# 漏洞分析视角下的CVE-2020-0796漏洞

深澜深蓝/stxletto

## 概述

* 2020年3月10日是微软补丁日，安全社区注意到Microsoft发布并立即删除了有关CVE-2020-0796的信息；
* 2020年3月11日早上，[Microsoft发布](https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-0796)了可纠正SMBv3协议如何处理特制请求的[修补程序](https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-0796)；
* 2020年03月12日微软发布安全公告声称Microsoft 服务器消息块 3.1.1 (SMBv3) 协议处理某些请求的方式中存在远程执行代码漏洞。成功利用此漏洞的攻击者可以获取在目标服务器或客户端上执行代码的能力。要利用针对服务器的漏洞，未经身份验证的攻击者可以将特制数据包发送到目标 SMBv3 服务器。要利用针对客户端的漏洞，未经身份验证的攻击者将需要配置恶意的 SMBv3 服务器，并说服用户连接到该服务器。此安全更新通过更正 SMBv3 协议处理这些特制请求的方式来修复此漏洞。
* 此缺陷可影响SMB协商中的客户端和服务端。服务端漏洞位于srv2.sys中，客户端漏洞位于mrxsmb.sys中，这两个漏洞最终都在SmbCompressDecompress中调用了相同的代码。

本文试以CVE-2020-0796为例，为读者呈现漏洞分析工作视角。

## 受影响的系统

Windows 10 Version 1903 for 32-bit Systems

Windows 10 Version 1903 for ARM64-based Systems

Windows 10 Version 1903 for x64-based Systems

Windows 10 Version 1909 for 32-bit Systems

Windows 10 Version 1909 for ARM64-based Systems

Windows 10 Version 1909 for x64-based Systems

Windows Server, version 1903 (Server Core installation)

Windows Server, version 1909 (Server Core installation)

## 分析

首先我们来执行CVE-2020-0796的PoC

1. PS C:\Users\admin\CVE-2020-0796\> python .\poc.py 192.168.0.10
2. Connected
3. Sent negotiate packet 1
4. Target responded with 452 bytes
5. Sent negotiate packet 2
6. Target responded with 534 bytes
7. Crash bytes sent
8. winexcept timed out



图 1

如果目标系统未处于调试状态，我们将观察到目标设备进入蓝屏状态。待Windows系统重启后，我们会使用WinDBG打开C:\Windows\System32\MEMORY.DMP文件，通过分析内存转储文件尝试找到触发蓝屏的原因。

如果目标系统处于调试状态，将会在WinDBG中观测到如下图所示的中断：

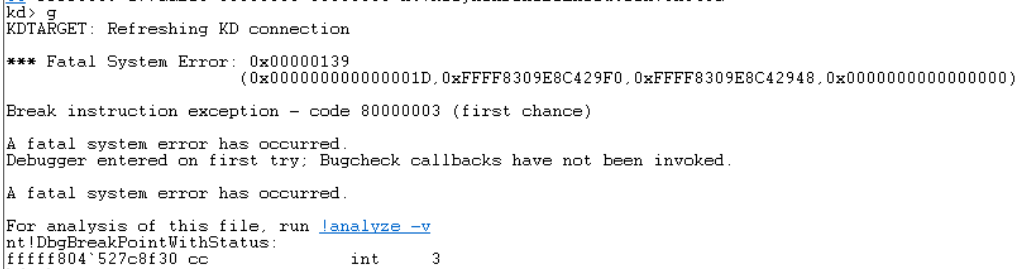


图 2

### 释放内存的错误

无论是任何一种情况，大多时候在WinDBG中首选执行!analyze -v，尝试由WinDBG自动分析导致问题的模块。

或者查看栈回溯

kd> kn

# Child-SP RetAddr Call Site

..

0b fffff904`bd3a2dd0 fffff806`1b97e5ae nt!ExFreePool+0x9

0c fffff904`bd3a2e00 fffff806`1b9d7f41 srvnet!SmbCompressionDecompress+0xfe

0d fffff904`bd3a2e70 fffff806`1b9d699e srv2+0x17f41

0e fffff904`bd3a2ed0 fffff806`1ba19a9f srv2+0x1699e

0f fffff904`bd3a2f00 fffff806`1cdc496e srv2+0x59a9f

..

如上文0x0C号栈帧所示，srvnet模块中的SmbCompressionDecompress函数在调用ExFreePool时是触发蓝屏的直接因素。

同时，我们注意到上文0x0D号栈帧所示的返回函数是模块名+偏移量的形式，这是因为WinDBG没有加载srv2模块的的符号文件。加载srv2模块的符号之后，栈回溯更有可读性：

kd> lml

start end module name

fffff806`1b960000 fffff806`1b9b3000 srvnet (pdb symbols) c:\symbol\srvnet.pdb\CFE2BF7A30464E7FCE0CC805AA1C96CB1\srvnet.pdb

fffff806`1b9c0000 fffff806`1ba85000 srv2 (pdb symbols) c:\symbol\srv2.pdb\E423CC65395AE603B3F59D9322DB98F31\srv2.pdb

fffff806`1cc00000 fffff806`1d6b5000 nt (pdb symbols) c:\symbol\ntkrnlmp.pdb\CE7FFB00C20B87500211456B3E905C471\ntkrnlmp.pdb

..

kd> kn

# Child-SP RetAddr Call Site

00 fffff904`bd3a1f28 fffff806`1cea92a2 nt!DbgBreakPointWithStatus

01 fffff904`bd3a1f30 fffff806`1cea8992 nt!KiBugCheckDebugBreak+0x12

02 fffff904`bd3a1f90 fffff806`1cdc11a7 nt!KeBugCheck2+0x952

03 fffff904`bd3a2690 fffff806`1cdd2ee9 nt!KeBugCheckEx+0x107

04 fffff904`bd3a26d0 fffff806`1cdd3310 nt!KiBugCheckDispatch+0x69

05 fffff904`bd3a2810 fffff806`1cdd16a5 nt!KiFastFailDispatch+0xd0

06 fffff904`bd3a29f0 fffff806`1cdfa745 nt!KiRaiseSecurityCheckFailure+0x325

07 fffff904`bd3a2b88 fffff806`1cc44380 nt!RtlRbRemoveNode+0x1b6145

08 fffff904`bd3a2ba0 fffff806`1cc43e3a nt!RtlpHpVsChunkCoalesce+0xb0

09 fffff904`bd3a2c10 fffff806`1cc460ad nt!RtlpHpVsContextFree+0x18a

0a fffff904`bd3a2cb0 fffff806`1cf6e0a9 nt!ExFreeHeapPool+0x56d

0b fffff904`bd3a2dd0 fffff806`1b97e5ae nt!ExFreePool+0x9

0c fffff904`bd3a2e00 fffff806`1b9d7f41 srvnet!SmbCompressionDecompress+0xfe

0d fffff904`bd3a2e70 fffff806`1b9d699e srv2!Srv2DecompressData+0xe1

0e fffff904`bd3a2ed0 fffff806`1ba19a9f srv2!Srv2DecompressMessageAsync+0x1e

0f fffff904`bd3a2f00 fffff806`1cdc496e srv2!RfspThreadPoolNodeWorkerProcessWorkItems+0x13f

10 fffff904`bd3a2f80 fffff806`1cdc492c nt!KxSwitchKernelStackCallout+0x2e

11 fffff904`bd3478f0 fffff806`1cc6a33e nt!KiSwitchKernelStackContinue

12 fffff904`bd347910 fffff806`1cc6a13c nt!KiExpandKernelStackAndCalloutOnStackSegment+0x18e

13 fffff904`bd3479b0 fffff806`1cc69fb3 nt!KiExpandKernelStackAndCalloutSwitchStack+0xdc

14 fffff904`bd347a20 fffff806`1cc69f6d nt!KeExpandKernelStackAndCalloutInternal+0x33

15 fffff904`bd347a90 fffff806`1ba197f7 nt!KeExpandKernelStackAndCalloutEx+0x1d

16 fffff904`bd347ad0 fffff806`1d316917 srv2!RfspThreadPoolNodeWorkerRun+0x117

17 fffff904`bd347b30 fffff806`1cd2a715 nt!IopThreadStart+0x37

18 fffff904`bd347b90 fffff806`1cdc86ea nt!PspSystemThreadStartup+0x55

19 fffff904`bd347be0 00000000`00000000 nt!KiStartSystemThread+0x2a

根据函数名称字面理解或参考DDK文档ExFreePool是释放内存的函数，一般不会有什么问题。这个涉及Windows内核的Pool内存管理机制及结构。过往经验告诉我们，ExFreePool需要操作的内存结构被破坏掉了，即这可能是个Windows内核中的内存破坏漏洞(Memory Corruption)。

人生终极三问：你是谁？从哪里来？到哪里去？在漏洞分析领域同样适用。

为搞明白ExFreePool要释放的内存，来自哪里，又是被谁搞坏的。我们需要在IDA Pro中看看srvnet模块中的SmbCompressionDecompress函数。



图 3

当然如果你那边IDA Pro显示的和上图所示不同，没有这些可读性较好的变量名，而是像下图这样

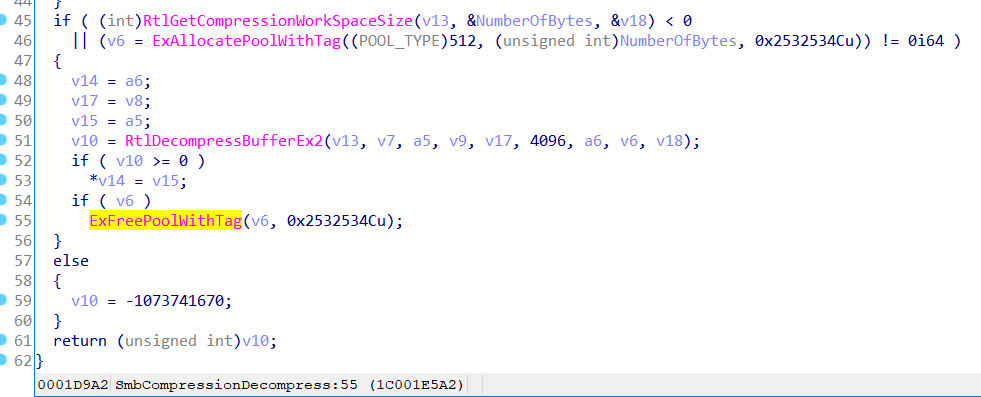


图 4

也不必惊讶，后续我们会解释，如何通过公开的文档、符号文件或者数据流，注解IDA Pro函数名或者变量名，使得显示更加友好，以便开展分析工作。这个过程有点像Windows系统自带的扫雷游戏。

IDA Pro显示srvnet模块中的SmbCompressionDecompress函数主要流程十分清晰：申请内存(ExAllocatePoolWithTag)、解压处理(RtlDecompressBufferEx2)、释放内存(ExFreePoolWithTag)。

我们现在已知蓝屏的直接原因是释放内存的操作引起的，那么问题就显然出现在成功申请内存之后，到释放内存之间的这个过程中。我们看到这个过程中只有一个处理函数，即RtlDecompressBufferEx2。

现在所有的疑点都集中在了RtlDecompressBufferEx2函数上，

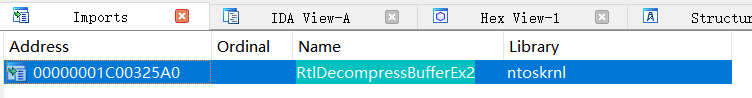


图 5

我们来看看这个ntoskrnl模块中的RtlDecompressBufferEx2函数。

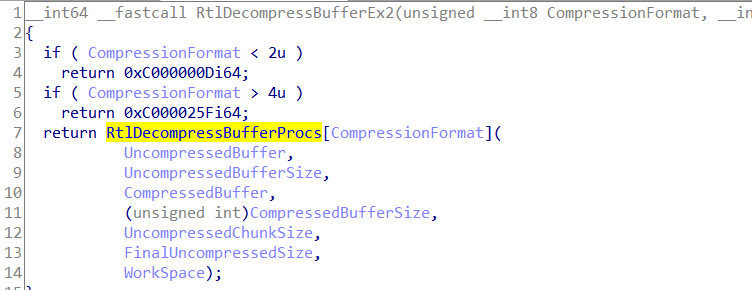


图 6

IDA Pro显示RtlDecompressBufferEx2函数是根据参数CompressionFormat的一个跳转函数。

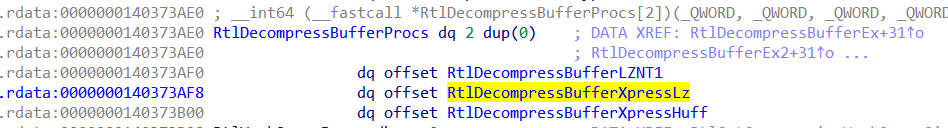


图 7

RtlDecompressBufferProcs数组前2个QWORD元素为0。即当CompressionFormat取值为3时，函数最终转向RtlDecompressBufferXpressLz函数中。

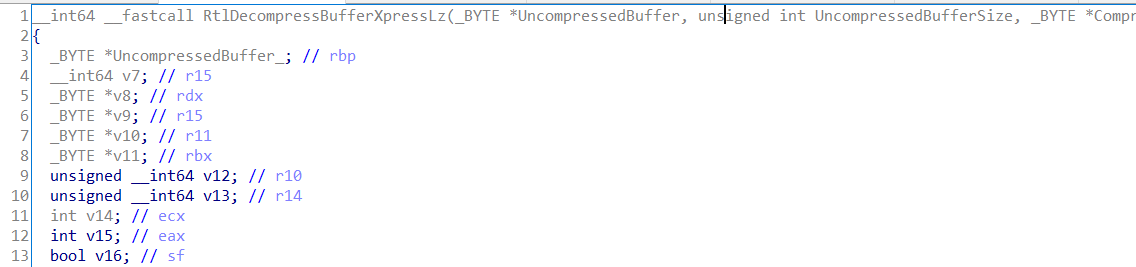


图 8

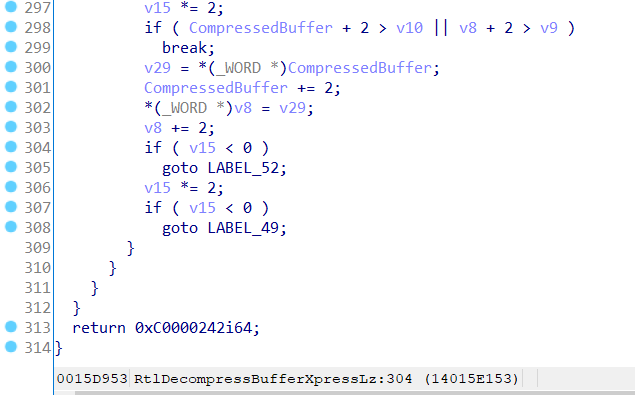


图 9

IDA Pro显示RtlDecompressBufferXpressLz函数是一个300多行伪代码的复杂函数。

静态分析有点吃力，为了快速定位问题，让我们来试试用WinDBG动态调试一下。

还是执行PoC，windbg中断时执行kn或者!analyze -v。这次我们试试!analyze -v。

FOLLOWUP\_IP:

nt!RtlDecompressBufferXpressLz+2d0

fffff800`4575e3c0 f3a4 rep movs byte ptr [rdi],byte ptr [rsi]

FAULT\_INSTR\_CODE: c085a4f3

SYMBOL\_STACK\_INDEX: 7

SYMBOL\_NAME: nt!RtlDecompressBufferXpressLz+2d0

FOLLOWUP\_NAME: MachineOwner

MODULE\_NAME: nt

IMAGE\_NAME: ntkrnlmp.exe

DEBUG\_FLR\_IMAGE\_TIMESTAMP: 0

STACK\_COMMAND: .thread ; .cxr ; kb

BUCKET\_ID\_FUNC\_OFFSET: 2d0

FAILURE\_BUCKET\_ID: AV\_INVALID\_nt!RtlDecompressBufferXpressLz

BUCKET\_ID: AV\_INVALID\_nt!RtlDecompressBufferXpressLz

PRIMARY\_PROBLEM\_CLASS: AV\_INVALID\_nt!RtlDecompressBufferXpressLz

太棒了，我们和WinDBG达成了共识。它直接提示可能是nt!RtlDecompressBufferXpressLz+2d0处出了问题。

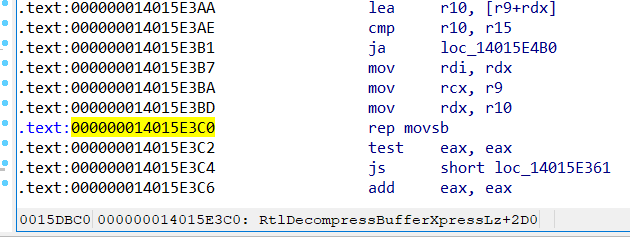


图 10

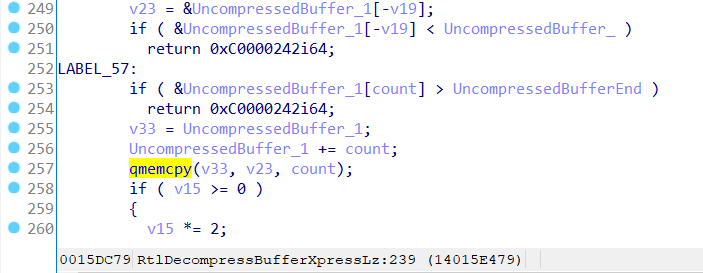


图 11

现在我们了解到nt!RtlDecompressBufferXpressLz+2d0处是一个内存复制函数qmemcpy。这符合往常的漏洞构成的元素。

我们需要再了解一下qmemcpy里面的这3个参数。

kd> !pool 0xffffe402f06b3000

Pool page ffffe402f06b3000 region is Nonpaged pool

\*ffffe402f06b3000 : large page allocation, tag is LS2%, size is 0xef30 bytes

Pooltag LS2% : LM server allocations

kd> !pool 0xffffe402f06b3000+0xef30

Pool page ffffe402f06c1f30 region is Nonpaged pool

\*ffffe402f06c1f30 size: b0 previous size: 0 (Free) \*...&

Owning component : Unknown (update pooltag.txt)

ffffe402f06c1fe0 size: 10020 previous size: 0 (Free) ...&

我们设置一个这样的断点：

bp nt!RtlDecompressBufferXpressLz+0x2D0 ".printf \"RtlDecompressBufferXpressLz(), qmemcpy(dst=0x%I64x, src=0x%I64x, count=0x%I64x)\", rdi, rsi, r9;.echo"

kd> r

rax=00000000fffffffe rbx=ffffe402e90a544f rcx=000000008483ffff

rdx=ffffe40371234438 rsi=ffffe402ec9f4438 rdi=ffffe402ec9f4439

rip=fffff8015a75e3c0 rsp=ffff890c4ed8ad98 rbp=ffffe402ec9f4438

r8=ffffe402e90a5457 r9=000000008483ffff r10=ffffe40371234438

r11=ffffe402e90a5457 r12=0000000000000000 r13=ffffe402e373bd00

r14=ffffe402e90a5401 r15=ffffe403ec9f4437

iopl=0 nv up ei ng nz na pe cy

cs=0010 ss=0018 ds=002b es=002b fs=0053 gs=002b efl=00040283

nt!RtlDecompressBufferXpressLz+0x2d0:

fffff801`5a75e3c0 f3a4 rep movs byte ptr [rdi],byte ptr [rsi]

可得我们感兴趣的qmemcpy的3个参数：

qmemcpy(dst=0xffffe402ec9f4439, src=0xffffe402ec9f4438, count=0x8483ffff)

查看一下目的内存的pool信息：

kd> !pool 0xffffe402ec9f4439

Pool page ffffe402ec9f4439 region is Nonpaged pool

\*ffffe402ec9f4000 : large page allocation, tag is LS00, size is 0x1280 bytes

Pooltag LS00 : SRVNET LookasideList level 0 allocation 256 Bytes, Binary : srvnet.sys

这是一个0x1280大小的非分页池内存。qmemcpy函数准备向其中写入0x8483ffff大小的数据。很显然会溢出。

kd> !pool 0xffffe402ec9f4000+0x1280

Pool page ffffe402ec9f5280 region is Nonpaged pool

\*ffffe402ec9f5280 size: 700 previous size: 0 (Free) \*...&

Owning component : Unknown (update pooltag.txt)

ffffe402ec9f5990 size: 290 previous size: 0 (Allocated) MmCi

ffffe402ec9f5c20 size: 3c0 previous size: 0 (Free) ...&

对于Pool内存的大小不超过一个页面长度(PAGE\_SIZE，即4K字节)时，可以通过使用POOL\_HEADER结构体来查看pool块信息。

我们注意到0xffffe402ec9f4000之后在ffffe402ec9f5280 处是一个0x700大小的空闲块，再之后ffffe402ec9f5990 处是一个0x290 大小的已被分配使用的块。

kd> !poolval 0xffffe402ec9f4000+0x1280

Pool page ffffe402ec9f5280 region is Nonpaged pool

Validating Pool headers for pool page: ffffe402ec9f5280

Pool page [ ffffe402ec9f5000 ] is INVALID.

Analyzing linked list...

[ ffffe402ec9f5000 ]: invalid previous size [ 0x41 ] should be [ 0x0 ]

Scanning for single bit errors...

None found

在qmemcpy函数执行后，我们发现ffffe402ec9f5280处的\_POOL\_HEADER确实被写入了数据。

### 复制数据的大小

现在我们需要搞明白，复制数据大小和目的地址的来源。

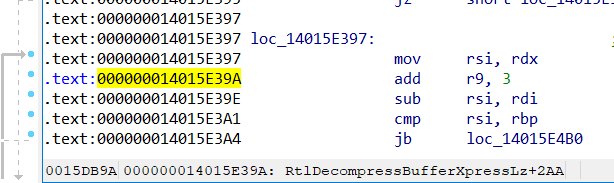


图 12

经过类似的断点和调试，我们在nt!RtlDecompressBufferXpressLz+0x2AA处，观察到qmemcpy中的count数据来自于RtlDecompressBufferXpressLz收到的参数CompressedBuffer的最后4个字节与3的和。因此操作压缩数据末尾的4个字节，可以控制复制数据的大小。

复制数据大小的来源已经清楚了，就剩下最后一个谜团--目的地址的来源。

### 目的地址的来源



图 13

我们根据设置的WinDBG断点日志，整理了上图所示的函数调用及数据传递过程。也顺便介绍了前文所述的如何通过公开的文档、符号文件或者数据流，注解IDA Pro函数名或者变量名，使得显示更加友好，以便开展分析工作。入手点是<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-rtldecompressbufferex2>查阅到的关于RtlDecompressBufferEx2的定义或NT之前泄露的源码中的相关函数定义。

从日志上来看，qmemcpy的目的地址正是UncompressedBuffer偏移1的地方。

Srv2DecompressData+0x85处的ExAllocatePoolWithTag() 返回值是0xffffa28f92503000，位于UncompressedBuffer之后0x370CBC8的位置。

即qmemcpy写入数据大小范围内有其他的Pool块时，将会导致ExFreePoolWithTag()时出错。

### 任意地址写入

如果size大小合适或者其范围内没有在用的Pool块，如0x1100+0n24大小时，则会有下述情况：

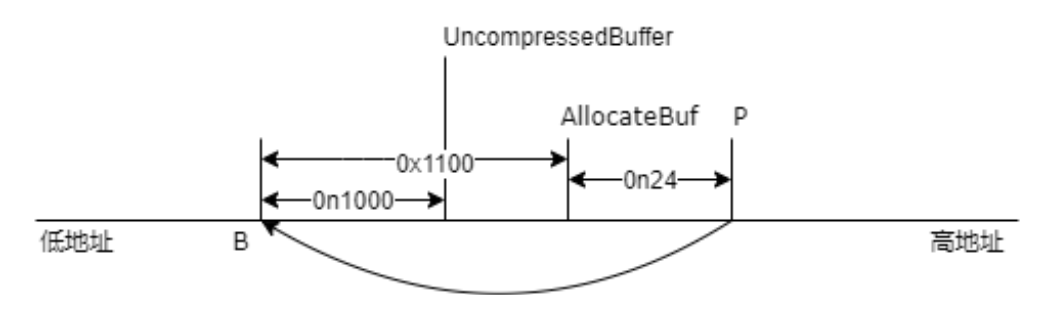


图 14

我们根据相关函数调用，绘制了上图所示的内存布局图。

当srv2!Srv2DecompressData+0x79处 SrvNetAllocateBuffer((unsigned int)(hdr.OriginalCompressedSegmentSize + offset)申请内存时，返回值设定AllocateBuf，简称A点。B点至U点正是SMB协议头中的offset值0x03e8(0n1000)。

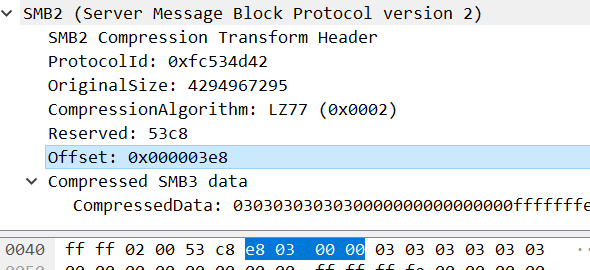


图 15

OriginalCompressedSegmentSize值（Wireshark中所示的OriginalSize）过大，与offset相加导致整数溢出。最终申请了一个较小的内存。即B点至A点的内存。内存的起始地址被写在AllocateBuf+0n24的P点。

当解压函数把超量数据写入U点时，如果超过了之前申请的内存（B点至A点的内存），也会覆盖原本存放在P处的指针。

srv2!Srv2DecompressData+0x108处的memmove会读取P点的指针作为目的地址，写入原始数据中offset之前的数据，从而完成预定的解压逻辑。当P处的指针可以被改写后，攻击者就获得了一次任意地址写入任意数据的能力。

srv2!memmove(Src=0xffffcb0558c5f060, Dst=0x4141414141414141, Size=1000)

kd> db 0x0xffffcb0558c5f060

ffffcb05`58c5f060 03 03 03 03 03 03 00 00-00 00 00 00 00 00 ff ff ................

ffffcb05`58c5f070 ff fe 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

ffffcb05`58c5f080 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

ffffcb05`58c5f090 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

ffffcb05`58c5f0a0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

ffffcb05`58c5f0b0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

ffffcb05`58c5f0c0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

ffffcb05`58c5f0d0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

kd> !pool 0x0xffffcb0558c5f060

Pool page ffffcb0558c5f060 region is Nonpaged pool

\*ffffcb0558c5f000 : large page allocation, tag is LS00, size is 0x1280 bytes

Pooltag LS00 : SRVNET LookasideList level 0 allocation 256 Bytes, Binary : srvnet.sys

至此漏洞分析视角下的工作基本完成，撰写分析报告时，我们会用倒叙的方法，就是大家经常看到的文章形式。后续文章我们再谈谈漏洞补丁分析和漏洞利用。

## 解决方案

尽快安装微软官方补丁或在网络出入口上阻止TCP端口445，以防止SMB流量进出互联网。此外，我们建议您进行内部网络分段，并禁止终端之间的SMB连接，以防止横向移动。

禁用SMBv3压缩将防止利用易受攻击的SMB服务器。要禁用SMBv3压缩，可以在PowerShell中运行以下命令：

1. Set-ItemProperty -Path
2. "HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters"
3. DisableCompression -Type DWORD -Value 1 -Force

## 综述

此漏洞对攻击者具有很高的价值，可使得攻击者很容易触及分配内存的函数，并且可以控制触发溢出的数据大小。蓝屏(BSOD)一般是远程代码执行的前兆，从其进化到远程代码执行(RCE)会更具挑战性，因为需要借助其他漏洞以便绕过Windows最新的缓解技术(KASLR)。应警惕漏洞利用难度稍小的本地权限提升情景，请尽快安装官方补丁。

## 参考 &引用

<https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/1d435f21-9a21-4f4c-828e-624a176cf2a0>

<https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/5606ad47-5ee0-437a-817e-70c366052962>

<http://yiiyee.cn/blog/2013/12/11/large-pool-1/>

<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-rtlgetcompressionworkspacesize>

<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-rtldecompressbufferex2>

## 命令&断点

sxe ld srv2

!pool

!poolval

dt nt!\_POOL\_HEADER

.writemem c:\test.log 0x20000000 L1000

bp nt!RtlDecompressBufferEx2 ".printf \"RtlDecompressBufferEx2(CompressionFormat=%d, UncompressedBuffer=0x%I64x, UncompressedBufferSize=%d, CompressedBuffer=0x%I64x, CompressedBufferSize=0x%I64x, UncompressedChunkSize=0x%I64x, FinalUncompressedSize=0x%I64x, WorkSpace=%d)\", cl, rdx, r8d, r9, poi(rsp+0x70), poi(rsp+0x78), poi(rsp+0x80), poi(rsp+0x88);.echo;g"

bp nt!RtlDecompressBufferXpressLz+0x2D0 ".printf \"RtlDecompressBufferXpressLz (), qmemcpy(dst=0x%I64x, src=0x%I64x, count=0x%I64x)\", rdi, rsi, r9;.echo"

bp srvnet!SmbCompressionDecompress ".printf \"srvnet!SmbCompressionDecompress(CompressionFormat=%d, CompressedBuffer=0x%I64x, CompressedBufferSize=%d, UncompressedBuffer=0x%I64x, UncompressedBufferSize=0x%I64x\", ecx, rdx, r8d, r9, poi(rsp+0x90);.echo;g"

bp srvnet!SmbCompressionDecompress+0x85 ".printf \"ExAllocatePoolWithTag(POOL\_TYPE=512, NumberOfBytes=%d(0x%I64x), Tag=2SL)\", edx, edx;.echo;g"

bp srvnet!SmbCompressionDecompress+0x91 ".printf \"ExAllocatePoolWithTag() return 0x%I64x\", rax;.echo;g"

bp srvnet!SmbCompressionDecompress+0xDF ".printf \"RtlDecompressBufferEx2() return0x%I64x, FinalUncompressedSize=%d(0x%I64x)\", ebx, poi(rsp+0x98), poi(rsp+0x98);.echo;g"

bp srvnet !SmbCompressionDecompress+0xF2 ".printf \"ExFreePoolWithTag(WorkSpace=%d(0x%I64x))\", rcx, rcx;.echo;g"

bp srv2!Srv2DecompressData ".printf \"srv2!Srv2DecompressData(buf=0x%I64x)\n\", rcx;db rcx L200;.echo;g"

bp srv2!Srv2DecompressData+0x79 ".printf \"srv2!SrvNetAllocateBuffer(Size=%d, Unknown=%d) \n\", rcx, rdx;db esp+0x30 LF;dd esp+0x30 L4;.echo;g"

bp srv2!Srv2DecompressData+0x85 ".printf \"srv2!SrvNetAllocateBuffer() =0x%I64x \n\", rax;db rax;.echo;g"

bp srv2!Srv2DecompressData+0xEC ".printf \"FinalUncompressedSize=0x%I64x, Size.m128i\_i32[1]=0x%I64x\", eax, r14d;.echo;g"

bp srv2!Srv2DecompressData+0x108 ".printf \"srv2!memmove(Src=0x%I64x, Dst=0x%I64x,Size=%d) \n\", rdx, rcx, r8d;db rdx;!pool rdx;.echo"

bp nt!RtlDecompressBufferXpressLz ".printf \" nt!RtlDecompressBufferXpressLz(UncompressedBuffer=0x%I64x, UncompressedBufferSize=%d, CompressedBuffer=0x%I64x, CompressedBufferSize=%d,0x%I64x,0x%I64x)\", rcx, edx, r8, r9d, rsp+0x38, poi(rsp+0x40);.echo;g"

bp srvnet!PplGenericAllocateFunction+0x35

bp nt!RtlDecompressBufferXpressLz+0x2AA

bp srvnet!SrvNetAllocateBuffer+0xD59F".printf \"srvnet!SrvNetAllocateBuffer(), SrvNetAllocateBufferFromPool()=0x%I64x\", rax;.echo"

## 问题&思考

1. 如何像查看小于PAGE\_SIZE的内存使用dt nt!\_POOL\_HEADER那样，查看large page allocation内存的size结构？
2. 对于较大的内存范围，如大于8字节，使用内存访问断点？
3. 对于较复杂的执行流程，从最终的数据，如何便捷的向上溯源？有点类似反向污点数据追踪。

2020年3月29日