LAB № 4

潘亦晟, 515021910384 05/25/2018

1 Android APK 逆向

- 实验环境：Ubuntu 17.10 amd64
- 实验工具：dex2jar、id-gui、apktool、IDA
- 实验目的：在提供的 sample.apk 文件中，有四个被隐藏起来的字符串，它们的格式为 cApwN{this_is_secret_string}

我们对 sample.apk 进行解压，然后使用 **dex2jar** 将 class.dex 转化为 class.jar。最后使用 **id-gui** 反编译 jar 为 java 代码。

Secret One

Listing 1: TabFragment1.class

```
1  public void onClick(View paramAnonymousView) {
2      String str = TabFragment1.this.mInput.getText().toString();
3      paramAnonymousView = str;
4      if (str.isEmpty())
5      {
6          int i = new Random().nextInt();
7          paramAnonymousView = "asset" + (i % 10 + 1);
8      }
9      TabFragment1.this.loadDataFromAsset(paramAnonymousView);
10 }
```

我们在 **TabFragment1.class** 中 **onCreate** method 中发现每次随机载入图片，我们在 *assets* 文件夹中发现了第一个 secret。第一个 secret 被隐藏在 **tHiS_iS_nOt_tHE_SeCrEt_lEveL_1_fiLE** 图片中。见 1

图 1: tHiS_iS_nOt_tHE_SeCrEt_lEveL_1_fiLE

First Secret: cApwNWELL_THAT_WAS_SUPER_EASY

Secret Two

我们接着在 **TabFragment2.class** 中寻找第二个 secret。

Listing 2: TabFragment2.class

```
1  public void onClick(View paramAnonymousView) {
2      try {
3          paramAnonymousView = InCryption.hashOfPlainText();
4          TabFragment2.this.mHashView.setText(paramAnonymousView);
5          TabFragment2.this.mHashView.setBackgroundColor(-1);
6          return;
7      }
8      catch (Exception paramAnonymousView) {
9          paramAnonymousView.printStackTrace();
10     }
11 }
```

我们发现其调用了 **InCryption.class** 中的 **hashOfPlainText.method**。我们发现这个函数会返回一个明文字符串的 hash 值。我们重新新建 JAVA 代码 (见) 来获取该明文字符串。

Listing 3: recreated JAVA code

```
1  import java.math.BigInteger;
2  import java.security.MessageDigest;
3  import java.security.NoSuchAlgorithmException;
4  import javax.crypto.Cipher;
5  import javax.crypto.spec.SecretKeySpec;
6
7  public class Solve {
8
9  public static void main(String[] args) {
```

```
10      try {
11          System.out.println(Solve.plainText());
12      } catch (Exception e) {
13
14      }
15  }
16  static String encryptedHex = "XXXXXXXXXXXXXXXXXXX";
17
18  static String bin2hex(byte[] paramArrayOfByte)
19  {
20      return String.format("%0" + paramArrayOfByte.length * 2 + "X", new ↩
            Object[] { new BigInteger(1, paramArrayOfByte) });
21  }
22
23  private static byte[] decrypt(byte[] paramArrayOfByte1, byte[] ↩
        paramArrayOfByte2)
24      throws Exception
25  {
26      SecretKeySpec a = new SecretKeySpec(paramArrayOfByte1, "AES");
27      Cipher localCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
28      localCipher.init(2, a);
29      return localCipher.doFinal(paramArrayOfByte2);
30  }
31
32  public static String getHash(String paramString)
33  {
34      try
35      {
36          MessageDigest localMessageDigest = MessageDigest.getInstance("SHA↩
                -256");
37          localMessageDigest.reset();
38          return bin2hex(localMessageDigest.digest(paramString.getBytes()));
39      }
40      catch (NoSuchAlgorithmException e)
41      {
42          e.printStackTrace();
43      }
44      return "";
45  }
46
47  public static String plainText()
48      throws Exception
49  {
50      return new String(hex2bytes(new String(decrypt(hex2bytes("0123456789↩
            ABCDEF0123456789ABCDEF"), hex2bytes(encryptedHex))).trim()));
51  }
52
```

```
53
54 static byte[] hex2bytes(String paramString)
55 {
56     byte[] arrayOfByte = new byte[paramString.length() / 2];
57     int i = 0;
58     while (i < arrayOfByte.length)
59     {
60         int j = i * 2;
61         arrayOfByte[i] = ((byte)Integer.parseInt(paramString.substring(j, ↩
               j + 2), 16));
62         i += 1;
63     }
64     return arrayOfByte;
65 }
66 }
```

我们运行代码，得到结果是由 DASH，DOT，SPACE 的组合，我们猜测该结果是摩斯密码。将摩斯密码进行转化，我们得到第二个 secret。

Second Secret: cApwNCRYP706R4PHY_15_H4RD_BR0

Secret Three

我们在 id-gui 中找到 **Level3Activity.class** 中，我们发现除了 **MonteCarlo.start()** 无其他明显异常；我们继续追踪至 **MonteCarlo.class**，该 class 似乎在逼近 π 的值，除了调用了一个 **ArraysArraysArrays.start()** 不知其作用，无其他异常; 我们继续追踪至 **ArraysArraysArrays.start()** 函数，发现其调用了一个不知其作用的 JNI method **x()**。为了分析这个 JNI method，我们直接在 IDA 软件中反编译 **libnative-lib.so**。反编译出的 C 代码如下：

Listing 4: JNI method ArraysArraysArrays.x()

```
1  int __fastcall Java_com_h1702ctf_ctfone_ArraysArraysArrays_x(int a1)
2  {
3      int v1; // r4@1
4      int v2; // r0@2
5      int v3; // r6@4
6      int v4; // r0@5
7      int v5; // r0@8
8      int result; // r0@10
9
10     v1 = a1;
11     if ( !byte_C113[0] )
12     {
13         v2 = 0;
14         do
15         {
```

```
16              byte_C113[v2] = byte_C05B[v2] ^ 0x3D;
17              ++v2;
18          }
19          while ( v2 != 29 );
20      }
21      v3 = (*(int (__fastcall **)(int, char *))(*(_DWORD *)v1 + 24))(v1, ↩
            byte_C113);
22      if ( !byte_C131[0] )
23      {
24          v4 = 0;
25          do
26          {
27              byte_C131[v4] = aIYi_x[v4] ^ 0x2C;
28              ++v4;
29          }
30          while ( v4 != 7 );
31      }
32      if ( !byte_C139[0] )
33      {
34          v5 = 0;
35          do
36          {
37              byte_C139[v5] = byte_C081[v5] ^ 0x58;
38              ++v5;
39          }
40          while ( v5 != 3 );
41      }
42      result = (*(int (__fastcall **)(int, int, char *, char *))(*(_DWORD *)↩
            v1 + 452))(v1, v3, byte_C131, byte_C139);
43      if ( result )
44          result = _JNIEnv::CallStaticVoidMethod(v1, v3);
45      return result;
46  }
```

很明显在这个函数中一些常量被一个固定的 key 加密了，我们使用 python 将加密过程还原，代码如下：

```
1  from pwn import *
2  first = [0x5E,0x52,0x50,0x12,0x55, 0xC, 0xA, 0xD, 0xF,0x5E,0x49,0x5B,0x12↩
       ,0x5E,0x49,0x5B,0x52,0x53,0x58,0x12,0x6F,0x58,0x4C,0x48,0x58,0x4E,0x49↩
       ,0x52,0x4F]
3  print xor(first,0x3d)
4  second = [0x5E, 0x49, 0x5D, 0x59, 0x49, 0x5F, 0x58]
5  print xor(second,0x2c)
6  third = [0x70,0x71,0xE]
7  print xor(third,0x58)
```

运行该 python 脚本，我们得到了如下结果：

```
com/h1702ctf/ctfone/Requestor
request
()V
```

根据上述结果，我们可以推断出 JNI method **x()** 会调用 **com/h1702ctf/ctfone/Requestor** 中的 **request.method**; 我们继续在 id-gui 中追踪 **com/h1702ctf/ctfone/Requestor.class** , 我们发现该 method 发出对 **https://h1702ctf.com/About** 的请求，其中一对本地 {key-value} 由两个 JNI methods **hName**，**hVal** 生成。

我们同样使用 IDA 软件中反编译 **libnative-lib.so** 找到这两个 JNI method。

```
int __fastcall Java_com_h1702ctf_ctfone_Requestor_hName(int a1)
{
    int v1; // r1

    if ( !byte_C0BC[0] )
    {
        v1 = 0;
        do
        {
            byte_C0BC[v1] = byte_C004[v1] ^ 0x37;
            ++v1;
        }
        while ( v1 != 13 );
    }
    return (*(int (**)(void))(*(_DWORD *)a1 + 668))();
}



int __fastcall Java_com_h1702ctf_ctfone_Requestor_hVal(int a1)
{
    int v1; // r1

    if ( !byte_C0CA[0] )
    {
        v1 = 0;
        do
        {
            byte_C0CA[v1] = byte_C012[v1] ^ 0x3E;
            ++v1;
        }
        while ( v1 != 72 );
```

```
33      }
34      return (*(int (**)(void))(*(_DWORD *)a1 + 668))();
35  }
```

我们发现同样的在这两个 JNI method 中，常量被固定的 key 加密，我们类似地使用 python 代码加密这两个常量：

```
1  headerKey =[0x6F,0x1A,0x7B ,0x52,0x41,0x52,0x5B,   4,0x1A,0x71,0x5B,0x56,0↩
       x50]
2  print xor(headerKey, 0x37)
3  headerValue =[  0x68, 0x0F, 0x6C, 0x7D, 0x6C, 0x0C, 0x6F, 0x47, 0x6B, 0x66↩
       ,
4  0x5A, 0x71, 0x68, 0x79, 0x6C, 0x71, 0x68, 0x53, 0x4E, 0x50,
5  0x5A, 0x0F, 0x52, 0x4D, 0x69, 0x6A, 0x68, 0x55, 0x68, 0x0F,
6  0x74, 0x67, 0x6A, 0x68, 0x5A, 0x4D, 0x6A, 0x55, 0x0E, 0x49,
7  0x5D, 0x79, 0x0F, 0x6B, 0x5F, 0x55, 0x4E, 0x48, 0x64, 0x68,
8  0x6B, 0x46, 0x70, 0x52, 0x6C, 0x4F, 0x5C, 0x7B, 0x6C, 0x5F,
9  0x5B, 0x54, 0x7F, 0x0B, 0x6F, 0x0C, 0x5D, 0x07, 0x6E, 0x6F,
10 0x51, 0x03]
11 print xor(headerValue, 0x3e)
```

运行 python 脚本，我们得到：

```
1  X-Level3-Flag
2  V1RCR2QyUXdOVGROVmpnd1lsWTVkV1JYTVdsTk0wcG1UakpvZVUxNlRqbERaejA5Q2c9PQo=
```

我们猜测第是三个 secret 是 64 进制数，我们将其解码得到：

Secret Three: cApwN1_4m_numb3r_7hr33

Fourth Secret

我们在 **MonteCarlo.class** 中，我们发现了一个被声明但是未被使用的 JNI method **functionnameLeftbraceOneCommaTwoCommaThreeCommaRightbraceFour**。这个函数名暗示我们需要使用前三个 secret 作为参数调用该函数得到第四个 secret。我们创建一个新的 apk 来直接调用该 JNI method。

```
1  package com.h1702ctf.ctfone;
2
3  public class MonteCarlo {
4      public native String ↩
           functionnameLeftbraceOneCommaTwoCommaThreeCommaRightbraceFour(↩
           String paramString1, String paramString2, String paramString3);
5      public String test() {
```

```
6           String a="cApwN{WELL_THAT_WAS_SUPER_EASY}";
7           String b="CAPWN{CRYP706R4PHY_15_H4RD_BR0}";
8           String c="cApwN{1_4m_numb3r_7hr33}";
9           return ↩
                functionnameLeftbraceOneCommaTwoCommaThreeCommaRightbraceFour(↩
                a,b,c);
10      }
11  }
```

运行代码，我们得到：

Secret Four: cApwNw1nn3r_w1nn3r_ch1ck3n_d1nn3r!

2 UPX 脱壳

- 实验环境：Windows XP 32bit
- 实验目的：了解 upx 壳的脱壳过程。
- 实验要求：在 Windows 32 位系统下，对于任意 exe 文件使用 upx 进行加壳然后使用脱壳工具进行脱壳。

使用 PEID 等查壳工具分析程序加壳情况

我们使用 PEID 工具对 **notepad_upx.exe** 进行分析，我们发现该 exe 文件被加了 **UPX 壳。**

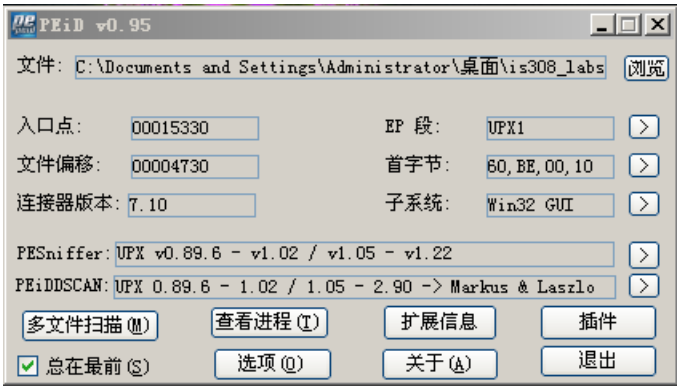

图 2: PEID analysis

比较加壳前后 PE 文件结构的变化

我们使用 **LordPE** 软件对加壳前后软件 (notepad_upx.exe 和 notepad.exe) 的结构进行分析比较。其中图 2,3 显示了 PE 文件整体信息的比较；图 4,5 展示了加壳前后区段表的比较；1展示了加壳前后 Optional Header 的比较；2 展示了加壳前后 Data Directories 的比较。

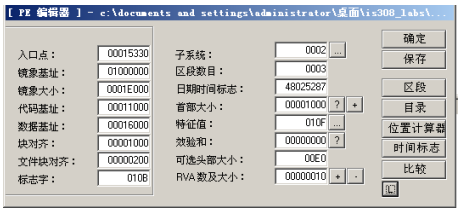

图 3: PE information of notepad_upx.exe

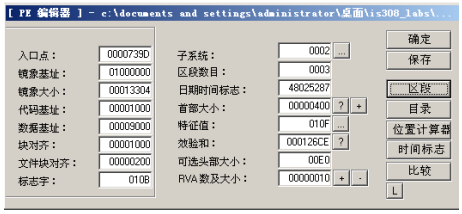

图 4: PE information of notepad.exe

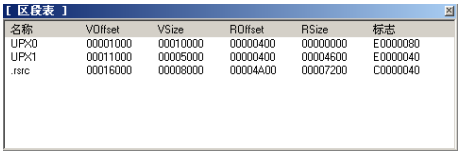

图 5: Section information of notepad_upx.exe

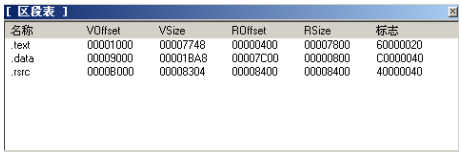

图 6: Section information of notepad.exe

结论

- 我们发现加壳后 exe 文件大小变小，说明 UPX 壳 (压缩壳) 对文件实现了压缩功能。
- 我们发现文件三个区段的 RVA 都发生了变化，其中.text,.data 区段变为了 UPX0，UPX1 区段。(见图 5,6)
- 我们发现文件的入口点发生了变化，因为 UPX 壳需要将文件的真正 OEP 更改为壳的入口，实现代码执行时先运行壳。
- 由于文件发生了变化，所以文件的检验和也发生了变化。

表 1: Optional Header

Item to compare	notepad_upx.exe	notepad.exe
Size Of Code	5000	7800
Size Of Initialized Data	8000	8C00
Size Of Uninitialized Data	10000	0000
Address Of Entry Point	15330	739D
Base Of Code	11000	1000
Base Of Data	16000	9000
Size Of Image	1E000	14000
Size Of Header	1000	400
Check Sum	0000	126CE

表 2: Data Directories

Item to compare	notepad_upx.exe	notepad.exe
Import Table -RVA	1CE04	7604
Import Table -Size	24C	C8
Resource -RVA	16000	B000
Resource -Size	6E04	8304
Debug -RVA	0000	1350
Debug -Size	0000	001C
LordConfig -RVA	154C0	18A8
LordConfig -Size	48	40
BoundImport -RVA	0000	250
BoundImport -Size	0	D0
IAT -RVA	0000	1000
IAT -Size	0000	348

使用堆栈平衡原理的方式脱壳

- 运行 pushad 指令后，我们发现 ESP 突变 (0006FFC4 → 0006FFA4)，命令行执行 dd 0006FFA4，下硬件断点 (word)。
- 在 F9 运行后，结果如图 7

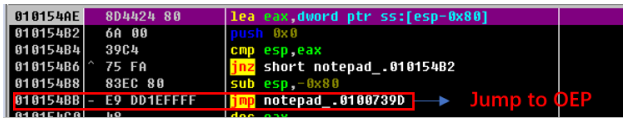

图 7: ESP method

- 找到 OEP 后我们就可以进行脱壳，我们使用 **LordPE** 进行脱壳 (修复镜像大小、选择完整转存)
- 使用 **ImportREConstructor** 进行修复，根据 OD 中的地址，计算出 OEP，OEP=VA(0100739D)-加载基址 (01000000)，自动查找 IAT，然后获取输入表，显示无效函数，最后进行修复。步骤见 8。

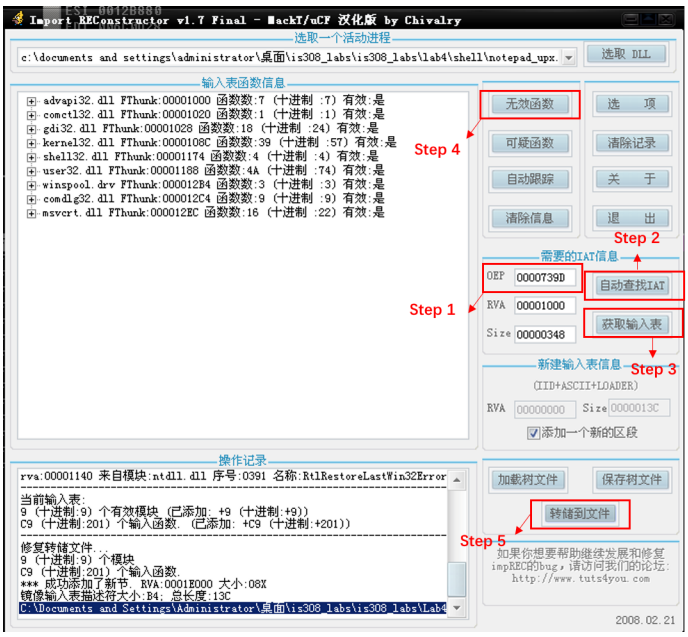

图 8: ImportREConstructor

- 使用 **PEID** 验证脱壳结果，我们发现该 exe 文件为 Microsoft Visual C++ v7.0，脱壳成功。结果见 9

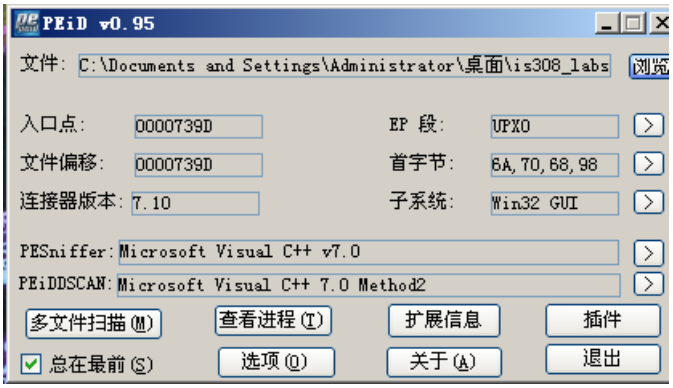

图 9: PEID check

使用其他方法破解 UPX 壳

单步追踪法

将程序拖入到 OD 中，一步一步单步跟踪执行，原则向上跳转不能让其实现，向下跳转可以实现。

- 当遇见向上跳转时，使用 F4 将代码运行到跳转的下一行；
- 当遇见 jmp 指令下面是 call 指令的情况，使用 F4 将代码运行到 call 指令的下一行；
- 当遇见 popad 出栈指令，在其后面不远处存在一个大的 jmp 指令，那么一般就跳转到了 OEP。

二次内存镜像法

- 在内存窗口中找到程序段中的第一个.rsrc， 然后使用 F2 下断点, 见 10。

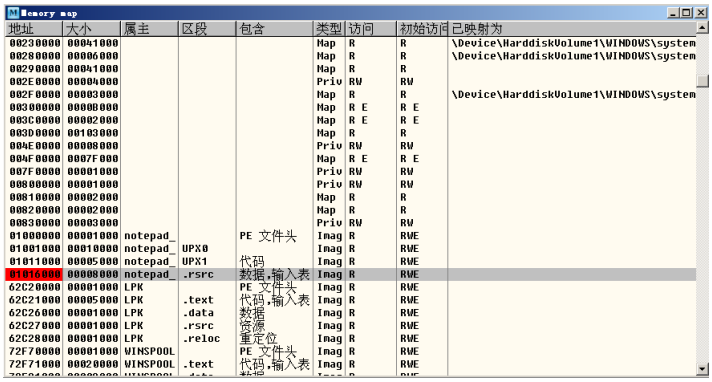

图 10: the first break point

- 然后使用 F9 运行，运行结果见 11。

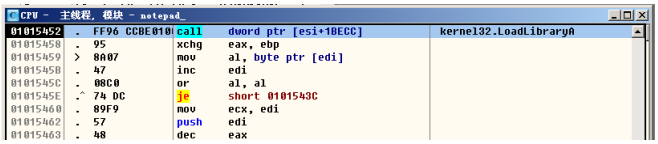

图 11: result of the first break point

- 在内存窗口中找到程序段中的 UPX0， 然后使用 F2 下断点, 见 12。

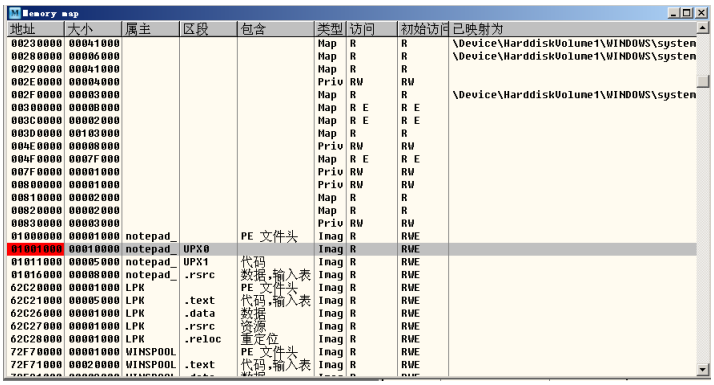

图 12: the second break point

- 然后使用 F9 运行，运行结果见 13。我们发现了一个大的 jmp 指令 (跳转至 OEP)

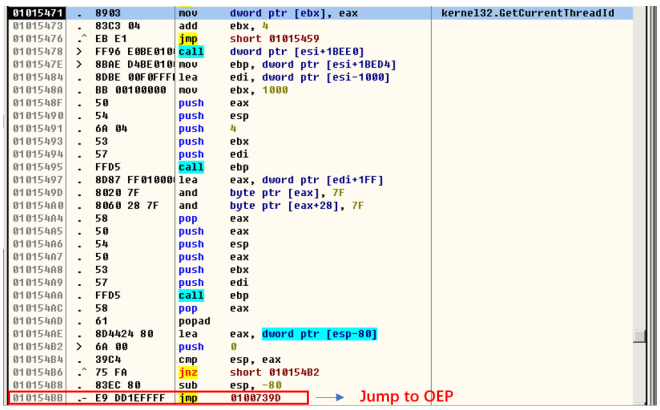

图 13: result of the second break point

一步直达法

我们在程序的开头见到有 pushad，那么肯定会有 popad，我们直接使用 ctrl+F 查询 popad 命令 (注意不要选取整个块) 即可, 结果见 14

图 14: search for popad