

LOG4SHELL 4 MONTHS LATER:

LogShell 4 Months Later: Are You Still Vulnerable?

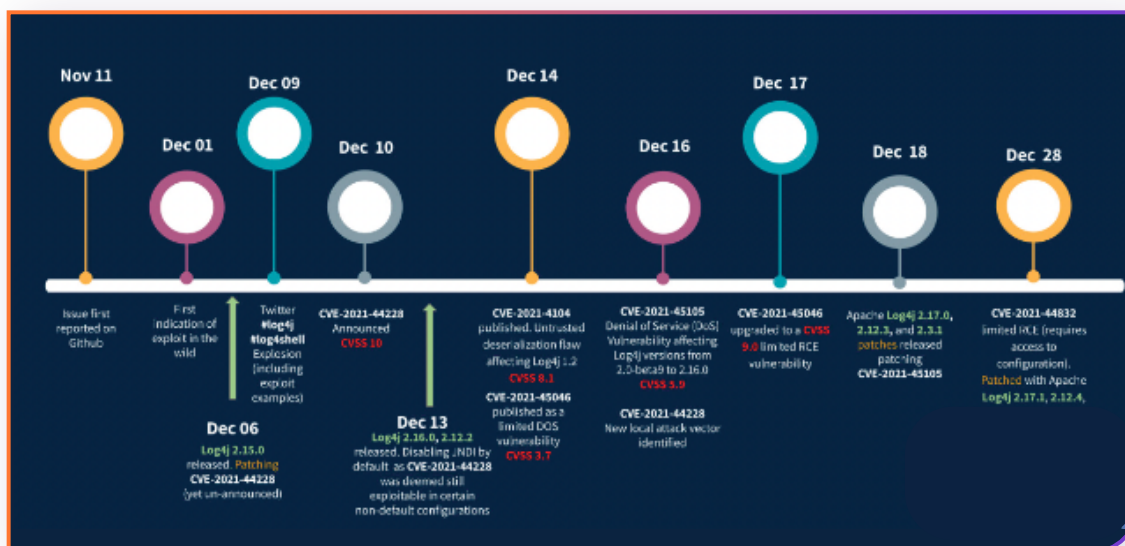


BY YOTAM PERKAL, HEAD OF VULNERABILITY RESEARCH

It has been over 4 months since the first publication of Log4Shell (CVE-2021-44228), one of the most critical vulnerabilities in recent years, due to a deadly trifecta of factors:

1. **Huge attack surface** — Millions of vulnerable Java applications worldwide according to most estimates.
2. **Ease of exploitation** — No privileged access or special configuration is needed, all an attacker has to do in order to exploit the vulnerability is to locate an input field that gets logged and write a simple string.
3. **Severe potential impact** — With a CVSS v3 score of 10, a successful exploit of the Log4Shell vulnerability will provide an attacker the ability to execute arbitrary code and potentially take full control of the system.

Log4Shell Timeline



Rezilion researchers decided to assess the current potential attack surface of this vulnerability today, 4 months later, after the dust settled. Due to the massive amount of media coverage that the Log4Shell vulnerability has received, we hoped that the majority of applications would already be patched. We assumed finding services that are still vulnerable would be challenging.

Unfortunately, we were wrong. We learned that the landscape is far from ideal and many applications vulnerable to Log4Shell still exist in the wild.

In this report, we will examine our findings and discuss the need for widespread industry support in the continued battle to minimize future potential exploitation.

A Big Hill to Climb

Before we dive into the actual research there are few data points that emphasize how much work is yet to be done.

The first is from [Open Source Insights](#), a Google service which scans millions of open source packages, builds their dependency graphs, and annotates them with information about ownership, licenses, popularity, and other metadata. When exploring the components affected by the Log4Shell vulnerability, i.e components using `org.apache.logging.log4j:log4j-core`, it appears that out of a total of 17.84K affected packages, only 7.14K are patched for Log4Shell. **This means that almost 60% of vulnerable packages are not yet patched!**

Remote code injection in Log4j

Overview		Summary		
Source	GHSA	17.84k TOTAL PACKAGES AFFECTED ⓘ	7.14k PACKAGES WITH A KNOWN FIX ⓘ	3.89% TOTAL ECOSYSTEM AFFECTED ⓘ
ID	GHSA-jfh8-c2jp-5v3q			
Aliases	CVE-2021-44228			
Affected package	org.apache.logging.log4j:log4j-core			

SOURCE: <https://deps.dev/advisory/GHSA/GHSA-jfh8-c2jp-5v3q>, April 20, 2022

At this point you might be thinking, "Ok, but perhaps the majority of these unpatched packages are simply not being used (old, esoteric, or unpopular packages) and therefore do not pose an actual risk." Let's explore that hypothesis.

Sonatype maintains a [Log4j vulnerability resource center](#) which tracks the amount of downloads of Log4j and provides statistics based on version, geographic location, and more.

As you can see in the below image, as of April 20, 2022, **36%** of the Log4j versions **actively**

Log4j Download Dashboard

SOURCE: IBM AND THE PONEMON INSTITUTE

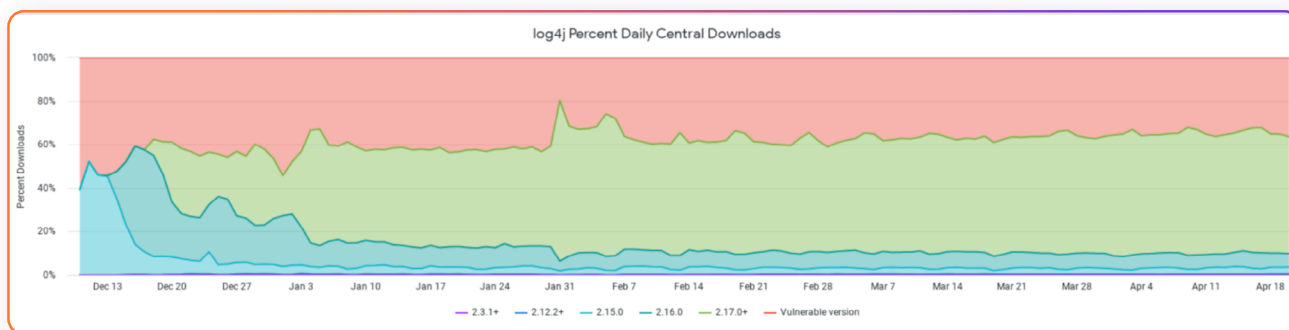
LOG4J LATEST STATISTICS

45,169,939

Total Downloads Since Dec. 10, 2021
39% vulnerable

36%

Vulnerable Downloads Last 24 Hours
412,686 total downloads



The red area marks the percentage of Log4j downloads vulnerable to Log4Shell and it has pretty much plateaued since early February.

So what can explain these concerning statistics? What are the services that are still pulling these vulnerable Log4j versions?

Two months after the initial publication, prior to the Senate hearing about the topic, industry and open source representatives have [attempted to explain](#) the cause for this high percentage of vulnerable components downloads. Potential reasons include:

- **Downloads by security researchers trying to analyze the vulnerability.** In my opinion this doesn't account for such a high percentage of continuous downloads, especially at this point in time when public interest has subsided.
- **Instances where Log4j was deployed in applications or networks that aren't accessible from the internet, so they were less of a priority to fix.** This is a misconception as it has been shown that even systems that are not directly exposed to the internet can be vulnerable in certain scenarios. See for example [this](#) blog post from Akamai which states:

"Note, however, that data centers where all Java servers are internal (namely, not internet-facing) cannot be considered safe. While Log4Shell has been mainly perceived as a means to breach networks, some cases showed how Java applications running on internal servers received logs from internet-facing servers and ended up being compromised. Log4Shell can thus be used as easily for lateral movement as for initial breach."

SOURCE: <https://www.sonatype.com/resources/log4j-vulnerability-resource-center>, April 20, 2022

We believe that one of the main reasons we still see a high number of vulnerable component downloads is the fact that people are unknowingly still using software that relies on vulnerable versions of Log4j.

This could be attributed to several factors:

- Many organizations lack mature vulnerability management processes and/or visibility into their software components (Software Bill of Materials, known as SBOM).
- Log4j has proven to be challenging to detect in production environments.
- Use of vulnerable third party software.

Validating Our Assumption: Log4Shell in Open Source Software

One of the reasons patching Log4j is challenging is because detecting it isn't always straightforward. 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, Java applications can be packaged in many different ways which creates a real challenge for tools trying to analyze them as they need to support each and every creative (yet possible) packaging format. That creates challenges for tools aiming to detect vulnerable Log4j components as we have highlighted in a [previous research](#).



The problem doesn't end there. Even if you are able to detect and patch Log4j in proprietary applications, when it comes to third party software (whether commercial or open source), detection isn't enough. Even if you are able to detect a vulnerable third party software, you will need to wait for the vendor to release an updated patched version of the software.

As an organization, there are two scenarios in which you will consume such vulnerable third party software:

1. You are using the latest version of the software, yet the vendor hasn't issued a patched release.
2. The vendor has issued a patched release, yet you are still using an outdated version. In the context of a containerized environment for example, this can happen by design if you use version pinning — explicitly setting the version of the container being pulled. The main advantage to this approach is stability and reproducibility of the deployed software, though when it comes to patching, it cements the use of the vulnerable version of the container and requires active action in order to upgrade as opposed to using the 'latest' tag. In this case, once a new version is released, it will pull the updated version with no human interaction.

Methodology

During the first weeks since the publication of Log4Shell several lists of potential vulnerable applications were compiled and shared. See, for example, [this list](#) maintained by the Dutch National Cyber Security Center (NCSC) or [this repository](#) maintained by CISA. Our approach was to try to examine the latest versions of public containers of these applications and see whether the version of Log4j in these containers is indeed up-to-date. Using [dive](#), an open source tool for exploring docker container images, we have verified exactly what version of Log4j is present in each of the containers.

In the second phase, for each container image still containing vulnerable Log4j versions we used [Shodan.io](#), a popular search engine for internet-connected devices, in order to see how many vulnerable applications are exposed to the internet.

For that purpose, we ran the containers and identified distinct characteristics of the application banner and/or the HTTP headers. We applied those to a Shodan.io search for some estimation of the commonality of these vulnerable applications.

Keep in mind that the majority of this research focuses on a **specific aspect** of the potential current attack surface for Log4Shell, servers running open source software. We must assume that there are also proprietary applications as well as commercial products still running vulnerable versions of Log4j. We will demonstrate that risk by analyzing publicly facing Minecraft servers.

Note: Actual exploitability of the potentially vulnerable servers was not tested. The results presented below are based on data from Shodan.io which attempts to associate vulnerabilities with services based on the software version assuming it is present in the service metadata (see [here](#)).

Results

Overall, we identified over 90,000 potential vulnerable internet facing applications, and believe that is just the tip of the iceberg in terms of the actual vulnerable attack surface.

The results are divided to 3 categories:

1. Containers that in their latest version, still contain obsolete versions of Log4j.
2. Containers which their latest version is up-to-date yet there is still evidence of previous versions usage.
3. Publicly facing Minecraft servers, which highlight the risks with outdated proprietary software.

For each container, we noted the total number of high and critical vulnerabilities in the container as reported by Gripe, an open source vulnerability scanner.

Note: The list of container images highlighted in this research is far from exhaustive as we focused on actively maintained, popular container images for applications. There are of course hundreds of obsolete container images on DockerHub which still contain vulnerable versions of Log4j, yet are out of scope for this research.

1. Containers that in their latest version, still contain obsolete versions of Log4j

Container Name	Log4j version	Last Updated	Total Shodan results	Total/High/Crit Vulnerabilities	# Vulnerabilities from Prior to 2020
Apache druid	Log4j-core-2.15.0.jar **	Dec 11, 2021	149	365/81/26	147/365 (40%)
Apache Solr	Log4j-core-2.16.0.jar ***	Mar 30, 2022	1657	93/14/9	46/93 (49%)
Apache Ozone	log4j-core-2.16.0.jar ***	Jan 23, 2022	115	750/41/6	592/750 (78%)

** Vulnerable to [CVE-2021-45046](#)

*** Vulnerable to [CVE-2021-45105](#)

In addition to the Log4j related vulnerabilities, all containers also have a significant number of additional vulnerabilities even though they are the 'latest' versions of their respective docker images.

On average, more than 55% of the vulnerabilities detected in these container images were published prior to 2020, yet are still alive and kicking.

2. Containers which their latest version is up-to-date yet there are still evidence of previous versions usage

Container Name	Log4j version prior to patch	Patched Version Release Date	Time to Patch	Total Shodan results
Apache Storm	Log4j-core-2.11.2.jar *	April 1, 2022	101 days	1309
Elasticsearch	Log4j-core-2.11.1.jar *	Dec. 21, 2021	11 days	21417
Apache skywalking-oap	Log4j-core-2.14.1.jar *	April 9, 2022	109 days	101
WSO2 API Manager	Log4j-core-2.13.3.jar*	March 28, 2022	97 Days	3309

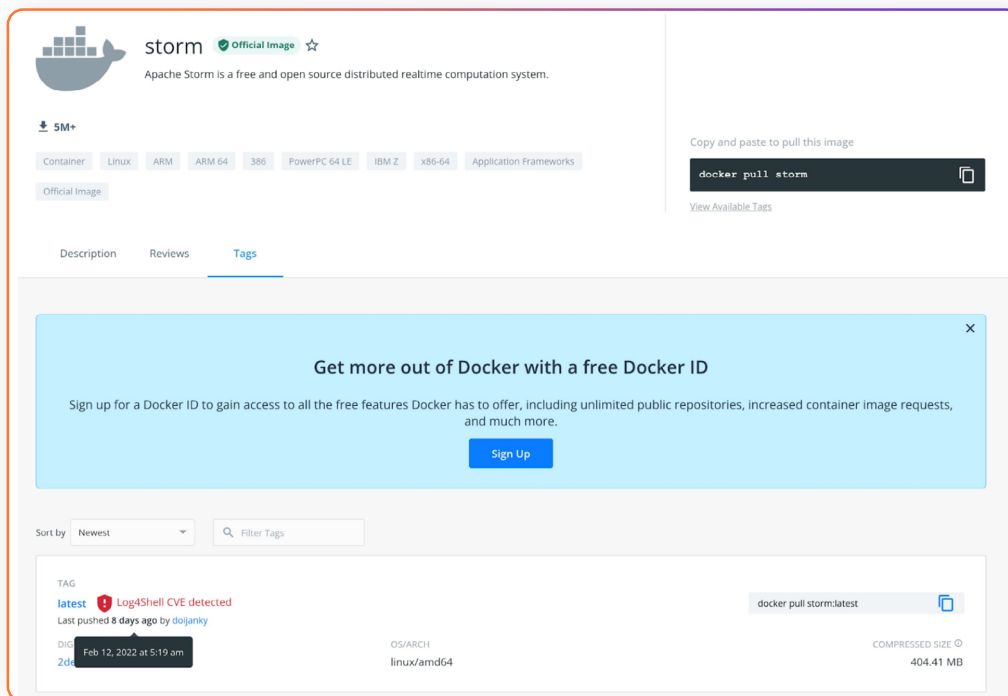
* Vulnerable to [CVE-2021-44228](#)

For the reviewed containers the average time it took for a patched container image to be published to DockerHub is almost **80** days!

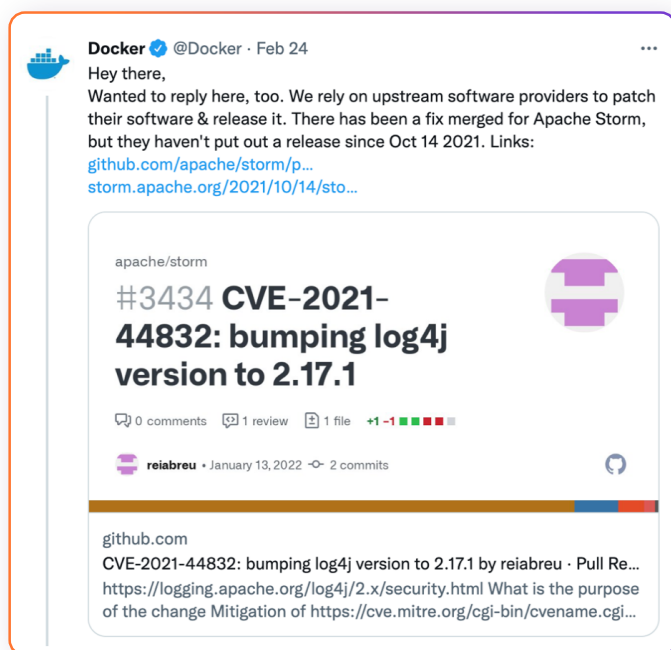
During that time period, anyone pulling the latest version of these containers to their environment would have effectively pulled and ran services that are vulnerable to Log4Shell.

Let's take the Apache Storm example to understand why it took so long for an updated docker image to be published.

While conducting research on a different topic in mid-February 2022, I noticed that the Apache Storm official image was still marked as vulnerable to Log4Shell.



I [reached out to Docker](#) and was told that they rely on upstream releases in order to publish updated official images:



And in fact, up until March 25, 2022 no official Storm release was issued:

Latest News

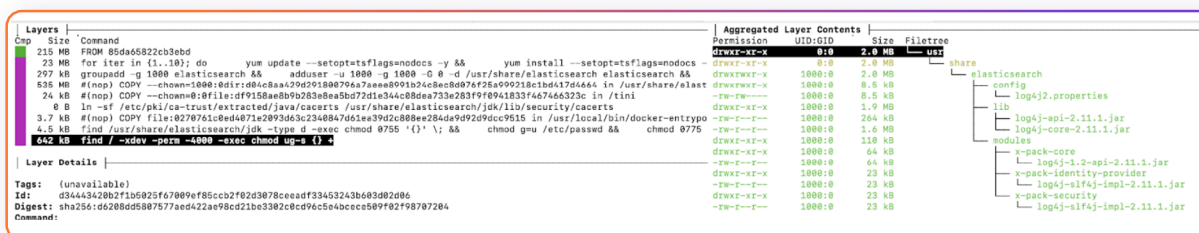
- ❑ [Apache Storm 2.4.0 Released](#) (25 Mar 2022)
- ❑ [Apache Storm 2.1.1 Released](#) (14 Oct 2021)
- ❑ [Apache Storm 2.2.1 Released](#) (11 Oct 2021)
- ❑ [Apache Storm 1.2.4 Released](#) (11 Oct 2021)
- ❑ [Apache Storm 2.3.0 Released](#) (27 Sep 2021)
- ❑ [Apache Storm 2.2.0 Released](#) (30 Jun 2020)
- ❑ [Apache Storm 2.1.0 Released](#) (31 Oct 2019)
- ❑ [Apache Storm 1.2.3 Released](#) (18 Jul 2019)

[More News](#)

This led to the fact that for over 100 days, every Apache Storm consumer using their containerized application had remained vulnerable to Log4Shell.

But let's take the case of Elasticsearch for which the patched image took 11 days to be published to DockerHub. Now the ball is in the consumers hands to update their software and, as we reveal here, that might be an unrealistic expectation.

According to Shodan.io, there are over 25.6k Elasticsearch servers publicly accessible. Since for Elasticsearch, Shodan provides accurate version information we were able to determine exactly how many servers contain vulnerable versions of Log4j. We excluded all [non vulnerable versions of Elasticsearch](#) from the search results (only took into account all 5.x versions, from 6.x.x to 6.8.21, and from 7.0.0 to 7.15.x) and verified using Dive that the vulnerable versions, in fact, contain the affected Log4j-core library:



Layers

Size	Command
215 MB	FROM 85da6822cb3ebd
23 MB	for iter in {1..10}; do
297 kB	groupadd -g 1000 elasticsearch && adduser -u 1000 -p 1000 -s /bin/bash elasticsearch &&
535 MB	#(nop) COPY --chown=1000:0d1r:d0c8aa42d20180079aa7aeae8091b34c8e8d076f25e99218c1bd417d4644 in /usr/share/elast
24 kB	#(nop) COPY --chown=0:0file:d9f158ae8b7b283e8a5b72d1e344c08dea733e283f9f8941833f467466323c in /tmp
0 B	ln -sf /etc/pki/ca-trust/extracted/java/cacerts /usr/share/elasticsearch/jdk/lib/security/cacerts
3.7 kB	#(nop) COPY file:0278761c8d4d071e2073d63c23468a7d61ee39d2c88ae25d4d992d9d9c9515 in /usr/local/bin/docker-entryp
4.5 kB	find /usr/share/elasticsearch/jdk -type d -exec chmod 0755 '{}' \; && chmod g+u /etc/passwd && chmod 0775
642 kB	find / -xdev -perm -4000 -exec chmod ug-s {} \;

Aggregated Layer Contents

Permission	UID:GID	Size	Filetree
drwxr-xr-x	0:0	2.0 MB	usr
drwxr-xr-x	0:0	2.0 MB	share
drwxr-xr-x	1000:0	2.0 MB	elasticsearch
drwxr-xr-x	1000:0	8.5 kB	config
-rw-rw-r--	1000:0	8.5 kB	log4j2.properties
drwxr-xr-x	1000:0	1.9 MB	lib
-rw-rw-r--	1000:0	264 kB	log4j-api-2.11.1.jar
-rw-rw-r--	1000:0	1.6 MB	log4j-core-2.11.1.jar
drwxr-xr-x	1000:0	110 kB	modules
drwxr-xr-x	1000:0	64 kB	x-pack-core
-rw-rw-r--	1000:0	64 kB	log4j-1.2-api-2.11.1.jar
drwxr-xr-x	1000:0	23 kB	x-pack-identity-provider
-rw-rw-r--	1000:0	23 kB	log4j-slf4j-impl-2.11.1.jar
drwxr-xr-x	1000:0	23 kB	x-pack-security
-rw-rw-r--	1000:0	23 kB	log4j-slf4j-impl-2.11.1.jar

Layer Details

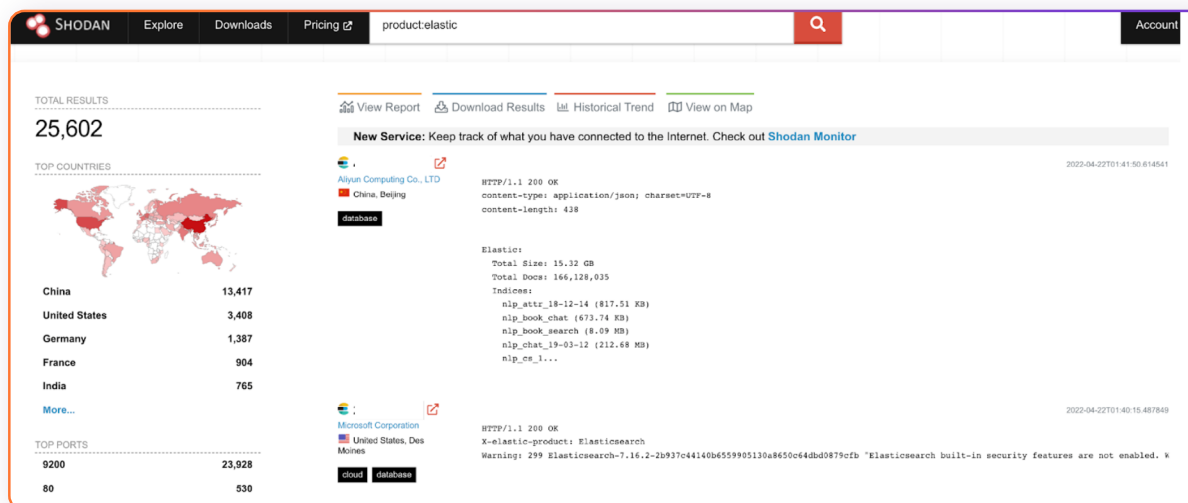
Tags: (unavailable)

Id: d34443428b2f1b56925f67009ef85ccb2f62d3078ceeadf33453243b603d82d06

Digest: sha256:d6288d5807577aed422ae98cd21be3302c8cd96c5e4bccc509f02f98707204

Command:

We were left with **21417** Elasticsearch servers which are still vulnerable to Log4Shell. That's almost 83%.



SHODAN Explore Downloads Pricing product:elastic Account

TOTAL RESULTS
25,602

TOP COUNTRIES

Country	Count
China	13,417
United States	3,408
Germany	1,387
France	904
India	765

TOP PORTS

Port	Count
9200	23,928
80	530

New Service: Keep track of what you have connected to the Internet. Check out [Shodan Monitor](#)

Aliyun Computing Co., LTD
China, Beijing

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 438

Elastic:
Total Size: 15.32 GB
Total Docs: 166,128,035
Indices:
nlp_attr_18-12-14 (817.51 KB)
nlp_book_chat (673.74 KB)
nlp_book_search (8.09 MB)
nlp_chat_19-03-12 (212.68 MB)
nlp_cs_1...

Warning: 299 Elasticsearch-7.16.2-2b937c44140b6559905130a8650c64dbd0879c7b "Elasticsearch built-in security features are not enabled."

There are also a lot of applications that are still using Log4j version 1.x ([this source](#) says even up to 10 times more), and falsely assume that that means they do not need to patch as the original Log4Shell vulnerability (CVE-2021-44228) does not apply for Log4j 1.x.

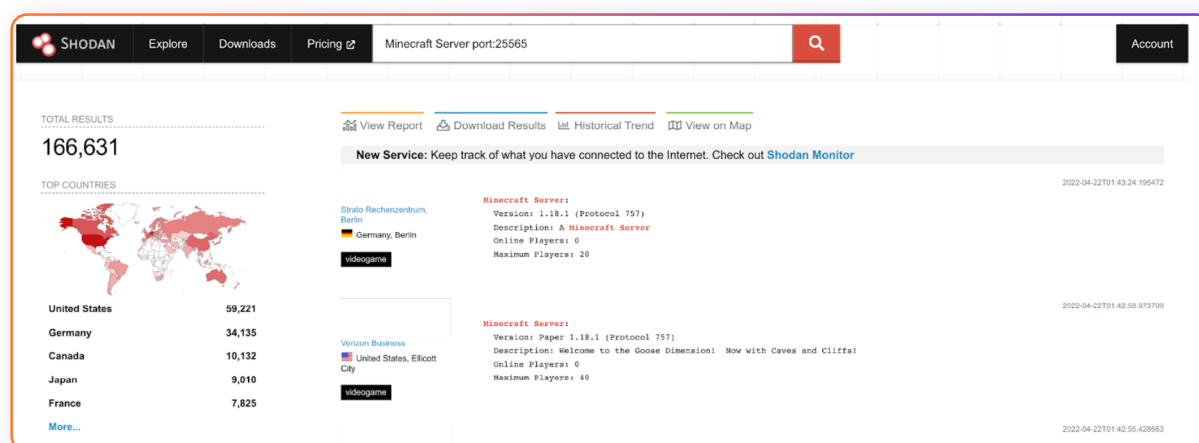
This is a misconception as version 1.x has been in an **end-of-life state since August 2015** (which means it does not get any security updates), and contains plenty of other vulnerabilities, including RCE vulnerabilities (trickier to exploit, yet still valid) such as [CVE-2021-4104](#). This should definitely worry organizations that are still using it. Also see this [stack-overflow question](#) that implies this misunderstanding.

Examples for applications still using Log4j 1.x include:

Container Name	Log4j version	# DockerHub Downloads	Total/High/Crit Vulnerabilities	# Vulnerabilities from Prior to 2020
Bitnami Spark	log4j-1.2.17.jar	Apr 20, 2021	141/43/11	75/141 (53%)
Bitnami Kafka	log4j-1.2.17.jar	Apr 20, 2022	92/16/7	50/92 (54%)
Apache zeppelin	log4j-1.2.17.jar	Feb 28, 2022	249/89/49	108/249 (43%)
Atlassian Crucible	log4j-1.2.17.jar	March 13, 2022	365/81/26	147/365 (40%)

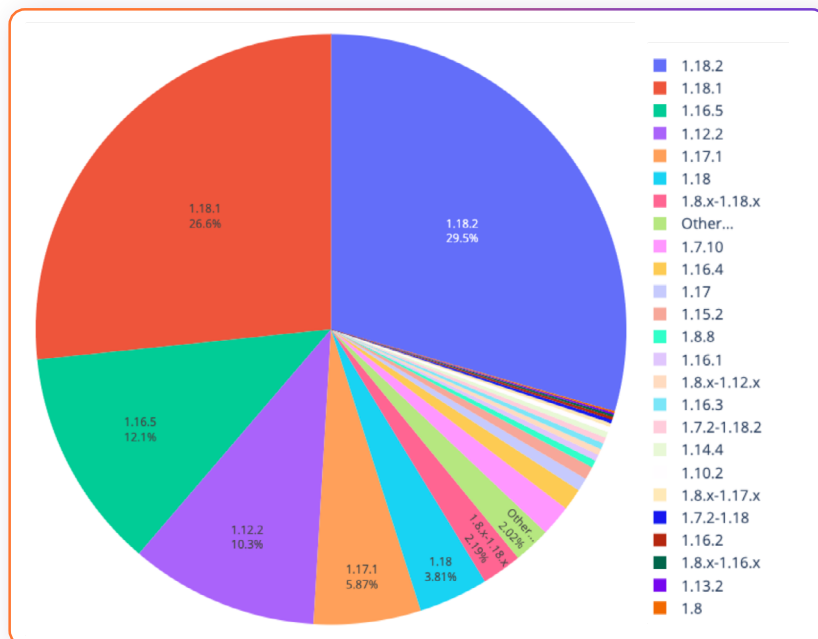
3. Publicly Facing Minecraft Servers

Using Shodan.io we have identified over 166,000 publicly facing Minecraft servers:



These numbers align with [independent research](#) recently conducted by Adrian Zhang. According to [Minecrafts' official security advisory](#), the vulnerable versions for Log4Shell are between 1.7 and 1.18.

This means that over 68,000 potentially vulnerable servers are still publicly exposed:



There is a possibility that some of these servers' maintainers have applied the mitigations which are recommended by Minecraft. These mitigations do decrease the risk of exploitation, but they are by no means a comprehensive defense as they rely on disabling `MsgLookups`, which is stated in several [blog posts](#) as well as in the official [Log4j documentation](#):

"This page previously mentioned other mitigation measures, but we discovered that these measures only limit exposure while leaving some attack vectors open.

Other insufficient mitigation measures are: setting system property `log4j2.formatMsgNoLookups` or environment variable `LOG4J_FORMAT_MSG_NO_LOOKUPS` to `true` for releases ≥ 2.10 , or modifying the logging configuration to disable message lookups with `%m{nolookups}`, `%msg{nolookups}` or `%message{nolookups}` for releases ≥ 2.7 and $\leq 2.14.1$.

The reason these measures are insufficient is that, in addition to the Thread Context attack vector mentioned above, there are still code paths in Log4j where message lookups could occur: known examples are applications that use `Logger.printf("%s", userInput)`, or applications that use a custom message factory, where the resulting messages do not implement `StringBuilderFormattable`. There may be other attack vectors.

The safest thing to do is to upgrade Log4j to a safe version, or remove the `JndiLookup` class from the `log4j-core` jar."



Exploitation

We have established the fact that there is still a significant attack surface vulnerable to Log4Shell even 4 months after its initial publication. But should we worry? Are there actually any active exploitation attempts? The answer is yes.

Active exploitation attempts were discovered very early on (even [days prior to the official publication](#) of the vulnerability). Initial exploitation attempts seen in the wild were aimed at deploying ransomware and various coin miners, but as time went on evidence of active exploitation of Log4Shell by various APT groups [started to accumulate](#).

Examples include:

- The [Chinese state-sponsored espionage group HAFNIUM](#)
- Iranian-backed groups [APT35](#) (aka Newscaster) and [Tunnel Vision](#)

More recently, active exploitation was also tied to the Chinese [APT 41 group](#) and [Deep Panda](#). The [SANS Internet Storm Center](#) has created a honeypot for the detection of exploitation attempts. To this date, there are still dozens of recorded daily exploitation attempts. See:

<https://isc.sans.edu/api/webhoneypotreportsbyurl/jndi:/2022-04-20?json>

Historical Precedents

What can we learn from past vulnerabilities' exposure on the future of Log4Shell? There are quite a few historical precedents of vulnerabilities that were discovered years ago, yet are still being exploited to this day.

[ShellShock](#), a family of security vulnerabilities affecting the Unix bash shell that was discovered in September 2014 and allowed an attacker to achieve arbitrary code execution was identified almost 6 years later "in the wild" as there was still [active malware attempting to exploit it](#). A notorious 2008 vulnerability in Microsoft Server Service ([MS08-67](#)) has been known to be exploited by the [Conficker Worm](#) reaching an estimated peak of around 9 million infected devices.

A report from Trend Micro in 2017 shows that even **10 years after its initial release** about 300,000 infections occurred worldwide. In 2020, Palo-Alto estimated that the number of infected devices is around 500,000.

Another prominent example of a vulnerability that refuses to go away is [HeartBleed](#). [CVE-2014-0160](#), made public in April 2014, affects the heartbeat extension (RFC 6520) implemented in OpenSSL 1.0.1 through 1.0.1f. It can result in the leak of memory contents from the server to the client and from the client to the server when exploited. Effectively, allowing anyone on the internet to read the memory of the systems using the vulnerable versions of the OpenSSL software.

The fact that exploitation of the vulnerability isn't very complex and leaves no visible trace on the attacked machine, made the HeartBleed vulnerability one of the most impactful vulnerabilities in recent years.

Eight years after the initial publication, according to Shodan.io, the number of potentially vulnerable internet facing applications is over **61,000**.



Looking Ahead...

As we have seen Log4Shell is still here, and as history suggests, that might be the case for a while. In this section we will try to give a few actionable recommendations you can take in order to better protect your environment as well as raise some potential methods of operation that we, as an industry, can adopt in order to minimize the time and scope of exposure for future vulnerabilities such as Log4Shell.

Recommended Immediate Steps

- It is important to continuously scan your environment. Scanning and patching vulnerable applications periodically does not ensure your environment is clear of vulnerabilities. You need to have processes in place that continuously monitor your environment for critical vulnerabilities with an emphasis on third-party code.
- If you don't have a vulnerability management function in your organization there are quite a few open source tools that you can integrate into your pipeline that will give you insight to your potential attack surface. Keep in mind that these tools have [their limitations](#). It is important to be aware of what the tool you chose can and can't do.
- In the event you do find vulnerable assets, especially given the amount of time elapsed, assume compromise, try to identify common post-exploit activity, and hunt for signs of malicious activity.

On a Broader Level

Given the fact that Log4j is so widespread yet not always easy to detect, especially in a production setting, it is evident that maintaining a Software Bill of Materials (SBOM) is a significant first step in order to get visibility of potentially vulnerable applications. However, what is also clear, is that an SBOM (as it is perceived today) is only a first step. In order for security teams to be able to effectively track software components across the supply chain, and use the SBOM as a tool for triaging Log4Shell like vulnerabilities (reducing time-to-detect and time-to-patch), an SBOM must be:

- **Machine Readable** in order to allow for automation. Some positive strides are being made with standards such as [SPDX and CycloneDX](#).
- **Contain Security Context**, an ingredient list of the software is not enough. Ideally, an SBOM should be able to report on things like: if the package is actively maintained, if it is in an end-of-life status, if it has a critical vulnerability, if it is exploitable in the context of the environment in which it is running etc. Adoption of standards such as Vex (Vulnerability-Exploitability eXchange) could help drive this change.
- **Dynamic**, not all packages in the development or CI environment find their way to production, even those who do are not necessarily used. A dynamic oriented SBOM should have the ability to identify which components are actually being used (loaded to memory) and which are not, thus providing valuable context for tasks such as code debloat or risk assessment.
- **Continuous**, an SBOM should not only be generated once at a given point in time, at a specific point in the SDLC pipeline. Instead, it should be a continuous ongoing process.

There is a constant tradeoff between stability, reliability and reproducibility, and security in the context of using outdated packages with the balance usually falling in the side of the former.

The question is to what extent? Is there a balance that can be achieved? Should Maven Central, in this case, or other package managers/main code repositories actively deny downloads of vulnerable package versions for specific critical vulnerabilities? What about packages like Log4j.l.x which is almost seven years past its end of life date? At the very least, should anyone one pulling such vulnerable components be alerted in the form of a warning message? There is no clear answer, but we feel it is a discussion worth conducting.



ABOUT REZILION

Rezilion's platform automatically secures the software you deliver to customers. Rezilion's continuous runtime analysis detects vulnerable software components on any layer of the software stack and determines their exploitability, filtering out up to 95% of identified vulnerabilities. Rezilion then automatically mitigates exploitable vulnerabilities across the SDLC, reducing vulnerability backlogs and remediation timelines from months to hours, while giving DevOps teams time back to build.

Learn more about Rezilion's software attack surface management platform at www.rezilion.com and get your 30 day free trial www.rezilion.com.

© Rezilion 2022 | Updated April 2022