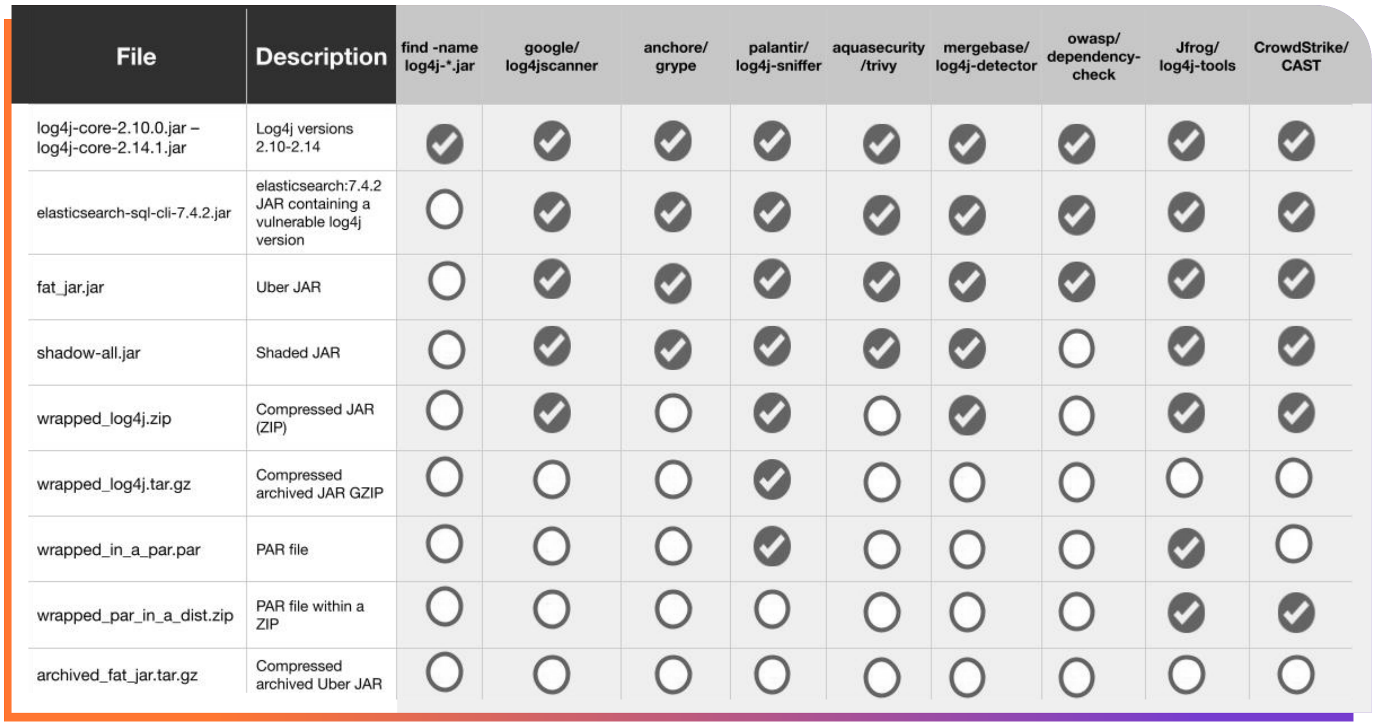
Log4j Blindspots: What Your Scanner Is Still Missing

Rezilion Research Report

THE POPULARITY OF THE LOG4J LIBRARY, coupled with the ease of exploitability and severe potential impact, means Log4Shell's blast radius is enormous — that's old news by now. However, what's being revealed these last few days is not just how popular it is, but **how deeply rooted it is in the software we use** — and this depth is creating some unique challenges in detecting it.

The biggest challenge lies in detecting Log4Shell within packaged software in production environments: Java files (such as Log4j) can be nested a few layers deep into other files — which means that a shallow search for the file won't find it. Furthermore, they may be packaged in many different formats which creates a real challenge in digging them inside other Java packages.

In order to estimate how big the industry's current blindspot is Rezilion's vulnerability research team conducted a survey where multiple open source and commercial scanning tools were assessed against a dataset of packaged Java files where Log4j was nested and packaged in various formats. All formats are commonly used by developers and IT teams. Some scanners did better than others, but **none was able to detect all formats**. A detailed analysis can be found in the research below.



File	Description	find -name log4j-*.jar	google/ log4jscanner	anchore/ grype	palantir/ log4j-sniffer	aquasecurity /trivy	mergebase/ log4j-detector	owasp/ dependency-check	Jfrog/ log4j-tools	CrowdStrike/ CAST
log4j-core-2.10.0.jar – log4j-core-2.14.1.jar	Log4j versions 2.10-2.14	✓	✓	✓	✓	✓	✓	✓	✓	✓
elasticsearch-sql-cli-7.4.2.jar	elasticsearch:7.4.2 JAR containing a vulnerable log4j version	○	✓	✓	✓	✓	✓	✓	✓	✓
fat_jar.jar	Uber JAR	○	✓	✓	✓	✓	✓	✓	✓	✓
shadow-all.jar	Shaded JAR	○	✓	✓	✓	✓	✓	○	✓	✓
wrapped_log4j.zip	Compressed JAR (ZIP)	○	✓	○	✓	○	✓	○	✓	✓
wrapped_log4j.tar.gz	Compressed archived JAR GZIP	○	○	○	✓	○	○	○	○	○
wrapped_in_a_par.par	PAR file	○	○	○	✓	○	○	○	✓	○
wrapped_par_in_a_dist.zip	PAR file within a ZIP	○	○	○	○	○	○	○	✓	✓
archived_fat_jar.tar.gz	Compressed archived Uber JAR	○	○	○	○	○	○	○	○	○

Another alternative that was assessed is using source-code analysis in order to detect dependencies on Log4j. While this approach will effectively detect nested and packaged instances of the vulnerable component, not all vulnerable findings can be addresses since most of them will probably be in third party software (which still constitutes for most of the code in production). Furthermore, it has major blindspots when it comes to transient dependencies, which we also discuss in the research.

Rezilion's findings in this initiative demonstrate the limitations of static scanning in detecting Log4j instances, and highlights the need for code-level visibility in runtime memory where the code isn't packaged or nested. It also reminds us that detection abilities are only as good as your detection method. Every scanner has blindspots. Security leaders cannot blindly assume that one open source or even commercial-grade tool will be able to detect every edge case — and in the case of Log4j there are a lot of edge cases in many places.

This report details the findings of our research.

Details
Log4Shell is up there with HeartBleed and ShellShock when it comes to potential impact and the time it is expected to keep affecting systems worldwide. Rezilion researchers have been watching it unfold. As mentioned in a previous blog post, the popularity of the Log4j library, coupled with the ease of exploitability and severe potential impact means Log4Shell's blast radius is enormous.

Over the last few days, exploitation attempts of various Log4j vulnerabilities have increased exponentially, emphasizing the need to identify and patch vulnerable systems as fast as possible. Open source maintainers, information security teams, and ops teams are working around the clock to contain the event, but there is still a lot of work to be done and the longer systems remain unpatched, the likelihood of exploitation rises.

Exactly what are some of the challenges around effective detection of vulnerable Log4j components? The most concerning is that without proper scanning, you still have blindspots in the form of undetected/unpatched vulnerable components. In other words, your scanner can give you a false sense of security about whether or not you are exposed.

We will address this detection challenge in two separate scopes:

1. **Production Environment** — source code is usually unavailable and detection has to occur at the binary/process level
2. **Development, continuous integration/continuous delivery (CI/CD), and staging environments** — detection will usually be based on application source code

Log4Shell Within a Production Environment

IN PRODUCTION, vulnerable artifacts will likely be present in their binary form. Therefore, the source code will not be available for analysis. In the Log4Shell scenario, we are talking about Java Archive (JAR) files.

Before we dive into the challenges in effectively identifying vulnerable components in production let's first understand the JAR format.

A JAR, is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution.

JAR files are built on the ZIP format and as such can be viewed as a compressed archive file. So much so that you can rename a JAR file to end with ".zip", double-click on it, and see its content. Simply put, a JAR file is a file that contains a compressed version of .class files, audio files, image files, or directories.

What are the Limitations of Binary Based Identification?

The flexibility of the JAR format allows for JAR files to take many shapes and forms.

They can have several levels of hierarchy (a JAR, within a JAR, within a JAR, etc.) and have different extensions. Each type usually reflects a different packaging and deployment schema and can have specific requirements.

Common examples include:

- **TAR —** Tape Archive (uncompressed)
- **WAR —** Web Application Archive. Should contain a web.xml file in the WEB-INF folder
- **EAR —** Enterprise Application Archive
- **SAR —** Service Application Archive
- **PAR —** Portlet Archive
- **RAR —** Resource Adapter. A system-level driver that connects a Java application to an enterprise information system. Contains source code and an ra.xml that serves as a deployment descriptor
- **KAR —** Apache Karaf Archive. A special type of artifact that package a features XML and all resources described in the features of this XML (bundle JARs)
- **Uber JAR/Fat JAR —** a JAR that contains a Java program as well as all of its dependencies. The JAR serves as an all-in-one distribution of the software, without needing any other Java code. It can be shaded, unshaded, or nested
    - **Unshaded JAR —** Unpack all JAR files, then repack them into a single JAR
    - **Shaded JAR —** Same as unshaded, but rename (i.e., "shade") all packages of all dependencies
    - **Nested JAR —** A JAR file that contains all other required JAR files (can have several levels of hierarchy)

Going back to our Log4j identification efforts, what this means is that the vulnerable component can be located in any one of these different packaging formats and in varying levels of nesting. **Can you be sure that the tools you use to locate the Log4j vulnerable components take that into account?**

We have conducted a comparison between various open source tools and three commercially-available scanners, all aiming at detecting the recent Log4j vulnerabilities.

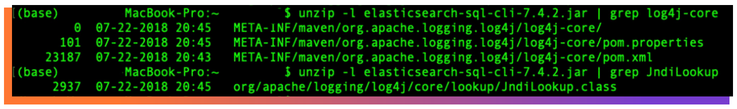
In our analysis, we compared the results of the following detectors:

- *ind`* command searching for vulnerable files based on file name matching. This was one of the earliest detection methods recommended on Twitter
- anchore/grype
- palantir/log4j-sniffer
- Trivy
- mergebase/log4j-detector
- OWASP Dependency-Check
- Jfrog/log4j-tools
- Three Leading, Commercial Web Application Scanners

All open source tools were measured against the same baseline. Each scanner was used to analyze one of 20 sample binaries provided in one of these open source projects and one example from an open source ElasticSearch container.

Effective identification of the Log4Shell vulnerability (CVE-2021-44228) was based on the existence of the *JndiLookup* class in Java binaries which contain Log4j-core code below version 2.17. In our research, we attempted to simulate the variety of tactics used to package Java applications and deploy them to production, analyzing each of the scanning tools' ability to detect vulnerable Log4j instances in multiple types of Java binaries with a range
of file extensions.

This is significant to note, as we know that vulnerable components can be located in any one of these different packaging formats and in varying levels of nesting. For example, in the ElasticSearch binary detection evaluation, the JAR name we examined was *elastic-search-sql-cli-7.4.2.jar,* a filename that does not meet the basic identification pattern used by many scanners to detect Log4j (e.g.: log4j-core-<version>.jar), yet it still contained vulnerable pieces of Log4j code.



```
(base)        MacBook-Pro:~        $ unzip -l elasticsearch-sql-cli-7.4.2.jar | grep log4j-core
        0  07-22-2018 20:45   META-INF/maven/org.apache.logging.log4j/log4j-core/
      101  07-22-2018 20:45   META-INF/maven/org.apache.logging.log4j/log4j-core/pom.properties
    23187  07-22-2018 20:43   META-INF/maven/org.apache.logging.log4j/log4j-core/pom.xml
(base)        MacBook-Pro:~        $ unzip -l elasticsearch-sql-cli-7.4.2.jar | grep JndiLookup
     2937  07-22-2018 20:45   org/apache/logging/log4j/core/lookup/JndiLookup.class
```

As can be seen in the table on page one, there are quite a few cases that are missed by the various scanners. Specifically, compressed archives as well as Uber JARs (which are known to present difficulties to SCA solutions) were not effectively detected by any of the tools used in our evaluation.

Analysis results are based on scans conducted on December 20, 2021. It is possible that some of the tools examined have since enhanced their detection capabilities.

The three leading commercial web application scanners were also included in our assessment. These scanners were evaluated based on inferred capabilities according to each tool's publicly-documented methodology and open source components specifically aimed at detecting Log4Shell vulnerabilities.

According to our research, commercial scanners were also susceptible to false negatives as they often rely on files names in order to detect vulnerable components and fail to address most of the above-mentioned edge cases. Details of the findings of our analysis of these tools are as follows.

Scanner #1

DETECTION METHODOLOGY: Looks for files matching: `log4j-core.*.jar`.

RESULT: Gaps identified.

This scanner will miss both vulnerable JAR files that don't contain log4j-core (e.g.: "elasticsearch-sql-cli-7.4.2.jar") as
well as other Java binary types detailed above.

Scanner #2

DETECTION METHODOLOGY: Scans for files ending in *.jar and/or JAR files that are nested within ZIP files.

RESULT: Gaps identified.

Scanner has separate detection scripts for both Linux and Windows environments.
The Linux based detection script will miss vulnerable components inside formats such as WAR, EAR, PAR, RAR, etc. In addition, the Linux script does not support various file systems such as Network File System (NFS), Andrew File System (AFS), and more, making this tool's detection capability ineffective for certain environments (Solaris for example, which is usually NFS based).

This scanner's Windows-based detection script does a slightly better job, as it addresses WAR and EAR files. Per above, however, there are still gaps in the tool's ability to detect vulnerable instances in other relevant file types such as PAR, KAR, SAR, etc.

Scanner #3

DETECTION METHODOLOGY: Checks for vulnerable versions of Log4j based on installed RPM packages and searches file system paths for known vulnerable Log4j matches.

RESULT: Gaps identified.

This tool will miss any vulnerable JAR whose name doesn't contain Log4j (for example: elasticsearch-sql-cli-7.4.2.jar).

It can potentially miss vulnerable components installed by other package managers. Mainly Maven, which will usually be used to install Log4j as well as libraries which call Log4j as a dependency.

Additionally, this scanner will miss vulnerable Log4j components that are nested within Java binary formats other than JAR (detailed above).

Log4Shell within a Development, CI/CD and Staging Environment

Although detection abilities in these types of environments are better due to the fact that source code is available, there are still a few detection challenges worth noting.

Let's first explain two basic concepts:

1. Direct Dependencies are simply libraries/packages that your code is calling
2. Indirect/Transitive Dependencies are libraries/packages that the direct dependencies are calling

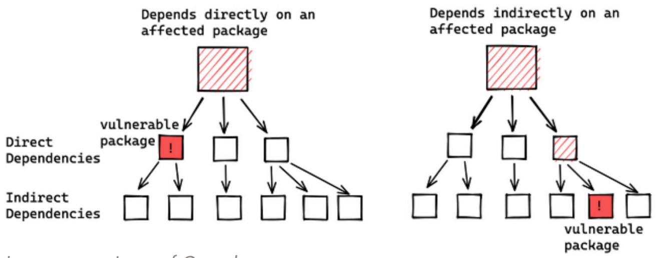

Depends directly on an affected package
vulnerable package
Direct Dependencies
Indirect Dependencies
Depends indirectly on an affected package
vulnerable package

Image courtesy of Google

Currently, according to Google Security Blog, nearly 36,0000 of the available Java artifacts from Maven Central depend on affected Log4j components:

"This means more than 8% of all packages on Maven Central have at least one version that is impacted by this vulnerability. As far as ecosystem impact goes, 8% is enormous. The average ecosystem impact of advisories affecting Maven Central is 2%"

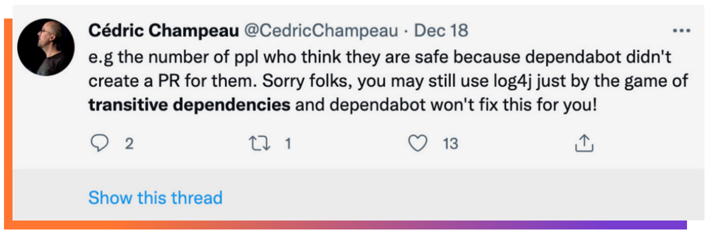
What is even more interesting is that the majority of these affected artifacts stem from indirect (transitive) dependencies.

That means that in most cases, Log4j is not explicitly defined as a dependency of the artifact, but gets pulled in as a transitive dependency. Furthermore, for more than 80% of the packages, the vulnerability is deeper than one level down, with a majority affected five levels down (for some packages even as many as nine levels down).

Rick Hunter 🚀 @CarlMarsalis · Dec 16
I know what my personal hell will look like and it will be solving **transitive dependencies** for all eternity.
The struggle is real 🥵 #Programming

Why are Transitive Dependencies an Issue?

Transitive dependencies are an issue because not all tools are equipped to handle them effectively. To avoid these challenges—and to avoid costly delays or inaccuracies in remediating Log4j vulnerabilities—you first need to make sure that your scanner can account for transitive dependencies within your codebase. While there are many tools available that can do this with accuracy, such as Mavens' d*ependency:tree,* not all scanners live up to their promises. Be sure to educate yourself on how to properly evaluate these solutions before making a selection.


Cédric Champeau @CedricChampeau · Dec 18
e.g the number of ppl who think they are safe because dependabot didn't create a PR for them. Sorry folks, you may still use log4j just by the game of **transitive dependencies** and dependabot won't fix this for you!

Transitive dependencies are particularly challenging when working with open source components. Aside from the obvious detection issues, which are potentially more complex and/or more prevalent in open source code, remediation of known vulnerabilities will only be possible once a package maintainer makes a patch available. Patches can be notoriously slow to roll out, since all third party libraries will need to first patch their code and release up-to-date fixed versions. In these scenarios, whenever possible, consider applying mitigations or have some form of compensating control in place in order to manage immediate risk.

In the case of Log4Shell, one possible mitigation could be explicitly deleting the vulnerable class (JndiLookup) from every vulnerable Log4j component in your environment by running a command like:

```
`zip -q -d log4j-core-*.jar org/apache/logging/log4j/core/lookup/JndiLookup.class`
```

On the other hand, there is also a strong possibility that some of these transitive dependencies are not being used by the application and thus are not actually vulnerable. As reflected in this quote from a study conducted by SAP in 2018:

> *"From our analysis of a sample of over 550 OSS libraries used by SAP projects,* ***as many as 20% of all the dependencies are nondeployed****. Any metrics reporting the "danger" of using OSS libraries that do not discriminate between those two classes would* ***lead to a wrong allocation of costly development and audit resources****."*

By validating the exploitability of detected vulnerabilities through establishing their presence in runtime memory, organizations can more efficiently filter out false positives, prioritize remediation efforts and reduce overall time to patch.

What Should You Take Away From This to Minimize Risk from Log4Shell?

Remember that your detection abilities are only as good as your detection method. As we've seen, scanners have blindspots. Security leaders cannot blindly assume that various open source or even commercial-grade tools will be able to detect every edge case.

In order to fully understand the risk, and be able to effectively mitigate it, it is imperative to understand these limitations. Consider what might be absent from your scanning tools' results, as they all have blindspots. Ensure that your scanner takes transitive dependencies into account when you scan your codebase for vulnerable components. And lastly, complement static scanning with code-level detection in runtime memory, where the code isn't packaged or nested.

It is evident: one tool alone cannot do it all when it comes to Log4Shell. We need complementary technologies to best tackle this threat.