**0x01 基本思路**

这里以Java Rasp的实现原理为例。Rasp想要将自己注入到被保护的应用中，基本思路类似于Java中的AOP技术，将RASP的探针代码注入到需要进行检测的地方。

Java的AOP主要可以从几个层面来实现：

* 编译期
* 字节码加载前
* 字节码加载后

在编译期进行AOP织入，一般需要编写静态代理，导致灵活性差，对原有的应用代码有修改。

在字节码加载后进行AOP织入，一般使用动态代理，为接口动态生成代理类。动态代理虽然灵活性高，但仍然需要使用相关的类库，进行动态代理的配置，并融合到应用的源代码中，不是理想的解决方案。

最后只剩下了在字节码加载前进行AOP织入。在字节码加载前进行织入，一般有两种方法，重写ClassLoader或利用Instrumentation。如果重写ClassLoader，仍然对现有代码进行了修改，不能做到对应用无侵入。所以只有利用Java的Instrumentation。

**0x02 Instrumentation**

“java.lang.instrument”包的具体实现依赖于JVMTI。JVMTI（Java Virtual Machine Tool Interface）是一套由 Java 虚拟机提供的，为 JVM 相关的工具提供的本地编程接口集合。在 Instrumentation 的实现当中，存在一个 JVMTI 的代理程序，通过调用 JVMTI 当中 Java 类相关的函数来完成 Java 类的动态操作。

我们可以Instrumentation的代理，并让其在main函数之前运行，这里需要实现的主要是premain函数。

public static void premain(String agentArgs, Instrumentation inst)

throws ClassNotFoundException, UnmodifiableClassException {

Console.log("init");

init();

inst.addTransformer(new ClassTransformer());

}

private static boolean init() {

Config.initConfig();

return true;

}

在premain函数中，我们将类转换器添加到了Instrumentation，这样在类加载前，我们便有机会对字节码进行操作，织入Rasp的安全探针。

若想使用带有Instrumentation代理的程序，需要在JVM的启动参数中添加-javaagent启动参数。

-javaagent:[编译好的agent jar文件路径]XXX.jar

**0x03 利用ClassTransformer进行探针织入**

在运行了Instrumentation代理的Java程序中，字节码的加载会经过我们自定义的ClassTransformer，在这里我们可以过滤出我们关注的类，并对其字节码进行相关的修改

public class ClassTransformer implements ClassFileTransformer {

public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,

ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassFormatException {

byte[] transformeredByteCode = classfileBuffer;

if (Config.moudleMap.containsKey(className)) {

try {

ClassReader reader = new ClassReader(classfileBuffer);

ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE\_MAXS);

ClassVisitor visitor = Reflections.createVisitorIns((String)Config.moudleMap.get(className).get("loadClass"), writer, className);

reader.accept(visitor, ClassReader.EXPAND\_FRAMES);

transformeredByteCode = writer.toByteArray();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

} catch (Exception e) {

e.printStackTrace();

}

}

return transformeredByteCode;

}

}

实现中使用了使用了Map将关注的类进行保存，一旦命中我们关心的类，便利用反射生成asm的ClassVisitor ，使用asm操作字节码，进行探针织入，最终返回修改后的字节码。

这里的ClassVisitor以Struts 2的Ognl表达式执行漏洞为例：

public class OgnlVisitor extends ClassVisitor {

public String className;

public OgnlVisitor(ClassVisitor cv, String className) {

super(Opcodes.ASM5, cv);

this.className = className;

}

@Override

public MethodVisitor visitMethod(int access, String name, String desc,

String signature, String[] exceptions) {

MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);

if ("parseExpression".equals(name) && "(Ljava/lang/String;)Ljava/lang/Object;".equals(desc)) {

mv = new OgnlVisitorAdapter(mv, access, name, desc);

}

return mv;

}

}

在这个ClassVisitor中，我们关心的是ognl.Ognl类中的parseExpression方法。只要在Ognl的parseExpression执行之前对Ognl表达式中的恶意参数进行过滤，可以对Struts 2的Ognl表达式执行漏洞进行有效的防护。具体的字节码操作封装在了OgnlVisitorAdapter中

public class OgnlVisitorAdapter extends AdviceAdapter {

public OgnlVisitorAdapter(MethodVisitor mv, int access, String name, String desc) {

super(Opcodes.ASM5, mv, access, name, desc);

}

@Override

protected void onMethodEnter() {

Label l30 = new Label();

mv.visitLabel(l30);

mv.visitVarInsn(ALOAD, 0);

mv.visitMethodInsn(INVOKESTATIC, "xbear/javaopenrasp/filters/rce/OgnlFilter", "staticFilter", "(Ljava/lang/Object;)Z", false);

Label l31 = new Label();

mv.visitJumpInsn(IFNE, l31);

Label l32 = new Label();

mv.visitLabel(l32);

mv.visitTypeInsn(NEW, "ognl/OgnlException");

mv.visitInsn(DUP);

mv.visitLdcInsn("invalid class in ognl expression because of security");

mv.visitMethodInsn(INVOKESPECIAL, "ognl/OgnlException", "<init>", "(Ljava/lang/String;)V", false);

mv.visitInsn(ATHROW);

mv.visitLabel(l31);

}

@Override

public void visitMaxs(int maxStack, int maxLocals) {

super.visitMaxs(maxStack, maxLocals);

}

}

在ognl.Ognl类中的parseExpression方法执行前，Hook了方法的执行，跳转至执行自定义的OgnlFilter。OgnlFilter中定义了如何对Ognl表达式进行过滤。如果出现了威胁的表达式，将进行log记录并抛出异常，若正常将放过，继续进行parseExpression。

**0x04 结尾**

这里介绍了Java Rasp实现的基本原理，除了ognl.Ognl类中的parseExpression方法加探针外，还有很多的地方可以加探针，比如：java/io/ObjectInputStream、java/lang/ProcessBuilder、com/mysql/jdbc/StatementImpl等等。重点关注数据的关键流转节点加入Rasp探针，进行安全过滤。

如果探针部署的足够充分，可以有效的防御XSS、CSRF、RCE、SQL注入等Web攻击。如果Rasp与云端结合，不但能够采集应用的安全日志，也能够对发现的漏洞进行迅速的修补，甚至抵御0Day攻击。

Demo: <https://github.com/xbeark/javaopenrasp>

这里只实现了使用了Instrumentation的premain进行代理，其实还可以使用agentmain进行虚拟机启动后的动态instrument，具体就不在这里研究啦~