Crack Reverse2020.exe

梁晨

3180102160

PARTI：破解过程

- 断点一： WinMain at 4012C0

  用IDA Pro的Quick View查看Function，可以看到WinMain在地址4012C0。用OllyDBG打开程序进行动态调试，跑到4012C0后选择“自动步过”。

- 断点二：Forms::TApplication::Run at 494D34

  选择“自动步过”后，发现程序在401306处停下，且窗口弹出。除非关闭窗口，否则WinMain里401306之后的指令不会执行。401306处的指令为：

  ```
  call 00494D34
  ```

  由此可以推测要跟的计算、比较SerialNumber的函数在494D34里。先在401306处设置断点，然后重新开始，F7步入，在494D34里设置断点，禁止401306处的断点，继续选择“自动步过”。

- 断点三：Forms::TCustomForm::SetFocusedControl at 490390及其他软件断点

  在Forms::TApplication::Run函数中可以看到494DB2至494DC4的几条指令：

  ```
  mov eax,[ebp-0x4]
  call 00494BA0
  mov eax,[ebp-0x4]
  cmp byte ptr [eax+0x8C],0x0
  je short 00494DB
  ```

  这个循环一直做除非关闭窗口，由此可以推测需要再从494DB5步入494BA0函数。同理，可以一步步往里进，这是一个漫长而痛苦的过程……其实找到了494BA0，就可以直接按照下面的步骤设置硬件断点了。但主要问题是494BA0还不够“靠里”，此处窗口的行为难以预知，在输入SerialNumber后可能跑很多次上面的循环才回显一个字符，在点击Register之后，可能跑很多次上面的循环也没有反应，需要再点一次。经过漫长而痛苦的跟进后，终于找到一个行为很合理的函数，即Forms::TCustomForm::SetFocusedControl at 490390。

在窗口状态正常的情况下，只要鼠标放在窗口里，EIP就会跳到490390。窗口出现后，在490390设置断点，在SerialNumber处进行输入，跑一次即可使字符串全部回显。点击Register后，跑两次可以弹框。行为可以预知。所以选择此断点作为找到计算、比较验证码的函数前最后一个软件断点，下面开始设置硬件断点。

- 断点四：硬件byte访问断点 at 159868

  打开内存窗口，查找刚刚自己输入的SerialNumber，可以发现其首字节仅被存放在159868处。推测计算、比较验证码时要从此处读SerialNumber，故在首字节处设置硬件byte访问断点。在设置硬件断点前，我试过在150000这个内存段设置读写的内存断点，发现很难断住，因为这个内存段存放了很多数据，包括机器、文件的信息等等。在点击Register后开启硬件断点，继续跑软件，试图找到计算、比较验证码的函数，

  为了保险，可以在SerialNumber字符串里多设几个硬件byte断点，或把性质改为读写。

- 断点五：_TfrmReverse2020_btnRegisterClick at 4015F8及附近软件断点

  找到这个函数的过程其实也漫长而痛苦。我用“一前一后”的方法去逐步逼近计算、校验验证码的函数。第一，充分运用IDA Pro可以显示函数名字的功能，设置完硬件断点，不断往前跑时，通过函数的名字判断找到的函数与计算、校验验证码是否有关。第二，暂时不设置硬件断点，在点击Register后或在弹框后，从490390处使用Contrl+F9和Alt+F9不断“回退”。可以发现“回退”和”前进“过程中有很多重合的函数，名字基本都和Window、Handle、Control等等有关，基本可以断定这些函数仅仅是处理窗口的函数。同时，在”回退“的过程中，如果从用户函数进入了系统函数（地址前两位从00跳到77或其他），基本可以断定这些用户函数仅仅是处理窗口的函数，在调用系统函数时被当作函数指针传入，在系统函数内部进一步被调用。

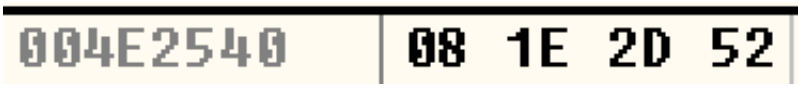
  通过上述方法，终于定位到_TfrmReverse2020_btnRegisterClick at 4015F8。看这个名字就觉得应该找对了，且这个函数在从490390开始的有限步的”回退“内没有出现过。在4015F8设置断点，同时disable其他所有断点。输入SerialNumber后，点击Register，程序被中断。点击Control+F9可以发现直接弹框。于是基本断定 _TfrmReverse2020_btnRegisterClick就是要找的计算、校验验证码的函数。稍作阅读后可以在4016F4除发现一条指令：

  ```
  call 004D5F8C
  ```

  在IDA Pro中查询，惊喜地发现004D5FBC是sscanf函数！此函数把验证码作为32bit16进制整数读入stack区的12FB88处。（如不满八位，左端用0补齐，如超过八位，取最右端八位，输入0-9A-F之外的字符，则立即停止读入。其实就是sscanf读32bit16进制整数，即读int或unsigned int的规则。）读入后则开始计算与校验。另外，在401737处开始两条指令为：

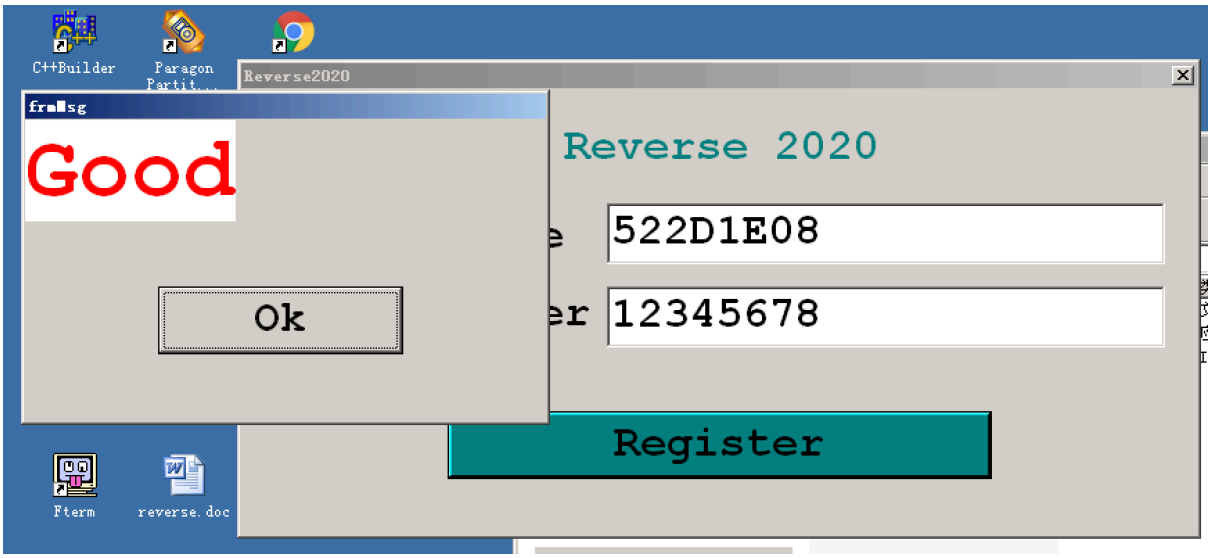
  ```
  cmp ecx,[0x4E2540]
  jnz short 0040174B
  ```

  查看40E2540的内存，发现其中存放的恰是Machine Code(我的是522D1E08)：

  尝试把jnz的指令改为nop，可以发现弹窗为Good！成功破解！

```
0040173D    90          nop
0040173E    90          nop
```



PARTII：分析计算过程

分析_TfrmReverse2020_btnRegisterClick函数在4016F4之后401737之前的代码如下：

```
;4016F9-4010731代码
004016F9  |.  83C4 0C       add     esp, 0xC
004016FC  |.  8D5D C4       lea     ebx, [ebp-0x3C] ;ebx=12FB88，即读入的十六进制数的地址
004016FF  |.  6A 01         push    0x1       ;为调用4015D8做准备

00401701  |.  8A43 01       mov     al, [ebx+0x1]  ;al=读入的十六进制数的第二个字节（注意在内存窗中用小端规则存放）
00401704  |.  50            push    eax          ;为调用4015D8做准备

00401705  |.  E8 CEFEFFFF   call    004015D8
0040170A  |.  83C4 08       add     esp, 0x8 ;结束4015D8的调用
0040170D  |.  8843 01       mov     [ebx+0x1], al ; 替换读入的十六进制数的第二个字节为此字节循环左移1位的结果
00401710  |.  6A 02         push    0x2           ;为调用4015E8做准备

00401712  |.  8A53 02       mov     dl, [ebx+0x2]  ;dl=读入的十六进制数的第三个字节（注意在内存窗中用小端规则存放）
00401715  |.  52            push    edx       ;为调用4015E8做准备

00401716  |.  E8 CDFEFFFF   call    004015E8
0040171B  |.  83C4 08       add     esp, 0x8 ;结束4015E8的调用
0040171E  |.  8843 02       mov     [ebx+0x2], al ;替换读入的十六进制数的第三个字节为此字节循环右移2位的结果
```

```
00401721  |.  8003 42       add     byte ptr [ebx], 0x42 ;读入的十六进制数
+0x42（注意不能进位！+为无进位加法！）
00401724  |.  806B 03 57    sub     byte ptr [ebx+0x3], 0x57; 读入的十六进制
数-0x57000000
00401728  |.  8B4D C4       mov     ecx, [ebp-0x3C] ;ecx=此时的十六进制数
0040172B  |.  81C1 DEC0AD0B add     ecx, 0xBADC0DE ;ecx+=0xBADC0DE(注意不做
sign extension，常数高位为0)
00401731  |.  81F1 EFBEADDE xor     ecx, 0xDEADBEEF ;ecx^=0xEFBEADDE
;4015D8函数代码
sub_4015D8 proc near
arg_0= byte ptr  8
arg_4= byte ptr  0Ch
push    ebp
mov     ebp, esp
xor     eax, eax ;将eax置0
mov     al, [ebp+arg_0] ;al=读入的十六进制数的第二个字节
mov     cl, [ebp+arg_4] ;cl=1
rol     al, cl ;al循环左移一位
pop     ebp
retn
sub_4015D8 endp
;4015E8函数代码
sub_4015E8 proc near
arg_0= byte ptr  8
arg_4= byte ptr  0Ch
push    ebp
mov     ebp, esp
xor     eax, eax ;将eax置0
mov     al, [ebp+arg_0] ;al=读入的十六进制数的第三个字节
mov     cl, [ebp+arg_4] ;cl=2
ror     al, cl ;al 循环右移2位
pop     ebp
retn
sub_4015E8 endp
```

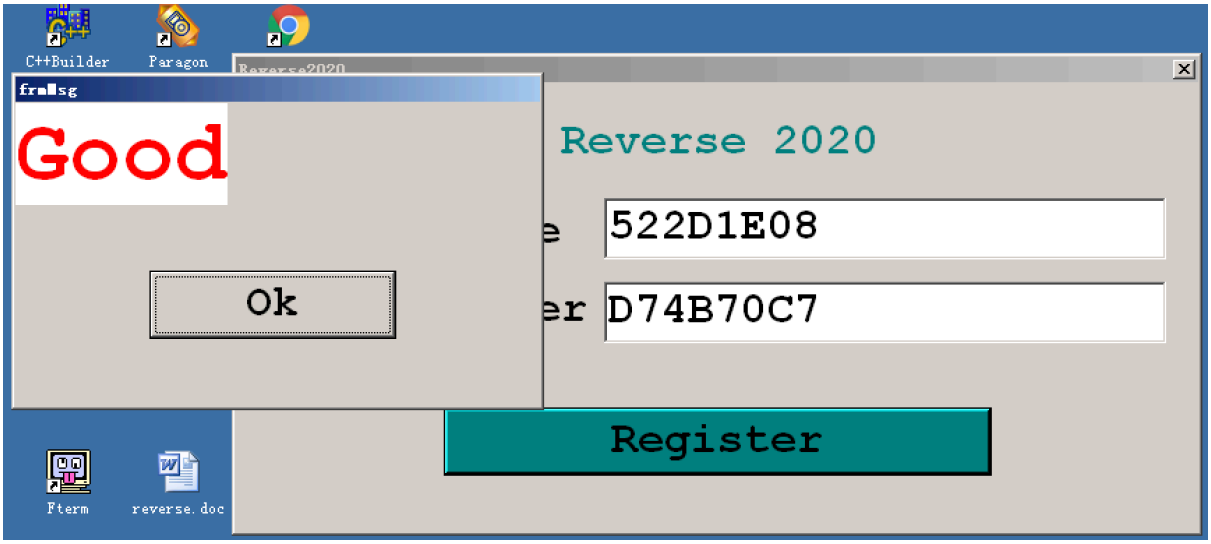
根据以上的分析过程，可以写出从MachineCode算出SerialNumber的函数（即为以上代码的逆过程）：

```
#include <stdio.h>
int main()
{
  unsigned int machine_code;
  scanf("%x",&machine_code);
  machine_code^=0xDEADBEEF;
  machine_code+=~0xBADC0DE;
  machine_code+=1;
  unsigned int m1=machine_code&0xFF000000;
  unsigned int m2=machine_code&0x000000FF;
```

```
    machine_code&=0x00FFFF00;
    m2+=~0x042;
    m2+=1;
    m2&=0x0FF;
    m1>>=24;
    m1+=0x057;
    m1&=0x0FF;
    m1<<=24;
    machine_code|=m1|m2;
    unsigned int t1=machine_code&0x00FF0000;
    unsigned int t2=machine_code&0x0000FF00;
    machine_code&=0xFF0000FF;
    t1>>=16;
    t2>>=8;
    for(int i=0;i<2;i++){
      unsigned int first_bit=t1&0x080;
      t1=((t1<<1)+(first_bit>>7))&0x0FF;
    }
    unsigned int last_bit=t2&0x01;
    t2=(t2>>1)+(last_bit<<7);
    t1<<=16;
    t2<<=8;
    machine_code|=t1|t2;
    printf("%x\n",machine_code);
    return 0;
}
```

为了避免不必要的混乱，以上解密函数中没有使用减法，都用加上补码加1来代替。同时，对于只涉及byte的操作，我们用不同的mask（0xFF000000等等）取相应的byte，同时用此mask的补码作用在machine_code上，将对应byte置0。再对相应byte做操作，操作后再按位或回machine_code中。这样做保证了其中的加法操作不涉及进位。

根据machine_code=522D1E08，算出SerialNumber=D74B70C7。破解成功！



Oct. 27th,2020