily on supply-chain related vulnerabilities.

### Attacker Behavioral Summary

- An attacker might want to abuse any flaws present in the library to attack applications that integrate it.
- An attacker might want to inject malicious code into builds of the library hosted on Maven Central in order to attack applications that integrate it.
- An attacker might like to trick consumers of the library into integrating a non-genuine version of the library containing malicious code.
- An attacker might want to inject code into a dependency of **slf4j** that is less popular than **slf4j** itself, thereby magnifying the potential impact.
- An attacker might want to compromise the **slf4j** website and change the build links to point to malicious packages.
- An attacker might want to "squat" names that are similar to **slf4j** on Maven Central and GitHub to trick users who misspell the project into downloading the incorrect packages.

### Application Threats

In it's current state as of April 2022 **slf4j** has a very small attack surface area. It does not support advanced logging capabilities that may be cause for security concern (such as the log4j vulnerabilities published in 2021.) Furthermore despite the specific integrations (or "wrapped implementation" in slf4j parlance) for many loggers such as **JUL**, **log4j**, etc. **slf4j** itself is very lightweight and does not add any notable attack surface area between the application and the logging framework. The logging frameworks themselves are separate dependencies and subject to their own vulnerabilities as discussed further below in this section of the report.

The most probable way that flaws in **slf4j** itself would manifest would be flaws in the handling of log messages containing malicious data, since this is likely the only untrusted data passed to **slf4j** by integrating applications.

An attacker wishing to insert malicious code into the library might attempt this in several ways.

- They might attempt to compromise the maintainer's computer.
- They could physically threaten the package's maintainer.
- They could compromise the project maintainer's GitHub account.
- They could make seemingly innocuous commits to the repository that introduce subtle vulnerabilities, perhaps by compromising the GitHub accounts of previous contributors to the project, lending credibility.
- They could attempt to compromise the account used to update the **slf4j** packages on Maven Central.

Some of the ways that someone attempting to compromise the **slf4j** web presence might proceed include:

- Directly obtaining the appropriate credentials to log in to the server, e.g., via compromise of the project maintainer's computer, phishing, or similar methods.
- Hijacking the slf4j.org domain name by attacking the domain register, abusing lax security on the domain, or opportunistically taking advantage of an accidental domain expiration.

**Application Mitigations**

*Current Mitigations:*

The fact that **slf4j** builds are reproducible adds significant defense against code tampering, although it relies on third parties to actually do the verification. The **slf4j** builds are also signed with GPG, meaning an attacker who is able to tamper with the final builds but who has not compromised the signing key could be discovered. Once again, though this process is not automatically performed by Maven, it relies on a third party to manually do this verification.

The project maintainer takes sensible precautions with the various accounts involved in **slf4j**, such as enabling two-factor authentication and using a password manager.

*Possible Improvements:*

Some possible improvements to the build process are noted in **Evaluation of Continuous Integration Pipeline and Automated Security Testing**.

## Review of Supply Chain Security Against SLSA

The assessment team evaluated the current state of the supply chain security in the **slf4j** project against the SLSA model v0.1. SLSA defines four levels with increasing degrees of confidence in the ultimate integrity of the build artifacts. A discussion of the results can be found in section **Evaluation of Continuous Integration Pipeline and Automated Security Testing**.

**L1 Requirements:**

| Category | Requirement | Satisfied? | Notes |
|---|---|---|---|
| **Build** | Scripted build | YES | **slf4j had a fully automated build process using Maven.** |
| **Provenance** | Available | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Identifies artifact | NO | **slf4j did not include a formal SLSA provenance with releases.** |

| | | | |
|---|---|---|---|
| **Provenance** | Identifies builder | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Identifies build instructions | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Includes metadata (optional) | NO | **slf4j did not include a formal SLSA provenance with releases.** |

**L2 Requirements:**

| Category | Requirement | Satisfied? | Notes |
|---|---|---|---|
| **Source** | Version controlled | YES | **The framework uses Git for version control.** |
| **Build** | Build service | NO | **Builds of the framework were created manually by the project maintainer.** |
| **Provenance** | Authenticated | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Service generated | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Identifies source code | NO | **slf4j did not include a formal SLSA provenance with releases.** |

**L3 Requirements:**

| Category | Requirement | Satisfied? | Notes |
|---|---|---|---|
| **Source** | Verified history | YES | **The application uses signed commits.** |
| **Source** | Retained at least 18 months | NO | **By default, git history is not immutable, and the slf4j repository was not configured to change this.** |
| **Build** | Build as code | NO | **The build script is a text file in the VCS, but is not invoked by a build service** |
| **Build** | Ephemeral environment | NO | **Builds were created on the project maintainer's system.** |
| **Build** | Isolated | NO | **Builds were created on the project maintainer's system.** |
| **Provenance** | Non-falsifiable | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Identifies as source code (authenticated) | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Identifies entry point | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Includes all build parameters | NO | **slf4j did not include a formal SLSA provenance with releases.** |

**L4 Requirements:**

| Category | Requirement | Satisfied? | Notes |
|---|---|---|---|
| **Source** | Retained indefinitely | NO | **By default, git history is not immutable, and the slf4j repository was not configured to change this.** |
| **Source** | Two-person reviewed | NO | **Pull requests were reviewed by the project maintainer, satisfying the requirement. However commits directly from the maintainer were not reviewed by a second person.** |
| **Build** | Parameterless | NO | **The build process accepted a parameter related to GPG signing.** |
| **Build** | Hermetic | NO | **The application downloaded several dependencies and plugins from the Maven Central during build time, including 39 .jar files** |
| **Build** | Reproducible (optional) | YES | **Starting from v1.7.36, slf4j builds are reproducible.** |
| **Provenance** | Dependencies complete | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Identifies source code (complete) | NO | **slf4j did not include a formal SLSA provenance with releases.** |

| | | | |
|---|---|---|---|
| **Provenance** | Includes all transitive dependencies | NO | **slf4j did not include a formal SLSA provenance with releases.** |
| **Provenance** | Includes reproducible info | NO | **slf4j did not include a formal SLSA provenance with releases.** |

## Evaluation of Continuous Integration Pipeline and Automated Security Testing

The **slf4j** project has already implemented reproducible builds. This means that the open-source community can build the project from the publicly available source code repository and obtain the exact same output as the builds supplied in Maven. This is a significant security win that, in the SLSA model, is not required until the highest level of confidence (level 4).

One major source of risk in the current setup revolves around the fact that builds are created manually on the project maintainer's computer. Compromise of that computer could lead to compromise of the build artifacts. This is somewhat mitigated by the fact that builds are reproducible — as discussed above, the community could detect when builds do not match the current state of the public source code repository. However, it was not clear that such verification was actually performed by the community, or how long detection might take in the event of a compromise. The tampered build artifacts might be automatically downloaded and included in many software projects before detection.

Because of these factors, offloading the build process to a dedicated build machine (such as GitHub Actions) would confer significant gains in supply chain confidence. This coupled with the inclusion of a signed SLSA provenance file containing metadata about the build such as builder and build instructions would allow the project to achieve SLSA level 2.

After that, by making some changes to the repository to disallow modification of the git history via force pushes, along with some additional fields and properties to the build provenance, the library could be further upgraded to SLSA level 3. Git can be made immutable repository-wide by setting the **receive.denyNonFastForwards** and **receive.denyDeletes** configurations. Alternatively, particular branches can be made immutable using a git hook such as this example one (Example git.kernal.org Hook Link) provided on kernel.org.

Due to the simplicity of the framework, the assessment team believes it is unlikely to significantly benefit from fuzzing and most automated security tests. However, one possible improvement here would be to include some automated tests that look for known vulnerabilities in the framework's dependencies. One such test would be DependencyCheck, which is part of the OWASP project. This tool has a Maven plugin for automatically running the tests during builds. No vulnerabilities were found in dependencies using this tool at the time of assessment.

## LOW-RISK FINDINGS

### L1: GPG Signing Passphrase Supplied via Command-Line Argument

***Description:***

The **slf4j** library contained a script used for creating builds. The script optionally accepts a password as a command-line parameter. When a password is supplied, the script in turn invokes Maven, and passes the password again as a command line argument. The password is then used by Maven as the GPG passphrase for signing the release.

All arguments specified via a command-line are available in the process table to all other users on the system. They are also usually recorded in a user's shell history.

***Impact:***

An attacker who compromises the computer of a project maintainer that has used this script might be able to recover the GPG passphrase from the user's shell history. This, in combination with the GPG key, would allow the attacker to sign their own releases. If the release is performed on a multiuser system, all users would be able to view the password by inspecting the process table at the correct time.

***Reproduction:***

The script in question is **release.sh**. On line 42, it uses the value accepted on the command line as the GPG passphrase:

```
PASS=$1
echo $PASS
[...]
if [ ! -z "$PASS"  ]
then
  echoRunAndCheck "$MVN deploy -P javadocjar,sign-artifacts -Dgpg.passphrase=$PASS"
fi
```

When passed, the password is stored in the user's shell history:

```
$ bash
$ ./release.sh hunter2
[...]
$ exit
exit
$ bash
$ tail ~/.bash_history
[...]
./release.sh hunter2
exit
```

***Recommended Remediation:***

The assessment team recommends not passing the password via the command line. If a passphrase isn't specified to Maven, it will prompt for one interactively. Alternatively, secrets can be passed as environment variables in the POM. In addition, the **slf4j** team could consider clearing any lines containing the password from maintainers' shell histories and/or rotating the GPG passphrase for the signing key.

***Remediation Notes:***

This finding was retested and found to be closed. As can be seen below, the original script was modified to eliminate the requirement for a GPG passphrase argument in favor of an interactive prompt on the TTY.

```
if [ ! -z "$PASS"  ]
then
  export GPG_TTY=$(tty)
  echoRunAndCheck "$MVN deploy -P javadocjar,sign-artifacts"
fi
```

***References:***

Passing Passwords
POM Documentation

## INFORMATIONAL FINDINGS

### I1: Project Does Not Define Security Policy

***Description:***

GitHub allows projects to create a SECURITY.md file that defines the project's security policy, such as the process for reporting security vulnerabilities. The assessment team noted that the **slf4j** project did not define a security policy.

***Impact:***

Community members who discover a vulnerability in **slf4j** may not know how to report it. This might lead to them reporting it in a public way (such as GitHub issues or the **slf4j** JIRA), or over unencrypted email, leading to disclosure of the vulnerability before the vulnerability can be remediated.

***Reproduction:***

To reproduce this finding, visit https://github.com/qos-ch/slf4j/security/policy and note that there is no security policy.

***Recommended Remediation:***

The assessment team recommends implementing a security policy that directs users with regard to how to submit vulnerabilities privately to the project's maintainers over a secure mechanism, such as via GPG encrypted emails.

***Remediation Notes:***

This finding was retested and found to be closed. The repository now contained a security policy built from the standard template with additional relevant details, found by visiting the link included in the reproduction section.

***References:***

Adding a Security Policy to Your Repository

## I2: Infrastructure Information Disclosure in Public Repository

***Description:***

The assessment team found instances where technical information about infrastructure used by the **slf4j** project (such as the web host) was available within the publicly available **slf4j** repository.

***Impact:***

An attacker might leverage the technical information when attempting to attack **slf4j** project infrastructure. The technical information included the hostname used when uploading the **slf4j** website via SCP, along with the full path to the web root on that host.

***Reproduction:***

The snippet below from file **slf4j/release.sh**, line 53, contains the web host information in a comment:

```
# for 1.7.x series, scp slf4j-1.7.*.tar.gz yvo.qos.ch:/var/www/www.slf4j.org/htdocs/dist/
```

***Recommended Remediation:***

The assessment team recommends, removing all infrastructure related information from the public repositories. Optionally (for more assurance) change all infrastructure hostnames and no longer using the IPs/hosts mentioned in the public docs or firewalling them off from the Internet completely.

***Remediation Notes:***

This finding was retested and found to be closed. The technical information originally extracted from a commented line was no longer present. The remaining source code was further analyzed to find other occurrences of similar information disclosure, and none could be found.

***References:***

Information Disclosure