jar包加密保护解决方案

2016-05-25 | 安全 | 0 Comments

为什么需要保护？

我用java写了一个程序如下：

```
1  package com.monkey.demo;
2
3  // App.java
4
5  public class App
6  {
7    static public void main( String args[] ) throws Exception {
8      System.out.println( "This is your application." );
9      System.out.print( "Args: " );
10     for (int a=0; a<args.length; ++a)
11       System.out.print( args[a]+" " );
12     System.out.println( "" );
13
14     new App().new AppChild().print();
15
16     new Foo();
17     new Bar();
18   }
19
20   public class AppChild{
21           public void print(){
22                   System.out.println("haha ....");
23           }
24   }
25 }
```

然后编译生成 .class 文件后，我发布出去了，别人拿到我的 .class 文件拖到 JD-GUI 里面看到的是这样：

App.class

```
package com.monkey.demo;

import java.io.PrintStream;

public class App
{
  public static void main(String[] args)
    throws Exception
  {
    System.out.println("This is your application.");
    System.out.print("Args: ");
    for (int a = 0; a < args.length; a++) {
      System.out.print(args[a] + " ");
    }
    System.out.println(""); void

      tmp76_73 = new App();tmp76_73.getClass();new AppChild(tmp76_73).print();

    new Foo();
    new Bar();
  }

  public class AppChild
  {
    public AppChild() {}

    public void print()
    {
      System.out.println("haha ....");
    }
  }
}
```

玩毛线，看源代码一样，当然你也可以使用 ProGuard 混淆，不过别人有点耐心还是能分析出来的。另外你还可以修改class文件里面的某些字段，这些字段对运行没有影响，但是能导致别人无法反编译。这里我们暂且不讨论这种方式，分别讨论下使用 ClassLoader 和 jvmti 对class文件加密解密的方案。

ClassLoader

Java运行时装入字节码的机制隐含地意味着可以对字节码进行修改。JVM每次装入类文件时都需要一个称为ClassLoader的对象，这个对象负责把新的类装入正在运行的JVM。JVM给ClassLoader一个包含了待装入类（比如java.lang.Object）名字的字符串，然后由ClassLoader负责找到类文件，装入原始数据，并把它转换成一个Class对象。

所以我们可以通过自定义一个ClassLoader，然后先对class进行解密之后，再加载到JVM。大致流程如下：

```
// 首先创建一个ClassLoader对象
ClassLoader myClassLoader = new myClassLoader();
// 利用定制ClassLoader对象装入类文件
// 并把它转换成Class对象
Class myClass = myClassLoader.loadClass( "mypackage.MyClass" );
// 最后，创建该类的一个实例
Object newInstance = myClass.newInstance();
```

loadClass

在创建自定义的ClassLoader时，只需覆盖其中的一个，即loadClass，获取加密后的文件数据解密加载。

```
public Class loadClass( String name, boolean resolve )
      throws ClassNotFoundException {
        try {
          // 我们要创建的Class对象
           Class clasz = null;
          // 如果类已经在系统缓冲之中，不必再次装入它
          clasz = findLoadedClass( name );
          if (clasz != null)
            return clasz;
          // 下面是定制部分
          byte classData[] = /* 解密加密后的字节数据 */;
          if (classData != null) {
            // 成功读取字节码数据，现在把它转换成一个Class对象
            clasz = defineClass( name, classData, 0, classData.length );
          }
```

```
            // 如果上面没有成功，尝试用默认的ClassLoader装入它
            if (clasz == null)
              clasz = findSystemClass( name );
            //如有必要，则装入相关的类
            if (resolve && clasz != null)
              resolveClass( clasz );
            // 把类返回给调用者
            return clasz;
          } catch( IOException ie ) {
            throw new ClassNotFoundException( ie.toString() );
          } catch( GeneralSecurityException gse ) {
            throw new ClassNotFoundException( gse.toString() );
          }
  }
```

上面是一个简单的loadClass实现，其中涉及到如下几个方法：

- findLoadedClass: 检查当前要加载的类是否已经加载。
- defineClass: 获得原始类文件字节码数据后，调用defineClass转换成一个Class对象。
- findSystemClass: 提供默认ClassLoader支持。
- resolveClass: 当JVM想要装入的不仅包括指定的类，而且还包括该类引用的所有其他类时，它会把loadClass的resolve参数设置成true。这时，必须在返回刚刚装入的Class对象给调用者之前调用resolveClass。

加密、解密

直接使用java自带加密算法，比如DES。

生成密钥

```
//DES算法要求有一个可信任的随机数源
SecureRandom sr = new SecureRandom();
//为选择的DES算法生成一个KeyGenerator对象
KeyGenerator kg = KeyGenerator.getInstance( "DES" );
kg.init( sr );
// 生成密匙
SecretKey key = kg.generateKey();
// 获取密匙数据
byte rawKeyData[] = key.getEncoded();
```

加密数据

```
// DES算法要求有一个可信任的随机数源
SecureRandom sr = new SecureRandom();
byte rawKeyData[] = /* 用某种方法获得密匙数据 */;
// 从原始密匙数据创建DESKeySpec对象
DESKeySpec dks = new DESKeySpec( rawKeyData );
// 创建一个密匙工厂，然后用它把DESKeySpec转换成一个SecretKey对象
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance( "DES" );
SecretKey key = keyFactory.generateSecret( dks );
// Cipher对象实际完成加密操作
Cipher cipher = Cipher.getInstance( "DES" );
// 用密匙初始化Cipher对象
cipher.init( Cipher.ENCRYPT_MODE, key, sr );
// 现在，获取数据并加密
byte data[] = /* 用某种方法获取数据 */
// 正式执行加密操作
byte encryptedData[] = cipher.doFinal( data );
// 进一步处理加密后的数据
doSomething( encryptedData );
```

解密数据

```
// DES算法要求有一个可信任的随机数源
```

```
SecureRandom sr = new SecureRandom();
byte rawKeyData[] = /* 用某种方法获取原始密匙数据 */;
// 从原始密匙数据创建一个DESKeySpec对象
DESKeySpec dks = new DESKeySpec( rawKeyData );
// 创建一个密匙工厂，然后用它把DESKeySpec对象转换成
// 一个SecretKey对象
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance( "DES" );
SecretKey key = keyFactory.generateSecret( dks );
// Cipher对象实际完成解密操作
Cipher cipher = Cipher.getInstance( "DES" );
// 用密匙初始化Cipher对象
cipher.init( Cipher.DECRYPT_MODE, key, sr );
// 现在，获取数据并解密
byte encryptedData[] = /* 获得经过加密的数据 */
// 正式执行解密操作
byte decryptedData[] = cipher.doFinal( encryptedData );
// 进一步处理解密后的数据
doSomething( decryptedData );
```

实际案例

这里写了一个简单的例子，代码在github。

首先生成密钥：

```
javac FileUtil.java
javac GenerateKey.java
java GenerateKey key.data
```

然后加密class:

```
javac EncryptClasses.java
java EncryptClasses key.data App.class Foo.class Bar.class
```

运行加密后的应用:

```
javac MyClassLoader.java -Xlint:unchecked
java MyClassLoader key.data App
```

总的来说ClassLoader在类非常多的情况还是比较麻烦，而且这样一来自定义的ClassLoader本身就成为了突破口。下面介绍另外一种加密保护的方案。

jvmti

jvmti(JVMTM Tool Interface)是JDK提供的一套用于开发JVM监控，问题定位与性能调优工具的通用变成接口。通过JVMTI，我们可以开发各式各样的JVMTI Agent。这个Agent的表现形式是一个以c/c++语言编写的动态共享库。

JVMTI Agent原理: java启动或运行时，动态加载一个外部基于JVM TI编写的dynamic module到Java进程内，然后触发JVM源生线程Attach Listener来执行这个dynamic module的回调函数。在函数体内，你可以获取各种各样的VM级信息，注册感兴趣的VM事件，甚至控制VM的行为。

这里我们只需要监控class的加载信息，而jvmti也提供了这样的接口，通过下面的方式我们就能监控到class的加载：

```
JNIEXPORT jint JNICALL
Agent_OnLoad(
    JavaVM *vm,
    char *options,
    void *reserved
)
```

```
{
        ......
        //设置事件回调
    jvmtiEventCallbacks callbacks;
    (void)memset(&callbacks,0, sizeof(callbacks));

    callbacks.ClassFileLoadHook = &MyClassFileLoadHook;
    error = jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
    ......
}

void JNICALL
MyClassFileLoadHook(
    jvmtiEnv *jvmti_env,
    JNIEnv* jni_env,
    jclass class_being_redefined,
    jobject loader,
    const char* name,   //class名字
    jobject protection_domain,
    jint class_data_len,  //class文件数据长度
    const unsigned char* class_data,   //class文件数据
    jint* new_class_data_len,   //新的class文件数据长度
    unsigned char** new_class_data   //新的class文件数据
)
{
        ......
}
```

通过这样的方式就能监控到class的加载然后再对其进行解密。

加密、解密

加密class文件

这里简单的通过遍历class文件，然后对每个字节进行一个异或处理，具体的加密方法可以自己扩展：

```
extern"C" JNIEXPORT jbyteArray JNICALL
Java_Encrypt_encrypt(
    JNIEnv * _env,
    jobject _obj,
    jbyteArray _buf
)
{
    jsize len =_env->GetArrayLength(_buf);

    unsigned char* dst = (unsigned char*)_env->GetByteArrayElements(_buf, 0);

        for (int i = 0; i < len; ++i)
        {
                dst[i] = dst[i] ^ 0x07;
        }

    _env->SetByteArrayRegion(_buf, 0, len, (jbyte *)dst);
    return _buf;
}
```

解密class文件

在运行jar文件的时候，加载我们的jvmti agent动态库进行动态解密：

```
void JNICALL
MyClassFileLoadHook(
    jvmtiEnv *jvmti_env,
    JNIEnv* jni_env,
    jclass class_being_redefined,
    jobject loader,
    const char* name,
    jobject protection_domain,
```

```
    jint class_data_len,
    const unsigned char* class_data,
    jint* new_class_data_len,
    unsigned char** new_class_data
)
{
    *new_class_data_len = class_data_len;
    jvmti_env->Allocate(class_data_len, new_class_data);

    unsigned char* my_data = *new_class_data;

    if(name&&strncmp(name,"com/monkey/",11)==0){
        for (int i = 0; i < class_data_len; ++i)
        {
            my_data[i] = class_data[i] ^ 0x07;
        }
    }else{
        for (int i = 0; i < class_data_len; ++i)
        {
            my_data[i] = class_data[i];
        }
    }
}
```

实际案例

这里写了一个简单的例子，代码在github。

首先加密jar包：

```
javac Encrypt.java
java -Djava.library.path=. -cp . Encrypt -src jardemo.jar
```

```
➜  encrypt git:(master) ✗ java -Djava.library.path=. -cp . Encrypt -src ../../Example/jardemo.jar
encode jar file: [../../Example/jardemo.jar ==> ../../Example/jardemo_encrypt.jar ]
encrypt com.monkey.demo.App$AppChild.class
encrypt com.monkey.demo.App.class
encrypt com.monkey.demo.Bar.class
encrypt com.monkey.demo.Foo.class
```

然后会得到一个 `jardemo_encrypt.jar` 文件，如果现在直接去运行该文件的话肯定是会出错的，所以要做解密。

先编译生成一个解密的动态库 `libdecrypt.dylib` 。然后运行：

```
java -jar -agentlib:decrypt jardemo_encrypt.jar
```

```
➜  Example git:(master) ✗ java -jar -agentlib:decrypt jardemo_encrypt.jar 1 2 3
This is your application.
Args: 1 2 3
haha ....
foo
bar
```

总结

总的来说，使用jvmti提供的监控api，方便了我们直接对class的操作，所以第二个方案更好一些，当然其中具体使用怎么样的加密，以及如何去保证加密不被破解就需要各位发挥自己的空间了。

欢迎您扫一扫上面的微信公众号，订阅我的博客！

↑

感谢你的支持，我会继续努力！

#jvmti

‹ DES和AES算法详解

Mac下载编译chromium源码 ›