The Log4Shell Cyber Pandemic

Anna Schmult

Department of Computing Sciences, Villanova University

CSC 3010: Cybersecurity

Professor Mark Anderson

April 28, 2022

The Log4Shell vulnerability, officially CVE-2021-44228, found in the Log4j logging package is one of the biggest, if not the single biggest, software vulnerabilities of the decade. Apache gave it a 10/10 in severity, and cybersecurity organizations and researchers around the world talked about its enormous potential for devastating consequences. Reports were made of teams of developers pulling repeated all-nighters, pouring through codebases, trying to find and fix instances of the vulnerability. The frenzy of activity its announcement kicked off led IT security firm Check Point to dub it a “cyber pandemic”.

Factors leading to the potential for widespread devastating damage from Log4Shell included the widespread usage of Log4j, the vulnerability’s hidden nature to end users, and the severity of its potential usage. Log4Shell can allow remote code execution and the leakage of sensitive environmental variables. A known timeline of major Log4Shell events (as of 4/13/22) is as follows:

2013: Apache releases the first Log4j version which had the Log4Shell vulnerability.

November 24, 2021: Alibaba, a Chinese technology and e-commerce company, first reports the issue to Apache. Alibaba later gets in trouble with the Chinese government for not reporting the flaw to them first.

Around December 1, 2021: It is believed that malicious actors first started using the Log4Shell vulnerability as a zero-day exploit around this date, although no major breaches were recorded.

December 6, 2021: Apache releases Log4j 2.15.0, which provides an initial fix for the vulnerability.

December 9, 2021: Alibaba reports the vulnerability to the world via Twitter.

Understanding Log4Shell starts with understanding Log4j. Log4j is an open-source Java logging application, maintained by the Apache Software Foundation. It is one of the most common logging libraries used in Java projects for a variety of reasons, including its ease of use, its robust features, and the software development principal of avoiding "recreating the wheel".

One of the first questions that arises regarding Log4j and its vulnerabilities is why logging functionality is needed in the first place. Maintaining logs is a crucial part of maintaining a secure environment. It allows for verification that procedures are being followed and documentation when things go wrong, and serves as a tool to figure out exactly how and why something went wrong. Logging is the central part of Auditing, and critical to Accountability, two of the AAA services of cybersecurity. Good software should log just about everything that goes on, with particular focus on where things can go wrong. These points include any points of user input or user decision, automated decisions with weight or impact on a business, as well as any time something is known to have gone wrong, such as when an error is thrown and caught.

One of the factors leading to the popularity of Log4j among developers is its ease of use. A developer working on a Java project using Maven, a common software project management and build automation tool, can add Log4j to their project using four lines of code. Once Log4j is a dependency in the project, adding something to a log requires a call to a single function. Logs can easily be configured to be sent to various places, including the console, text files, databases, etc. Along with ease of use, a second factor making Log4j so popular is its robust and powerful functionality.

The robust functionality that contributes to making Log4j popular is the same functionality that led to the Log4Shell vulnerability. The vulnerability comes about through the Log4j string replacement functionality, which allows a string to be passed in and substituted, much in the form of a variable being evaluated. String replacement is triggered by a portion of a string passed into a log which contains the syntax `${prefix:value}`. These string replacements constitute a convenience to users of Log4j - it allows them to easily manipulate logged values. Even more conveniently, developers can easily pull outside resources or information into their logs, without having to write scripts to do so themselves.

Some of the simplest prefixes are `upper` and `lower`, which replace the value with the all-uppercase or all-lowercase versions of the text. Another simple prefix is `date`, which, when given a formatting string as a value, will return the current date, formatted in that style. Some of the slightly more complex prefixes allow for documentation of environmental variables, such as the `java` prefix, which will allow for information like the Java version or the operating system to be pulled into the log. When trying to document a known failure, like an error being thrown, it's very important to know things like what version of Java is running, making this prefix important. Another prefix available to users is the `env` prefix, an abbreviation for environmental lookup, which allows users to pull environmental variables, which often include configuration settings and credentials, from global files. Like with the `java` prefix, this is very useful when attempting to record the exact details of something that went wrong so it can be replicated later.

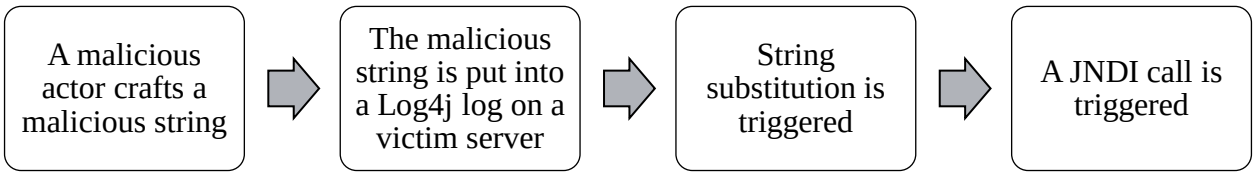
One of the prefixes available to use within Log4j is the `jndi` prefix, which allows Log4j to make JNDI (Java Naming and Directory Interface) calls. JNDI is a Java-specific naming service, with a naming service being an interface which associates an object with a name and allows an

object to be retrieved given its corresponding name. JNDI provides a way for Java applications to interact with various directory-based services, including LDAP and DNS.

LDAP, Lightweight Directory Access Protocol, is a non-Java-specific protocol which serves as a directory service. Directory services, which are often used to extend naming services, allow attributes to be assigned to objects (and their assigned names), and allows any characteristic (object, name, attribute) to be retrieved based on any other characteristic. In addition to serving as a directory service, LDAP also provides authentication, although it can be used anonymously.

JNDI and LDAP are often used together when working with Java. Java interfaces with JNDI, which in turn interfaces with LDAP. One common usage of JNDI and LDAP, which is of relevance regarding Log4Shell, allows a user to, given a location of a resource, go to that location, find the resource there, and invoke it locally. This can be triggered by the Log4j `jndi` prefix, using the lookup type `ldap`. This constitutes a significant convenience to users of Log4j, as it allows resources to be stored remotely and centrally, and then accessed only when needed. This can help reduce redundancy and free up storage space on servers.

Log4Shell occurs due to a failure to sanitize user input. Sanitization of user input generally ensures that any input provided by a user is always treated as a string (or other safe data type) and never executed as a script or a function. Log4Shell allows user input to trigger string substitution, which in turn triggers a JNDI call. The general process of the vulnerability is as follows:



Malicious agents have found several different methods of leveraging the Log4Shell vulnerability. First, they can craft the input string using nested or recurring string replacement so

that the inner strings get replaced with sensitive environment variables. Then, the outer string creates a JNDI call to a server controlled by the malicious actor, with sensitive variables as part of a URL path. This allows malicious actors to gain private information, which can either be used directly in a later attack, or, in the case of credentials, sold. Honeypot servers, designed to be attacked for research purposes, experienced several instances of attacks using strings specifically aiming to steal AWS credentials.

Second, a malicious actor can craft a string that makes a call to a server they control, load a Java object from that server, and instantiate that object, which begins running malicious code on the victim server. This allows for remote code execution, which can be used for any number of malicious purposes, including installation of malware.

The above two methods of leveraging the vulnerability are the most common, but there are a large number of others, most with technical details far beyond the scope of this paper. In addition, based on the extreme scrutiny of Log4j in the days following the announcement of Log4Shell, a number of other very similar vulnerabilities were discovered. These vulnerabilities also relied on a lack of user input sanitization triggering Log4j string substitution, which in turn triggered some sort of action or call. By and large, these other vulnerabilities were extremely complex, and are not thought to have been discovered or used by malicious actors, but only by security researchers, leading to their rapid patching.

The malicious strings needed to leverage the Jog4Shell vulnerability can be relatively simple. The string `${jndi:ldap://x.x.x.x/file}`, when put into a Log4j log, will cause Log4j to send an LDAP-type JNDI call to `x.x.x.x` looking for the object at the path `/file`, and, assuming a Java object is returned, it will be instantiated, triggering whatever code is in the object's

constructor method. Obviously, a malicious actor could have a Java object at `x.x.x.x/path` which, when instantiated, loads malware on the system.

However, in many cases, the goal of the JNDI call was neither to steal sensitive information nor to install malware. Instead, the JNDI call served simply to log that the machine it came from is vulnerable to Log4Shell, perhaps as reconnaissance for later. Some of the exploit strings used for reconnaissance probing for vulnerable systems also included information about how the string was successfully able to get itself into the Log4j log.

The most common way for a malicious string to find its way into a Log4j log seems to be in the user agent header of an HTTP request. HTTP headers allow clients and servers to pass information with HTTP requests or responses and are a form of metadata. The user agent is a type of request header (meaning it contains information about the resource being requested or the client requesting it), which specifies the application, operating system, vendor, and/or version of the client making the request. The user agent field may be used by the server to tailor the response it sends to the client, such as sending different versions of a page to mobile versus desktop web clients. Malicious actors would use a malicious string as the value in the user agent header, with the hope that the server, seeing an unknown user agent value, would log the event (including the unknown user agent value). This was largely witnessed for reconnaissance probing, and the malicious actor would simply wait and see which probed servers made a JNDI call to their server.

When not using the user agent header, malicious actors often used other HTTP headers in largely the same way. However, there were other, even simpler methods of getting malicious strings into Log4j logs, with some of the simplest being setting an iPhone name or a Twitter username to contain a malicious string or putting a malicious string into the chat of a Minecraft server.

A chaotic period followed the public disclosure of Log4Shell, in which malicious actors worked to use the exploit before it was patched, and vulnerable companies aimed to patch the exploit before it was used against them. During this period, many of the exploits of Log4Shell were simply probing, with the aim to merely to find and record vulnerable servers. Within 24 hours of the disclosure, there had been 200,000 recorded attempted exploitations of Log4Shell, made by 60 different implementations of exploits. By December 14, five days after the announcement of the vulnerability, 46% of corporate networks globally had witnessed some form of attempted probing or another attempted exploit of the attack. Companies operating in the IT supply chain were most heavily targeted, followed by the educational and research sectors. The widespread scale of the probing was unlike what has been seen following the disclosures of other recent vulnerabilities, leading some to dub it a “cyber pandemic”.

In addition to probing, which doesn’t have any negative impacts on an organization in and of itself, there were various inherently damaging attacks perpetrated or attempted. Some of the most basic were attempts to steal credentials, such as the honeypot servers which recorded attempts to exfiltrate AWS credentials. Very early on, the most common attack witnessed was cryptojacking, which refers to malicious mining of cryptocurrencies without the knowledge and consent of the owners of the machines. Many of these cryptojacking attacks also involved the victim server being added to a botnet, with the Muhstik botnet being most active in these attacks. Other attacks included an assortment of ransomware attacks, as well as a targeted intrusion by Iranian-backed APT 35 against several Israeli targets.

As soon as the vulnerability became publicly known, developers began working furiously to patch it. Three days before the announcement of the vulnerability, Apache had released Log4j 2.15.0, and then in the days following the announcement they released versions 2.16.0, 2.17.0, and

2.17.1 (which as of 4/13/22 is the most recent stable release). 2.15.0 fixed the original vulnerability, and the subsequent releases addressed other, largely more technical issues, which were subsequently discovered by researchers. 2.17.1 mitigates Log4Shell by disabling JNDI by default and implements a whitelist for JNDI calls if they are enabled at all. By default, the whitelist permits only calls to localhost. Updating the Log4j version used by a project will provide sufficient protection against all currently known Log4Shell exploits. In addition, a primitive patch is to unzip a projects' JAR and remove the `JndiLookup.class` file, or to remove the file from the classpath. However, if an application needs Log4j to actually make JNDI calls, it is the responsibility of the developer of that application to develop and test input sanitization to ensure that malicious strings put into Log4j logs are sanitized.

The fixes to Log4Shell are reasonably easy to implement, especially because most projects using Log4j are not using JNDI or other advanced features. However, in the real world, implementation was slowed by the fact that, in many cases, Log4j was introduced into a project indirectly, such as a dependency of a dependency. This led to two problems. First, some end software owners or developers had no idea if Log4j was used in their project, and struggled to figure that out, or to figure out what version was used. Second, even if they knew a vulnerable version of Log4j was used in their software, they were often unable to do much of anything about it, as they were often provided with compiled software. Many end users or developers had to wait for a patch to be applied by the developers of their dependencies, who may have in turn been waiting on others to apply the patch.

For various reasons, including those discussed above and the typical but concerning lack of patching of vulnerable software, the mitigation of Log4Shell was rolled out somewhat slowly, especially considering how many attempted exploitations companies were facing. By December

20, ten days after the announcement of the vulnerability, only 45% of vulnerable cloud-based enterprise systems had been patched. In addition, large companies often had more resources to patch their software, leaving smaller companies with fewer resources more vulnerable to attack.

Log4Shell raises several important lessons and issues that must be considered within the realm of software management. The most obvious, and arguably the most important, is the critical nature of user input sanitization. Input sanitization ensures that any time a user is able to provide input, that input is treated in such a way that it cannot execute any sort of code or script, unless a code or script can be verified to be safe (although these situations are not very common). Log4Shell is not alone in being a failure to sanitize user inputs- SQL injection and cross-site scripting are other examples.

A second important lesson from the Log4Shell vulnerability is the importance of understanding the code in a project or software under your development, control, or ownership. Only a small fraction of Log4j users were using complicated functionality, such as JNDI calls, and many of the users not using advanced features had no idea what they were, how they worked, or what their possible security implications were. Many end developers and users weren't sure if Log4j was in use in their software at all.

One tool that can be of assistance in knowing what packages are used in a project is a Software Bill of Materials (SBOM, which is a nested inventory of all software components and dependencies in a project or software. An SBOM makes sure that everyone working with a given project knows exactly what other projects are contained within it. Adoption of SBOMs within the U.S. has been strongly encouraged by various federal agencies, including an executive order in May of 2021. It is believed by many cybersecurity researchers that Log4Shell will advance the adoption of SBOMs.

On top of technical lessons, Log4Shell also raises more general questions about responsibilities and roles in software development. Large companies all over the world make billions and billions of dollars off software and processes which in turn use Log4j, along with thousands of other pieces of open-source software. Do these large companies bear some level of responsibility to provide some form of benefit to Log4j and other open-source products, such as financial backing or assistance with testing? Ultimately, whose responsibility is it when massive vulnerabilities like Log4Shell exist?

References:

Actual CVE-2021-44228 payloads captured in the wild. (2021, December 10). The Cloudflare Blog. http://blog.cloudflare.com/actual-cve-2021-44228-payloads-captured-in-the-wild/

Almost half of networks probed for Log4Shell weaknesses. (n.d.). ComputerWeekly.Com. Retrieved April 13, 2022, from https://www.computerweekly.com/news/252510939/Almost-half-of-networks-probed-for-Log4Shell-weaknesses

China disciplines Alibaba Cloud for handling of Log4j bug. (2021, December 22). South China Morning Post. https://www.scmp.com/tech/big-tech/article/3160670/apache-Log4j-bug-chinas-industry-ministry-pulls-support-alibaba-cloud

Directory and LDAP Packages (The Java™ Tutorials > Java Naming and Directory Interface > Overview of JNDI). (n.d.). Retrieved April 12, 2022, from https://docs.oracle.com/javase/tutorial/jndi/overview/dir.html

Executive Order on Improving the Nation's Cybersecurity. (2021, May 12). The White House. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

Greig, J. (n.d.). *Second Log4j vulnerability discovered, patch already released*. ZDNet. Retrieved April 13, 2022, from https://www.zdnet.com/article/second-Log4j-vulnerability-found-apache-Log4j-2-16-0-released/

How to Fix the New Log4j DoS Vulnerability: CVE-2021-45105 - FOSSA. (2021, December 19). Dependency Heaven. https://fossa.com/blog/how-fix-new-Log4j-dos-vulnerability-cve-2021-45105/

HTTP headers—HTTP | MDN. (n.d.). Retrieved April 12, 2022, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

Inside the Log4j2 vulnerability (CVE-2021-44228). (2021, December 10). The Cloudflare Blog. http://blog.cloudflare.com/inside-the-Log4j2-vulnerability-cve-2021-44228/

Java Naming and Directory Interface. (2022). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Java_Naming_and_Directory_Interface&oldid=1077629539

JNDI as an LDAP API (The Java™ Tutorials > Java Naming and Directory Interface > Advanced Topics for LDAP Users). (n.d.). Retrieved April 12, 2022, from https://docs.oracle.com/javase/tutorial/jndi/ldap/jndi.html

LDAP and JNDI: Together forever | Computerworld. (n.d.). Retrieved April 12, 2022, from https://www.computerworld.com/article/2076073/ldap-and-jndi--together-forever.amp.html

Lesson: Overview of JNDI (The Java™ Tutorials > Java Naming and Directory Interface). (n.d.). Retrieved April 12, 2022, from https://docs.oracle.com/javase/tutorial/jndi/overview/index.html

Lightweight Directory Access Protocol. (2022). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Lightweight_Directory_Access_Protocol&oldid=1074161707

Log4j – Log4j 2 Lookups. (n.d.). Retrieved April 12, 2022, from https://logging.apache.org/Log4j/2.x/manual/lookups.html

Log4Shell. (2022). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Log4Shell&oldid=1081069130

Log4Shell Explained. (n.d.). Cynet. Retrieved April 12, 2022, from https://www.cynet.com/Log4Shell/

Log4Shell: RCE 0-day exploit found in Log4j 2, a popular Java logging package | LunaSec. (2021, December 19). https://www.lunasec.io/docs/blog/Log4j-zero-day/

Request header - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. (n.d.). Retrieved April 12, 2022, from https://developer.mozilla.org/en-US/docs/Glossary/Request_header

Software Bill of Materials | CISA. (n.d.). Retrieved April 13, 2022, from https://www.cisa.gov/sbom

The 'most serious' security breach ever is unfolding right now. Here's what you need to know. (2021, December 20). Washington Post. https://www.washingtonpost.com/technology/2021/12/20/Log4j-hack-vulnerability-java/

The Numbers Behind Log4j CVE-2021-44228. (2021, December 13). Check Point Software. https://blog.checkpoint.com/2021/12/13/the-numbers-behind-a-cyber-pandemic-detailed-dive/

User agent. (2022). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=User_agent&oldid=1079594226

User-Agent—HTTP | MDN. (n.d.). Retrieved April 12, 2022, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent

Woodyard, C. (n.d.). *"Critical vulnerability": Smaller firms may find it harder to stop hackers from exploiting Log4j flaw*. USA TODAY. Retrieved April 13, 2022, from https://www.usatoday.com/story/money/business/2021/12/16/log-4-j-vulnerability-small-business/8910567002/

Zugec, M. (n.d.). *Technical Advisory: Zero-day critical vulnerability in Log4j2 exploited in the wild*. Retrieved April 13, 2022, from https://businessinsights.bitdefender.com/technical-advisory-zero-day-critical-vulnerability-in-Log4j2-exploited-in-the-wild