




强网杯 - JustRe - writeup

原创

咕君  于 2019-05-29 15:28:34 发布  970  收藏 1

分类专栏: [CTF](#) 文章标签: [Reverse SMC](#) [反调试](#) [CTF](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: https://blog.csdn.net/weixin_42151611/article/details/90671904

版权



[CTF 专栏收录该内容](#)

16 篇文章 0 订阅

订阅专栏

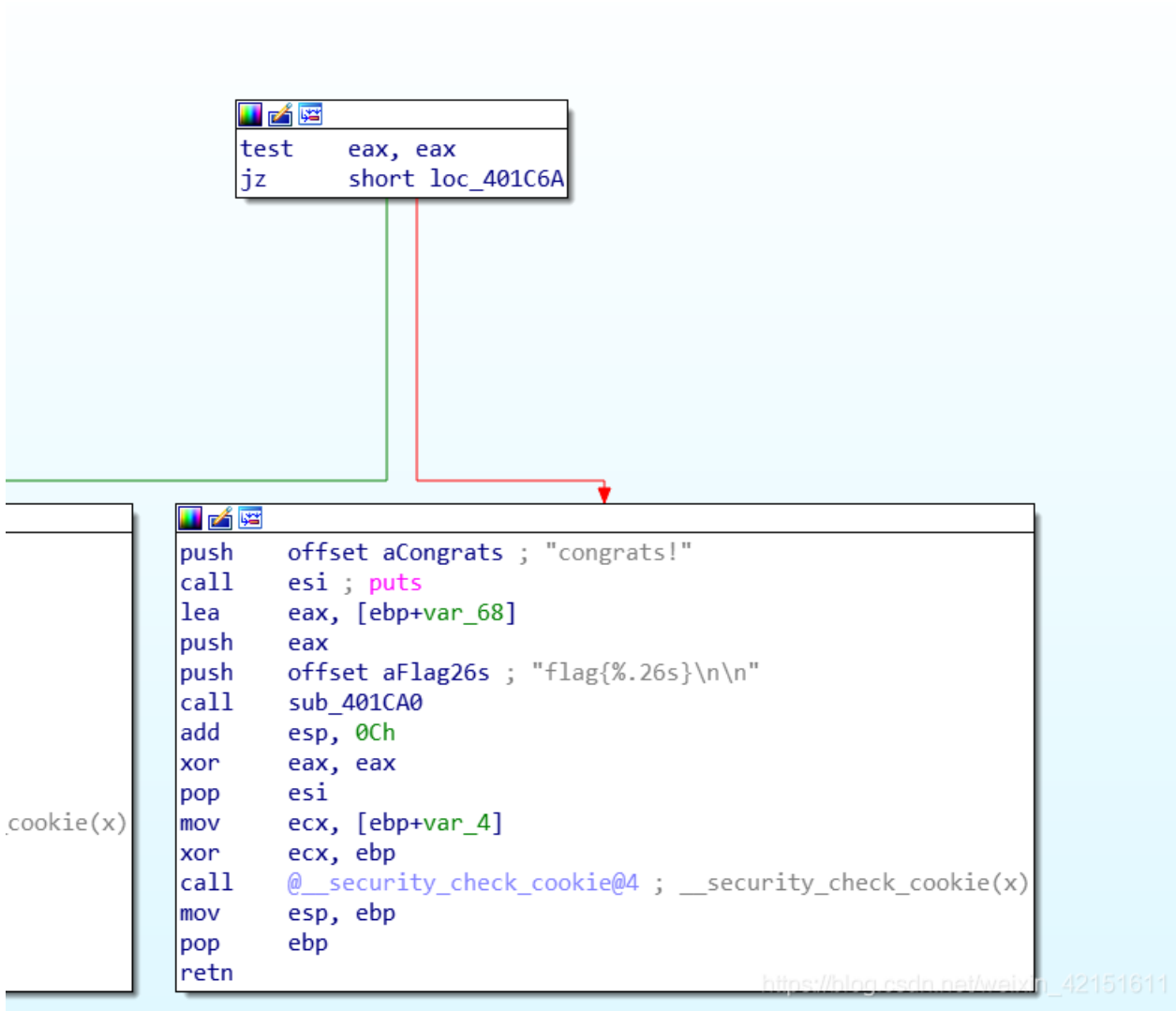
JustRe

[文件链接 -> Github](#)

初步分析

使用32位的IDA打开文件

按 `shift+F12` 搜索字符串, 找到了带有 `flag{%.26s}` 字样的字符串。根据交叉引用找到该字符串被引用的代码段



发现 `test eax, eax` 这句代码上方的代码IDA识别不出来。按下 `[space]`，查看上方的代码是什么样的

```

.text:00401C29      lea    ecx, [ebp+var_68]
.text:00401C2C      call   sub_401610
.text:00401C31      test   eax, eax
.text:00401C33      jz     short loc_401C6A
.text:00401C35      lea    ecx, [ebp+var_68]
.text:00401C38      call   sub_4018A0
.text:00401C3D      ; -----
.text:00401C3D      test   eax, eax
.text:00401C3F      jz     short loc_401C6A
.text:00401C41      push   offset aCongrats ; "congrats!"
.text:00401C46      call   esi ; puts
.text:00401C48      lea    eax, [ebp+var_68]
.text:00401C4B      push   eax
.text:00401C4C      push   offset aFlag26s ; "flag{%.26s}\n\n"
.text:00401C51      call   sub_401CA0
.text:00401C56      add    esp, 0Ch
.text:00401C59      xor    eax, eax

```

我们发现它调用了这个函数，跟进去看一下

```

.text:004018A0
.text:004018A0 sub_4018A0      proc near          ; CODE XREF: sub_401BD0+68↓p
.text:004018A0                                     ; DATA XREF: sub_401610+25E↑o
.text:004018A0      sbb     edx, [edi-3F74814Ah]
.text:004018A6      xchg   eax, ebx
.text:004018A7      in     eax, dx
.text:004018A8      lock or eax, 14C80112h
.text:004018AE      bound eax, [ecx+6Ah]
.text:004018B1      dec    ebp
.text:004018B3      mov    bh, ah
.text:004018B5      hlt
.text:004018B5 sub_4018A0      endp
.text:004018B5 ; -----
.text:004018B6      dw    0FF3Dh
.text:004018B8      dd    0F119106Ch, 41537868h, 40D2706Eh, 0A6010AFh, 2CEE5481h
.text:004018B8      dd    5B19BFC2h, 4153886Fh, 0B3193273h, 0A51F4B0h, 671C5185h
.text:004018B8      dd    254E98B6h, 258E9DB3h, 1121164h, 0A6210CFh, 0FCEE548Bh
.text:004018B8      dd    1196853h, 8C52367Ch, 0FF59F3F9h, 6A0000h, 0F47E850h
.text:004018B8      dd    748D0000h, 0C4834824h, 14E8D18h, 8446068Ah, 2BF975C0h
.text:004018B8      dd    24448DF1h, 8D505630h, 1E82484h, 0E8500000h, 0FC3h, 18B8h
.text:004018B8      dd    50C62B00h, 0F024848Dh, 3000001h, 50006AC6h, 0F08E8h, 42151611

```

发现端倪了，这里使用了 SMC 技术。那么返回，查看上方的函数调用

经过分析，得知 `sub_401CE0` 的作用是获取输入；`sub_401610` 的作用就是对函数 `sub_4018A0` 进行解密

在 `sub_401610` 中看到这两个调用

```

.text:00401855      cmp    eax, 0FFFFFFFh
.text:0040185E      jb     short loc_401864
.text:00401860 loc_401860:      ; CODE XREF: sub_401610+242↑j
.text:00401860      xor    eax, eax
.text:00401862      mov    [eax], eax
.text:00401864 loc_401864:      ; CODE XREF: sub_401610+24E↑j
.text:00401864      popa
.text:00401865      push  0          ; lpNumberOfBytesWritten
.text:00401867      push  60h        ; nSize
.text:00401869      push  offset xmmword_405018 ; lpBuffer
.text:0040186E      push  offset sub_4018A0 ; lpBaseAddress
.text:00401873      call  ds:GetCurrentProcess
.text:00401879      push  eax        ; hProcess
.text:0040187A      call  ds:WriteProcessMemory
.text:00401880      mov    eax, 1
.text:00401885      pop    edi
.text:00401886      pop    esi
.text:00401887      mov    esp, ebp
.text:00401889      pop    ebp
.text:0040188A      mov    esp, ebx
.text:0040188C      pop    ebx
.text:0040188D      retn

```

https://blog.csdn.net/weixin_42151611

`GetCurrentProcess` 的作用是获取当前进程的一个伪句柄

`WriteProcessMemory` 的作用就是向进程的指定偏移写入内容

在这两个调用的上方就是解密的过程，其内使用了大量的 `xmm` 寄存器，IDA与OD都不能很好地识别这些指令，于是就转战 x64debug

分析SMC解密函数

在 `401C2C` 处下断点，根据 `ECX` 的值得到了输入在内存中的位置，同时得知输入将会在 `sub_401610` 被使用

00401C29	8D4D 98	lea ecx,dword ptr ss:[ebp-08]	EBX	00303000
→ 00401C2C	E8 DFF9FFFF	call justre.401610	ECX	0019FED0 "1234567890"
00401C31	85C0	test eax,eax	EDX	00000001
00401C33	74 35	je justre.401C6A	EBP	0019FF38
00401C35	8D4D 98	lea ecx,dword ptr ss:[ebp-68]	ESP	0019FECC
00401C38	E8 63FCFFFF	call justre.4018A0	ESI	76802DA0 <ucrtbase.puts>
断点未设置	85C0	test eax,eax	EDI	00000000

跟进函数一步步查看，就得到了函数的逻辑：

将输入的前八位转换成数字，比如输入为“12345678”，就将它转换为 0x12345678，将其看作Num_0

将转换后的数字扩展到128位，比如将 0x12345678 拓展为 0x12345678123456781234567812345678，将其看作Num_1

将输入的第 9、10 位转换成数字，比如“90”转换成 0x90

将转换后的数字扩展到128位，比如 0x90 拓展为 0x90909090909090909090909090909090，将其看作Num_2

在 `403040` 有一个表，表中有总共 128 * 4 字节的数据，将其看作数组A，共四项，每项128字节

00404340	00 00 00 00	01 00 00 00	02 00 00 00	03 00 00 00
00404350	04 00 00 00	04 00 00 00	04 00 00 00	04 00 00 00
00404360	08 00 00 00	08 00 00 00	08 00 00 00	08 00 00 00
00404370	0C 00 00 00	0C 00 00 00	0C 00 00 00	0C 00 00 00

6. `405018` 有一个表，该表有总共 128 * 6 字节的数据，将其看作数组B，共六项，每项128字节，其中每项也是一个数组，该数组每项32字节

00405018	1B 97 B6 7E	8B C0 93 ED	F0 0D 12 01	C8 14 62 41
00405028	6A FF CD 88	E7 F4 3D FF	6C 10 19 F1	68 78 53 41
00405038	6E 70 D2 40	AF 10 60 0A	81 54 EE 2C	C2 BF 19 5B
00405048	6F 88 53 41	73 32 19 B3	80 F4 51 0A	85 51 1C 67
00405058	B6 98 4E 25	B3 9D 8E 25	64 11 12 01	CF 10 62 0A
00405068	8B 54 EE FC	53 68 19 01	7C 36 52 8C	F9 F3 59 FF

进行如下运算

$$B[0] = (B[0] + Num_2) ^ (A[0] + Num_1)$$

$$B[1] = (B[1] + Num_2) ^ (A[0] + A[1] + Num_1)$$

$$B[2] = (B[2] + Num_2) ^ (A[0] + A[2] + Num_1)$$

$$B[3] = (B[3] + Num_2) ^ (A[0] + A[3] + Num_1)$$

将输入的第 9、10 位拓展到32位，就此例，0x90 拓展为 0x90909090，将其看作 Num_3

进行如下运算

```
for(int i = 0; i < 8; i++)
{
    B[4][i] = (B[4][i] + Num_3) ^ (Num_0 + 0x10 + i);
    B[5][i] = (B[5][i] + Num_3) ^ (Num_0 + 0x10 + 4 + i);
}
```

将运算完成的数据与 `404148` 的数据进行比对，如果全部比对成功，就开始将数据写入 `sub_4018A0` 的指定位置

00404148	55	8B	EC	83	E4	F0	81	EC	78	02	00	00	A1	04	50	40	U.i.äð.ìx...j.P@
00404158	00	33	C4	89	84	24	74	02	00	00	0F	10	05	A8	41	40	.3Ä..\$t.....A@
00404168	00	A0	C0	41	40	00	56	0F	11	44	24	2C	57	F3	0F	7E	.AA@.v...D\$,wó.~
00404178	05	B8	41	40	00	66	0F	D6	44	24	40	0F	10	41	0A	6A	.,A@.f.ÖD\$@..A.j
00404188	40	88	44	24	4C	8D	84	24	FC	01	00	00	6A	00	50	0F	@.D\$L...\$ü...j.P.
00404198	11	44	24	1C	E8	58	0F	00	00	6A	40	8D	84	24	48	02	.D\$.èx...j@...\$H.

得到前十位

使用Z3来解决这个方程，就实际而言，只需要解出第7步中第一个方程即可得到结果

由于Python的Z3模块我不会使用，所以我学习了一下Z3，用Z3官方教程页面的在线IDE [rise4fun](#) 进行了求解

代码如下

```

(define-fun a() (_ BitVec 32) #x416214C8)
(define-fun b() (_ BitVec 32) #x01120DF0)
(define-fun c() (_ BitVec 32) #xED93C08B)
(define-fun d() (_ BitVec 32) #x7EB6971B)
(define-fun e() (_ BitVec 32) #x00000003)
(define-fun f() (_ BitVec 32) #x00000002)
(define-fun g() (_ BitVec 32) #x00000001)
(define-fun h() (_ BitVec 32) #x00000000)
(define-fun e0 () (_ BitVec 32) #x254e98b6)
(define-fun e1 () (_ BitVec 32) #x258e9db3)
(define-fun e2 () (_ BitVec 32) #x01121164)
(define-fun e3 () (_ BitVec 32) #x0a6210cf)
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const x1 (_ BitVec 2))
(declare-const x2 (_ BitVec 2))
(assert (bvugt x #x00000000))
(assert (bvule x #xffffffff))
(assert (bvugt y #x00000000))
(assert (bvule y #x000000ff))
(define-fun c1 ((b1 (_ BitVec 32)) (b2 (_ BitVec 32)) (b3 (_ BitVec 32)) (b4 (_ BitVec 32))) (_ BitVec 32)
  (bvnot (bvxnor (bvadd b3 (bvmul b2 #x01010101)) (bvadd b4 b1))))
)
(define-fun c2 ((b1 (_ BitVec 32)) (b2 (_ BitVec 32)) (b3 (_ BitVec 32)) (b4 (_ BitVec 32))) (_ BitVec 32)
  (bvnot (bvxnor (bvadd b3 (bvmul b2 #x01010101)) (bvadd b1 b4))))
)
(assert (= (c1 x y a e) #x405004A1))
(assert (= (c1 x y b f) #x00000278))
(assert (= (c1 x y c g) #xEC81F0E4))
(assert (= (c1 x y d h) #x83EC8B55))
(check-sat)
(get-model)

```

最终得到了结果：1324227812

接下来重新启动程序，在 401C38 下断点，并清除原来的断点，接着进入 sub_4018A0 分析该函数的内容

3DES解密

sub_4018A0 是3DES解密的一个过程，找到key以后直接解密即可

```

0019FE18|41 46 53 41|46 43 45 44|41 46 53 41|46 43 45 44|AFSAFCEDAFSAFCED|
0019FE28|59 43 58 43|58 41 43 4E|44 46 4B 44|43 51 58 43|YCXCXACNDFKDCQXC|

```

上图从第9个字节开始即为key

解密脚本

总结

这题我没直接做出来，前面SMC部分还好说，后面这个3DES直接就不认识，也因为见识不广吃了大亏。不过还好，下次就认识了