SMB v3_RCE_POC 分析报告

简介

2020 年 3 月 12 日，微软官方纰漏了 SMB V3（CVE-202-0796）漏洞，并表示该漏洞存在远程代码执行的可能。在 4 月份就有人证明并演示了如何利用该漏洞进行远程代码执行。而在 2020 年 6 月 2 日，国外安全研究员便将远程代码执行的 POC 公开了，使得漏洞利用风险骤然升级。本文章便结合公开的 POC 来简单的介绍一下如何利用 CVE-202-0796 来进行远程代码执行的（由于本人能力有限，后续复杂的寻找指令指针寄存器的相关内容仍需进一步分析和研究）

CVE-2020-0796 相关信息

SMB v3 远程代码执行 POC：
https://github.com/chompie1337/SMBGhost_RCE_PoC
SMB v3 本地提权 POC：
https://github.com/danigargu/CVE-2020-0796
相关分析文档：
https://mp.weixin.qq.com/s/rKJdP_mZkaipQ9m0Qn9_2Q
https://blog.zecops.com/vulnerabilities/exploiting-smbghost-cve-2020-0796-for-a-local-privilege-escalation-writeup-and-poc/
https://ricercasecurity.blogspot.com/2020/04/ill-ask-your-body-smbghost-pre-auth-rce.html

漏洞原理分析

SMB v3 支持数据压缩，当收到经过压缩的数据包时，srv2.sys 中的 Srv2DecompressData 函数便会对数据进行解压缩，但在解压的时候，对数据的大小计算出现了整型溢出，从而导致开辟的空间小于要实际要拷贝的数据，最终导致拷贝时产生溢出。

数据包协议分析

SMB v3 Compression 数据包传送时需要使用 SMB2COMPRESSION_TRANSFORM_HEADER 头，微软文档对该头结构描述如图：

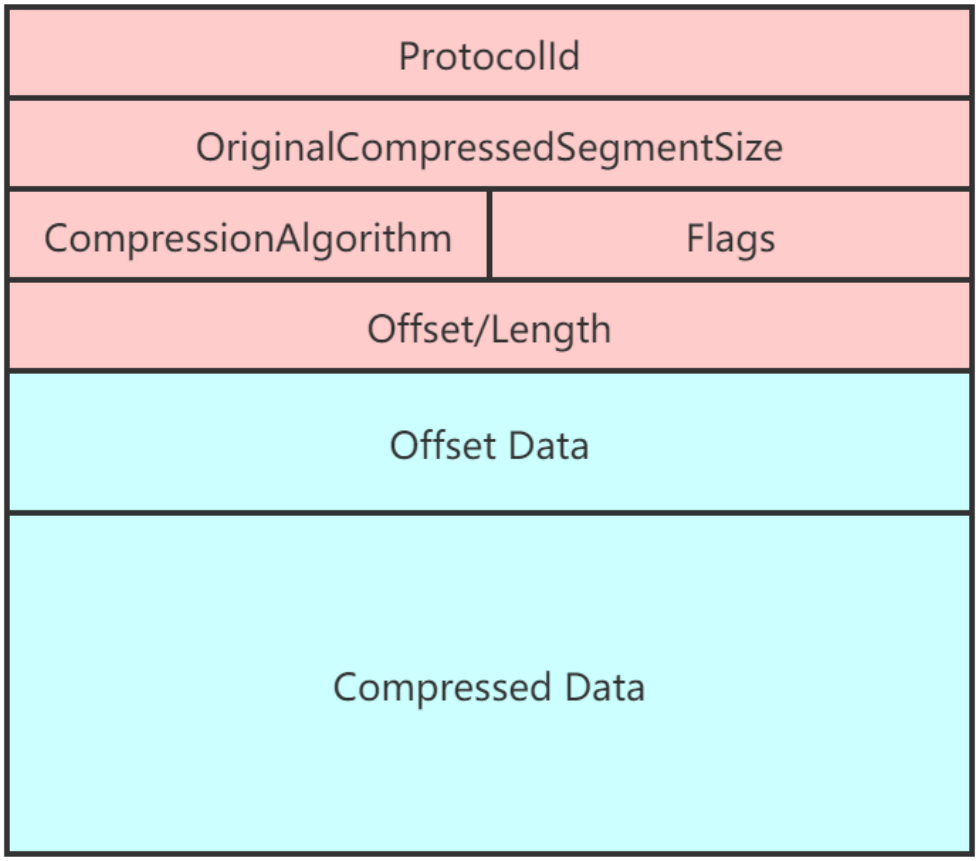
2.2.42 SMB2 COMPRESSION_TRANSFORM_HEADER

03/02/2020 • 2 minutes to read

The SMB2 COMPRESSION_TRANSFORM_HEADER is used by the client or server when sending compressed messages. This optional header is only valid for the SMB 3.1.1 dialect<69>.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
ProtocolId																															
OriginalCompressedSegmentSize																															
CompressionAlgorithm																				Flags											
Offset/Length																															

SMB2COMPRESSION_TRANSFORM_HEADER 头中需要我们留意两个值，一个是描述压缩数据大小的 OriginalCompressedSegmentSize，另一个是描述压缩数据相对包头的偏移 Offset。正是这两个值相加导致的整型溢出。SMB v3 Compression 数据包结构大致如图：



溢出点分析

srv2!Srv2DecompressData 函数接受到客户端发送的压缩数据包，分配所需内存空间，然后解压数据并放置分配的内存空间中。如果 Compression 数据包 Offset 字段不为空时，则将 Offset

Data 原样放置在解压缩数据之前，也就是缓冲区的开头。以下是 Srv2DecompressData 函数的简化代码：

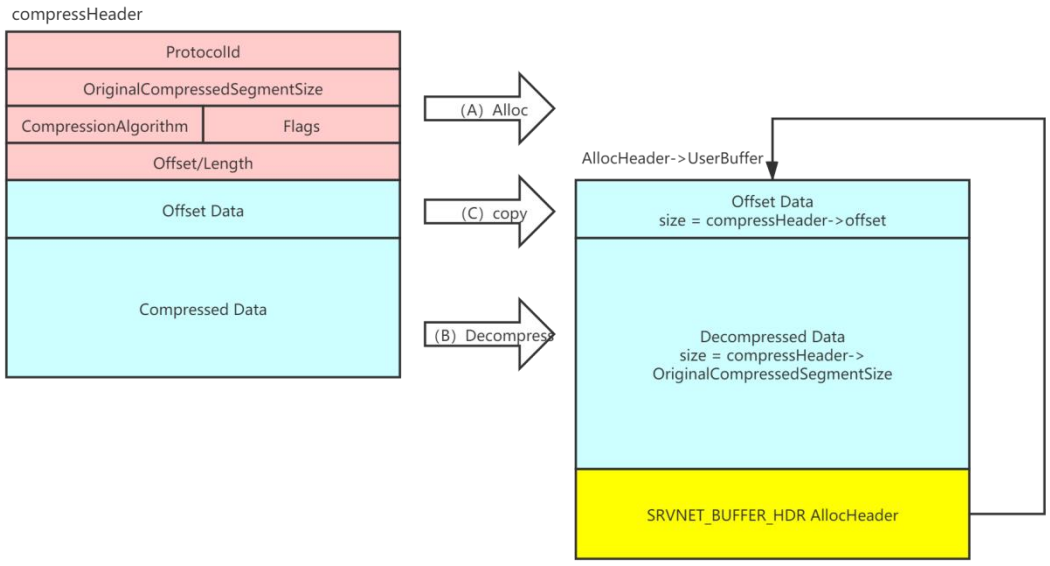
```
NTSTATUS Srv2DecompressData(PCOMPRESSION_TRANSFORM_HEADER PackageHeader, SIZE_T TotalSize)
{
    ...
    request = PackageHeader->psbhRequest;
    if ( request->dwMsgSize < 0x10 )
        return 0xC000090Bi64;
    compressHeader = *(CompressionTransformHeader *)request->pNetRawBuffer;
    ...
    // (A) 整型溢出
    //当OriginalCompressedSegmentSize+offset小于等于0x1100时，将会申请长度为0x1100的UncompressedBuffer缓冲区
    PSRVNET_BUFFER_HDR AllocHeader = SrvNetAllocateBuffer(                    //如果offset=0xffffffff时，开辟的空间变小
        (ULONG)(compressHeader->OriginalCompressedSegmentSize + compressHeader->Offset),
        NULL);
    If (!AllocHeader)
        return STATUS_INSUFFICIENT_RESOURCES;

    ULONG FinalCompressedSize = 0;
    // (B) 第一次缓冲区溢出，解压缩数据,溢出覆盖newHeader->pNetRawBuffer后的数据（为了弄清楚溢出覆盖的内容看函数SrvNetAllocateBufferFromPool）
    //该解函数在解压缩成功后将更新finalDecompressedSize = CompressedBufferSize，从而导致不走SrvNetFreeBuffer分支
    NTSTATUS Status = SmbCompressionDecompress(
        compressHeader->CompressionAlgorithm,
        (PUCHAR)compressHeader + sizeof(COMPRESSION_TRANSFORM_HEADER) + compressHeader->Offset, //CompressedBuffer
        (ULONG)(TotalSize - sizeof(COMPRESSION_TRANSFORM_HEADER) - compressHeader->Offset),    //CompressedBufferSize 溢出点
        (PUCHAR)AllocHeader->UserBuffer + compressHeader->Offset,       //UncompressedBuffer，会传入SmbCompressionDecompress函数进行Decompress处理，
        compressHeader->OriginalCompressedSegmentSize,
        &FinalCompressedSize);
    //如果走这个分支，可以通过覆盖NewHeader来实现free-after-free,UAF
    if (Status < 0 || FinalCompressedSize != compressHeader->OriginalCompressedSegmentSize)
    {
        SrvNetFreeBuffer(AllocHeader);
        return STATUS_BAD_DATA;
    }

    if (compressHeader->Offset > 0)
    {
        // (C) 第二次缓冲区溢出
        memcpy(
            AllocHeader->UserBuffer,
            (PUCHAR)compressHeader + sizeof(COMPRESSION_TRANSFORM_HEADER), //指向offset data
            compressHeader->Offset);
    }

    Srv2ReplaceReceiveBuffer(some_session_handle, AllocHeader);
    return STATUS_SUCCESS;
}
```

数据包被函数 Srv2DecompressData 处理的示意图：



A.开辟内存空间

在代码 A 处 SrvNetAllocateBuffer 函数申请内存空间，第一个参数为 originalCompressedSegSize 加上 offset 的值表示要开辟空间的大小。由于该函数是直接使用 eax 和 ecx 来将两数相加，

然后取结果的 4 字节作为参数开辟空间，没有考虑到进位的情况，产生整型溢出从而导致开辟的空间比实际的小。例如 originalCompressedSegSize 大小为 0xffffffff，Offset 大小为 0x10，两个数相加结果应该为 0x0x10000000f，而由于没有考虑进位，实际作为参数传进去的大小便为 0xf，导致开辟的内存空间远远小于实际大小。反汇编代码如图：

```
loc_1C0017EC8:                          ; CODE XREF: Srv2DecompressData+5C↑j
                mov     rax, qword ptr [rsp+58h+Size]
                xor     edx, edx
                shr     rax, 20h
                shr     rcx, 20h
                add     ecx, eax
                call    cs:__imp_SrvNetAllocateBuffer
                nop     dword ptr [rax+rax+00h]
                mov     rbx, rax
                test    rax, rax
                jnz     short loc_1C0017EF7
                mov     eax, 0C000009Ah
                jmp     loc_1C0017F99
```

深入分析 SrvNetAllocateBuffer

Srvnet!SrvNetAllocateBuffer 会根据所需分配空间大小进行判断。当所需大小大于 16MB 时将不进行分配，所需大小大于 1MB 小于 16MB 时，调用函数 SrvNetAllocateBufferFromPool 进行分配空间。其余小于 1MB 的数据使用后备列表（Lookaside List）来进行分配。以下是 SrvNetAllocateBuffer 函数的简化代码：

```
PSRVNET_BUFFER_HDR SrvNetAllocateBuffer(SIZE_T AllocSize, PALLOCATION_HEADER SourceBuffer)
{
    // ...

    if (SrvDisableNetBufferLookAsideList || AllocSize > 0x100100) {
        if (AllocSize > 0x1000100) {
            return NULL;
        }
        Result = SrvNetAllocateBufferFromPool(AllocSize, AllocSize);
    } else {
        int LookasideListIndex = 0;
        if (AllocSize > 0x1100) {
            LookasideListIndex = /* some calculation based on AllocSize */;
        }

        SOME_STRUCT list = SrvNetBufferLookasides[LookasideListIndex];
        Result = /* fetch result from list */;
    }

    // Initialize some Result fields...

    return Result;
}
```

其中值得关注的是后备列表，是 Windows 内核提供的一种机制，用于有效地为驱动程序保留一组可重用的固定大小的缓冲区。由于每次调用 ExAllocatePoolWithTag 和 ExFreePoolWithTag 都花费大量时间，因此内核驱动程序通常会为其自身的数据结构提供一个后备列表，当后备列表数据结构初始化完成时，后续从列表中检索元素时就无需再次初始化了，以提高效率。这表明我们可以破坏表头，将其添加到列表中，然后在以后的请求中从列表中检索复用，从而达到漏洞利用的目的。

函数 SrvNetAllocateBuffer 这里用到的后备列表 SrvNetBufferLookasides 也是如此，在函数 SrvNetCreateBufferLookasides 初始化之后，便可以从列表中检索元素复用，并且在初始化也同样是调用 SrvNetAllocateBufferFromPool 来开辟空间的只不过后备列表是提前开辟好的固

定大小。SrvNetAllocateBuffer 函数这里提供了 9 个后备列表，其中大小分别为：[0x1100,0x2100,0x4100,0x8100,0x10100,0x20100,0x40100,0x80100,0x100100]
SrvNetAllocateBuffer 最终都是通过调用 SrvNetAllocateBufferFromPool 来开辟空间的，但该函数返回的并不是指向开辟缓冲区的指针，而是 SRVNET_BUFFER_HDR 结构体，开辟的缓冲区就位于该结构体上方。这种布局为后续溢出提供了可能。布局可见上面 Srv2DecompressData 处理的示意图。SRVNET_BUFFER_HDR 结构信息如图：

```
struct __declspec(align(8)) SRVNET_BUFFER_HDR
{
  LIST_ENTRY List;
  USHORT Flag;
  BYTE unknown0[4];
  WORD unknown1;
  PBYTE pNetRawBuffer; //point to userbuffer
  DWORD dwNetRawBufferSize;
  DWORD dwMsgSize;
  DWORD dwNonPagedPoolSize;
  DWORD dwPadding;
  PVOID pNonPagedPoolAddr;
  PMDL pMDL1; // points to mdl1
  DWORD dwByteProcessed;
  BYTE unknown2[4];
  _QWORD unknown3;
  PMDL pMDL2; // points to mdl2
  PSRVNET_RECV pSrvNetWskStruct;
  DWORD unknown4;
  char unknown5[12];
  char unknown6[32];
  MDL mdl1; // variable size
  char unknow7[24];
  MDL mdl2; // variable size
};
```

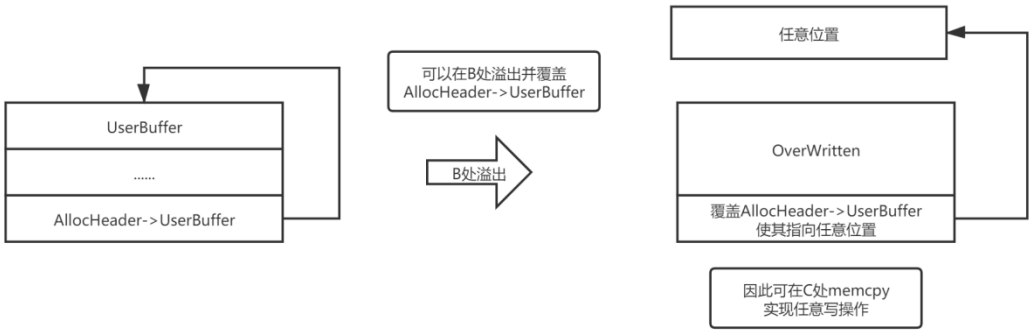
B. 解压缩数据

在代码 B 处 SmbCompressionDecompress 函数解压缩收到的压缩数据。解压后的 Decompress Data 数据被放到 AllocHeader->UserBuffer+Offset 处 (OriginalCompressedSegmentSize 的值不影响函数的运行结果)。由于在 A 处的整型溢出开辟了 0x1100（后备列表中的最小值）的缓冲区空间，因此只要构造 Offset Data + Decompress Data 大于 0x1100 的数据就能覆盖紧跟在缓冲区后面的 AllocHeader 结构。被覆盖的 AllocHeader 结构中有两个值需要关注，后面漏洞利用会用上。

- 偏移为 0x18 的 AllocHeader->UserBuffer,指向存放 Offset Data + Decompress Data 的缓存区。
- 偏移为 0x38 的 AllocHeader->PMDL1,指向 MDL1 结构。

C. 拷贝 Offset Data 数据

在代码 C 处 memcpy 函数将压缩数据包中的 Offset Data 拷贝到 AllocHeader->UserBuffer 中（解压缩数据之前）。不过在 B 处执行 SmbCompressionDecompress 时存在缓冲区溢出的可能，如果将 AllocHeader->UserBuffer 的值覆盖为指定地址，并且 Offset Data 数据和大小，以及 Decompress Data（未压缩的数据）的值也都是我们可以控制的。因此就可以实现任意内存写操作了。



漏洞利用分析

本地权限提升

本地提权（Exp）利用前面所描述的任意内存写操作修改 SEP_TOKEN_PRIVILEGES 结构中的关键数据来提升自身权限。首先提权程序先获取自身 token，通过 0x40 偏移取得 SEP_TOKEN_PRIVILEGES 结构的首地址 XXXX,结构如图：

```
0: kd> dt nt!_token
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr64 _ERESOURCE
   +0x038 ModifiedId       : _LUID
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x078 SessionId        : Uint4B
   +0x07c UserAndGroupCount : Uint4B
   +0x080 RestrictedSidCount : Uint4B
   +0x084 VariableLength   : Uint4B
```

SEP_TOKEN_PRIVILEGES 结构中包含与令牌相关的特权信息。其中 Present 为令牌当前可用权限；Enable 为已启用的权限；EnabledByDefault 为默认情况下已启用的权限。它们都采用 8 字节数据来存储特权的 flag，从低位起每一个 bit 位代表一个指定的特权。结构如图：

```
   +0x490 VariablePart     : Uint8B
0: kd> dt nt!_SEP_TOKEN_PRIVILEGES
   +0x000 Present          : Uint8B
   +0x008 Enabled          : Uint8B
   +0x010 EnabledByDefault : Uint8B
```

然后通过该漏洞精准覆盖 AllocHeader->UserBuffer 地址为 XXXX，将 Offset Data 的值设置为两个 0x1ff2ffffbc（system 进程中的 Present 为此值），分别用于覆盖当前进程的 Present 和 Enable 值，从而实现权限提升。

远程代码执行

远程代码执行相对于本地权限提升就要复杂的多。当想要实现远程代码执行需要两个条件，一个条件是远程主机内存中写 shellcode，另一个条件是控制指令指针寄存器指向这段 shellcode 中去执行。

远程写 shellcode

往远程主机内存中写 shellcode，根据之前分析的任意内存写操作已经可以实现了，只需要将内核态的 shellcode 和用户态的 shellcode 写入 KUSER_SHARED_DATA 中，这是一个在用户域和内核域中映射的结构（和页面）。它的地址是 0x7ffe0000 和 0xfffff78000000000，在用户域和内核域分别设置为 r--和 rw-。

控制指令指针寄存器

要想控制指令指针寄存器，首先需要实现任意内存读取操作。

任意内存读取

由于我们通过溢出覆盖 AllocHeader->UserBuffer 来实现任意内存写操作，是通过请求数据包来实现的，服务器将保持沉默或最多返回正常相应不会立即提供任何信息。幸运的是 srv2.sys 提供了一个可以利用的函数 srv2!Srv2SetResonseBufferToReceiveBuffer:

```
struct __declspec(align(16)) SRV2_WORKITEM
{
  ...
  PSRVNET_BUFFER_HDR psbhRequest; // offset +0xf0
  PSRVNET_BUFFER_HDR psbhResponse; // offset +0xf8
  ...
};

void __fastcall Srv2SetResponseBufferToReceiveBuffer(SRV2_WORKITEM *workitem)
{
  ...
  workitem->psbhResponse = workitem->psbhRequest;
  ...
}
```

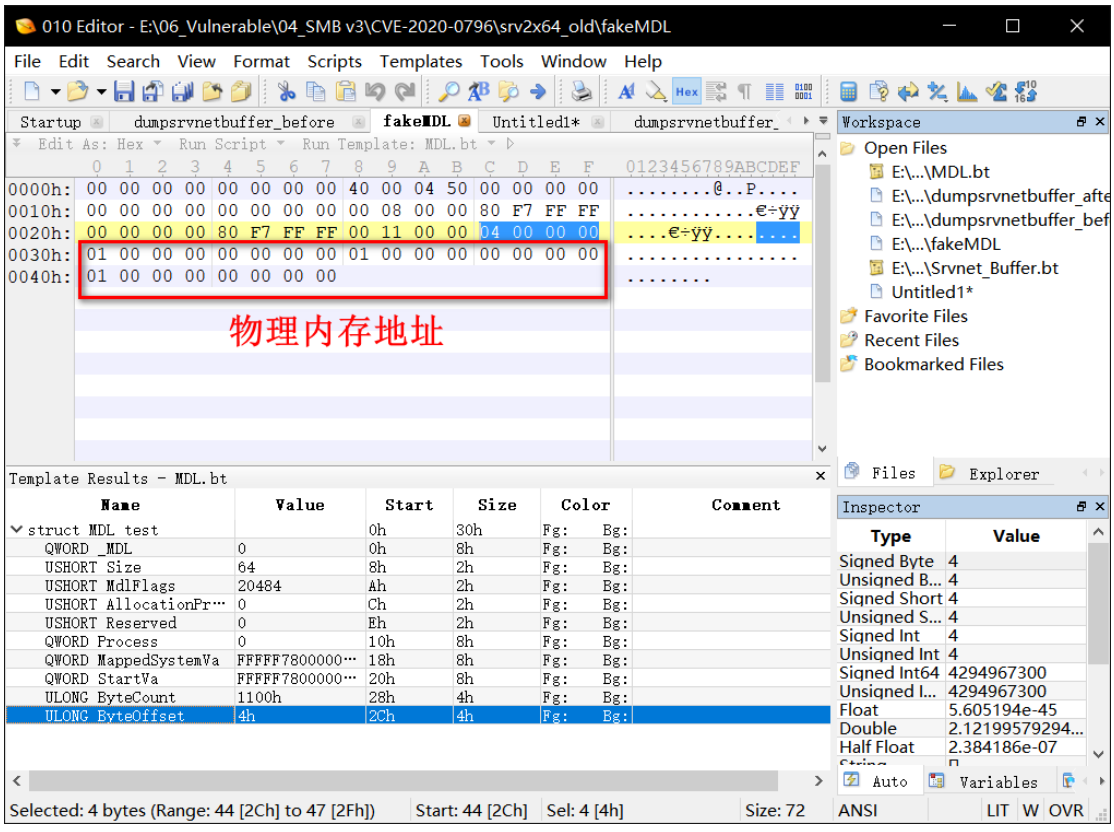
请求和响应在有效负载中共享许多公共部分，使用此功能能有效地重用缓冲区，因此提供了利用的可能。正如 srv2!Srv2SetResponseBufferToReceiveBuffer 函数在准备响应缓冲区时，不会再次初始化缓冲区（SRVNET_BUFFER_HDR）。因此我们只需要调用此函数，便能通过控制请求缓冲来实现控制响应缓冲区。

伪造 MDL

现在可以控制响应缓冲区（SRVNET_BUFFER_HDR）了，剩下便是如何控制要读取的内容了。由于 tcpip.sys 最终依赖 DMA(Direct Memory Access 直接内存访问)来传输数据包的，因此驱动程序会维护 MDL 中缓冲区的物理地址。MDL 结构如下：

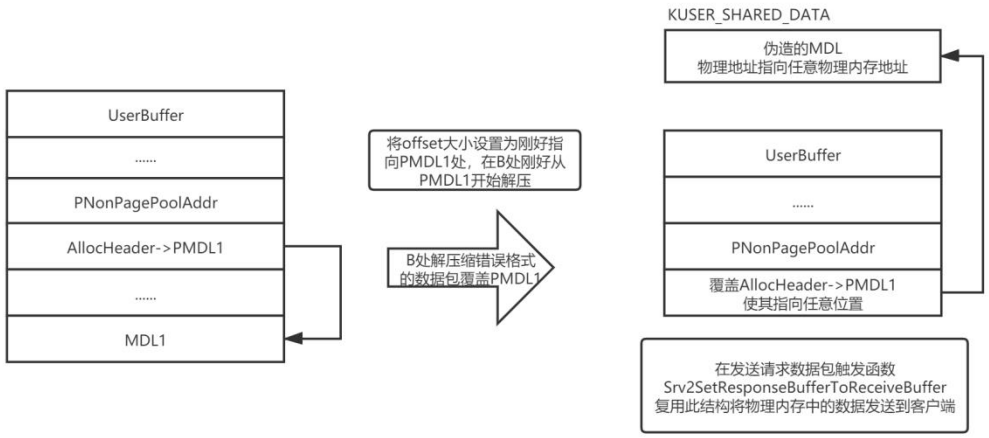
```
typedef struct{
  struct           _MDL;
  CSHORT           Size;
  CSHORT           MdlFlags;
  USHORT           AllocationProcessorNumber;
  USHORT           Reserved;
  QWORD            Process;
  QWORD            MappedSystemVa;
  QWORD            StartVa;
  ULONG            ByteCount;
  ULONG            ByteOffset;
  // Actually physical addresses follow.
  // Therefore, the size of this struct is variable
} MDL, PMDL;
```

POC 中伪造的 MDL 如图：



在 SRVNET_BUFFER_HDR 中，PMDL1 和 PMDL2 是执行 MDL 结构的指针，这些结构描述了包含 tcpip.sys 发送给客户端的数据内存。因此我们可以通过任意内存写，先将构造好的伪造的 MDL 写到 KUSER_SHARED_DATA 中，然后在通过溢出覆盖 PMDL1 指针，使其指向写入

KUSER_SHARED_DATA 中伪造的 MDL，再通过触发函数 Srv2SetResponseBufferToReceiveBuffer 复用此 SRVNET_BUFFER_HDR 缓冲区结构，就能通过构造不同的 MDL 来实现从服务端任意物理内存读取数据的操作，示意图如下：



覆盖 PMDL1 以及触发函数 Srv2SetResponseBufferToReceiveBuffer 细节

如果我们像之前那样在 B 处直接覆盖 AllocHeader->PMDL1 势必会覆盖到 PNonPagePoolAddr，给 PNonPagePoolAddr 一个无效的值，迟早会在 srvnet!SrvNetFreeBuffer 中产生崩溃。因此可以通过将 Offset Data 的大小设置为 AllocHeader->PMDL1 相对于 AllocHeader->UserBuffer 的偏移。这样在 B 处进行解压缩时便是从 AllocHeader->UserBuffer+Offset = AllocHeader->PMDL1 处开始的，这样就可以在 B 处不影响 PNonPagePoolAddr 的值来实现覆盖 AllocHeader->PMDL1。不过这样在 C 处拷贝 Offset Data 时依旧会覆盖 PNonPagePoolAddr 导致崩溃，因此选择在 B 处构造错误的 LZNT1（压缩格式）数据包。在 B 处解压错误的 LZNT1 数据包时，依旧能够覆盖 PMDL，并直到解压缩到损坏模块为止，然后在发送请求数据包来触发 Srv2SetResponseBufferToReceiveBuffer 来实现任意物理内存读取（这里调试没有完全跟到是如何读取的物理内存地址数据的）。

构造并执行 ShellCode

下面的内容由于时间有限能力有限，对如何从内存地址中找到指令指针寄存器不是很清楚。根据自己的理解简单的描述了一下。

有了任意物理内存读，然后通过在物理页面暴力搜索找到 HAL 的堆地址，然后从 HAL 的堆地址中找到 HalpInterruptController 和 HalpApicRequestInterrupt 两个值来构造内核态的 Shellcode，。如图（其中 HALP_APIC_REQ_INTERRUPT_OFFSET 的值是写死的 0x78）：

```
def build_shellcode():
    global KERNEL_SHELLCODE
    KERNEL_SHELLCODE += struct.pack("<Q", PHALP_INTERRUPT +
                                    HALP_APIC_REQ_INTERRUPT_OFFSET)
    KERNEL_SHELLCODE += struct.pack("<Q", PHALP_APIC_INTERRUPT)
    KERNEL_SHELLCODE += USER_PAYLOAD
```

然后再通过任意内存写操作将构造好的 ShellCode 写到 KUSER_SHARED_DATA 地址上。最终通过任意内存写操作将指令指针（PHALP_INTERRUPT + HALP_APIC_REQ_INTERRUPT_OFFSET）

指向 KUSER_SHARED_DATA 地址上的 ShellCode 去执行。写 Shellcode 和修改 EIP 操作如图：

```
# TODO: figure out why we can't write the entire shellcode data at once. There is a check before srv2!Srv2Decompres
to_write = len(KERNEL_SHELLCODE)
write_bytes = 0
while write_bytes < to_write:
    write_sz = min([write_unit, to_write - write_bytes])
    write_primitive(ip, port, KERNEL_SHELLCODE[write_bytes:write_bytes + write_sz], pshellcodeva + write_bytes)
    write_bytes += write_sz

print("[+] Wrote shellcode at %lx!" % pshellcodeva)

input("[+] Press a key to execute shellcode!")

write_primitive(ip, port, struct.pack("<Q", pshellcodeva), PHALP_INTERRUPT + HALP_APIC_REQ_INTERRUPT_OFFSET)
print("[+] overwrote HalpInterruptController pointer, should have execution shortly...")
```

远程代码执行读写操作逻辑，如下图：

```
[+] found low stub at phys addr 13000!
[+] PML4 at 1ad000
[+] base of HAL heap at fffff7f980000000
[+] found PML4 self-ref entry 112
[+] found HalpInterruptController at fffff7f980000570
[+] found HalpApicRequestInterrupt at fffff80411c0bbb0
[+] built shellcode!
[+] KUSER_SHARED_DATA PTE at ffff897bc0000000
[+] KUSER_SHARED_DATA PTE NX bit cleared!
[+] Wrote shellcode at fffff78000000950!
[+] Press a key to execute shellcode!
```