

Windows SMB Ghost (CVE-2020-0796) 漏洞分析

启明星辰 ADLab 4月9日

更多安全资讯和分析文章请关注启明星辰ADLab微信公众号及官方网站
(adlab.venustech.com.cn)



漏洞介绍

2020年3月10日，微软在其官方SRC发布了CVE-2020-0796的安全公告（ADV200005，Microsoft Guidance for Disabling SMBv3 Compression），公告表示在Windows SMBv3版本的客户端和服务端存在远程代码执行漏洞。同时指出该漏洞存在于MicroSoft Server Message Block 3.1.1协议处理特定请求包的功能中，攻击者利用该漏洞可在目标SMB Server或者Client中执行任意代码。

启明星辰ADLab安全研究人员在对该漏洞进行研究的过程中发现目前流传的一些漏洞分析存在某些问题，因此对该漏洞进行了深入的分析，并在Windows 10系统上进行了复现。

漏洞复现

采用Windows 10 1903版本进行复现。在漏洞利用后，验证程序提权结束后创建了一个system权限的cmd shell，如图1所示。

```
命令提示符 - cve-2020-0796-local.exe
Microsoft Windows [版本 10.0.18362.476]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\Jack>cd Desktop

C:\Users\Jack\Desktop>cve-2020-0796-local.exe
Sending SMB negotiation request...
Finished SMB negotiation
Sending compressed buffer...
SEP_TOKEN_PRIVILEGES changed
Success! A new cmd shell should popup with system privilege!

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.18362.476]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>
```

图1 CVE-2020-0796本地提权

漏洞基本原理

CVE-2020-0796漏洞存在于受影响版本的Windows驱动srv2.sys中。Windows SMB v3.1.1 版本增加了对压缩数据的支持。图2所示为带压缩数据的SMB数据报文的构成。



图2 带压缩数据的SMB数据报文结构

根据微软MS-SMB2协议文档，SMB Compression Transform Header的结构如图3所示。

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
ProtocolId																															
OriginalCompressedSegmentSize																															
CompressionAlgorithm																Flags															
Offset/Length																															




图3 SMB Compression Transform Header数据结构

- ProtocolId: 4字节, 固定为0x424D53FC
- OriginalComressedSegmentSize: 4字节, 原始的未压缩数据大小
- CompressionAlgorithm: 2字节, 压缩算法
- Flags : 2字节, 详见协议文档
- Offset/Length: 根据Flags的取值为Offset或者Length, Offset表示数据包中压缩数据相对于当前结构的偏移

srv2.sys 中处理 SMBv3 压缩数据包的解压函数 Srv2DecompressData 未严格校验数据包中 OriginalCompressedSegmentSize 和 Offset/Length字段的合法性。而这两个字段影响了Srv2DecompressData中内存分配函数SrvNetAllocateBuffer的参数。如图4所示的Srv2DecompressData函数反编译代码, SrvNetAllocateBuffer实际的参数为OriginalCompressedSegmentSize+Offset。这两个参数都直接来源于数据包中SMB Compression Transform Header中的字段, 而函数并未判断这两个字段是否合法, 就直接将其相加后作为内存分配的参数(unsigned int类型)。

```

1 signed __int64 __fastcall Srv2DecompressData(__int64 smb_packet)
2 {
3     __int64 v1; // rdi
4     __int64 v2; // rax
5     __m128i payload_size; // xmm0
6     __m128i v4; // xmm0
7     unsigned int CompressAlgo; // ebp
8     __int64 alloc_buffer; // rax
9     __int64 _alloc_buffer; // rbx
10    __int32 OriginalCompressSegSize; // ST20_4
11    int v10; // eax
12    __m128i Size; // [rsp+30h] [rbp-28h]
13    int FinalUnCompressedSize; // [rsp+60h] [rbp+8h]
14
15    FinalUnCompressedSize = 0;
16    v1 = smb_packet;
17    v2 = *(_QWORD *)(&smb_packet + 0xF0);
18    if ( *(_DWORD *)(&v2 + 36) < 16u )
19        return 0xC000090B164;
20    payload_size = *(__m128i *)(&v2 + 24); // TCP payload size
21    Size = payload_size;
22    v4 = _mm_srli_si128(payload_size, 8);
23    CompressAlgo = *(_DWORD *)(&v4 + 0) * *(_QWORD *)(&smb_packet + 80) + 496164 + 140164;
24    if ( CompressAlgo != v4.m128i_u16[0] )
25        return 3221225659164;
26    alloc_buffer = SrvNetAllocateBuffer((unsigned int)(Size.m128i_i32[1] + v4.m128i_i32[1]), 0164); // v4指向了Algo, 偏移一个4字节就是OffsetOrLen;
27    // Size偏移4字节指向了OriginalCompressSeqSize;

```

这里，OriginalCompressedSegmentSize+Offset可能小于实际需要分配的内存大小，从而在后续调用解压函数SmbCompressionDecompress过程中存在越界读取或者写入的风险。

提权利用过程

目前已公开的针对该漏洞的本地提权利用包含如下的主要过程：

- (1) 验证程序首先创建到SMS server的会话连接（记为session）。
- (2) 验证程序获取自身token数据结构中privilege成员在内核中的地址（记tokenAddr）。
- (3) 验证程序通过session发送畸形压缩数据（记为evilData）给SMB server触发漏洞。其中，evilData包含tokenAddr、权限数据、溢出占位数据。
- (4) SMS server收到evilData后触发漏洞，并修改tokenAddr地址处的权限数据，从而提升验证程序的权限。
- (5) 验证程序获取权限后对winlogon进行控制，来创建system用户shell。

漏洞内存分配分析

首先，看一下已公开利用的evilData数据包的内容，如图5所示。

```

> Frame 89: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface \Device\NPF_{Loopback}, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 2659, Dst Port: 445, Seq: 201, Ack: 519, Len: 55
  NetBIOS Session Service
    Message Type: Session message (0x00)
    Length: 51
  SMB2 (Server Message Block Protocol version 2)
    SMB2 Compression Transform Header
    ProtocolId: 0xfc534d42
    OriginalSize: 4294967295 0xffffffff
    CompressionAlgorithm: LZ77 (0x0002)
    Reserved: 0000
    Offset: 0x00000010
  Compressed SMB3 data
    CompressedData: bcffff21f000000bcffff21f000000ffff3f404107000f...

```

0000	02 00 00 00	45 00 00 5f	cc 14 40 00 40 06 00 00E.._..@.@...
0010	7f 00 00 01	7f 00 00 01	0a 63 01 bd 70 33 ec 9ac..p3..
0020	24 24 be 47	50 18 27 f7	d1 87 00 00 00 00 00 33	\$\$GP.'.....3
0030	fc 53 4d 42	ff ff ff ff	02 00 00 00 10 00 00 00	.SMB.....
0040	bc ff ff f2	1f 00 00 00	bc ff ff f2 1f 00 00 00
0050	ff ff 3f 40	41 07 00 0f	ff 04 11 f0 c6 b5 4f 09	..?@A.....O.
0060	b2 ff ff			...

SMB header
 special token value
 real compressed data

ADLab

图5 提权poc发送的带压缩数据的SMB数据包

数据包的内容很简单，其中几个关键字段数据如下：

- OriginalSize: 0xffffffff
- Offset: 0x10
- Real compressed data: 13字节的压缩数据，解压后应为1108字节'A'加8字节的token地址。
- SMB3 raw data: 实际上是由2个8字节的0x1FF2FFFFBC（总长0x10）加上0x13字节的压缩数据组成。

从上面的漏洞原理分析可知，漏洞成因是Srv2DecompressData函数对报文字段缺乏合法性判断造成内存分配不当。在该漏洞数据包中，OriginalSize 是一个畸形值。OriginalSize + Offset = 0xffffffff + 0x10 = 0xf 是一个很小的值，其将会传递给SrvNetAllocateBuffer进行调用，下面具体分析内存分配情况。SrvNetAllocateBuffer的反编译代码如图6。

```

1 __int64 __fastcall SrvNetAllocateBuffer(unsigned __int64 a1, __int64 a2)
2 {
3     unsigned __int16 v2; // bp
4     int v3; // esi
5     __int64 v4; // r14
6     signed __int16 v5; // di
7     __int64 v6; // rcx
8     unsigned int v7; // eax
9     __int64 v8; // rdx
10    __int64 v9; // rax
11    __int64 v10; // rdi
12    __int64 v11; // rbx
13    unsigned __int64 v13; // rcx
14    __int64 v14; // rdx
15    __int64 v15; // rax
16    unsigned int v16; // eax
17    void *v17; // rcx
18    __int16 v18; // ax
19
20    v2 = __readgsdword(0x1A4u);
21    v3 = 0;
22    v4 = a2;
23    v5 = 0;
24    if ( SrvDisableNetBufferLookAsideList || a1 > 0x100100 ) // 大于1M采用函数SrvNetAllocateBufferFromPool分配
25        // SrvDisableNetBufferLookAsideList默认为False
26    {
27        if ( a1 > 0x1000100 ) // 大于16M不分配
28            return 0i64;
29        v11 = SrvNetAllocateBufferFromPool(a1, a1);
30    }
31    else // 小于1M的内存分配直接通过SrvNetBufferLookasides表
32    {
33        if ( a1 > 0x1100 )
34        {
35            v13 = a1 - 256;
36            _BitScanReverse64((unsigned __int64 *)&v14, v13);
37            _BitScanForward64((unsigned __int64 *)&v15, v13);
38            if ( (_DWORD)v14 == (_DWORD)v15 )
39                v3 = v14 - 12;
40            else
41                v3 = v14 - 11;
42        }
43        v6 = SrvNetBufferLookasides[v3];

```

图6 SrvNetAllocateBuffer内存分配过程

由于传给SrvNetAllocateBuffer的参数为0xf，根据SrvNetAllocateBuffer的处理流程可知，该请求内存将从SrvNetBufferLookasides表中分配。这里需要注意的是，变量SrvDisableNetBufferLookAsideList跟注册表项相关，系统默认状态下SrvDisableNetBufferLookAsideList为0。

```

1 if ( (signed int)SrvLibGetDWord(
2     L"\\Registry\\Machine\\System\\CurrentControlSet\\Services\\LanmanServer\\Parameters",
3     L"DisableNetBufferLookAside") >= 0 )
4     v2 = (_DWORD)v28 != 0;
5 else
6     v2 = 0;
7 SrvDisableNetBufferLookAsideList = v2;

```

图7 SrvDisableNetBufferLookAsideList变量初始化过程

SrvNetBufferLookasides表通过函数SrvNetCreateBuffer初始化，实际SrvNetCreateBuffer循环调用了SrvNetBufferLookasideAllocate分配内存，调用SrvNetBufferLookasideAllocate的参数分别为['0x1100', '0x2100', '0x4100', '0x8100', '0x10100', '0x20100', '0x40100', '0x80100', '0x100100']。在这里，内存分配参数为0xf，对应的lookaside表为0x1100大小的表项。

```
1 signed __int64 SrvNetCreateBufferLookasides()
2 {
3     _QWORD *v0; // rdi
4     __int64 v1; // r8
5     __int64 v2; // r9
6     unsigned int v3; // ebx
7     _QWORD *v4; // rax
8     ULONG v6; // [rsp+30h] [rbp-18h]
9
10    v0 = SrvNetBufferLookasides;
11    memset(SrvNetBufferLookasides, 0, 0x48ui64);
12    v3 = 0;
13    while ( 1 )
14    {
15        v4 = PplCreateLookasideList(
16            (__int64 (__fastcall *)())SrvNetBufferLookasideAllocate,
17            (__int64)SrvNetBufferLookasideFree,
18            v1,
19            v2,
20            (1 << (v3 + 12)) + 256, SrvNetBufferLookasideAllocate分配的大小
21            0x3030534C,
22            v6,
23            0x6662534Cu);
24        *v0 = v4;
25        if ( !v4 )
26            break;
27        ++v3;
28        ++v0;
29        if ( v3 ≥ 9 )
30            return 0i64;
31    }
32    SrvNetDeleteBufferLookasides();
33    return 0xC000009Ai64;
```

图8 SrvNetCreateBuffer反编译代码

SrvNetBufferLookasideAllocate函数实际是调用SrvNetAllocateBufferFromPool来分配内存，如图9所示。

```

1 unsigned __int64 __fastcall SrvNetBufferLookasideAllocate(__int64 a1, unsigned __int64 a2)
2 {
3     return SrvNetAllocateBufferFromPool(a1, a2);
4 }

```



图9 SrvNetBufferLookasideAllocate反编译代码

在函数SrvNetAllocateBufferFromPool中，对于用户请求的内存分配大小，内部通过ExAllocatePoolWithTag函数分配的内存实际要大于请求值（多出部分用于存储部分内存相关数据结构）。以请求分配0x1100大小为例，经过一系列判断后，最后分配的内存大小`allocate_size = 0x1100 + E8 + 2*(MmSizeOfMdl + 8)`。

```

v2 = a2; // a2=0x1100
if ( a2 > 0xFFFFFFFF )
    return 0i64;
if ( (unsigned int)a2 ≥ 0xFFFFFFFFFFFFFFFFB0ui64 )
    return 0i64;
if ( (unsigned __int64)(unsigned int)a2 + 88 < (unsigned __int64)(unsigned int)a2 + 80 )
    return 0i64;
v3 = (unsigned int)a2 + 0xE8i64;
if ( v3 < (unsigned __int64)(unsigned int)a2 + 88 )
    return 0i64;
v4 = MmSizeOfMdl(0i64, (unsigned int)a2 + 0xE8i64);
v5 = v4 + 8;
if ( v4 + 8 < v4 )
    return 0i64;
v6 = 2 * v5;
if ( !is_mul_ok(v5, 2ui64) )
    return 0i64;
allocate_size = v6 + v3;
if ( v6 + v3 < v3 )
    return 0i64;
if ( allocate_size < 0x1000 )
{
    allocate_size = 0x1000i64;
}
else if ( allocate_size > 0xFFFFFFFF )
{
    return 0i64;
}
ExAllocateBuf = (char *)ExAllocatePoolWithTag((POOL_TYPE)512, allocate_size, 0x30343434);
if ( !ExAllocateBuf )

```



图10 SrvNetAllocateBufferFromPool函数反编译代码

内存分配完毕之后，SrvNetAllocateBufferFromPool函数还对分配的内存进行了一系列初始化操作，最后返回了一个内存信息结构体指针作为函数的返回值。


```

ExAllocateBuf = (char *)ExAllocatePoolWithTag((POOL_TYPE)512, allocate_size, 0x3030534Cu);
if ( !ExAllocateBuf )
{
    _InterlockedIncrement((volatile signed __int32 *)&unk_1C002DE98);
    return 0i64;
}
v9 = allocate_size + _InterlockedExchangeAdd((volatile signed __int32 *)&unk_1C002DE94, allocate_size);
if ( (signed int)allocate_size > 0 )
{
    do
    {
        v10 = dword_1C002DE9C;
        while ( v9 > dword_1C002DE9C && v10 != _InterlockedCompareExchange(&dword_1C002DE9C, v9, dword_1C002DE9C) );
    }
    v11 = (signed __int64)(ExAllocateBuf + 0x50);
    return_buffer = (unsigned __int64)&ExAllocateBuf[request_size + 0x57] & 0xFFFFFFFFFFFFFFFF8ui64; // return_buffer =
    // 分配的ExAllocateBuf+0x1150
    *(_QWORD *)(return_buffer + 0x30) = ExAllocateBuf;
    *(_QWORD *)(return_buffer + 0x50) = (return_buffer + v5 + 151) & 0xFFFFFFFFFFFFFFFF8ui64;
    v13 = (return_buffer + 0x97) & 0xFFFFFFFFFFFFFFFF8ui64; // ExAllocateBuf+0x11E0
    *(_QWORD *)(return_buffer + 0x18) = ExAllocateBuf + 0x50; // return_buffer+0x18在后续解压过程中指向解压后数据的存储地址
    *(_QWORD *)(return_buffer + 0x38) = v13;
    *(_WORD *)(return_buffer + 0x10) = 0;
    *(_WORD *)(return_buffer + 0x16) = 0;
    *(_DWORD *)(return_buffer + 0x20) = request_size;
    *(_DWORD *)(return_buffer + 0x24) = 0;
    v14 = ((_WORD)ExAllocateBuf + 0x50) & 0xFFF;
    *(_DWORD *)(return_buffer + 0x28) = allocate_size;
    *(_DWORD *)(return_buffer + 0x40) = 0;
    *(_QWORD *)(return_buffer + 0x48) = 0i64;
    *(_QWORD *)(return_buffer + 0x58) = 0i64;
    *(_DWORD *)(return_buffer + 0x60) = 0;
    *(_QWORD *)v13 = 0i64;
    *(_WORD *)(v13 + 010) = 8 * (((unsigned __int16)v14 + (unsigned __int64)request_size + 4095) >> 12) + 6);
    *(_WORD *)(v13 + 0xA) = 0;
    *(_QWORD *)(v13 + 0x20) = v11 & 0xFFFFFFFFFFFFFFFF000ui64;
    *(_DWORD *)(v13 + 0x2C) = v14;
    *(_DWORD *)(v13 + 0x28) = request_size;
    MmBuildMdlForNonPagedPool(*(_PMDL *)(return_buffer + 56));
    MmMdlPageContentsState(*(_QWORD *)(return_buffer + 56), 1i64);
    *(_WORD *)((*_QWORD *)(return_buffer + 56) + 10i64) |= 0x1000u;
    v15 = *(_QWORD *)(return_buffer + 80);
    v16 = *(_QWORD *)(return_buffer + 24) & 0xFFFFFFFFFFFFFFFF000ui64;
    v17 = *(_QWORD *)(return_buffer + 24) & 0xFFFi64;

```

ADLab

图11 SrvNetAllocateBufferFromPool初始化内存数据

这里需要注意如下的数据关系：**SrvNetAllocateBufferFromPool**函数返回值**return_buffer**指向一个内存数据结构，该内存数据结构起始地址同实际分配内存（函数**ExAllocatePoolWithTag**分配的内存）起始地址的偏移为**0x1150**；**return_buffer+0x18**位置指向了实际分配内存起始地址偏移**0x50**位置处，而最终**return_buffer**会作为函数**SrvNetAllocateBuffer**的返回值。其内存布局关系如图12。

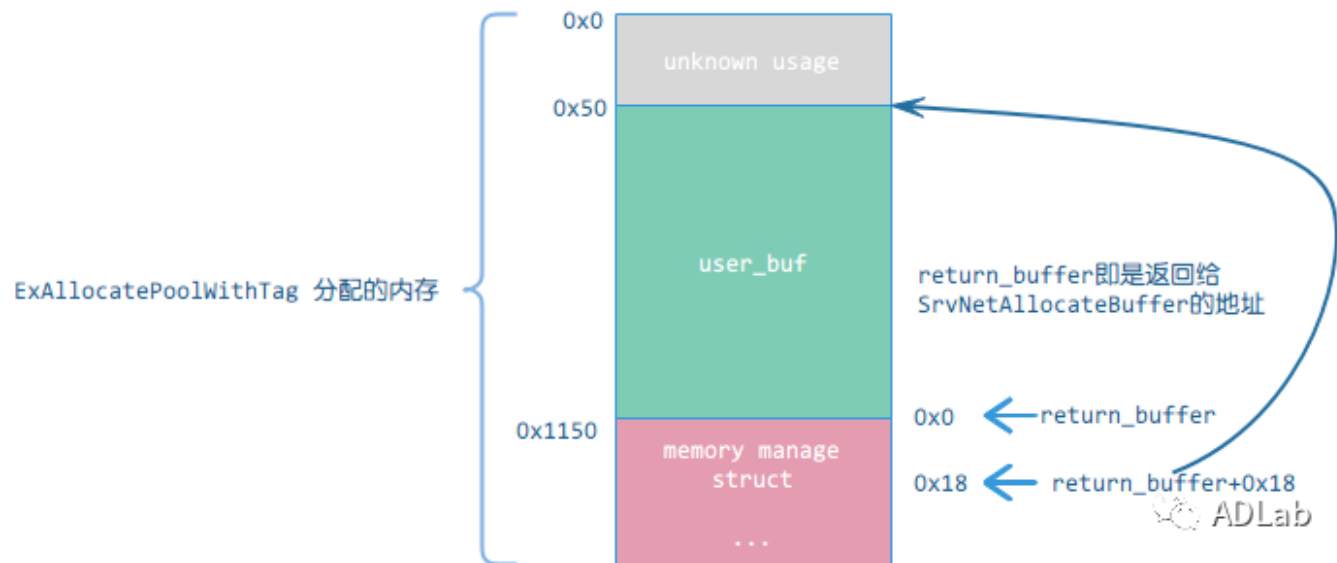


图12 SrvNetAllocateBuffer (0xf)返回的内存数据布局

漏洞内存破坏分析

回到漏洞解压函数 `Srv2DecompressData`，在进行内存分配之后，`Srv2DecompressData`调用函数 `SmbCompressionDecompress`开始解压被压缩的数据。其函数逻辑如图13所示。

```

return 3221225626i64;
alloc_buffer = SrvNetAllocateBuffer((unsigned int)(Size.m128i_i32[1] + v4.m128i_i32[1]), 0i64); // v4指向了Algo, 偏移一个4字节就是OffsetOrLen;
// Size偏移4字节指向了OriginalCompressSegSize;
// SrvNetAllocateBuffer(0xffffffff+0x10,0)
// 也就是SrvNetAllocateBuffer(0xf,0)

_alloc_buffer = alloc_buffer;
if ( !alloc_buffer )
    return 3221225626i64;
OriginalCompressSegSize = Size.m128i_i32[1];
if ( (signed int)SmbCompressionDecompress( // 这个函数实际调用RtlDecompressBufferEx2
    CompressAlgo,
    *(__QWORD *)(&_alloc_buffer + 0x18) + 24i64 + Size.m128i_u32[3] + 0x10i64, // CompressedBuf: payload+offset+0x10
    (unsigned int)(*(__DWORD *)(&_alloc_buffer + 0x18) + 36i64) - Size.m128i_i32[3] - 0x10, // CompressSize= PayloadSize-offset-0x10
    Size.m128i_u32[3] + *(__QWORD *)(&_alloc_buffer + 0x18), // UnCompressedBuf: alloc_buf+0x18 + offset
    OriginalCompressSegSize, // UnCompressedSize
    &FinalUnCompressedSize) < 0
    || (v10 = FinalUnCompressedSize, FinalUnCompressedSize != Size.m128i_i32[1])) // 实际上在Decompress函数中, 这个FinalUnCompressedSize
    // 赋值成了OriginalCompressedSegSize
{
    SrvNetFreeBuffer(_alloc_buffer);
    return 0xC000090Bi64;
}
if ( Size.m128i_i32[3] ) // if(offset)
{
    memmove( // 内存写入点
        *(void **)(alloc_buffer + 0x18),
        (const void *)(&_alloc_buffer + 0x18) + 24i64 + 16i64,
        Size.m128i_u32[3]); // memmove(*(alloc_buffer+0x18), SMBPayload, offset)
    v10 = FinalUnCompressedSize;
}
}

```

图13 Srv2DecompressData解压压缩数据

实际上，该函数调用了Windows库函数RtlDecompressBufferEx2来实现解压，根据RtlDecompressBufferEx2的函数原型来对应分析SmbCompressionDecompress函数的各个参数。

SmbCompressionDecompress(CompressAlgo, //压缩算法
Compressed_buf, //指向数据包中的压缩数据
Compressed_size, //数据包中压缩数据大小，计算得到
UnCompressedBuf, //解压后的数据存储地址，*(alloc_buffer+0x18)+0x10
UnCompressedSize, //压缩数据原始大小,源于数据包OriginalCompressedSegmentSize
FinalUnCompressedSize) //最终解压后数据大小

从反编译代码可以看出，函数SmbCompressionDecompress中保存解压后数据的地址为*(alloc_buffer+0x18)+0x10的位置，根据内存分配过程分析，alloc_buffer + 0x18指向了实际内存分配起始位置偏移0x50处，所以拷贝目的地址为实际内存分配起始地址偏移0x60位置处。

在解压过程中，压缩数据解压后将存储到这个地址指向的内存中。根据evilData数据的构造过程，解压后的数据为占坑数据和tokenAddr。拷贝到该处地址后，tokenAddr将覆盖原内存数据结构中alloc_buffer+0x18处的数据。也就是解压缩函数SmbCompressionDecompress返回后，alloc_buffer+0x18将指向验证程序的tokenAddr内核地址。拷贝过程如图14和15所示。

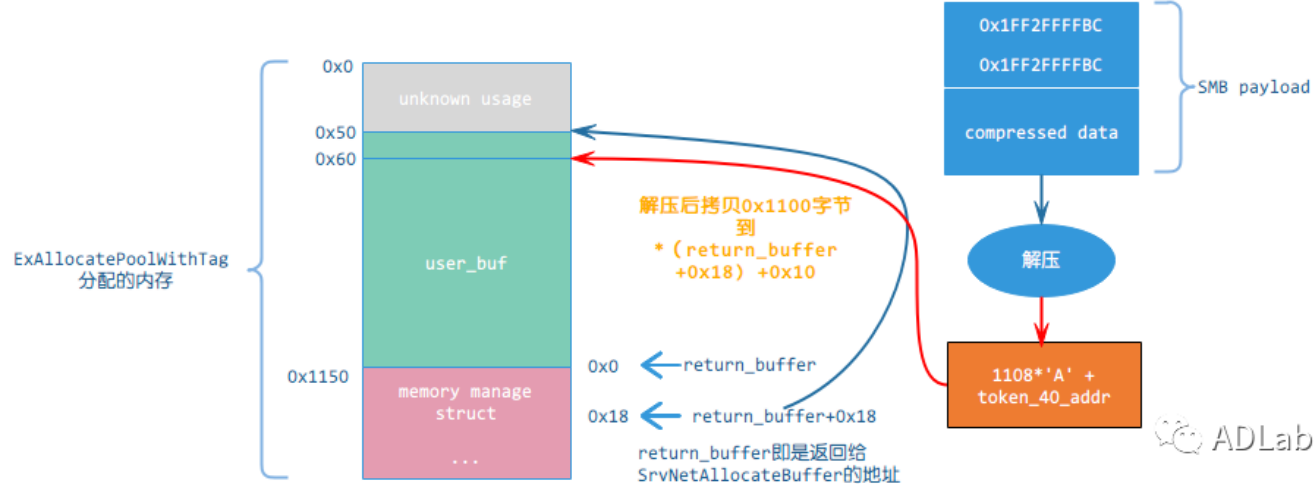


图14 解压拷贝过程

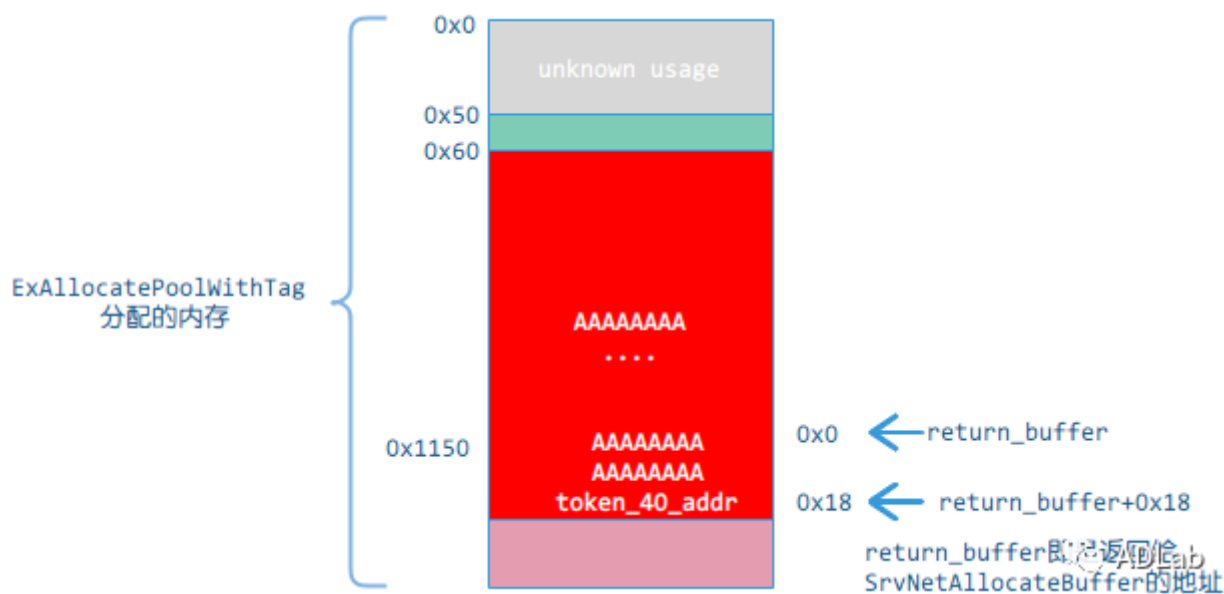


图15解压完成后内存布局

继续看Srv2DecompressData的后续处理流程，解压成功后，函数判断offset的结果不为0。不为0则进行内存移动，内存拷贝的参数如下：

```
memmove(*(alloc_buffer+0x18), SMB_payload, offset)
```

此时，alloc_buffer+0x18已经指向验证程序的tokenAddr内核地址，而SMB_payload此时指向evilData中的权限数据，offset则为0x10。因此，这个内存移动完成后，权限数据将写入tokenAddr处。这意味着，SMS Server成功修改了验证程序的权限，从而实现了验证程序的提权！

还有一个细节需要注意，在解压时，Srv2DecompressData函数会判断实际的解压后数据大小FinalUnCompressedSize是否和数据包中原始数据大小OriginalCompressedSegmentSize一致，如图16所示。

```
if ( (signed int)SmbCompressionDecompress( // 这个函数实际调用RtlDecompressBufferEx2
    CompressAlgo,
    *(_QWORD *)(*(_QWORD *) (v1 + 240) + 24i64) + Size.m128i_u32[3] + 0x10i64, // Com
    (unsigned int)(*_DWORD *)(*(_QWORD *) (v1 + 240) + 36i64) - Size.m128i_i32[3] -
    Size.m128i_u32[3] + *(_QWORD *) (alloc_buffer + 0x18), // UnCompressedBuf: alloc_
    OriginalCompressSegSize, // UnCompressedSize
    &FinalUnCompressedSize) < 0
|| (v10 = FinalUnCompressedSize, FinalUnCompressedSize != Size.m128i_i32[1]) )
```

图16 Srv2DecompressData检查压缩数据大小

按理来说实际解压后的数据大小为0x1100，不等于数据包中的原始压缩数据大小0xffffffff，这里应该进入到后面内存释放的流程。然而，实际上在函数SmbCompressionDecompress中，调用RtlDecompressBufferEx2成功后会直接将OriginalCompressedSegmentSize赋值给FinalUnCompressedSize。这也是该漏洞关于任意地址写入成功的关键之一。

```
FinalSize = FinalUnCompressedSize;
v15 = v8;
v16 = OriginalCompressSegSize;
v10 = RtlDecompressBufferEx2(
    v13,
    v7,
    OriginalCompressSegSize,
    v9,
    v15,
    4096,
    FinalUnCompressedSize,
    v6,
    *(_QWORD *)(&v18);
if ( v10 >= 0 )
    *FinalSize = v16;
if ( v6 )
    ExFreePoolWithTag(v6, 0x2532534Cu);
```

图17 SmbCompressionDecompress赋值FinalUnCompressedSize

漏洞修复建议

CVE-2020-0796是内存破坏漏洞，精心利用可导致远程代码执行，同时网络上已经出现该漏洞的本地提权利用代码。在此，建议受影响版本Windows用户及时根据微软官方漏洞防护公告对该漏洞进行防护。