SMB2: 351 Packets from the Trampoline

Wpis na blogu utworzony przez użytkownika rapid7-admin w 2009-10-04

"Lubię to" • 0   Komentarz • 3

Originally Posted by hdm

This a guest blog entry written by Piotr Bania .

Disclaimer

Prologue

About a month ago Laurent Gaffié released an advisory in which he described the SMB 2.0 NEGOTIATE PROTOCOL REQUEST Remote BSoD vulnerability. Fortunately for some and unfortunately for others this vulnerability is remotely exploitable. At the time of writing, there are only two exploits available for this flaw, one written by Immunity Inc., which only provides a copy to paying customers, and one written by Stephen Fewer and included in the Metasploit Framework. Unfortunately, Stephen Fewer's exploit seems to be unreliable against physical machines (vs VMs) due to a hardcoded address from the BIOS/HAL memory region (0xFFD00D09) which must be initiated to "POP ESI; RET". In this article I am going to describe a method for exploiting this vulnerability that only requires a stable absolute memory address (filled with NULL bytes).

Step One. Where to?

First, lets take a look at the vulnerable code, we will assume a Windows Vista SP2 operating system and SRV2.SYS version 6.0.6002.18005:

```
.text:000056B3 loc_56B3:                              ; CODE XREF: Smb2ValidateProviderCallback(x)+4D5↑j
.text:000056B3                                        ; Smb2ValidateProviderCallback(x)+4DE↑j
.text:000056B3                 movzx   eax, word ptr [esi+0Ch]
.text:000056B7                 mov     eax, _ValidateRoutines[eax*4]
.text:000056BE                 test    eax, eax
.text:000056C0                 jnz     short loc_56C9
.text:000056C2                 mov     eax, 0C0000002h
.text:000056C7                 jmp     short loc_56CC
.text:000056C9 ; ---------------------------------------------------------------------------
.text:000056C9
.text:000056C9 loc_56C9:                              ; CODE XREF: Smb2ValidateProviderCallback(x)+4F3↑j
.text:000056C9                 push    ebx
.text:000056CA                 call    eax ; Smb2ValidateNegotiate(x) ; Smb2ValidateNegotiate(x)
```

At offset 0x000056B3 EAX is initialized with a word from [ESI+0Ch]. The [ESI+0Ch] location points to the SMB2 packet, giving the attacker complete control on the lower 16 bits of the EAX register (AX). In the next instruction (0x000056B7) our controlled EAX is used as an array index. There is only one safety check on this value that verifies that *(DWORD*)ValidateRoutines[EAX*4] is not NULL. This is the cause of this vulnerability, since there is no check to determine if the EAX value (array index) exceeds the number of elements in the ValidateRoutines array. Further in the code, the location pointed to by ValidateRoutines[EAX*4] is executed by the "call EAX" instruction (0x000056CA).

In summary, we can redirect execution to any location (as long as it is not null) from ValidateRoutines to (ValidateRoutines + (0xFFFF

* 4)). This gives us about 2^16 potential memory locations to check. this is not completely accurate, since we cannot assume that any memory location outside the SRV2.SYS address space will be consistent across mul;tiple machines (device driver ImageBase addresses change on every boot). To make my life less miserable, I wrote a little program that dumps the SRV2.SYS address space from system memory, then disassembles every potential region that can be reached through ValidateRoutines[INDEX*4]. Additionally, I set some boundaries that ensure we are operating only on the SRV2.SYS address space. Here are the results I have obtained:

I must confess that I was confused at first, not because of the results obtained, but due to the Immunity exploit video that was released. In this video, they stated that exploitation is based on on time values. This led me to focus on any function that manipulated time values. I noticed that the SrvBalanceCredits function (index 0x31, 0x4b7) can be used to modify the CurrentTime structure (0x0001D320), which can then be used again later as the memory address for a "call EAX". However, since KeQuerySystemTime returns the time as a count of 100-nanosecond intervals since January 1, 1601 and the system time is typically updated approximately every ten milliseconds, it is very unlikely to use this as reliable offset. An alternative would be to use the BootTime variable and reboot the machine to reset it, however my results were still not satisfying (the BootTime and CurrentTime values are both returned as part of a normal SMB2 NEGOTIATE_RESPONSE packet, so it is possible to query these remotely).

I decided that the time approach was a dead end and that it was time to start over from scratch and never watch Immunity videos again :-) After leaving the time approach I decided to look into the functions that would corrupt the stack by using a accepting a different number of arguments than the original function. The following indexes showed the most promise: 0x217 (srv2!SrvSnapShotScavengerTimer), 0x237 (srv2!SrvScavengerTimer), 0x1e3 (srv2!SrvScavengeDurableHandlesTimer), and 0x1bb (srv2!SrvProcessOplockBreakTimer). Stephen Fewer's exploit uses the 0x217 (srv2!SrvSnapShotScavengerTimer) as a index value. All four of those indexes have something in common:

```
kd> u poi(srv2!ValidateRoutines+(0x217*4))
srv2!SrvSnapShotScavengerTimer:
97e0ef35 6a01            push    1
97e0ef37 68809ae197      push    offset srv2!SrvSnapShotScavengerState (97e19a80)
97e0ef3c ff151880e197    call    dword ptr [srv2!_imp__ExQueueWorkItem (97e18018)]
97e0ef42 c21000          ret     10h
97e0ef45 90              nop
97e0ef46 90              nop
97e0ef47 90              nop
97e0ef48 90              nop
kd> u poi(srv2!ValidateRoutines+(0x237*4))
srv2!SrvScavengerTimer:
97dfeeab 6a01            push    1
97dfeead 68009be197      push    offset srv2!SrvScavengerState (97e19b00)
97dfeeb2 ff151880e197    call    dword ptr [srv2!_imp__ExQueueWorkItem (97e18018)]
97dfeeb8 c21000          ret     10h
97dfeebb 90              nop
97dfeebc 90              nop
97dfeebd 90              nop
97dfeebe 90              nop
kd> u poi(srv2!ValidateRoutines+(0x1e3*4))
srv2!SrvScavengeDurableHandlesTimer:
97e0f4c3 6a00            push    0
97e0f4c5 68a099e197      push    offset srv2!Smb2Dur (97e199a0)
97e0f4ca ff151880e197    call    dword ptr [srv2!_imp__ExQueueWorkItem (97e18018)]
97e0f4d0 c21000          ret     10h
97e0f4d3 90              nop
97e0f4d4 90              nop
97e0f4d5 90              nop
97e0f4d6 90              nop
kd> u poi(srv2!ValidateRoutines+(0x1bb*4))
srv2!SrvProcessOplockBreakTimer:
97e0fb2f 6a00            push    0
97e0fb31 680099e197      push    offset srv2!SrvOplockState (97e19900)
97e0fb36 ff151880e197    call    dword ptr [srv2!_imp__ExQueueWorkItem (97e18018)]
97e0fb3c c21000          ret     10h
```

Each of those functions ends with a "ret 10h", indicating the function expects four arguments, and will adjust the stack to account for those when it returns. To see how this helps us, lets take one more look at the vulnerable code:

```
.text:000056C9 loc_56C9:                        ; CODE XREF: Smb2ValidateProviderCallback(x)+4F3↑j
.text:000056C9                 push    ebx
.text:000056CA                 call    eax ; Smb2ValidateNegotiate(x) ; Smb2ValidateNegotiate(x)
.text:000056CC
.text:000056CC loc_56CC:                        ; CODE XREF: Smb2ValidateProviderCallback(x)+94↑j
.text:000056CC                                  ; Smb2ValidateProviderCallback(x)+290↑j ...
.text:000056CC                 mov     ecx, [ebp+var_4]
.text:000056CF                 pop     edi
.text:000056D0                 pop     esi             ; esi=packet
.text:000056D1                 xor     ecx, ebp
.text:000056D3                 pop     ebx
.text:000056D4                 call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:000056D9                 leave
.text:000056DA                 retn    4               ; back to: 1fa7f
.text:000056DA _Smb2ValidateProviderCallback@4 endp    ;
```

As you can see, the procedure pointed to by EAX is called (0x000056CA) with one argument on the stack (see 0x000056C9 - PUSH EBX). SRV2.SYS assumes that the called function is using the stdcall convention (callee is responsible for cleanup of the stack). Since we forced EAX to point to one of the "ret 10" functions, the callee will clean the stack, but adjust it for for four parameters, not just the single parameter that was passed in (0x10=16 -> 16/4=4). How does this influence the execution flow? Take a look:

```
Breakpoint 0 hit
srv2!Smb2ValidateProviderCallback+0x4fe:
99e096c9 53              push    ebx
kd> d esp
9a7ead0c  00 00 00 00 e8 05 90 92-a4 ba 73 85 84 5b 8d 92  ..........s..[..
9a7ead1c  f0 65 7a 85 88 07 90 92-80 5c 8d 92 48 b5 73 85  .ez......\..H.s.
9a7ead2c  20 5b 8d 92 cc 3f e2 99-00 00 00 00 75 b6 9c 03   [...?......u...
9a7ead3c  50 ad 7e 9a 7f 3a e2 99-e8 05 90 92 58 ba 73 85  P.~..:......X.s.
9a7ead4c  e8 05 90 92 7c ad 7e 9a-9f 21 e2 99 e8 05 90 92  ....|.~..!......
9a7ead5c  00 00 00 00 48 b5 73 85-00 00 00 00 00 00 00 00  ....H.s.........
9a7ead6c  00 00 00 00 80 ad 7e 9a-01 00 00 00 01 00 00 00  ......~.........
9a7ead7c  c0 ad 7e 9a bd 65 9e 81-00 00 00 00 d1 14 1c 12  ..~..e..........
kd> p
srv2!Smb2ValidateProviderCallback+0x4ff:
99e096ca ffd0            call    eax
kd> p
srv2!Smb2ValidateProviderCallback+0x501:
99e096cc 8b4dfc          mov     ecx,dword ptr [ebp-4]
kd> d esp
9a7ead18  84 5b 8d 92 f0 65 7a 85-88 07 90 92 80 5c 8d 92  .[...ez......\..
9a7ead28  48 b5 73 85 20 5b 8d 92-cc 3f e2 99 00 00 00 00  H.s. [...?......
9a7ead38  75 b6 9c 03 50 ad 7e 9a-7f 3a e2 99 e8 05 90 92  u...P.~..:......
9a7ead48  58 ba 73 85 e8 05 90 92-7c ad 7e 9a 9f 21 e2 99  X.s.....|.~..!..
9a7ead58  e8 05 90 92 00 00 00 00-48 b5 73 85 00 00 00 00  ........H.s.....
9a7ead68  00 00 00 00 00 00 00 00-80 ad 7e 9a 01 00 00 00  ..........~.....
9a7ead78  01 00 00 00 c0 ad 7e 9a-bd 65 9e 81 00 00 00 00  ......~..e......
9a7ead88  d1 14 1c 12 00 00 00 00-00 00 00 00 00 00 00 00  ................
kd> d poi(esp+4)
857a65f0  ff 53 4d 42 72 00 00 00-00 18 53 c8 37 02 00 00  .SMBr.....S.7...
857a6600  00 00 00 00 00 00 00 00-ff ff ff fe 00 00 00 00  ................
857a6610  00 8e 03 02 53 4d 42 20-32 2e 30 30 32 00 90 90  ....SMB 2.002...
```

The first "d ESP" command shows the stack before the "CALL EAX" (where EAX points to on of the "ret 10" procedures). The second "d ESP" shows the stack after the "ret 10" function was executed. The important part is when the "POP ESI" (0x000056D0) instruction is executed, it will be exchanged with the pointer to our SMB packet (see "d poi(esp+4)") -- this will bring us some serious kudos later. Additionally, even if at the moment the stack pointer is invalid (because we haxored it) it will be reinitialized correctly by the instruction at 0x000056D9. As you probably know, the LEAVE instruction (also called High Level Procedure Exit), sets the ESP to EBP and pops EBP. In other words, despite the fact we have mangled the stack and forced ESI to point to our packed data, ESP will be "good" again. That is important, since otherwise it would cause an exception when executing the "ret 4". Lets assume we used 0x237 (srv2!SrvScavengerTimer) as an index, after few instructions we land here:

```
PAGE:0001FAB1 loc_1FAB1:                              ; CODE XREF: SrvProcessPacket(x)+76↑j
PAGE:0001FAB1                 setnl   cl
PAGE:0001FAB4                 push    ecx
PAGE:0001FAB5                 push    eax
PAGE:0001FAB6
PAGE:0001FAB6 loc_1FAB6:                              ; CODE XREF: SrvProcessPacket(x)+6B↑j
PAGE:0001FAB6                                         ; SrvProcessPacket(x)+7B↑j
PAGE:0001FAB6                 push    esi             ; esi=pakiet
PAGE:0001FAB7                 call    _SrvProcCompleteRequest@12 ; SrvProcCompleteRequest(x,x,x)
PAGE:0001FABC
```

As you can see, ESI still points to our packet. The instruction at 0x0001FAB1 (setnl cl) is also a key factor in the way I have chosen to exploit this, since the setnl result depends on the value our called "faked function", which is why a function like 0x1e3 (srv2!SrvScavengeDurableHandlesTimer) will not work), since the CL register must be 1 before the PUSH ecx is executed. This will be discussed later.

Step Two. Mum I want a Trampoline!

In this step we will create a trampoline that will transfer the code execution to the shellcode. Stephen's exploit code depended on a static "pop esi; ret" address that made it unreliable on many non-virtual machines. With my technique, we just need to find a stable 4-byte memory region filled with NULL bytes (or any other predictable value) and we will force the SMB code to build a trampoline for us, using just 351 packets. After some digging I found following piece of code interesting (located in the end of _SrvProcPartialCompleteCompoundedRequest@8 function):

```
PAGE:00021156 zajebistosc:                            ; CODE XREF: SrvProcPartialCompleteCompoundedRequest(x,x)+8C↑j
PAGE:00021156                 xor     ecx, ecx
PAGE:00021158                 lea     eax, [ebx+0BCh]
PAGE:0002115E                 inc     ecx
PAGE:0002115F                 lock xadd [eax], ecx    ; [eax]++
PAGE:00021163                 inc     ecx
PAGE:00021164                 cmp     ecx, [ebx+0CCh]
PAGE:0002116A                 jns     short loc_21176
PAGE:0002116C                 push    ebx
PAGE:0002116D                 call    _SrvProcCompleteCompoundedRequest@4 ; SrvProcCompleteCompoundedRequest(x)
PAGE:00021172                 mov     byte ptr [ebp+paket+3], 1
PAGE:00021176
PAGE:00021176 loc_21176:                              ; CODE XREF: SrvProcPartialCompleteCompoundedRequest(x,x)+19C↑j
PAGE:00021176                 mov     al, byte ptr [ebp+paket+3]
```

The instruction located at 0x0002115F is used to automically increase the value pointed to by the EAX register by ECX (=1). This is actually a variation of the InterlockedExchangeAdd function. The key point here is that the EAX register value is controlled by the SMB packet and ECX is set to 1. Lets review how the EBX register value is computed:

```
PAGE:00020FCE _SrvProcPartialCompleteCompoundedRequest@8 proc near
PAGE:00020FCE                                         ; CODE XREF: SrvProcCompleteRequest(x,x,x)+3E↑p
PAGE:00020FCE
PAGE:00020FCE ptr_moj         = dword ptr -4
PAGE:00020FCE paket           = dword ptr  8
PAGE:00020FCE arg_4           = byte ptr  0Ch
PAGE:00020FCE
PAGE:00020FCE                 mov     edi, edi
PAGE:00020FD0                 push    ebp
PAGE:00020FD1                 mov     ebp, esp
PAGE:00020FD3                 push    ecx
PAGE:00020FD4                 cmp     [ebp+arg_4], 0
PAGE:00020FD8                 push    ebx
PAGE:00020FD9                 push    esi
PAGE:00020FDA                 mov     esi, [ebp+paket]
PAGE:00020FDD                 mov     ebx, [esi+0ACh]
```

In the code above, you can that EBX is equal to the [packet+0xAC] field. This means that the memory region that is be increased by the xadd instruction is equal to [packet+0xAC]+0xBC (this offset changes among the different Vista versions). This provides us with full control of the area that will be increased by each request. So what we are going to do with it? We are going to build a trampoline, dumbass :-)

To do that we, must consider:

1) We need an absolute memory address that is executable (see DEP) and is filled with constant data (NULLs in our case, however thanks to the xadd arithmetic operation any stable value works). We
need four bytes of NULLs at the address and an additional three bytes before it to handle overlapping writes to reduce the number of packets required.

2) We need to know what value to compute and how many requests it will take to accomplish this.

Answers:

1) Lets use the same BIOS/HAL region chosen by Stephen's exploit, since the memory here is readable, writeable, and executable. NULL bytes in this region are much easier to find than a POP ESI;RET for sure!

2) It seems that the opcode sequence "INC ESI; POP ESI; RET" (0x46 0x5E 0xC3) would be the easiest way to bounce to our shellcode using this as a trampoline. However, writing the value 0x4656C3 with a single increment per require would require us to send 4,609,731 packets. Fortunately, there is a solution that reduces this to just 351 packets -- a much more reasonable number. The trick is to divide the process into three stages, where each stage is responsible for increasing only one byte. For example, we send 0x46 packets to increment address+0, 0x65 packets to increment address+1, and 0xC3 packets to increment loc+2.

Step Three. Code Execution

Now that the trampoline is ready we just need to jump to it, here is the code responsible for that:

```
PAGE:0001FB8E exec_proc:                              ; CODE XREF: SrvProcCompleteRequest(x,x,x)+B7↑j
PAGE:0001FB8E                                         ; SrvProcCompleteRequest(x,x,x)+BF↑j
PAGE:0001FB8E                 mov     eax, [esi+168h] ; fnction offset
PAGE:0001FB94                 cmp     eax, edi
PAGE:0001FB96                 jz      short func_pointer_null
PAGE:0001FB98                 push    esi
PAGE:0001FB99                 call    eax
PAGE:0001FB9B
```

EAX (call desitnation address) is fully controlled by the value from the SMB packet (ESI+168h). This offset changes does change between different Vista versions. Here's the general schema of my attack:

That is all for now, expect to see an updated Metasploit module in the near future that takes advantage of this technique.

Widoczność: Metasploit • **1224 wyświetleń**

Data ostatniej modyfikacji: 2011-04-05 20:42