攻防世界ReverseMe-120详解

原创

Casuall 于 2019-12-21 20:11:08 发布

分类专栏： Reverse 文章标签： 逆向 ctf 攻防世界

网上其他的WP写的都比较简略，所以写一篇比较详细的解题思路，供新手参考，大神请绕过。

首先大致看一下程序的逻辑

```
printf("please input your flah:");
v11 = 0;
memset(&v12, 0, 0x63u);
scanf("%s", &v11);
v13 = 0;
memset(&v14, 0, 0x63u);
sub_401000(&v15, &v13, (unsigned __int8 *)&v11, strlen(&v11));
v3 = v15;
v4 = 0;
if ( v15 )
{
  if ( v15 >= 0x10 )
  {
    v5 = _mm_load_si128((const __m128i *)&xmmword_414F20);// xr
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;                 // v7指向v13
    do
    {
      v8 = _mm_loadu_si128(v7);                 // 这条命令和上面
      v4 += 16;
      ++v7;                                     // v7++, 这也是下
      _mm_storeu_si128((__m128i *)&v7[-1], _mm_xor_si128(v8, v5
                                                // 也就是将v8和v5
    }
    while ( v4 < v6 );
  }
  for ( ; v4 < v3; ++v4 )
    *(&v13 + v4) ^= 0x25u;                      // 这两句根本没有
}
v9 = strcmp(&v13, "you_know_how_to_remove_junk_code");
if ( v9 )
  v9 = -(v9 < 0) | 1;
if ( v9 )
  printf("wrong\n");
else
  printf("correct\n");
system("pause");
return 0;
}
```

可以看到成功的条件是`v9`，而`v9`是`v13`与字符串"`you_know_how_to_remove_junk_code`"比较的结果。然后追一下`v13`的数据流，看看`v13`是怎么来的。

```
printf("please input your flah:");
v11 = 0;
memset(&v12, 0, 0x63u);
scanf("%s", &v11);                      // 你的输入
v13 = 0;
memset(&v14, 0, 0x63u);
sub_401000(&v15, &v13, (unsigned __int8 *)&v11, strlen(&v11));// b
v3 = v15;
v4 = 0;
if ( v15 )
{
  if ( v15 >= 0x10 )
  {
    v5 = _mm_load_si128((const __m128i *)&xmmword_414F20);// xmmwo
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;                 // v7指向v13
    do
    {
      v8 = _mm_loadu_si128(v7);                 // 这条命令和上面的_
      v4 += 16;
      ++v7;                                     // v7++，这也是下面将
      _mm_storeu_si128((__m128i *)&v7[-1], _mm_xor_si128(v8, v5));
                                                // 也就是将v8和v5按位
    }
    while ( v4 < v6 );
  }
  for ( ; v4 < v3; ++v4 )
    *(&v13 + v4) ^= 0x25u;                      // 这两句根本没有执行
}
```

可以看到`v13`的定义，以及一个关键函数`sub_401000`，为什么说是关键函数呢，因为函数的参数包含了**刚定义的**`v13`，以及**你的输入**`v11`。

我们跟进去看一看，注意我们想知道的是`v13`是怎么得到的，而`v13`作为**第二个参数**，在函数`sub_401000`里是`a2`，我们顺着`a2`去看。

```
signed int __usercall sub_401000@<eax>(unsigned int *a1@<edx>, _BYTE *a2@<ecx>, unsigned __int8 *a3, unsigned int a4)
{
  v4 = 0;
  v18 = a2;                                     // a2为返回指针
  v5 = 0;
  v6 = 0;
  v19 = a1;
  v20 = 0;
  if ( !a4 )                                    // a4=len(input)
    return 0;
  v7 = a3;                                      // a3为input的指针
  v12 = v18;
  v13 = ((unsigned int)(6 * v6 + 7) >> 3) - v4;
  if ( v18 && *v19 >= v13 )
  {
    v21 = 3;
    v14 = 0;
    for ( i = 0; v5; --v5 )
    {
      v15 = *v7;
      if ( *v7 != 13 && v15 != 10 && v15 != 32 )
      {
        v16 = byte_414E40[v15];                 // byte_414E40就是base64_suffix_map
        v21 -= v16 == 0x40;
        v14 = v16 & 0x3F | (v14 << 6);          // 将其依次放入一个int型中，占3字节
        if ( ++i == 4 )                         // 输入4个字节
        {
          i = 0;                                // 输出3个字节
          if ( v21 )
            *v12++ = BYTE2(v14);
          if ( v21 > 1 )
            *v12++ = BYTE1(v14);
          if ( v21 > 2 )
            *v12++ = v14;
        }
      }
      ++v7;
    }
```


上图为是经过拼接处理的，可以得到这样的关系`v12 = v18 = a2 = v13`,然后对`v12`进行了赋值处理。上面有一个数组`byte_414E40`，里面的内容如下图所示，有的WP说一看就知道是**base64解码表**，然而我比较菜，没有看出来为啥是base64解码表，去网上查阅资料，最后终于弄懂了，详情请参考C语言实现base64编码，这篇文章介绍了原理和代码实现，**把代码从头到尾研究一遍**就懂了下面的图是什么了。

```
00414E40  7F 7F 7F 7F 7F 7F 7F 7F  7F 7F 7F 7F 7F 7F 7F 7F  ................
00414E50  7F 7F 7F 7F 7F 7F 7F 7F  7F 7F 7F 7F 7F 7F 7F 7F  ................
00414E60  7F 7F 7F 7F 7F 7F 7F 7F  7F 7F 7F 3E 7F 7F 7F 3F  ...........>...?
00414E70  34 35 36 37 38 39 3A 3B  3C 3D 7F 7F 7F 40 7F 7F  456789:;<=...@..
00414E80  7F 00 01 02 03 04 05 06  07 08 09 0A 0B 0C 0D 0E  ................
00414E90  0F 10 11 12 13 14 15 16  17 18 19 7F 7F 7F 7F 7F  ................
00414EA0  7F 1A 1B 1C 1D 1E 1F 20  21 22 23 24 25 26 27 28  ....... !"#$%&'(
00414EB0  29 2A 2B 2C 2D 2E 2F 30  31 32 33 7F 7F 7F 7F 7F  )*+,-./0123.....
```

分析完整个函数，知道函数`sub_401000`是base64解码函数了，得到的信息是程序将`你的输入`---->`base64解码`---->`得到v13`。

```
v3 = v15;
v4 = 0;
if ( v15 )
{
  if ( v15 >= 0x10 )
  {
    v5 = _mm_load_si128((const __m128i *)&xmmword_414F20);// xmmwo
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;                 // v7指向v13
    do
    {
      v8 = _mm_loadu_si128(v7);                 // 这条命令和上面的_
      v4 += 16;
      ++v7;                                     // v7++，这也是下面将
      _mm_storeu_si128((__m128i *)&v7[-1], _mm_xor_si128(v8, v5));
                                                // 也就是将v8和v5按位
    }
    while ( v4 < v6 );
  }
  for ( ; v4 < v3; ++v4 )
    *(&v13 + v4) ^= 0x25u;                      // 这两句根本没有执行
}
```

然后继续往下分析，然后有一堆代码，看着就不是我能理解的，但是那堆看不懂的代码在一个`if语句`中，下面有个for循环，这个是我能看懂的，就是把`v13所指的字符串`挨个与`0x25`**异或**，好像看到了逆向的常规操作，异或。

那我就先猜测`v15>=0x10`这个条件不成立吧，毕竟七分逆向三分猜嘛，`你的输入`---->`base64解码`---->`得到v13`---->`与0x25异或`---->`与“you_know_how_to_remove_junk_code”比较`---->`结果`。按照这个思路逆着写一个脚本

```
import base64
s = "you_know_how_to_remove_junk_code"
f = ''
for i in s:
    f += chr(ord(i) ^ 0x25)
print(f)
print(base64.b64encode(f.encode()))
```

就得到了flag。

不过，我将正确的flag输入，用OD调试跟踪，发现，**程序根本没有进入for循环**，wtf，居然用错误的思路弄出了正确的答案，哎，只能继续分析那些令人头疼的代码了

```
.rdata:00414F20 xmmword_414F20  xmmword 25252525252525252525252525252525h
```

```
if ( v15 )
{
  if ( v15 >= 0x10 )
  {
    v5 = _mm_load_si128((const __m128i *)&xmmword_414F20);// xmmword_414F20是16字节的（0x25）数据
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;                 // v7指向v13
    do
    {
      v8 = _mm_loadu_si128(v7);                 // 这条命令和上面的_mm_load_si128是Intel SSE指令集中load系列，用于加载数据，从内存到暂存器
      v4 += 16;
      ++v7;                                     // v7++，这也是下面将异或后的结果放在v7[-1]的原因
      _mm_storeu_si128((__m128i *)&v7[-1], _mm_xor_si128(v8, v5));// store系列，用于将计算结果等SSE暂存器的数据保存到内存中；_mm_xor_si128计算128位的按位异或；
                                                // 也就是将v8和v5按位异或，然后存在v7[-1]中
    }
    while ( v4 < v6 );
  }
```


`xmmword`用于具有MMX和SSE (XMM)指令的**128位**多媒体操作数（也不知道翻译的对不对，官方解释是“Used for 128-bit multimedia operands with MMX and SSE (XMM) instructions.”）。

SEE指令，参考(https://www.jianshu.com/p/d718c1ea5f22)

1. load(set)系列，用于加载数据，从内存到暂存器。
   `__m128i _mm_load_si128(__m128i *p);`
   `__m128i _mm_loadu_si128(__m128i *p);`
2. store系列，用于将计算结果等SSE暂存器的数据保存到内存中。
   `void _mm_store_si128 (__m128i *p, __m128i a);`
   `void _mm_storeu_si128 (__m128i *p, __m128i a);`

`_mm_load_si128`函数表示从内存中加载一个**128bits**值到暂存器，也就是**16字节**，**注意：**p必须是一个16字节对齐的一个变量的地址。返回可以存放在代表寄存器的变量中的值。
`_mm_loadu_si128`函数和`_mm_load_si128`一样的，但是不要求地址p是16字节对齐。

store系列的`_mm_store_si128`和`_mm_storeu_si128`函数，与上面的load系列的函数是对应的。 表示将__m128i 变量a的值存储到p所指定的地址中去。

`_mm_xor_si128`用于计算128位（16字节）的按位异或，然后通过`v14`控制循环结束的条件，可以看到`v14`增长的步长为`16`，而且通过上面得到的flag值解码得到的字符串为`32个字节`大小，正好是`16的整数倍`。

所以，基本上逻辑已经清楚了，上面图片也已经注释了，发现实际上**和下面for循环的功能是一样的**，可能是为了降低难度给的提示吧。至于判断条件中的`v6`，以及`v15`还没有研究明白，有懂的大佬可以指点一下，估计研究反汇编代码可能知道他们的具体含义，先这样，以后有时间了再来研究这道题的反汇编代码。