CVE-2020-0796 利用SMBGhost进行本地特权升级：Writeup + POC

原创

子曰小玖 于 2020-04-08 15:15:49 发布
分类专栏： 漏洞

利用SMBGhost（CVE-2020-0796）进行本地特权升级：Writeup + POC

介绍

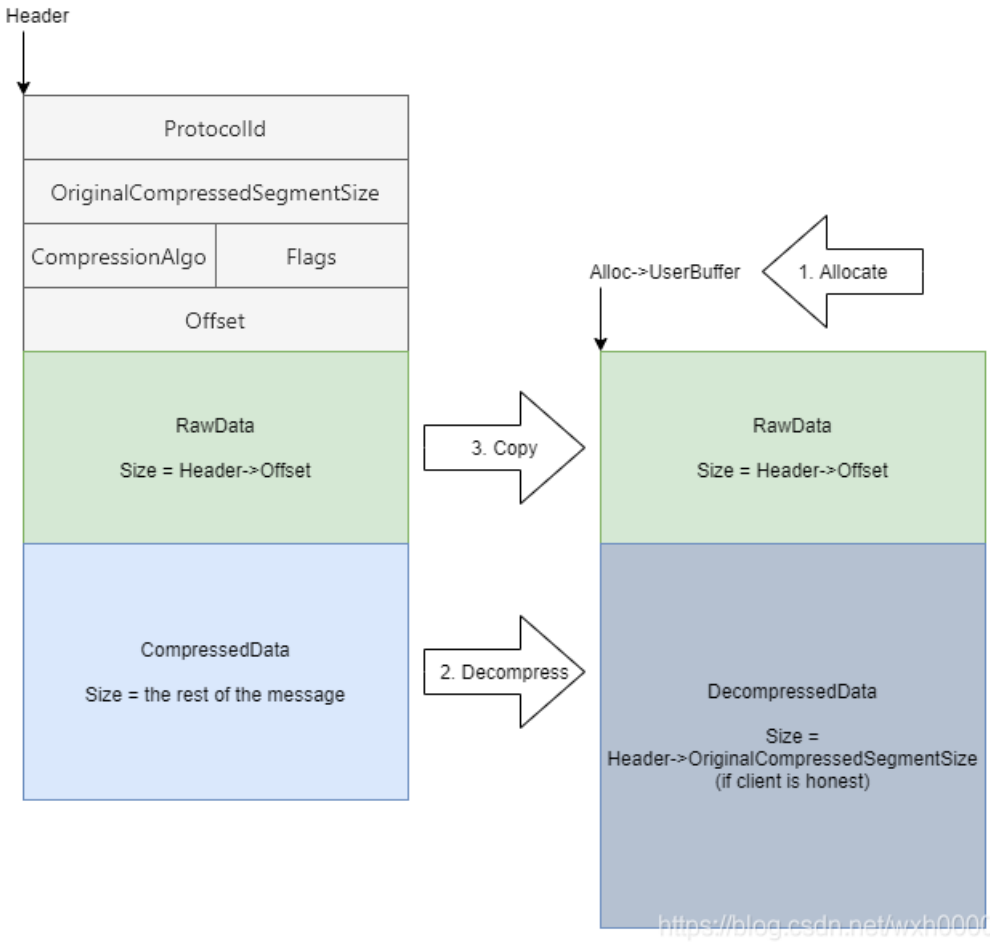
CVE-2020-0796是SMBv3.1.1（也称为“ SMBGhost”）的压缩机制中的错误。该错误影响Windows 10版本1903和1909，大约三周前由Microsoft 宣布并修补。得知此消息后，我们将浏览所有细节并创建一个快速的POC（概念验证），以演示如何通过引发BSOD（死亡蓝屏）而无需身份验证即可远程触发该错误。几天前，我们返回此错误的不仅仅是远程DoS。Microsoft安全公告将该错误描述为远程代码执行（RCE）漏洞，但是没有公共POC通过此错误来演示RCE。

初步分析

该错误是整数溢出错误，它发生在`Srv2DecompressData`srv2.sys SMB服务器驱动程序的函数中。这是该函数的简化版本，省略了不相关的细节：

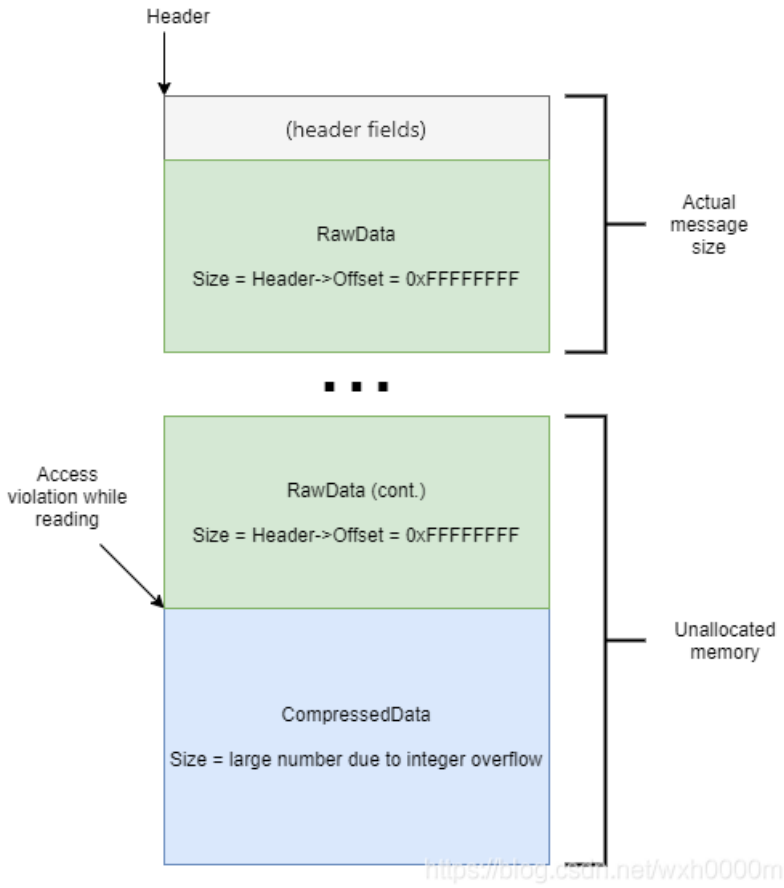
```
typedef struct _COMPRESSION_TRANSFORM_HEADER
{
    ULONG ProtocolId;
    ULONG OriginalCompressedSegmentSize;
    USHORT CompressionAlgorithm;
    USHORT Flags;
    ULONG Offset;
} COMPRESSION_TRANSFORM_HEADER, *PCOMPRESSION_TRANSFORM_HEADER;

typedef struct _ALLOCATION_HEADER
{
    // ...
    PVOID UserBuffer;
    // ...
} ALLOCATION_HEADER, *PALLOCATION_HEADER;

NTSTATUS Srv2DecompressData(PCOMPRESSION_TRANSFORM_HEADER Header, SIZE_T TotalSize)
{
    PALLOCATION_HEADER Alloc = SrvNetAllocateBuffer(
        (ULONG)(Header->OriginalCompressedSegmentSize + Header->Offset),
        NULL);
    If (!Alloc) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    ULONG FinalCompressedSize = 0;

    NTSTATUS Status = SmbCompressionDecompress(
        Header->CompressionAlgorithm,
        (PUCHAR)Header + sizeof(COMPRESSION_TRANSFORM_HEADER) + Header->Offset,
        (ULONG)(TotalSize - sizeof(COMPRESSION_TRANSFORM_HEADER) - Header->Offset),
        (PUCHAR)Alloc->UserBuffer + Header->Offset,
        Header->OriginalCompressedSegmentSize,
        &FinalCompressedSize);
    if (Status < 0 || FinalCompressedSize != Header->OriginalCompressedSegmentSize) {
        SrvNetFreeBuffer(Alloc);
        return STATUS_BAD_DATA;
    }

    if (Header->Offset > 0) {
        memcpy(
            Alloc->UserBuffer,
            (PUCHAR)Header + sizeof(COMPRESSION_TRANSFORM_HEADER),
            Header->Offset);
    }

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

该`Srv2DecompressData`函数接收客户端发送的压缩消息，分配所需的内存量，然后解压缩数据。然后，如果该`Offset`字段不为零，则将放置在压缩数据之前的数据照原样复制到分配的缓冲区的开头。



如果仔细看，我们会注意到第20和31行可能导致某些输入的整数溢出。例如，大多数在发布错误后不久并使系统崩溃的POC都使用`0xFFFFFFFF`该*`Offset`*字段的值。使用该值`0xFFFFFFFF`会在第20行触发整数溢出，结果分配的字节数减少了。

后来，它在第31行触发了另一个整数溢出。崩溃的发生是由于在第30行计算的地址处的内存访问，该地址与接收到的消息距离很远。如果代码在第31行验证了计算结果，则由于缓冲区长度恰好为负数且无法表示，因此它将提早解决，这也使第30行的地址本身也无效。

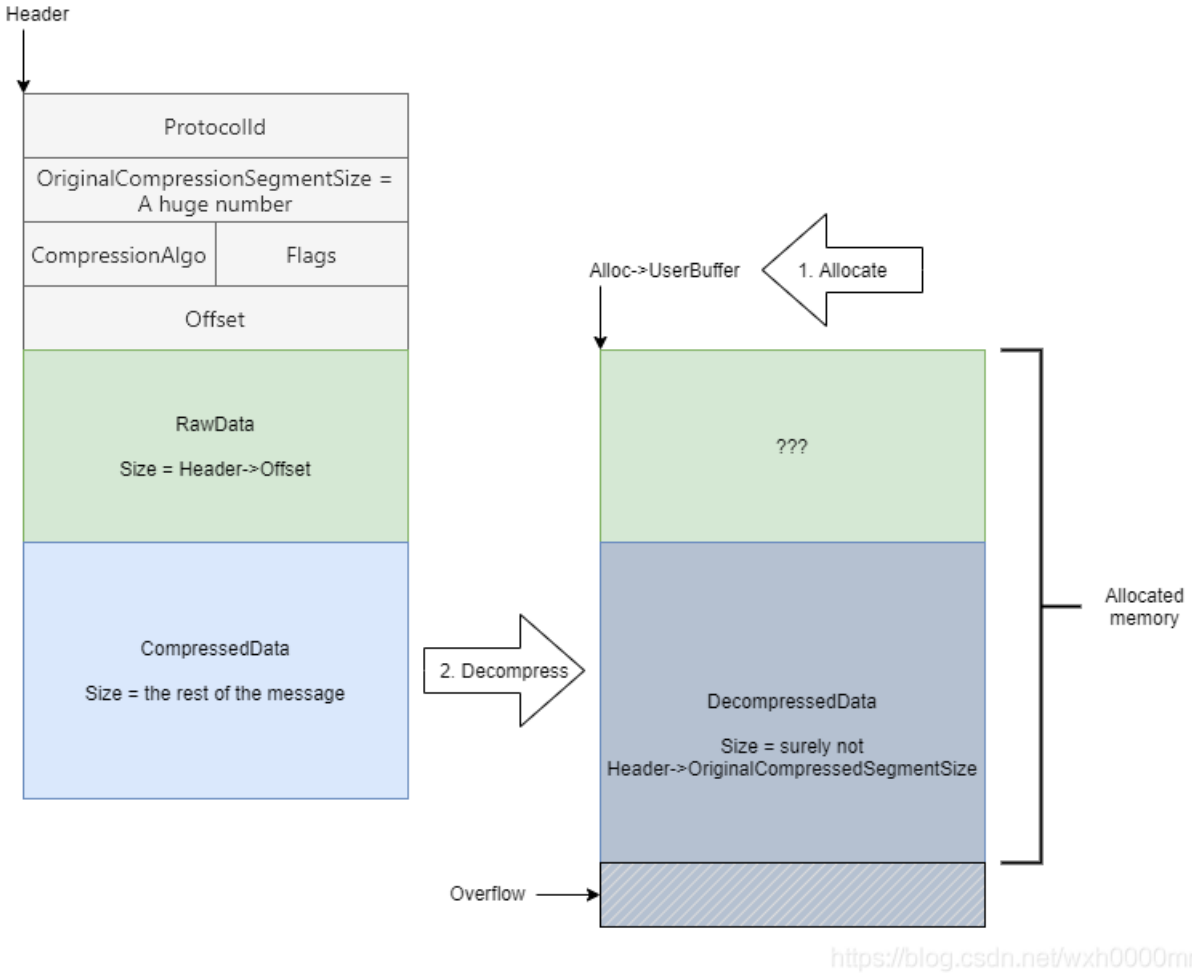


选择要溢出的内容

我们只有两个相关的字段可以控制以引起整数溢出：*`OriginalCompressedSegmentSize`*和*`Offset`*，因此没有太多选择。在尝试了几种组合之后，以下组合引起了我们的注意：如果我们发送合法`Offset`价值和巨大`OriginalCompressedSegmentSize`价值怎么办？让我们看一下代码将要执行的三个步骤：

1. **分配：**由于整数溢出，分配的字节数将小于两个字段的总和。
2. **解压缩：**解压缩将获得巨大的`OriginalCompressedSegmentSize`价值，将目标缓冲区视为实际上具有无限大小。所有其他参数均不受影响，因此将按预期工作。
3. **复制：**如果将要执行（会吗？），则复制将按预期工作。

无论是否要执行“复制”步骤，它都已经很有趣–我们可以在“解压缩”阶段触发越界写入，因为我们设法分配了比“分配”阶段所需的字节少的字节。



如您所见，使用这种技术，我们可以触发任何大小和内容的溢出，这是一个很好的开始。但是什么位于我们的缓冲区之外？让我们找出答案！

深入SrvNetAllocateBuffer

为了回答这个问题，在我们的案例中，我们需要查看分配函数`SrvNetAllocateBuffer`。这是函数的有趣部分：

```
PALLOCATION_HEADER 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;
}
```

我们可以看到分配函数根据所需的字节数执行不同的操作。大型分配（大于约16 MB）只会失败。中型分配（大于约1 MB）使用该`SrvNetAllocateBufferFromPool`功能进行分配。小分配（其余）使用后备列表进行优化。

注意：还有一个`SrvDisableNetBufferLookAsideList`标志会影响该功能的功能，但是它是由一个未记录的注册表设置来设置的，并且默认情况下处于禁用状态，因此不是很有趣。

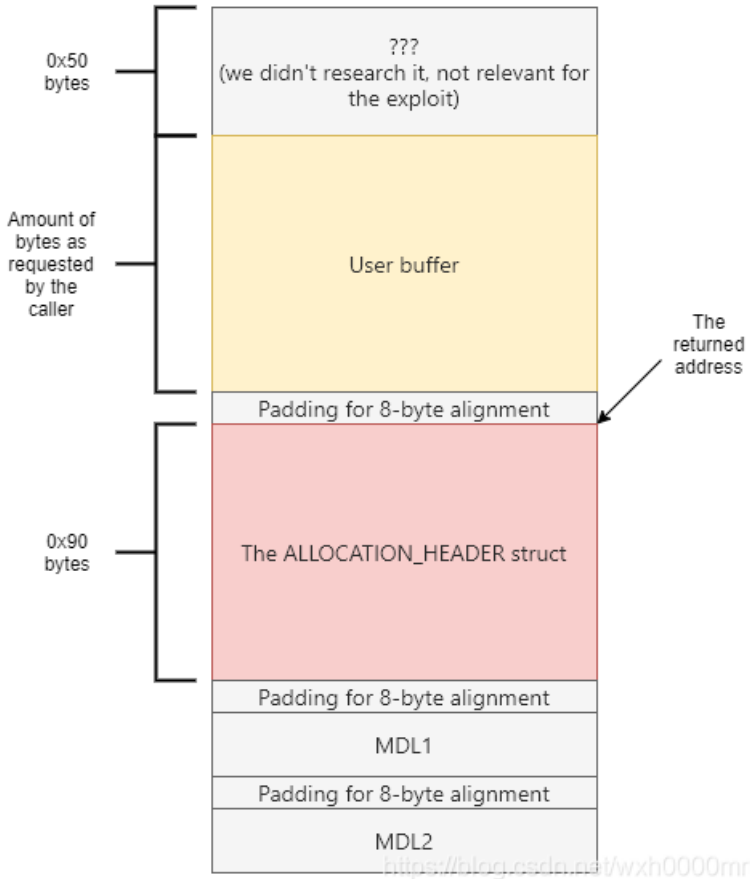
后备列表用于有效地为驱动程序保留一组可重用的固定大小的缓冲区。后备列表的功能之一是定义自定义的分配/空闲功能，这些功能将用于管理缓冲区。查看`SrvNetBufferLookasides`数组的引用，我们发现它已在`SrvNetCreateBufferLookasides`函数中初始化，并且通过查看它，我们学到了以下内容：

- 自定义分配函数定义为`SrvNetBufferLookasideAllocate`，它仅调用`SrvNetAllocateBufferFromPool`。
- 我们使用Python快速计算出了9个后备列表，它们的大小如下：
  >>> [hex（（1 <（（1 << (i + 12)）+ 256）for range（9）中的i]
  ['0x1100'，'0x2100”，“0x4100”，“0x8100”，“0x10100”，“0x20100”，“0x40100”，“0x80100”，“0x100100”]
  这与我们的发现相符，即`0x100100`不使用后备列表就分配了大于字节的分配。

结论是每个分配请求最终都出现在`SrvNetBufferLookasideAllocate`函数中，因此让我们来看一下。

SrvNetBufferLookasideAllocate和分配的缓冲区布局

该`SrvNetBufferLookasideAllocate`函数`NonPagedPoolNx`使用该`ExAllocatePoolWithTag`函数在池中分配一个缓冲区，然后用数据填充某些结构。分配的缓冲区的布局如下：



对于我们的研究范围，此布局的唯一相关部分是用户缓冲区和ALLOCATION_HEADER结构。我们可以立即看到，通过溢出用户缓冲区，我们最终将覆盖该ALLOCATION_HEADER结构。看起来很方便。

覆盖ALLOCATION_HEADER结构

这时我们的第一个想法是，由于SmbCompressionDecompress调用之后的检查：

```
<span style="color:#001c26">如果（状态<0 || FinalCompressedSize！=标头-> OriginalCompressedSegmentSize）{
    SrvNetFreeBuffer（Alloc）；
    返回STATUS_BAD_DATA;
}</span>
```

SrvNetFreeBuffer将被调用，并且该函数将失败，因为我们将其设计OriginalCompressedSegmentSize成一个很大的数字，并且FinalCompressedSize将成为一个较小的数字，代表解压缩的字节的实际数量。因此，我们分析了该SrvNetFreeBuffer函数，设法将分配指针替换为一个幻数，然后等待free函数尝试对其进行释放，以期以后将其用于free-after-free或类似用途。但是令我们惊讶的是，该memcpy函数崩溃了。这使我们感到高兴，因为我们根本不希望到达那里，但是我们必须检查它为什么发生。可以在SmbCompressionDecompress函数的实现中找到说明：

```
NTSTATUS SmbCompressionDecompress(
    USHORT CompressionAlgorithm,
    PUCHAR UncompressedBuffer,
    ULONG UncompressedBufferSize,
    PUCHAR CompressedBuffer,
    ULONG CompressedBufferSize,
    PULONG FinalCompressedSize)
{
    // ...

    NTSTATUS Status = RtlDecompressBufferEx2(
        ...,
        FinalUncompressedSize,
        ...);
    if (Status >= 0) {
        *FinalCompressedSize = CompressedBufferSize;
    }

    // ...

    return Status;
}
```

基本上，如果解压缩成功，则将其`FinalCompressedSize`更新以保留的值`CompressedBufferSize`，即缓冲区的大小。对`FinalCompressedSize`返回值的这种故意更新对于我们来说似乎非常可疑，因为这个小细节以及分配的缓冲区布局允许非常方便地利用此错误。

由于执行继续到复制原始数据的阶段，因此让我们再次查看该调用：

```
<span style="color:#001c26">memcpy（
    Alloc-> UserBuffer，
     （PUCHAR）标题+ sizeof（COMPRESSION_TRANSFORM_HEADER），
    Header-> Offset）;</span>
```

从`ALLOCATION_HEADER`结构中读取目标地址，我们可以覆盖该结构。缓冲区的内容和大小也由我们控制。大奖！远程写入内核中的任何内容！

远程“随处写”实现

我们做了一个快速实施的写什么，在哪里CVE-2020-0796漏洞利用 Python中，这是基于对CVE-2020-0796的DoS POC maxpl0it。该代码相当简短。

本地特权升级

现在我们有了写在哪里的漏洞，我们该怎么办？显然，我们可能会使系统崩溃。我们也许可以触发远程代码执行，但是还没有找到一种方法。如果我们在本地主机上使用该漏洞利用并泄漏其他信息，则可以将其用于本地特权升级，因为已经通过多种技术证明了它的可行性。

我们尝试的第一种技术是Morten Schenk在其《Black Hat USA 2017》演讲中提出的。该技术涉及覆盖驱动程序`.data`部分中的函数指针`win32kbase.sys`，然后从用户模式调用适当的函数以执行代码。j00ru撰写了一篇有关在WCTF 2018中使用此技术的出色文章，并提供了他的利用源代码。我们针对“在哪里写”漏洞进行了调整，但发现它不起作用，因为处理SMB消息的线程不是GUI线程。因此，`win32kbase.sys`它没有被映射，并且该技术也不相关（除非有一种使其成为GUI线程的方法，而我们没有研究过）。

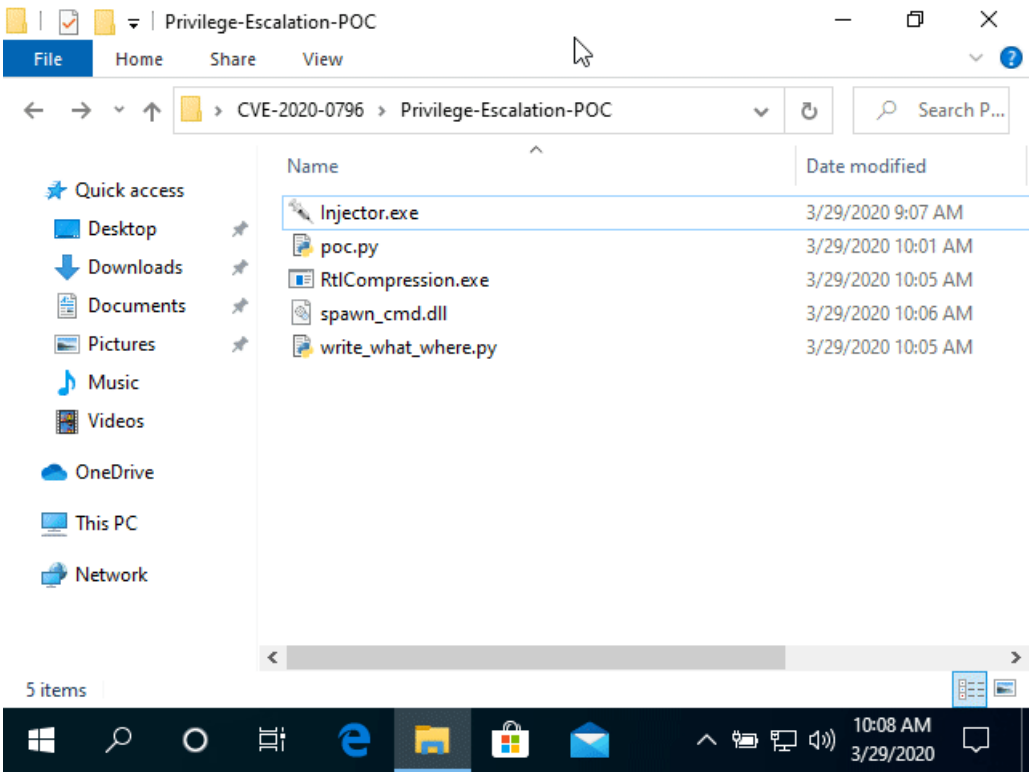
我们最终在2012年的Black Hat演讲“ Easy Local Windows Kernel Exploitation”中使用了cerarcer涵盖的众所周知的技术。该技术是关于通过使用API 泄漏当前进程令牌地址，然后对其进行覆盖，授予当前进程令牌特权，该特权随后可用于特权升级。该滥权令牌特权期末由布莱恩·亚历山大（研究dronesec）和斯蒂芬·布林（breenmachine）（2017）展示了使用权限提升各种令牌特权的几种方法。`NtQuerySystemInformation(SystemHandleInformation)`

我们的漏洞利用基于Alexandre Beaulieu在“ 利用任意写入权限升级特权”文章中友善地共享的代码。通过将DLL注入来修改进程的令牌特权后，我们完成了特权升级`winlogon.exe`。DLL的全部目的是启动的特权实例`cmd.exe`。我们的完整本地特权升级概念证明可在此处找到，仅可用于研究/防御目的。

摘要

我们设法证明可以利用CVE-2020-0796漏洞进行本地特权升级。请注意，我们的利用仅限于中等完整性级别，因为它依赖于较低完整性级别不可用的API调用。我们可以做得更多吗？也许可以，但是这需要更多的研究。我们可以在分配的缓冲区中覆盖许多其他字段，也许其中之一可以帮助我们实现其他有趣的事情，例如远程代码执行。

POC源代码



修复

1. 我们建议将服务器和端点更新到最新的Windows版本，以修复此漏洞。如果可能，请阻塞端口445，直到部署更新为止。无论CVE-2020-0796，我们建议在可能的情况下启用主机隔离。
2. 可以禁用SMBv3.1.1压缩以避免触发此错误，但是，如果可能的话，我们建议执行完全更新。