UTCTF逆向题详解

dotsu / 2019-03-16 01:11:00 / 浏览数 1444 安全技术 CTF 顶(0) 踩(0)

UTCTF是上周末国外的一个CTF比赛，逆向题中有几道质量还不错，简单整理了一下供大家参考。

Super Sucure Authentication

这是一道Java逆向题。Java逆向题在CTF里比较少见，主要是因为Java反编译太容易，没有太多trick。
其中考察比较多的有反射和动态加载类等，这道题就是使用动态加载类对代码进行了保护。

首先使用Jd-gui反编译Authenticator类，可以发现flag被分成了8份，并分别通过8个Verifier类进行检查（Verifier0 - Verifier7）：

```
if (!candidate.substring(0, 7).equals("utflag{")) {
 return false;
}
if (candidate.charAt(candidate.length() - 1) != '}') {
 return false;
}
StringTokenizer st = new StringTokenizer(candidate.substring(7, candidate.length() - 1), "_");
if (!Verifier0.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier1.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier2.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier3.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier4.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier5.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier6.verifyFlag(st.nextToken())) {
 return false;
}
if (!Verifier7.verifyFlag(st.nextToken())) {
 return false;
}
```

随便点开几个Verifier，发现逻辑都是一样的：

```
private static byte[] arr = jBaseZ85.decode(new String("+kO#^0000Q0ZE7[5DJ%U0u.ZH0S:wG0u.WG0S:CK00ifB2MU+E0v4*I...");
public static boolean verifyFlag(String paramString)
{
   Verifier0 localVerifier0 = new Verifier0();
   Class localClass = localVerifier0.defineClass("Verifier0", arr, 0, arr.length);
   Object localObject = localClass.getMethod("verifyFlag", new Class[] { String.class }).invoke(null, new Object[] { paramStri
   return ((Boolean)localObject).booleanValue();
}
```

可以看到这里使用了Java的动态加载类的方法，将一串常量字符串通过Base85解码，并加载为`Verifier0`类，并调用其中的`verifyFlag`函数。

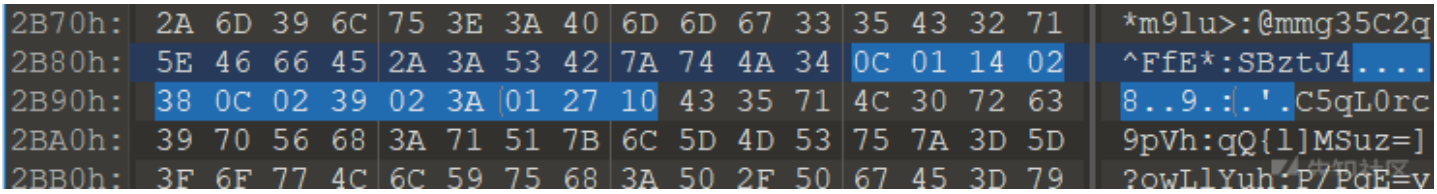
这里我们直接将代码在Java IDE中执行，发现Base85解码后得到的字符串开头就是Class文件头`CAFEBABE`。
将其保存到文件，然后用Jd-gui打开，发现代码跟上面基本一样，只是常量字符串发生了变化。由于文件有3MB之大，猜测之后还有很多层，于是需要写代码自动化脱壳。

这里我们需要做的就是从Class文件中提取出该字符串，使用Base85进行解码，然后再提取字符串，不断重复该过程。于是就需要从Class文件中提取字符串。

为了实现这个目标，我们可以考虑使用一些相关的库来Parse
Class文件，但对于这种简单的字符串提取，也可以研究一下文件结构，手动把字符串从Class文件中提取出来。

首先观察到字符串的开头都是相同的`+kO#`，对应了Java Class文件的文件头，这样我们就可以定位到字符串开头。
但是实际上最后的字符串是由多个字符串拼起来的，即类似于`new String("+kO#..") + new String("B2MU..") + ... + new`

`String("F9Kl..")`，体现在Class文件中就是两个字符串之间还有一段没有用的数据：



观察了一下可以发现，这段数据的长度是有规律的，基本上第一个间隔是13，后面的都是3，所以可以特判直接过滤掉。（我的特判写的比较丑陋就不放出来了，大家可以自

最后得到8个Verifier的class文件，都是简单的编码或者加密：

Verifier0，异或加密：

```
public class Verifier0
{
  private static byte[] encrypted = { 50, 48, 45, 50, 42, 39, 54, 49 };

  public static boolean verifyFlag(String paramString)
  {
    if (paramString.length() != encrypted.length) {
      return false;
    }
    for (int i = 0; i < encrypted.length; i++) {
      if (encrypted[i] != (paramString.charAt(i) ^ 0x42)) {
        return false;
      }
    }
    return true;
  }
}
```

Verifier1，字符串逆序：

```
public class Verifier1
{
  private static byte[] encrypted = { 115, 117, 111, 105, 120, 110, 97 };

  public static boolean verifyFlag(String paramString)
  {
    if (paramString.length() != encrypted.length) {
      return false;
    }
    for (int i = 0; i < encrypted.length; i++) {
      if (encrypted[i] != paramString.charAt(encrypted.length - 1 - i)) {
        return false;
      }
    }
    return true;
  }
}
```

Verifier2，hashCode，Java中爆破：

```
public class Verifier2
{
  private static int[] encrypted = { 3080674, 3110465, 3348793, 3408375, 3319002, 322962

  public static boolean verifyFlag(String paramString)
  {
    if (paramString.length() != encrypted.length) {
      return false;
    }
    for (int i = 0; i < encrypted.length; i++) {
      if (encrypted[i] != (paramString.substring(i, i + 1) + "foo").hashCode()) {
        return false;
      }
    }
    return true;
  }
}
```

```
private static int[] encrypted = { 3080674, 3110465, 3348793, 3408375, 3319002, 3229629, 3557330, 3229629, 3408375, 3378584 };

  public static void verifyFlag()
  {
      for(int i = 0; i < 10; i++)
      {
          for(char c = 32; c < 127; c++)
              if (encrypted[i] == (c + "foo").hashCode()) {
                  System.out.print(c);
              }
      }
  }
```

Verifier3，凯撒移位：

```
public class Verifier3
{
  private static String encrypted = "obwaohfcbwq";

  public static boolean verifyFlag(String paramString)
  {
    if (paramString.length() != encrypted.length()) {
      return false;
    }
    for (int i = 0; i < encrypted.length(); i++)
    {
      if (!Character.isLowerCase(paramString.charAt(i))) {
        return false;
      }
      if ((encrypted.charAt(i) - 'a' + 12) % 26 != paramString.charAt(i) - 'a') {
        return false;
      }
    }
    return true;
  }
}
```

Verifier4，简单数字运算：

```
public class Verifier4
{
  private static int[] encrypted = { 3376, 3295, 3646, 3187, 3484, 3268 };

  public static boolean verifyFlag(String paramString)
  {
    if (paramString.length() != encrypted.length) {
      return false;
    }
    for (int i = 0; i < encrypted.length; i++) {
      if (encrypted[i] != paramString.charAt(i) * '\033' + 568) {
        return false;
      }
    }
    return true;
  }
}
```

Verifier5，MD5，直接查cmd5

```
import java.security.MessageDigest;
import javax.xml.bind.DatatypeConverter;

public class Verifier5
{
  private static String encrypted = "8FA14CDD754F91CC6554C9E71929CCE7865C0C0B4AB0E063E5CAA3

  public static boolean verifyFlag(String paramString)
  {
    try
    {
      MessageDigest localMessageDigest = MessageDigest.getInstance("MD5");

      String str = "";
      for (int k : paramString.toCharArray())
      {
        localMessageDigest.update((byte)k);

        str = str + DatatypeConverter.printHexBinary(localMessageDigest.digest());
      }
      return str.equals(encrypted);
    }
    catch (Exception localException) {}
    return false;
  }
}
```

Verifier6，SHA1，同上

```java
import java.security.MessageDigest;
import javax.xml.bind.DatatypeConverter;

public class Verifier6
{
  private static String hash = "1B480158E1F30E0B6CEE7813E9ECF094BD6B3745";

  public static boolean verifyFlag(String paramString)
  {
    if (paramString.length() != 4) {
      return false;
    }
    try
    {
      MessageDigest localMessageDigest = MessageDigest.getInstance("SHA1");

      localMessageDigest.update(paramString.getBytes());

      String str = DatatypeConverter.printHexBinary(localMessageDigest.digest());

      return str.equals(hash);
    }
    catch (Exception localException) {}
    return false;
  }
}
```

Verifier7，flag直接送了

```java
public class Verifier7
{
  private static String flag = "goodbye";

  public static boolean verifyFlag(String paramString)
  {
    return paramString.equals(flag);
  }
}
```

连接起来，得到完整flag:
`utflag{prophets_anxious_demolition_animatronic_herald_fizz_stop_goodbye}`

Simple python script

给出了python源代码：

```
flag = input("> ")
for i in range(0, len(flag), int((544+5j).imag)):
   inputs = []
   (lambda ■■, _, __, ___, ____, _____, ■■, q, k: getattr(■■, "extend")...
   temp = getattr(__import__("ha"+"".__class__.__name__[0]+"hl"+(3)...
   getattr(temp, "update")(getattr(flag[i:i + 5], "encode")("utf-8"))
   if getattr(__import__("difflib"), "SequenceMatcher")(None, getattr(getattr(temp, "hexdigest")(), "lower")(), getattr(inputs
       exit()

print("correct")
```

可以看到使用了多种混淆方式：

1. `getattr(class, method)`就相当于`class.method`
2. `"".__class__.__name__[0]`那些用来隐藏字符串（即"String"[0]，"S"）
3. 最后的`SequenceMatcher(...).ratio() == 1`其实就是比较相等

于是整理出以下代码：

```
inputs = []
(lambda ... # ■■■■■■■■inputs
temp = hashlib.new(... # ■■■■■■■■■■■hashlib■■■■■
temp.update(flag[i:i + 5].encode("utf-8") ) # ■flag■5■■■■■■■
```

```
print(inputs[i // 5].decode("utf-8").lower()) # ■■■■■■■■■■■■■■■flag■■■■■■■■■inputs[i // 5]■■■■■■■
if temp.hexdigest().lower() == inputs[i // 5].decode("utf-8").lower(): # ■■■■!=■■■■■==■■exit
   exit()
```

运行可以打印出几个哈希值：

```
26d33687bdb491480087ce1096c80329aaacbec7
1c3bcf656687cd31a3b486f6d5936c4b76a5d749
11a3e059c6f9c223ce20ef98290f0109f10b2ac6
6301cb033554bf0e424f7862efcc9d644df8678d
95d79f53b52da1408cc79d83f445224a58355b13
```

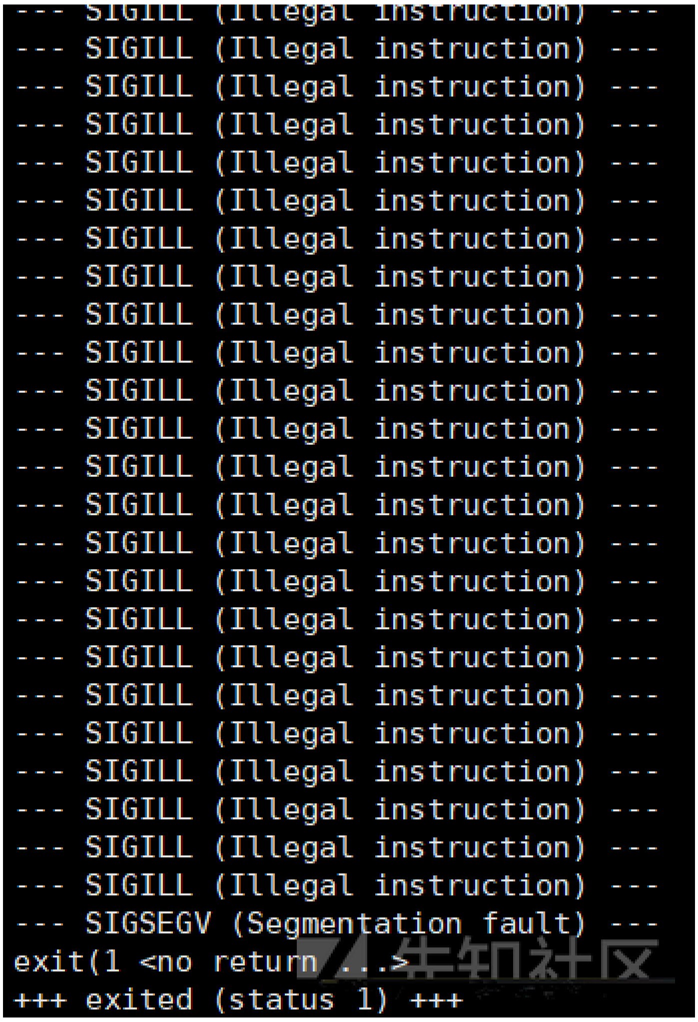
在CMD5上可以查到其中一部分，哈希算法是SHA1，剩余的可以在hashtoolkit上查到，连接得到完整flag：
`puppyp1zzaanimetoruskitty`

MOV

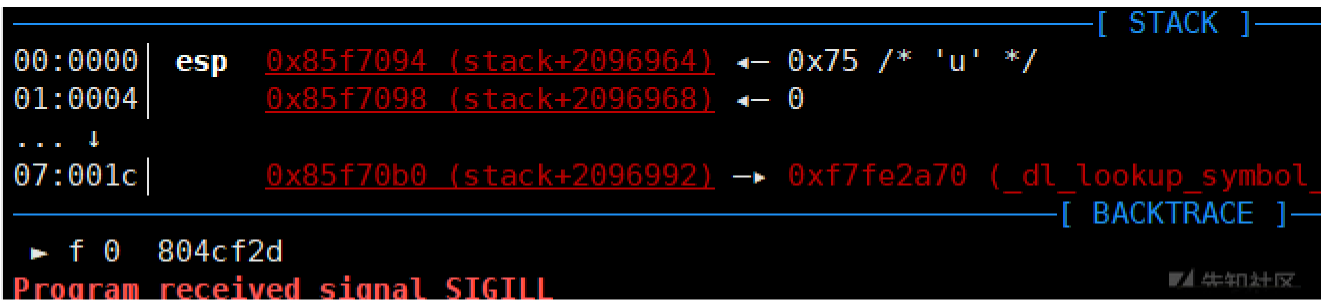
IDA加载发现全是MOV指令，可以大概知道使用了Movfuscator进行了混淆处理。

对于复杂一些的Movfuscator程序，可以尝试根据程序中字符串等信息，配合trace工具和下断点来追踪程序流程，并猜测程序逻辑（一般来说逻辑不会特别复杂）。此外，

不过对于这道题，程序中搜不到字符串信息，运行也没有反应，在strace和ltrace时发现很多SIGILL信号：



于是尝试使用gdb进行调试，在发出SIGILL信号时，gdb会断住，这是可以观察到栈顶有一个u字符：



继续跟踪下去，发现每次SIGILL时栈顶都会添加一个字符，逐渐形成一个完整的flag：

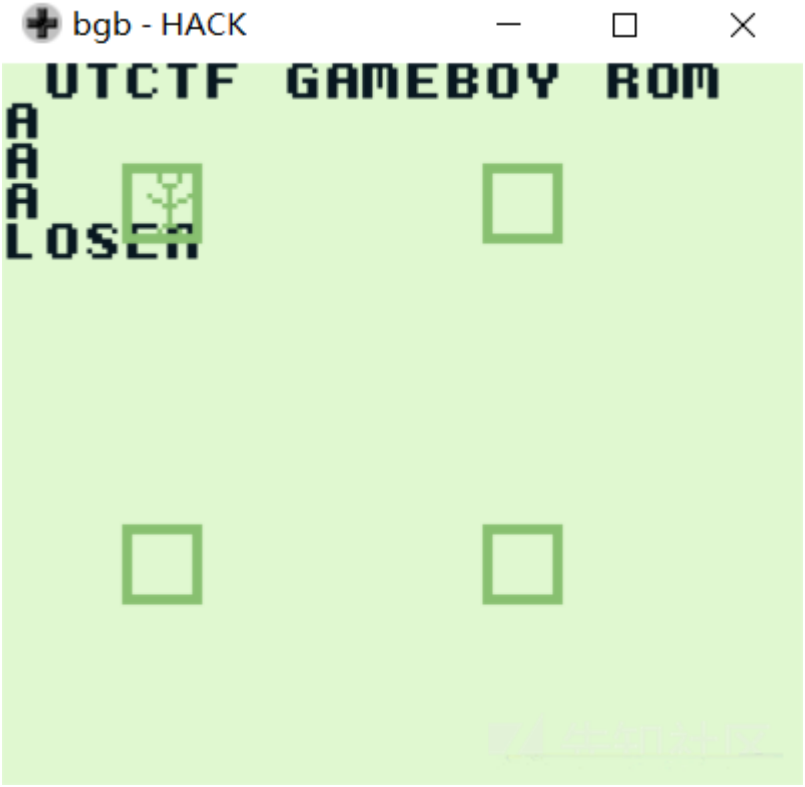
```
Program received signal SIGILL
pwndbg> xxd $esp 0x50
00000000: 7574 666c 6167 7b73 656e 7465 6e63 655f  utflag{sentence_
00000010: 7468 6174 5f69 735f 736f 6d65 7768 6174  that_is_somewhat
00000020: 5f74 616e 6765 6e74 6961 6c6c 795f 7265  _tangentially_re
00000030: 6c61 7465 645f 746f 5f74 68f7 0100 0000  lated_to_th.....
00000040: 0100 0000 0000 0000 3b00 0000 1a19 0b31  ........;.......1
```

```
utflag{sentence_that_is_somewhat_tangentially_related_to_the_challenge}
```

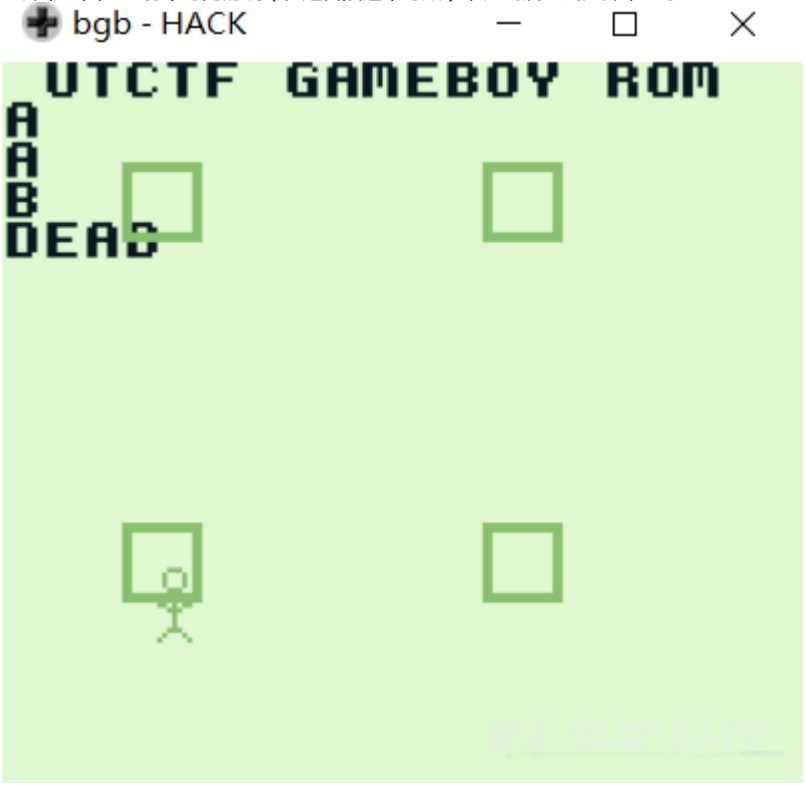
UTCTF adventure ROM

一道gameboy游戏逆向题，使用的工具是bgb（可以debug，非常方便）和IDA。

首先使用bgb运行游戏，可以看到有四个框，分别可以输入ABCD，输入错误会死掉，显示LOSER：



此外，地图上还有不可见的线（在题目描述中可知），碰到后也会死掉，显示DEAD：

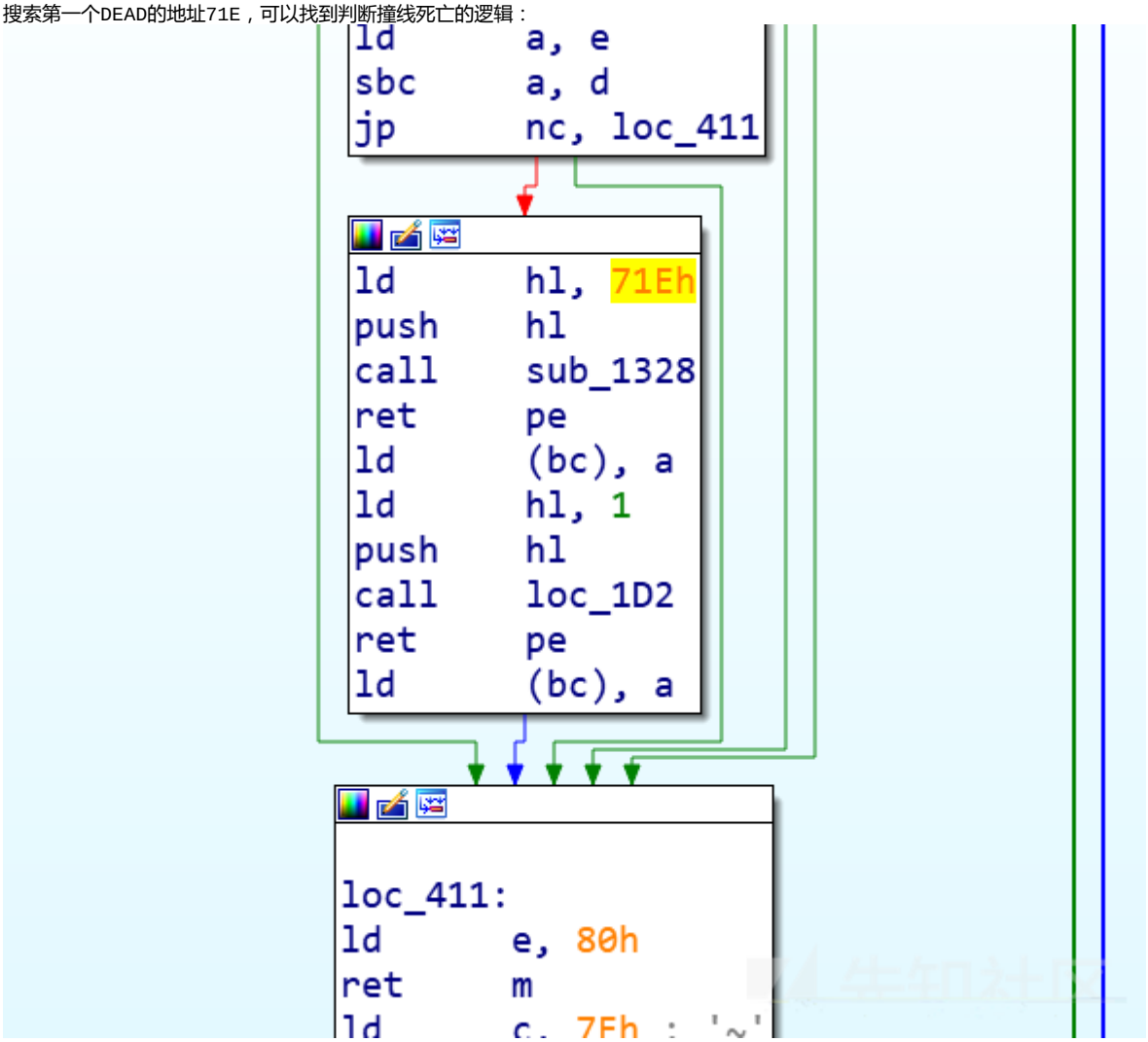


大体了解游戏逻辑后，我们就可以开始逆向了。在IDA中打开，处理器选择z80（具体可以参考这份wp）。

首先搜索字符串，可以找到LOSER和DEAD：

```
ROM:06F3
ROM:06F6 ; ---------------------------------------------------------------
ROM:06F6                 ret     pe
ROM:06F7                 ld      d, e
ROM:06F8                 ret
ROM:06F8 ; ---------------------------------------------------------------
ROM:06F9 aAaaabbbbccccdd:.ascii 'AAAABBBBCCCCDDDD',0
ROM:070A aUtctfGameboyRo:.ascii ' UTCTF GAMEBOY ROM\n',0
ROM:071E aDead:          .ascii 'DEAD\n',0
ROM:0724 aDead_0:        .ascii 'DEAD\n',0
ROM:072A aDead_1:        .ascii 'DEAD\n',0
ROM:0730 aC:             .ascii '%c\n',0
ROM:0734 aLoser:         .ascii 'LOSER\n',0
ROM:073B
```

搜索第一个DEAD的地址71E，可以找到判断撞线死亡的逻辑：

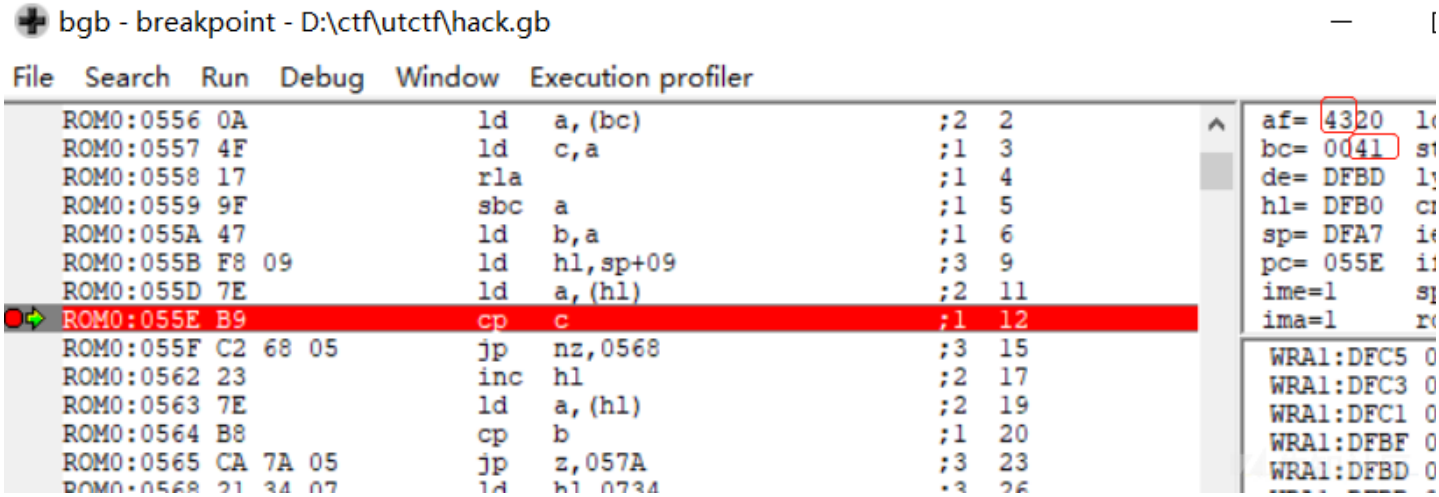


这里我们直接把这部分nop掉就不会再撞死了（注意这里的nop是\x00）

同样的方法搜索LOSER的地址568：

```
ROM:055C                add     hl, bc
ROM:055D                ld      a, (hl)
ROM:055E                cp      c
ROM:055F                jp      nz, loc_568
ROM:0562                inc     hl
ROM:0563                ld      a, (hl)
ROM:0564                cp      b
ROM:0565                jp      z, loc_57A
ROM:0568
ROM:0568 loc_568:                               ; CODE XREF: sub_33E+221↑j
ROM:0568                ld      hl, 734h
ROM:056B                push    hl
ROM:056C                call    sub_1328
```

找到了判断输入是否正确的判断，于是我们使用bgb在这里下断点：



可以看到我们的输入和正确值分别保存在a和c寄存器中。
于是我们就可以反复运行，随便输入一个值，然后修改我们的输入值为正确值，并记录下来，即可获得完整flag：
AABDCACBBDBCDCAD

其他

剩下的几道题比较简单，有问题可以留言交流

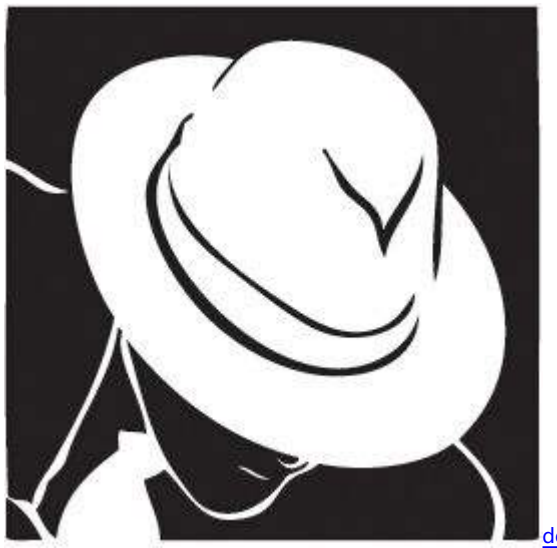
上一篇：MWeb For Mac 客户端从... 下一篇：Ruby on Rails 路径穿...

1. 4 条回复


snow146 2019-03-17 14:31:28

太强了，大佬，有没有Domain Generation Algorithm的wp

0 回复Ta


dotsu 2019-03-17 22:02:37

@snow146
这道好像是用PyInstaller打包的，可以用PyInstaller里的pyi-archive_viewer解包试试。另外官方也放了源码：https://github.com/UTISSS/UTCTF/

0 回复Ta


snow146 2019-03-25 21:49:33

我又来了，为什么我的ida处理器选择z80后打开的跟你的不一样，依旧是binary file，是我的ida缺少什么东西吗？

0 回复Ta


dotsu 2019-03-27 15:39:20

@snow146 IDA里edit->select
all，然后按C键选force，这样会强制解析整个文件，但是里面的数据和字符串也会被解析为指令，可以对照着hex窗口，按A转为字符串或者按D转为数据。

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板