CVE-2012-0179 分析报告

启明星辰 2012/5

漏洞概要

最近在某国外网站上放出了 CVE-2012-0179（MS12-032）的利用 POC，该漏洞为一个本地漏洞，当在 Windows7，Windows2008 等系统上绑定一个由 ipv4 扩展的 ipv6 地址的时候，如果绑定的地址未在本机设置（地址无效），则会出现一个内存的 double-free 错误，导致蓝屏。下面是该漏洞的详细分析。

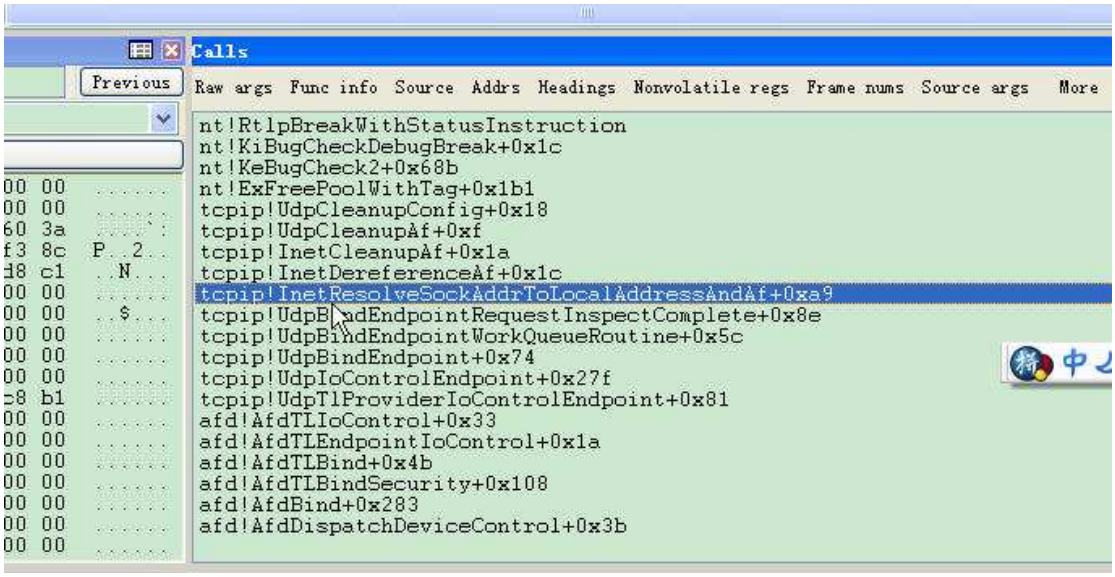
漏洞分析

漏洞利用代码（部分）如下图所示：

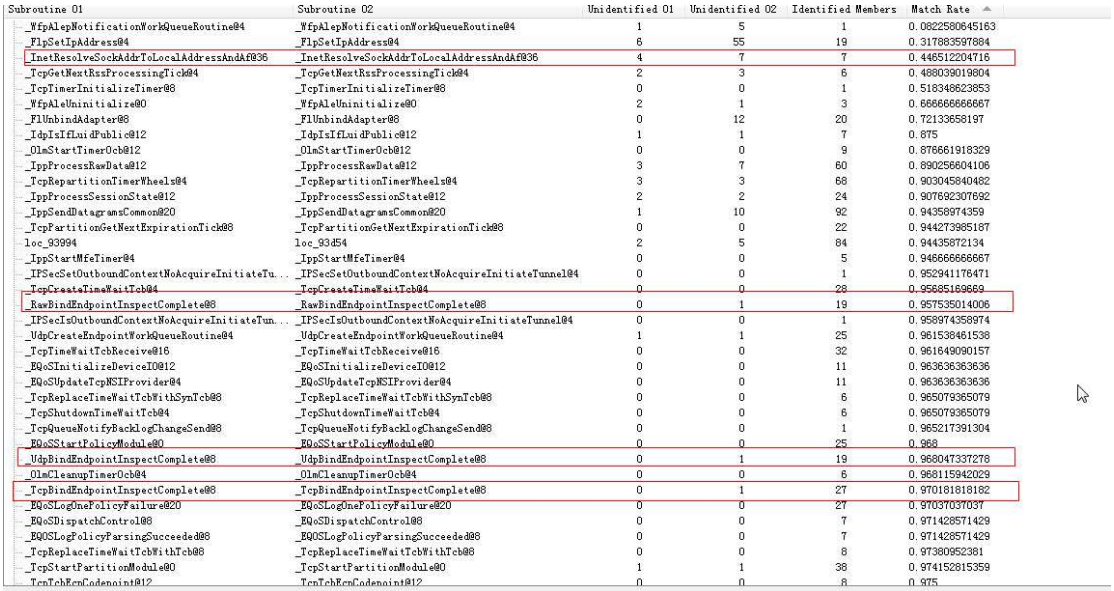
```
bool bindSuccess = false;

while(!bindSuccess)
{
        SOCKET sock = WSASocket(AF_INET6,
                SOCK_DGRAM,
                IPPROTO_UDP,
                NULL,
                0,
                WSA_FLAG_OVERLAPPED);
        if(sock == INVALID_SOCKET)
        {
                printf("WSASocket failed\n");
                exit(-1);
        }

        DWORD val = 0;
        if (setsockopt(sock,
                IPPROTO_IPV6,
                IPV6_V6ONLY,
                (const char*)&val,
                sizeof(val)) != 0)
        {
                printf("setsockopt failed\n");
                closesocket(sock);
                exit(-1);
        }

        sockaddr_in6 sockAddr;
        memset(&sockAddr, 0, sizeof(sockAddr));
        sockAddr.sin6_family = AF_INET6;
        sockAddr.sin6_port = htons(5060);

        // set address to IPV6-mapped 169.13.13.13 (not configured on the local machine)
        // that is [::FFFF:169.13.13.13]
        sockAddr.sin6_addr.u.Byte[15] = 1;
        sockAddr.sin6_addr.u.Byte[14] = 0;
        sockAddr.sin6_addr.u.Byte[13] = 0;
        sockAddr.sin6_addr.u.Byte[12] = 127;
        sockAddr.sin6_addr.u.Byte[11] = 0xFF;
        sockAddr.sin6_addr.u.Byte[10] = 0xFF;

        int size = 28; // 28 is sizeof(sockaddr_in6)
```

这里循环对于一个不存在的 ipv4 地址 169.13.13.13 扩展的 ipv6 地址进行绑定。在循环了几次后，便会出现蓝屏。

崩溃时候截图如下



对此漏洞，微软修补了 tcpip.sys 这个内核驱动，我们通过对比找到了线索



Subroutine 01	Subroutine 02	Unidentified 01	Unidentified 02	Identified Members	Match Rate
_WfpAlepNotificationWorkQueueRoutine@4	_WfpAlepNotificationWorkQueueRoutine@4	1	5	1	0.0822580645163
_FlpSetIpAddress@4	_FlpSetIpAddress@4	6	55	19	0.317883597884
_InetResolveSockAddrToLocalAddressAndAf@36	_InetResolveSockAddrToLocalAddressAndAf@36	4	7	7	0.446512204716
_TcpGetNextRssProcessingTick@4	_TcpGetNextRssProcessingTick@4	2	3	6	0.488039019804
_TcpTimerInitializeTimer@8	_TcpTimerInitializeTimer@8	0	0	1	0.518348623853
_WfpAleUninitialize@0	_WfpAleUninitialize@0	2	1	3	0.666666666667
_FlUnbindAdapter@8	_FlUnbindAdapter@8	0	12	20	0.72133658197
_IdpIsIfLuidPublic@12	_IdpIsIfLuidPublic@12	1	1	7	0.875
_OlmStartTimerOcb@12	_OlmStartTimerOcb@12	0	0	9	0.876661918329
_IppProcessRawData@12	_IppProcessRawData@12	3	7	60	0.890256604106
_TcpRepartitionTimerWheels@4	_TcpRepartitionTimerWheels@4	3	3	68	0.903045840482
_IppProcessSessionState@12	_IppProcessSessionState@12	2	2	24	0.907692307692
_IppSendDatagramsCommon@20	_IppSendDatagramsCommon@20	1	10	92	0.94358974359
_TcpPartitionGetNextExpirationTick@8	_TcpPartitionGetNextExpirationTick@8	0	0	22	0.944273985187
loc_93994	loc_93d54	2	5	84	0.94435872134
_IppStartMfeTimer@4	_IppStartMfeTimer@4	0	0	5	0.946666666667
_IPSecSetOutboundContextNoAcquireInitiateTu...	_IPSecSetOutboundContextNoAcquireInitiateTunnel@4	0	0	1	0.952941176471
_TcpCreateTimeWaitTcb@4	_TcpCreateTimeWaitTcb@4	0	0	28	0.95685169669
_RawBindEndpointInspectComplete@8	_RawBindEndpointInspectComplete@8	0	1	19	0.957535014006
_IPSecIsOutboundContextNoAcquireInitiateTun...	_IPSecIsOutboundContextNoAcquireInitiateTunnel@4	0	0	1	0.958974358974
_UdpCreateEndpointWorkQueueRoutine@4	_UdpCreateEndpointWorkQueueRoutine@4	1	1	25	0.961538461538
_TcpTimeWaitTcbReceive@16	_TcpTimeWaitTcbReceive@16	0	0	32	0.961640090157
_EQoSInitializeDeviceIO@12	_EQoSInitializeDeviceIO@12	0	0	11	0.963636363636
_EQoSUpdateTcpNSIProvider@4	_EQoSUpdateTcpNSIProvider@4	0	0	11	0.963636363636
_TcpReplaceTimeWaitTcbWithSynTcb@8	_TcpReplaceTimeWaitTcbWithSynTcb@8	0	0	6	0.965079365079
_TcpShutdownTimeWaitTcb@4	_TcpShutdownTimeWaitTcb@4	0	0	6	0.965079365079
_TcpQueueNotifyBacklogChangeSend@8	_TcpQueueNotifyBacklogChangeSend@8	0	0	1	0.965217391304
_EQoSStartPolicyModule@0	_EQoSStartPolicyModule@0	0	0	25	0.968
_UdpBindEndpointInspectComplete@8	_UdpBindEndpointInspectComplete@8	0	1	19	0.968047337278
_OlmCleanupTimerOcb@4	_OlmCleanupTimerOcb@4	0	0	6	0.968115942029
_TcpBindEndpointInspectComplete@8	_TcpBindEndpointInspectComplete@8	0	1	27	0.970181818182
_EQoSLogOnePolicyFailure@20	_EQoSLogOnePolicyFailure@20	0	0	27	0.97037037037
_EQoSDispatchControl@8	_EQoSDispatchControl@8	0	0	7	0.971428571429
_EQOSLogPolicyParsingSucceeded@8	_EQOSLogPolicyParsingSucceeded@8	0	0	7	0.971428571429
_TcpReplaceTimeWaitTcbWithTcb@8	_TcpReplaceTimeWaitTcbWithTcb@8	0	0	8	0.97380952381
_TcpStartPartitionModule@0	_TcpStartPartitionModule@0	1	1	38	0.974152815359
_TcpTcbEcnCodepoint@12	_TcpTcbEcnCodepoint@12	0	0	8	0.975

Subroutine 01	Subroutine 02	Unidentified 01	Unidentified 02	Identified Members	Match Rate
_TcpReplaceTimeWaitTcbWithTcb@8	_TcpReplaceTimeWaitTcbWithTcb@8	0	0	8	0.97380952381
_TcpStartPartitionModule@0	_TcpStartPartitionModule@0	1	1	38	0.974152815359
_TcpTcbEcnCodepoint@12	_TcpTcbEcnCodepoint@12	0	0	8	0.975
_OlmGetNextInterface@16	_OlmGetNextInterface@16	0	0	23	0.980156075808
_OlmQuerySendCallInProgress@8	_OlmQuerySendCallInProgress@8	0	0	6	0.980952380952
_InetResolveSockAddrToAf@36	_InetResolveSockAddrToAf@36	0	0	18	0.983333333333
_EQoSReadAndEstablishMachinePolicyTable@8	_EQoSReadAndEstablishMachinePolicyTable@8	0	0	13	0.984615384615
_IppInitializePathSet@4	_IppInitializePathSet@4	0	0	9	0.987739463602
_EQoSProcessMachinePolicyChange@12	_EQoSProcessMachinePolicyChange@12	0	0	17	0.988235294118
_OlmSetTcpInterfaceParametersByLuid@12	_OlmSetTcpInterfaceParametersByLuid@12	0	0	9	0.988846960168
_OlmFindAndReferenceInterfaceByLuid@4	_OlmFindAndReferenceInterfaceByLuid@4	0	0	7	0.989115646259
_IPSecValidateIPSecSa@12	_IPSecValidateIPSecSa@12	0	0	16	0.99

从上图中我们找出了和崩溃时堆栈回溯相关的几个函数。经过逆向，我们断在 UdpEndpointRequestInspectComplete 这个函数上分析这个漏洞。该函数是在上层执行 bind 的时候跑到的，因为源代码中绑定的是 UDP 端口，因此自然会跑到 Udp 相关的函数来。

这里有几个重要的结构

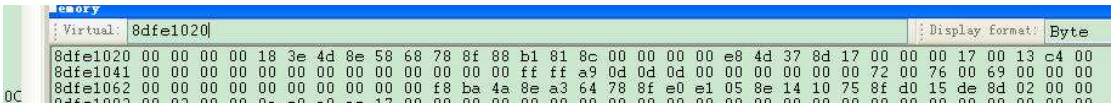
StructA（该结构为 Windows 初始化时候就确定的）

偏移 4 指向 StructB

偏移 0x14 StructIpv4/Ipv6

偏移 0x18 flag of Ipv4/ipv6
偏移 0x1a ipv4 to ipv6 flag
偏移 0x1cuserbuffer
StructB
偏移 0x14StructIpv4/Ipv6

tcpip!UdpBindEndpointRequestInspectComplete:
8f458a88 8bff mov edi,edi
8f458a8a 55 push ebp
8f458a8b 8bec mov ebp,esp
8f458a8d 83ec0c sub esp,0Ch
8f458a90 837d0c00 cmp dword ptr [ebp+0Ch],0
8f458a94 53 push ebx
8f458a95 56 push esi
8f458a96 8b7508 mov esi,dword ptr [ebp+8] ss:0010:9254b9b8=8dfe1020 ;这里有一个结构，该结构指针是从 UdpEndpoint 的一个工作队列中取得的，经过逆向发现是开机初始化的时候就会存下来的一个指针。因为没有具体文档参考，我们将其命名为 StructA
8f458a99 8b5e04 mov ebx,dword ptr [esi+4] ;在 structA 偏移 4 的位置存放一个指针
8f458aa2 807d1000 cmp byte ptr [ebp+10h],0
8f458aa6 8b5514 mov edx,dword ptr [ebp+14h] ss:0010:9254b9c4=8dfe103c(这里传进来一个参数，该参数为 StructA 偏移 0x1c 处的指针
8f458aa9 57 push edi
8f458aaa 7431 je tcpip!UdpBindEndpointRequestInspectComplete+0x55 (8f458add) ;这里跳走



之后会将 StructA 不同位置的指针传入 InetResolveSockAddrToLocalAddressAndAf 函数中

.text:00052AE7 lea edi, [esi+10h]；StructA 偏移 0x10
.text:00052AEA push edi
.text:00052AEB lea eax, [esi+14h] ; StructA 偏移 0x14
.text:00052AEE push eax
.text:00052AEF moveax, [ebx+0Ch] ; StructB 偏移 0xC
.text:00052AF2 lea ecx, [esi+1Ah] ;StructA 偏移 0x1A
.text:00052AF5 push ecx
.text:00052AF6 shreax, 4
.text:00052AF9 lea edx, [esi+18h] ;StructA 偏移 0x18
.text:00052AFC push edx
.text:00052AFD push [ebp+struct1c] ;StructA 偏移 0x1C
.text:00052B00 and al, 1
.text:00052B02 movzxeax, al
.text:00052B05 push eax
.text:00052B06 push dwordptr [ebx+28h] ;StructB 偏移 0x28

```
.text:00052B09                    push    dwordptr [ebx+14h] ;StructB 偏移 0x14
.text:00052B0C                    push    offset _UdpInetTransport；
.text:00052B11          call      InetResolveSockAddrToLocalAddressAndAf(x,x,x,x,x,x,x,x)
```

之后 InetResolveSockAddrToLocalAddressAndAf 又将传进来的参数传给了 InetResolveSockAddrToAf 函数

```
.text:00067457                    movedi, edi
.text:00067459                    push    ebp
.text:0006745A                    movebp, esp
.text:0006745C                    push    ebx
.text:0006745D                    movebx, [ebp+structA14]
.text:00067460                    push    esi
.text:00067461                    movesi, [ebp+structB28]
.text:00067464                    push    edi
.text:00067465                    movedi, [ebp+structA1c]
.text:00067468                    push    ebx
.text:00067469                    push    [ebp+structA18]
.text:0006746C                    lea     eax, [ebp+structA18+3]
.text:0006746F                    push    [ebp+structA1a]
.text:00067472                    push    eax
.text:00067473                    push    edi
.text:00067474                    push    [ebp+flag]
.text:00067477                    push    esi
.text:00067478                    push    [ebp+structB14]
.text:0006747B                    push    [ebp+arg_0]
.text:0006747E                    call    InetResolveSockAddrToAf(x,x,x,x,x,x,x,x,x)
```

进入 InetResolveSockAddrToAf 函数：

```
tcpip!InetResolveSockAddrToAf:
8f46d50e 8bff        mov    edi,edi
8f46d510 55          push   ebp
8f46d511 8bec        mov    ebp,esp
8f46d513 53          push   ebx
8f46d514 8b5d0c      mov    ebx,dword ptr [ebp+0Ch] ss:0010:9254b940=8d374de8 ； 取出 StructB 偏移 0x14 处的值
8f46d517 66837b0c17  cmp    word ptr [ebx+0Ch],17h ;这里是一个标志，用于比较是否是 Ipv6 地址格式
```

这里为 0x17，显示是一个 IPv6 地址格式；另外通过逆向得知 0x2 表示 ipv4 地址格式

```
.text:00067520          cmp     [ebp+flag], 0
```

；比较刚才传进来的标志位，猜测是表示该 IPv6 地址是否为 ipv4 地址扩展而来的标志

```
.text:00067524          jnz     short loc_6758E
```

；如果是 ipv4 扩展而来的 ipv6 地址，则不跳转

.text:00067526 movesi, [ebp+structA1c] ;StructA 偏移 0x1C 位置传进来的是用户态在执行 bind 函数时传进来的 UserBuffer

具体原因：

如图：AfdDispatchDeviceControl 函数通过从三环传进来的 IoControlCode 中计算处理函数的 index 值。这里调用的为 AfdBind 函数。

```
PAGEAFD:000314C9                mov     edi, edi
PAGEAFD:000314CB                push    ebp
PAGEAFD:000314CC                mov     ebp, esp
PAGEAFD:000314CE                mov     ecx, [ebp+arg_4]
PAGEAFD:000314D1                mov     edx, [ecx+60h]
PAGEAFD:000314D4                push    esi
PAGEAFD:000314D5                push    edi
PAGEAFD:000314D6                mov     edi, [edx+IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
PAGEAFD:000314D9                mov     eax, edi
PAGEAFD:000314DB                shr     eax, 2
PAGEAFD:000314DE                and     eax, 3FFh
PAGEAFD:000314E3                cmp     eax, 46h
PAGEAFD:000314E6                jnb     short loc_31506
PAGEAFD:000314E6
PAGEAFD:000314E8                mov     esi, eax
PAGEAFD:000314EA                shl     esi, 2
PAGEAFD:000314ED                cmp     ds:_AfdIoctlTable[esi], edi
PAGEAFD:000314F3                jnz     short loc_31506
PAGEAFD:000314F3
PAGEAFD:000314F5                mov     [edx+1], al
PAGEAFD:000314F8                mov     esi, ds:_AfdIrpCallDispatch[esi]
PAGEAFD:000314FE                test    esi, esi
PAGEAFD:00031500                jz      short loc_31506
PAGEAFD:00031500
PAGEAFD:00031502                call    esi
PAGEAFD:00031504                jmp     short loc_3151C
PAGEAFD:00031504
PAGEAFD:00031506 ; ---------------------------------------------------------------------------
PAGEAFD:00031506
```

_AfdIrpCallDispatch dd offset AfdBind(x,x) ; DATA XREF: AfdDispatchDeviceControl

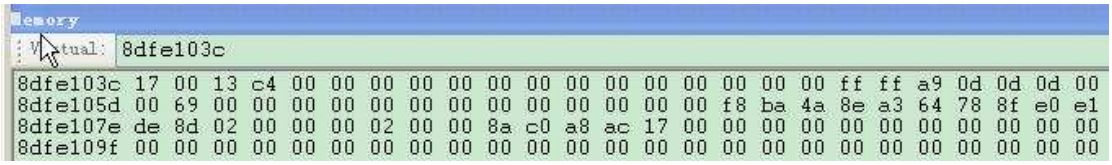
AfdBind 函数会首先检查输入的缓冲区长度和输出缓冲区长度是否合法

```
mov     edi, edx        ; Io_StackLocation
mov     [ebp+var_40], edi
mov     [ebp+Irp], ecx
xor     ebx, ebx
mov     [ebp+var_2C], ebx
mov     eax, [edi+IO_STACK_LOCATION.FileObject]
mov     esi, [eax+0Ch]
mov     [ebp+var_38], esi
mov     [ecx+1Ch], ebx
test    dword ptr [esi+8], 200h
jnz     short loc_24C98

mov     ecx, [edi+IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
cmp     ecx, 8
jb      short Failed

mov     eax, [edi+IO_STACK_LOCATION.Parameters.DeviceIoControl.OutputBufferLength]
cmp     eax, 0Ch
jb      short Failed

add     ecx, 0FFFFFFFCh
```

之后从用户缓冲区偏移 4 的位置拷贝了一个结构给刚申请的一个 buf，该 buf 正是 StructA 偏移 0x1c 处的结构。

```
PAGE:00024D49                                   ; AfdBind(x,x)+10F↑j
PAGE:00024D49                push    0EC646641h      ; Tag
PAGE:00024D4E                push    [ebp+NumberOfBytes] ; NumberOfBytes
PAGE:00024D51                push    10h             ; PoolType
PAGE:00024D53                call    ds:ExAllocatePoolWithQuotaTag(x,x,x)
PAGE:00024D59                mov     [ebp+P], eax
PAGE:00024D5C                push    [ebp+NumberOfBytes] ; size_t
PAGE:00024D5F                test    dword ptr [esi+8], 200h
PAGE:00024D66                jnz     short loc_24DAA
PAGE:00024D66
PAGE:00024D68                mov     eax, [edi+IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
PAGE:00024D6B                mov     ecx, [eax]
PAGE:00024D6D                mov     [ebp+var_30], ecx
PAGE:00024D70                add     eax, 4          ; 用户传进来的Buffer偏移4的位置为该结构
PAGE:00024D73                push    eax             ; void *
PAGE:00024D74                push    [ebp+P]         ; void *
PAGE:00024D77                call    ds:__imp__memmove
PAGE:00024D7D                add     esp, 0Ch
```

如下图，该结构开头的 2 字节大小 0x17 标识了 ip 地址格式，该结构偏移 8 字节开始是 Ip 地址。



继续 InetResolveSockAddrToAf 函数：

.text:00067529 lea eax, [esi+8]；取得 IP 地址指针

.text:0006752C push eax

.text:0006752D call IN6_IS_ADDR_V4MAPPED(x)；检验 IP 地址是否为 ipv4 地址映射而来。由于 IPv4 地址是 32 位，IPv6 地址为 128 位，因此在将 IPv4 地址扩展为 IPv6 地址的时候前面被填充了 0，后面的前 16 位填充为 1，最后的 32 位为 IPv4 地址。

.text:00067532 test al, al

.text:00067534 jz short loc_67569；如果是 IPv4 映射而来，则返回 1，不跳走

.text:00067536 moveax, [ebp+structA1a]

.text:00067539 movedi, [ebp+structA18]

.text:0006753C push 2

.text:0006753E mov byte ptr [eax], 1 ; StructA1a 也是一个标志位

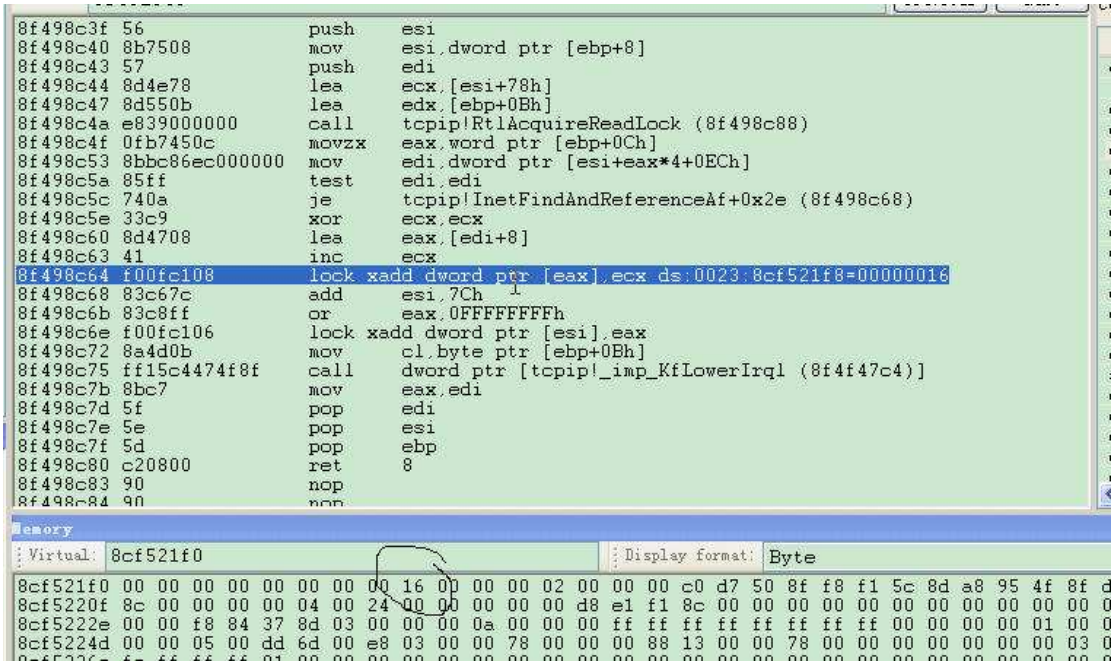
.text:00067541 pop eax

.text:00067542 push eax

.text:00067543 push [ebp+arg_0]

.text:00067546 mov [edi], ax ;因为是 ipv4 扩展而来，则将 structA 偏移 0x18 位置的标志改为 0x2

.text:00067549 call InetFindAndReferenceAf(x,x) ;这里查找 ipv4 相关的结构（StructIpv4/Ipv6），该结构记录了绑定的相关协议(ipv4)的引用计数，这里会查找到相应的结构（以之前的 UdpInetTransport 为基地址），并将其中的引用计数增 1，并返回相应的结构指针，我们暂且称该结构为 StructIPV4，如下图（圈住的为引用计数）



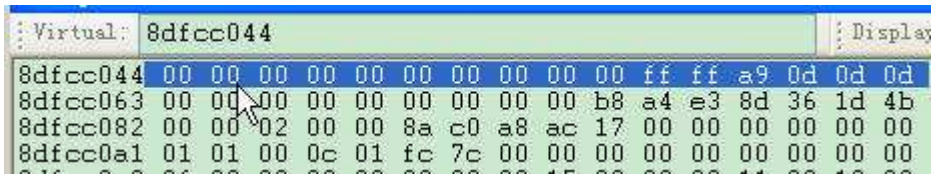
.text:0006754E movesi, [ebp+structA14]

.text:00067551 push [ebp+structA1c]

.text:00067554 mov [esi], eax ；将返回的 ipv4 相关结构 StructIPV4 赋给 StructA 偏移 0x14 位置

.text:00067556 call INETADDR_ADDRESS(x)；返回指向 IP 地址的指针

.text:0006755B cmpdwordptr [eax+0Ch], 0 比较是否得到了 IP 地址，因为该 IP 地址为 IPV4 地址映射而来。而 IPV6 地址为 128 位，IpV4 地址为 32 位，为了表示由 IPV4 地址映射而来，在将 IPv4 转成 IpV6 地址之后，会在其前面添加标志 0xffff。



这里返回的 IP 地址指针存在了 eax 里面，由于是 ipv4 转换而来，因此前面 10 个字节均为 0。因此比较 eax+0C 位置的值是否为 0，从而验证了 IP 地址是否已经设置。

.text:0006755F movecx, [ebp+setaddrflag] ;

.text:00067562 setz al

.text:00067565 mov [ecx], al ；设置相应标志为 0，表明 IP 地址已经设置

.text:00067567 jmp short loc_675B0

检查 StructA 偏移 0x14 处是否已经被赋值

.text:000675B0 movesi, [esi]

.text:000675B2 test esi, esi

.text:000675B4 jnz short loc_675BD

该函数成功则返回 0。

继续 InetResolveSockAddrToLocalAddressAndAf 函数

.text:00067483 test eax, eax

.text:00067485 jl short failed；查看 InetResolveSockAddrToAf 的返回值

.text:00067487 cmp byte ptr [ebp+structA18+3], 0 ；比较刚才设置的标志位，该标志位表明是否设置了 IPaddress，0 表示已经设置了 IP 地址

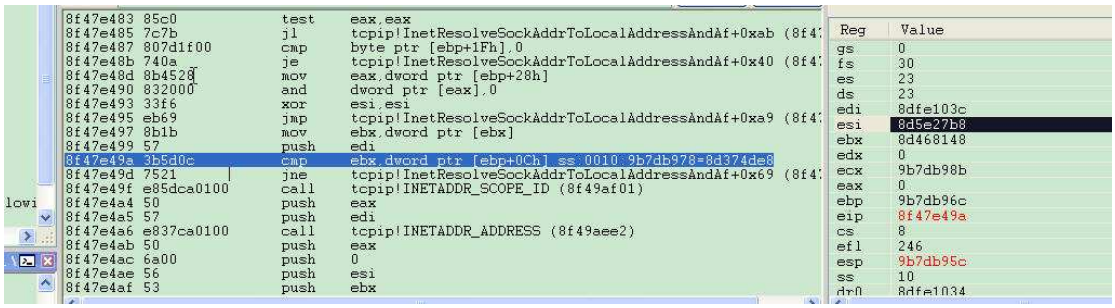
.text:0006748B jz short loc_67497

.text:00067497 movebx, [ebx]；取出 StructA 偏移 0x14 位置存储的值

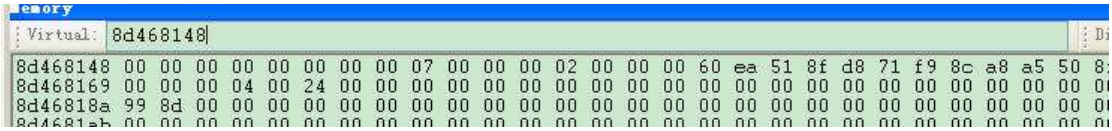
.text:00067499 push edi

.text:0006749A cmpebx, [ebp+structB14]；**比较和 StructB 偏移 0x14 存储的值是否相同，如果不同则为 ipv4 转换为 ipv6 地址的情况**

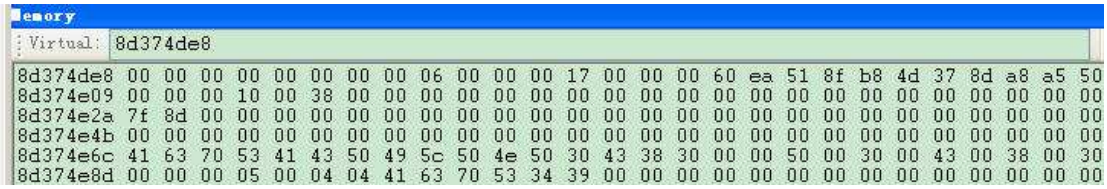
.text:0006749D jnz short ipv4



如上图，这里明显不同



Ipv4 Struct



Ipv6 Struct

.text:000674C0 movesi, [esi]

.text:000674C2 call INETADDR_ADDRESS(x)；取出 IP 地址结构的指针

.text:000674C7 moveax, [eax+0Ch]；指向 IP 地址

.text:000674CA push edi

.text:000674CB mov [ebp+structA18], eax；将其填充到 StructA 偏移 0x18 的位置

.text:000674CE call INETADDR_SCOPE_ID(x)

.text:000674CE

.text:000674D3 push eax

.text:000674D4 lea eax, [ebp+structA18]

.text:000674D7 push eax

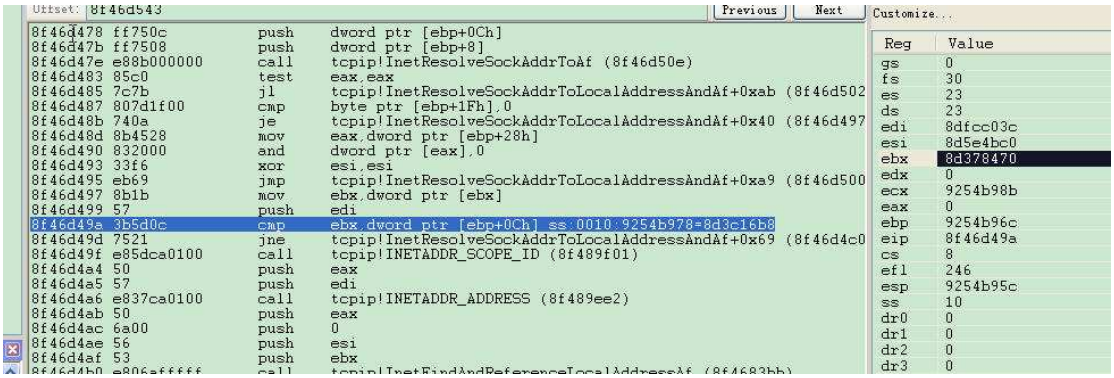
.text:000674D8 push esi

.text:000674D9 xoresi, esi

.text:000674DB push esi

.text:000674DC push ebx

.text:000674DD call InetFindAndReferenceLocalAddressAf(x,x,x,x,x)；此函数检测需要绑定的本地地址是否有效，如果有效则将其对应的结构引用计数增加 1，由于传入的是一个无效地址，因此此函数返回为 0，表示失败；如果该函数成功，则返回 StructLocalAddr，从函数名上来看应该是记录了绑定在本地地址上的一个结构



.text:000674E2 movecx, [ebp+structA10]

.text:000674E5 mov [ecx], eax；将返回的 StructLocalAddr 的结构指针赋给 StructA 偏移 0x10 的位置，但此时为 0

.text:000674E7 cmpeax, esi；检测是否为 0

.text:000674E9 jnz short loc_67500；返回 0 表示失败，所以这里不跳转

.text:000674EB moveax, [ebp+structA14]；将刚存储的 StructIPV4 指针取出来（因为刚才已经将其引用计数增 1，此时需要递减引用计数）

.text:000674EE moveax, [eax]

.text:000674F0 movesi, STATUS_INVALID_ADDRESS_COMPONENT；设置返回值

.text:000674F5 cmpeax, [ebp+structB14]

.text:000674F8 jz short loc_67500；前文述 StructA 偏移 0x14 处的的值和 StructB 偏移 0x14 处的值不同，因此此处不跳转

.text:000674FA push eax

.text:000674FB call InetDereferenceAf(x)；递减 StructIPV4 的引用计数，第一次递减。

InetDereferenceAf 函数如下：

.text:0009466C movedi, edi

.text:0009466E push ebp

.text:0009466F movebp, esp

.text:00094671 moveax, [ebp+arg_0]

.text:00094674 push esi

.text:00094675 lea ecx, [eax+8]

.text:00094678 or esi, 0FFFFFFFFh

.text:0009467B lock xadd [ecx], esi

.text:0009467F decesi ；引用计数减 1

.text:00094680 jnz short loc_94688；如果引用计数为 0 了，则将相应的内存释放

.text:00094682 push eax

.text:00094683 call InetCleanupAf(x)

之后便从 InetResolveSockAddrToLocalAddressAndAf 返回，继续 UdpBindEndpointRequestInspectComplete 函数：

.text:00052B16 mov [ebp+retnvalue], eax；获得返回值，返回值小于

0 的情况则表示失败

```
.text:00052B19                  test    eax, eax
.text:00052B1B                  jge     short success；这里不跳转

.text:00052B55                  movesi, [ebp+structA] ;StructA 指针赋值给 esi
.text:00052B58                  lea     edx, [ebp+var_C]
.text:00052B5B                  movecx, ebx
.text:00052B5D                  call    ds:KeAcquireInStackQueuedSpinLock(x,x)
```

最后跳转至此

```
.text:00052E7B                  push    [ebp+retnvalue]
.text:00052E7E                  push    esi；StructA
.text:00052E7F                  call    UdpBindEndpointInspectComplete(x,x)
```

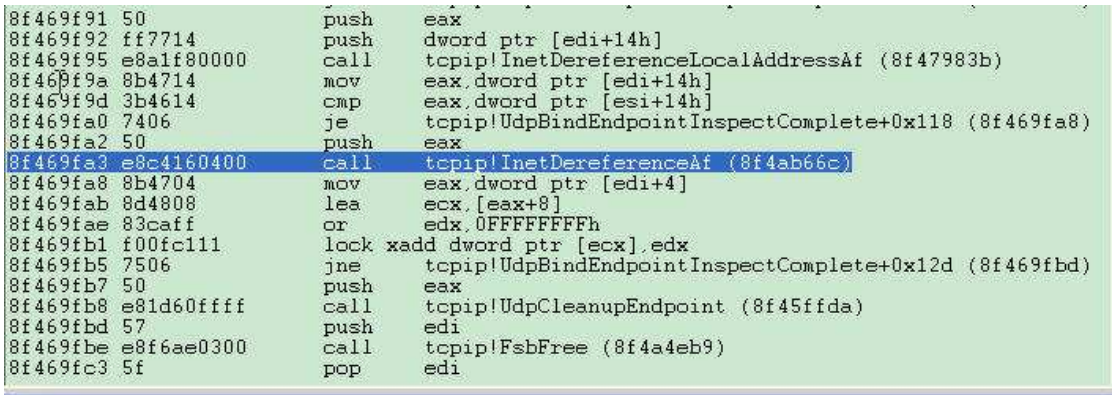
在 UdpBindEndpointInspectComplete 函数中同样有这么一段代码

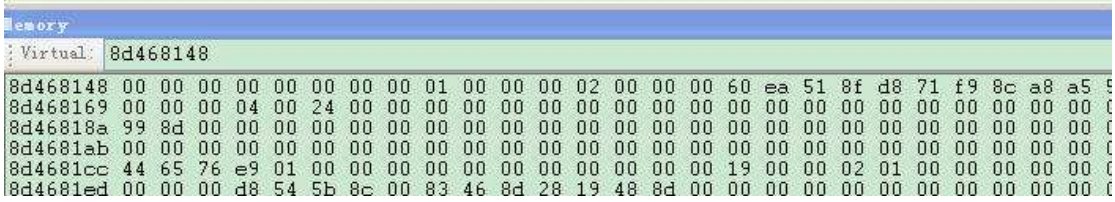
```
.text:00052E9B                  movedi, [ebp+StructA]
.text:00052E9E                  movesi, [edi+4] ;StructB
…
.text:00052F9A                  moveax, [edi+14h]
.text:00052F9D                  cmpeax, [esi+14h]
.text:00052FA0                  jz      short loc_52FA8
.text:00052FA2                  push    eax
```

.text:00052FA3 call InetDereferenceAf(x)；递减 StructIPV4 的引用计数，第二次递减。

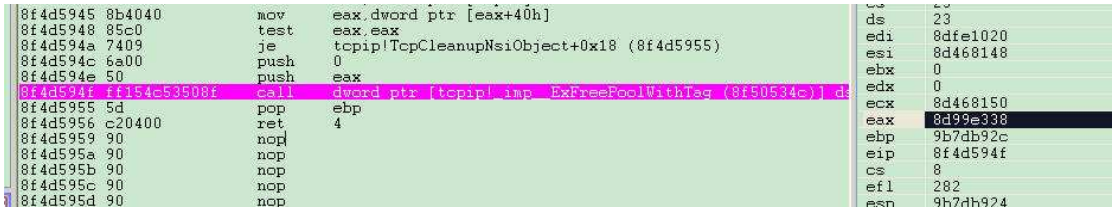
同样是比较 StructA 和 StructB 偏移 0x14 处的值，如果不同（也就是 Ipv4 扩展为 ipv6 的情况）则再一次减小引用计数。这里就出问题了，前面只有一次增加引用计数，却有两次减少引用计数。所以导致程序每执行一次 bind 函数，对应的 Ipv4 Struct 的引用计数则减 1，当减少到 0 的时候，根据 InetDereferenceAf 代码，其对应的一块内存将被释放。

具体地，当引用计数减少到 1 的时候，下一次再执行 bind 操作的时候则会由于第二次引用计数（UdpBindEndpointInspectComplete）调用 InetDereferenceAf 减少引用计数而最终释放掉相应的内存。如图





再一次执行则会由于引用计数归 0 而释放掉一块内存



内存释放

而再一次执行 bind 函数的时候，先增加一次引用计数，在其后 InetResolveSockAddrToLocalAddressAndAf 调用的 InetDereferenceAf 函数中会递减引用计数为 0，再一次释放内存。导致 Double-free 错误。