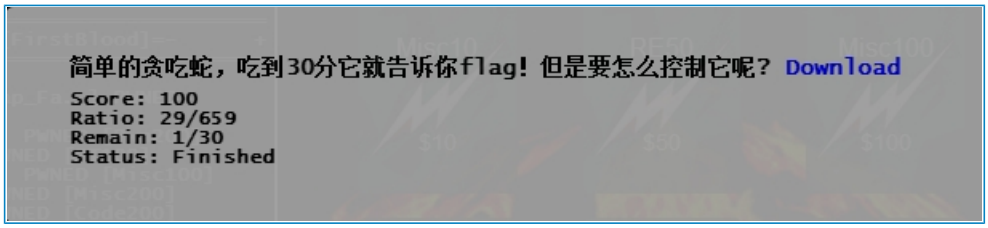
SCTF-2014 misc100 writeup（赛后分析）

转载

weixin_34255055 于 2014-12-13 15:31:29 发布 203 收藏
文章标签： c/c++
原文链接：http://blog.51cto.com/cugou/1589577



下载文件后，file命令看一下是ELF32程序，strings命令发现程序被UPX加壳了。

```
upx -d snake-final.exe
```

脱壳后扔到IDA里面分析，主函数发现调用signal注册了若干个回调函数：

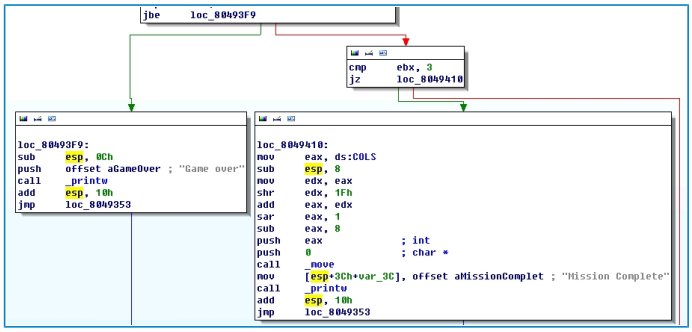
```
.text:08048C40                  lea     ecx, [esp+4]
.text:08048C44                  and     esp, 0FFFFFFF0h
.text:08048C47                  push    dword ptr [ecx-4]
.text:08048C4A                  push    ebp
.text:08048C4B                  mov     ebp, esp
.text:08048C4D                  push    ecx
.text:08048C4E                  sub     esp, 4
.text:08048C51                  call    _initscr
.text:08048C56                  call    sub_8048EA0
.text:08048C5B                  sub     esp, 8
.text:08048C5E                  push    offset handler
.text:08048C63                  push    0Eh
.text:08048C65                  call    _signal
.text:08048C6A                  pop     eax
.text:08048C6B                  pop     edx
.text:08048C6C                  push    offset loc_8048E10
.text:08048C71                  push    38h
.text:08048C73                  call    _signal
.text:08048C78                  pop     ecx
.text:08048C79                  pop     eax
.text:08048C7A                  push    offset loc_8048E70
.text:08048C7F                  push    32h
.text:08048C81                  call    _signal
.text:08048C86                  pop     eax
.text:08048C87                  pop     edx
.text:08048C88                  push    offset nullsub_1
.text:08048C8D                  push    5
.text:08048C8F                  call    _signal
.text:08048C94                  pop     ecx
.text:08048C95                  pop     eax
.text:08048C96                  push    offset loc_8048E10
.text:08048C9B                  push    0Ah
.text:08048C9D                  call    _signal
```

特别是几个38h、32h、34h、36h。实际上是定义了4个控制贪吃蛇行动的游戏按键。但显然就这个程序，按对应的键是产生不出对应的signal信号的。（有队伍使用向进程发送对应signal的方式，间接操控贪吃蛇，游戏成功可以获得flag）。经过进一步分析，发现最重要的回调函数是handler，同时在handler中存在"int 3"反调试，正常执行的时候会产生signal==5的信号，这里的处理函数为nullsub_1。实际上就是忽略，所以我们也可以直接 nop 掉 int 3。当然完全按照本文的静态分析方法，而不使用调试，完全可以忽略 int 3。

进一步分析handler，发现是一个极其复杂的函数。没心思看，找找有没有别的入口。

strings窗口发现"Mission Complete"，xref发现这个字符串引用位置sub_80492E0+14C。

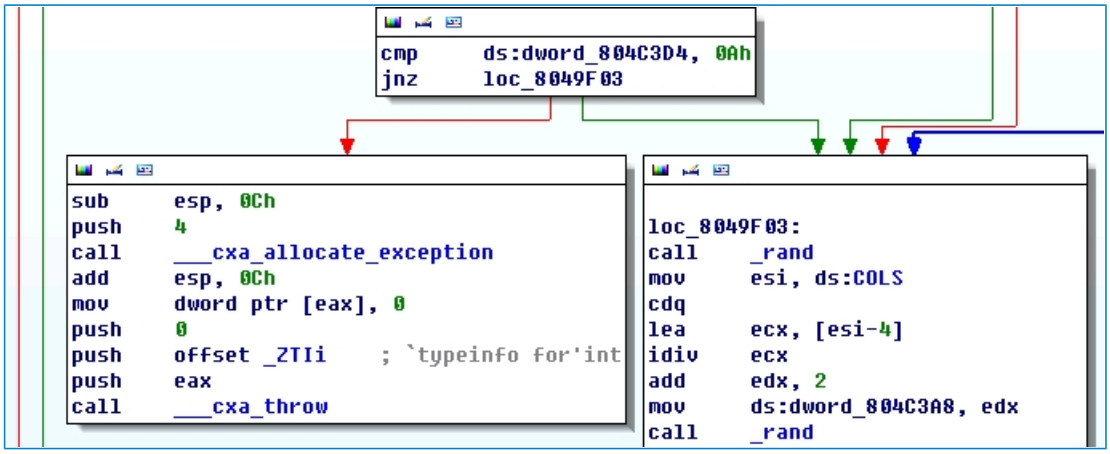
分析sub_80492E0：



这个函数确实是判断游戏成功与否的标志，成功的话，还需要判断一个ebx == 3，这里的ebx实际上是函数的传入参数。但问题是，找到成功的标志，但是后面没有跟着输出flag的地方，而仅仅是调用settimer重置了新的信号量。这样程序的执行流又回到handler里面。

根据目前的信息，寻找带参数3调用sub_80492E0的地方，xref只发现两处8049941、8049FF1。实际上这两处的代码是一样的（经过最后的分析实际上这里往后的代码就是输出flag的地方，但我在这里迷失了很久）。比较奇怪的是这两处都不在IDA识别的handle函数的作用域中。原因是IDA识别出了handler函数的Canary保护机制，以此作为handler函数的开始和结束。

进一步分析handler发现：



原来成功吃到30个食物成功后，是通过C++异常处理机制来调用sub_80492E0(3)的。

通过上面查找xref的两个地方，随便选取第二个，看到调用代码：

```
.text:08049FE1                  sub     esp, 0Ch
.text:08049FE4                  push    eax
.text:08049FE5                  call    ___cxa_begin_catch
.text:08049FEA                  mov     dword ptr [esp], 3
.text:08049FF1                  call    sub_80492E0
.text:08049FF6                  mov     dword ptr [esp], 4
.text:08049FFD                  call    ___cxa_allocate_exception
.text:0804A002                  add     esp, 0Ch
.text:0804A005                  mov     dword ptr [eax], 1
.text:0804A00B                  push    0
.text:0804A00D                  push    offset _ZTIi    ; `typeinfo for'int
.text:0804A012                  push    eax
.text:0804A013                  call    ___cxa_throw
```

由于sub_80492E0里面没有显示flag的地方，只是重置settimer，那么猜测很有可能flag也是通过异常处理链来打印的。上面的代码发现，调用sub_80492E0后，重新抛出了一个异常。所以继续观察下面的代码：

```
.text:0804A018                  mov     ebx, eax
.text:0804A01A
.text:0804A01A loc_804A01A:                             ; CODE XREF: .text:0804A125↓j
.text:0804A01A                  call    ___cxa_end_catch
.text:0804A01F                  sub     esp, 0Ch
.text:0804A022                  push    ebx
.text:0804A023                  call    __Unwind_Resume
.text:0804A028 ; ---------------------------------------------------------------------
.text:0804A028                  sub     edx, 1
.text:0804A02B                  jnz     short loc_804A018
.text:0804A02D                  sub     esp, 0Ch
.text:0804A030                  push    eax
.text:0804A031                  call    ___cxa_begin_catch
.text:0804A036                  add     esp, 10h
.text:0804A039                  cmp     ds:dword_804C3DC, 3Bh
.text:0804A040                  jg      short loc_804A051
.text:0804A042
.text:0804A042 loc_804A042:                             ; CODE XREF: .text:0804A156↓j
.text:0804A042                  call    ___cxa_end_catch
.text:0804A047                  call    ___cxa_end_catch
.text:0804A04C                  jmp     loc_8049B87
```

这里注意804A039处的一个比较，这是在打印flag前的异常处理链中的最后一道门槛。也是调试时候为什么sub_80492E0已经运行到"Mission Complete"，但还是没有出现flag的原因。

```
.text:0804A051
.text:0804A051 loc_804A051:                             ; CODE XREF: .text:0804A040↑j
.text:0804A051                  call    ___cxa_end_catch
.text:0804A056                  sub     esp, 0Ch
.text:0804A059                  push    4
.text:0804A05B                  call    ___cxa_allocate_exception
.text:0804A060                  add     esp, 0Ch
.text:0804A063                  mov     dword ptr [eax], 2Ah
.text:0804A069                  push    0
.text:0804A06B                  push    offset _ZTIi    ; `typeinfo for'int
.text:0804A070                  push    eax
.text:0804A071                  call    ___cxa_throw
```

```
.text:0804A076                  sub     edx, 1
.text:0804A079                  jnz     short loc_804A018
.text:0804A07B                  sub     esp, 0Ch
.text:0804A07E                  push    eax
.text:0804A07F                  call    ___cxa_begin_catch
.text:0804A084                  mov     ebx, [eax]
.text:0804A086                  mov     ecx, 2
.text:0804A08B                  mov     dword ptr [ebp-50h], 7Fh
.text:0804A092                  pop     eax
.text:0804A093                  mov     eax, ds:COLS
.text:0804A098                  mov     dword ptr [ebp-4Ch], 1Ah
.text:0804A09F                  pop     edx
.text:0804A0A0                  mov     dword ptr [ebp-48h], 64h
.text:0804A0A7                  mov     dword ptr [ebp-44h], 7Fh
.text:0804A0AE                  sub     eax, 20h
.text:0804A0B1                  mov     dword ptr [ebp-40h], 78h
.text:0804A0B8                  mov     dword ptr [ebp-3Ch], 44h
.text:0804A0BF                  cdq
.text:0804A0C0                  mov     dword ptr [ebp-38h], 5Eh
.text:0804A0C7                  mov     dword ptr [ebp-34h], 50h
.text:0804A0CE                  idiv    ecx
.text:0804A0D0                  mov     dword ptr [ebp-30h], 67h
.text:0804A0D7                  mov     dword ptr [ebp-2Ch], 7Dh
.text:0804A0DE                  mov     dword ptr [ebp-28h], 4Eh
.text:0804A0E5                  mov     dword ptr [ebp-24h], 5Fh
.text:0804A0EC                  mov     dword ptr [ebp-20h], 2Ah
.text:0804A0F3                  push    eax
.text:0804A0F4                  push    1
.text:0804A0F6                  call    _move
.text:0804A0FB                  lea     esi, [ebp-50h]
.text:0804A0FE                  add     esp, 10h
```

```
.text:0804A101 loc_804A101:                            ; CODE XREF: .text:0804A11C↓j
.text:0804A101                 mov     eax, [esi]
.text:0804A103                 cmp     ebx, eax
.text:0804A105                 jz      short loc_804A12A
.text:0804A107                 xor     eax, ebx
.text:0804A109                 push    edi
.text:0804A10A                 push    edi
.text:0804A10B                 push    eax
.text:0804A10C                 push    offset aC       ; "%c"
.text:0804A111                 call    _printw
.text:0804A116                 add     esp, 10h
.text:0804A119                 add     esi, 4
.text:0804A11C                 jmp     short loc_804A101
```

注意这里的一个循环，实际上就是printf "[ebp-50+i] xor 2Ah"。(2Ah是ascii的'*')

```
import sys
a='\x7f\x1a\x64\7f\x78\x44\x5e\x50\x67\x7d\x4e\x5f'
for i in range(0, len(a)):
    sys.stdout.write(chr(ord(a[i]) ^ ord('*')))
```

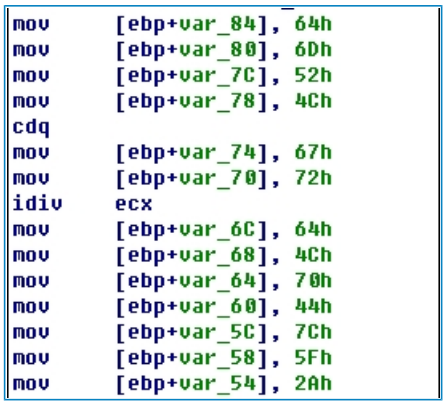
得到：U0N-LRntzMWdu

这个还没有完，继续往下看：

```
.text:0804A12A loc_804A12A:                            ; CODE XREF: .text:0804A105↑j
.text:0804A12A                 lea     esi, [ebp-84h]
.text:0804A130
.text:0804A130 loc_804A130:                            ; CODE XREF: .text:0804A14B↓j
.text:0804A130                 mov     eax, [esi]
.text:0804A132                 cmp     ebx, eax
.text:0804A134                 jz      short loc_804A14D
.text:0804A136                 xor     eax, ebx
.text:0804A138                 push    ecx
.text:0804A139                 push    ecx
.text:0804A13A                 push    eax
.text:0804A13B                 push    offset aC       ; "%c"
.text:0804A140                 call    _printw
.text:0804A145                 add     esp, 10h
.text:0804A148                 add     esi, 4
.text:0804A14B                 jmp     short loc_804A130
```

```
.text:0804A14D loc_804A14D:                            ; CODE XREF: .text:0804A134↑j
.text:0804A14D                 mov     esi, offset unk_804C0C0
.text:0804A152
.text:0804A152 loc_804A152:                            ; CODE XREF: .text:0804A171↓j
.text:0804A152                 mov     eax, [esi]
.text:0804A154                 cmp     ebx, eax
.text:0804A156                 jz      loc_804A042
.text:0804A15C                 xor     eax, ebx
.text:0804A15E                 push    edx
.text:0804A15F                 push    edx
.text:0804A160                 push    eax
.text:0804A161                 push    offset aC       ; "%c"
.text:0804A166                 call    _printw
.text:0804A16B                 add     esp, 10h
.text:0804A16E                 add     esi, 4
.text:0804A171                 jmp     short loc_804A152
```

可见flag还有两部分，其中[ebp-84h]就是handler函数开始处的值（说明异常处理实际上还是应该在handler当中的，只是IDA分析的使用因为canary的原因，被排除了）：



最后一部分的数据是804C0C0处的全局数据，三部分的算法是一样的，最后解码得到：

U0NURntzMWduNGxfMXNfZnVubnk6KX0=

```
echo
U0NURntzMWduNGxfMXNfZnVubnk6KX0= | base64 -d
```

得到flag。

后记：

此题看360首发的sctf writeup上，只讲了"Mission Complete"和sub_80492E0，然后就直接base64字符串了，不知道那位大神是如何“看”出来的。

C++的throw--catch的逆向分析是头一回遇到，还是应该写一个简单的C++相应程序，然后逆向研究一下它的结构。这样才知道handler里面canary和异常处理是人为的有意为之，还是编译器本来就是这样处理的。

由于信号量的处理有线程再入的特点，异常也会涉及到跨线程的问题，所以程序流程要完全研究透彻，需要理解的内容非常多。有待进一步学习。

这个题目如果调试，如果跟踪到异常处理的代码，还请大牛不吝赐教。

转载于:https://blog.51cto.com/cugou/1589577