The case of the very large memory blocks of the same size, mostly zero, but whose nonzero bytes follow a pattern

July 6, 2018

Raymond Chen

A program was exhibiting very high memory usage. It had reached 600MB by the time the memory dump was created. The program did not appear to be actively doing anything to explain the high memory usage, and the suspicion was that there was some sort of leak.

From the dump file, let's look at the memory usage.

```
0:000> !heap -s
LFH Key                    : 0x7bfd169cb0392edb
Termination on corruption : ENABLED
Affinity manager status:
   - Virtual affinity limit 16
   - Current entries in use 0
   - Statistics:  Swaps=0, Resets=0, Allocs=0

          Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                            (k)     (k)    (k)     (k) length      blocks cont. heap
-------------------------------------------------------------------------------------
0000000000490000 00000002  405268 343108 404876  16857  2076    35    0    23a   LFH
0000000000120000 00008000      64      4     64      2     1     1    0      0
0000000000860000 00001002   64360  43276  63968  14325   626    11    0      c   LFH
...
```

The first heap is up to 343MB, so that's a good place to start digging.

```
0:000> !heap -m
Index   Address  Name      Debugging options enabled
  1:   00490000
    Segment at 0000000000490000 to 000000000058f000 (000ff000 bytes committed)
    Segment at 0000000002a60000 to 0000000002b5f000 (000ff000 bytes committed)
    Segment at 0000000004800000 to 00000000049ff000 (001ff000 bytes committed)
    Segment at 0000000005580000 to 000000000597f000 (003ff000 bytes committed)
    Segment at 0000000010330000 to 0000000010b2f000 (007ff000 bytes committed)
    Segment at 0000000007000000 to 0000000007fcf000 (00fcf000 bytes committed)
    Segment at 0000000017b20000 to 0000000018aef000 (00fcf000 bytes committed)
    Segment at 000000001d8a0000 to 000000001e86f000 (00fcf000 bytes committed)
    Segment at 000000001fc70000 to 0000000020c3f000 (00fcf000 bytes committed)
    Segment at 0000000025f20000 to 0000000026eef000 (00fcf000 bytes committed)
    Segment at 0000000022f50000 to 0000000023f1f000 (00fcf000 bytes committed)
    Segment at 0000000026ef0000 to 0000000027ebf000 (00fcf000 bytes committed)
    Segment at 0000000029010000 to 0000000029fdf000 (00fcf000 bytes committed)
    Segment at 000000002e1c0000 to 000000002f18f000 (00fcf000 bytes committed)
    Segment at 0000000033620000 to 00000000345ef000 (00fcf000 bytes committed)
    Segment at 000000002c930000 to 000000002d8ff000 (00fcf000 bytes committed)
    Segment at 0000000031320000 to 00000000322ef000 (00fcf000 bytes committed)
    Segment at 00000000345f0000 to 00000000355bf000 (00fcf000 bytes committed)
    Segment at 000000003a6a0000 to 000000003b66f000 (00fcf000 bytes committed)
    Segment at 00000000365c0000 to 000000003758f000 (00fcf000 bytes committed)
    Segment at 0000000041fa0000 to 0000000042f6f000 (00c25000 bytes committed)
    Segment at 000000004bda0000 to 000000004cd6f000 (002fe000 bytes committed)
    Segment at 000000004d1a0000 to 000000004e16f000 (00086000 bytes committed)
    Segment at 000000004e520000 to 000000004f4ef000 (00086000 bytes committed)
    Segment at 000000004ff20000 to 0000000050eef000 (00b29000 bytes committed)
    Segment at 00000000546a0000 to 000000005566f000 (00f81000 bytes committed)
    Segment at 0000000066420000 to 00000000673ef000 (00f81000 bytes committed)
    Segment at 00000000673f0000 to 00000000683bf000 (00f81000 bytes committed)
    Segment at 00000000683c0000 to 000000006938f000 (007d5000 bytes committed)
    Flags:                00000002
    ForceFlags:           00000000
    Granularity:          16 bytes
    Segment Reserve:      18b80000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000400
    DeCommit Total Thres: 00001000
    Total Free Size:      0010766b
    Max. Allocation Size: 00007ffffffdefff
    Lock Variable at:     00000000004902a0
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   00490110
    Uncommitted ranges:   004900f0
            68371000: 0004e000  (319488 bytes)
            673a1000: 0004e000  (319488 bytes)
            55621000: 0004e000  (319488 bytes)
            50ea1000: 0004e000  (319488 bytes)
```

```
            4f4a1000: 0004e000  (319488 bytes)
            4e121000: 0004e000  (319488 bytes)
            42f00000: 0006f000  (454656 bytes)
            426f9000: 0011e000  (1171456 bytes)
            42cbf000: 0021d000  (2215936 bytes)
            68723000: 003ee000  (4120576 bytes)
            68f83000: 0040c000  (4243456 bytes)
            4ff21000: 00458000  (4554752 bytes)
            4c09e000: 00cd1000  (13438976 bytes)
            4e521000: 00efb000  (15708160 bytes)
            4d1a1000: 00efb000  (15708160 bytes)
    FreeList[ 00 ] at 0000000000490150: 00000000428c2020 . 0000000042371060   (2076
blocks)
...
```

Earlier segments will probably hold memory that was allocated a long time ago, possibly even at process startup, which are probably not the leaked memory. The most recent segment will contain transient allocations. So I grab a segment that's third-from-the-end. That will probably be rich in leaked data.

```
0:000> db 00000000`546a0000
00000000`546a0000  00 00 00 00 00 00 00 00-c1 f3 15 82 00 82 01 01  ................
00000000`546a0010  ee ff ee ff 02 00 00 00-18 00 42 66 00 00 00 00  ..........Bf....
00000000`546a0020  18 00 f2 4f 00 00 00 00-00 00 49 00 00 00 00 00  ...O......I.....
00000000`546a0030  00 00 6a 54 00 00 00 00-cf 0f 00 00 00 00 00 00  ..jT............
00000000`546a0040  70 00 6a 54 00 00 00 00-00 f0 66 55 00 00 00 00  p.jT......fU....
00000000`546a0050  4e 00 00 00 01 00 00 00-00 00 00 00 00 00 00 00  N...............
00000000`546a0060  e0 0f 62 55 00 00 00 00-e0 0f 62 55 00 00 00 00  ..bU......bU....
00000000`546a0070  00 00 00 00 00 00 00 00-07 88 15 3f 07 82 01 10  ...........?....
0:000> d
00000000`546a0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a0090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a00a0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a00b0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a00c0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a00d0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a00e0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546a00f0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
(lots of zeroes)
```

Okay, so it's a lot of zeroes. Let's ask the debugger where the heap block begins and how big it is.

```
0:000> !heap -p -a 00000000`546a0100
    address 00000000546a0100 found in
    _HEAP @ 490000
              HEAP_ENTRY Size Prev Flags            UserPtr UserSize - state
        00000000546a0070 7bc1 0000  [00]   00000000546a0080    7bc00 - (busy)
```

The heap block begins at `00000000546a0080` (`UserPtr`) and is of size `7bc00` (`UserSize`), or around half a megabyte of memory. If we keep dumping that memory, we see that it eventually starts containing nonzero memory.

```
00000000`546b0900  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0910  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0920  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0930  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0940  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0950  b1 b1 b1 b1 ff ff ff ff-ff ff ff ff ff ff ff ff  ................
00000000`546b0960  ff ff ff ff d1 d1 d1 d1-00 00 00 00 00 00 00 00  ................
00000000`546b0970  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0980  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0990  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b09a0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b09b0  00 00 00 00 da da da da-ff ff ff ff ff ff ff ff  ................
00000000`546b09c0  ff ff ff ff ff ff ff ff-a3 a3 a3 a3 00 00 00 00  ................
00000000`546b09d0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b09e0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b09f0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0a00  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0a10  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0a20  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0a30  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`546b0a40  00 00 00 00 00 00 00 00-93 93 93 93 da da da da  ................
00000000`546b0a50  da da da da da da da da-da da da da da da da da  ................
00000000`546b0a60  da da da da da da da da-da da da da da da da da  ................
00000000`546b0a70  b1 b1 b1 b1 82 82 82 82-00 00 00 00 00 00 00 00  ................
```

This looks like a 32bpp bitmap of monochrome data. I'm guessing a 32bpp bitmap because all the nonzero pieces come in four-byte chunks. Monochrome because the first three bytes are equal (indicating a grayscale value). The fourth byte would be the alpha channel, which matches the other three, so this could be an all-white bitmap with premultiplied alpha and varying per-pixel opacity. Not sure, though.

We don't know the dimensions of the bitmap, so we'll have to guess. I wrote a program that lets me interactively render a bitmap at different strides, so I can play with it until the result looks decent.

I started with the scratch program and made these changes:

```
DWORD bitmapSize;
BYTE* rawPixels;
int dir = 1;

void OnChar(HWND hwnd, TCHAR ch, int cRepeat)
{
 if (ch == ' ') {
  dir = -dir;
  InvalidateRect(hwnd, nullptr, FALSE);
 }
}

void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
  RECT rc;
  GetClientRect(hwnd, &rc);
  if (rc.right) {
    int cx = rc.right;
    int cy = bitmapSize / 4 / cx;
    BITMAPINFO bi = {};
    bi.bmiHeader.biSize = sizeof(bi.bmiHeader);
    bi.bmiHeader.biWidth = cx;
    bi.bmiHeader.biHeight = cy * dir;
    bi.bmiHeader.biPlanes = 1;
    bi.bmiHeader.biBitCount = 32;
    bi.bmiHeader.biCompression = BI_RGB;
    SetDIBitsToDevice(pps->hdc, 0, 0, cx, cy,
                      0, 0, 0, cy,
                      rawPixels, &bi,
                      DIB_RGB_COLORS);
  }
}

HANDLE_MSG(hwnd, WM_CHAR, OnChar);

BOOL
InitApp(void)
{
    WNDCLASS wc;

    wc.style = CS_HREDRAW;
    wc.lpfnWndProc = WndProc;
    ....
}
int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hinstPrev,
    LPSTR lpCmdLine, int nShowCmd)
{
    MSG msg;
    HWND hwnd;

    int argc;
```

```
    auto argv = CommandLineToArgvW(GetCommandLineW(), &argc);
    auto h = CreateFileW(argv[1], GENERIC_READ, 0, nullptr,
                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
    bitmapSize = GetFileSize(h, nullptr);
    rawPixels = new BYTE[bitmapSize];
    DWORD actual;
    ReadFile(h, rawPixels, bitmapSize, &actual, nullptr);
    CloseHandle(h);

    g_hinst = hinst;
    ...
}
```

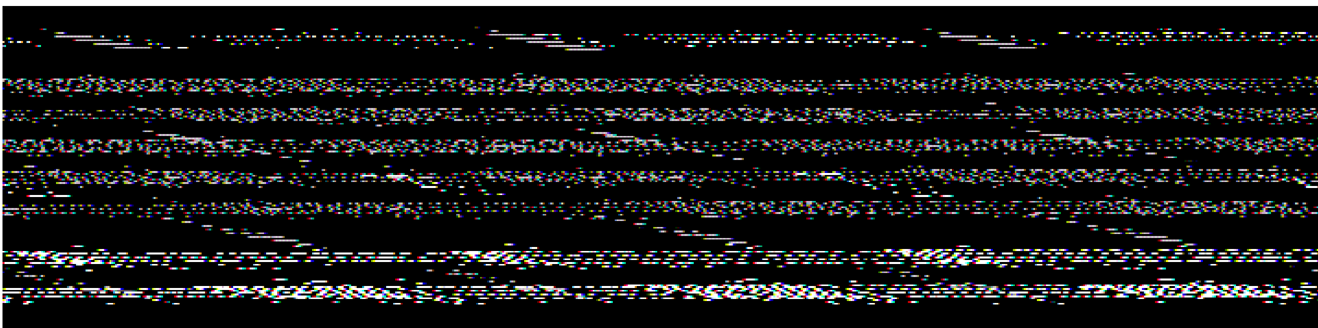
This program loads a 32bpp bitmap whose raw pixels are in a file whose name is given on the command line. It renders the contents of the bitmap assuming the current window width is the stride, and derives the proposed height from that width. The rendering is done with the `SetDIBitsToDevice` function, which lets us take a block of memory and render it directly into a device context without putting it into a bitmap first. The sign of the `biHeight` controls whether it is a bottom-to-top DIB or a top-to-bottom DIB. I don't know which kind of bitmap I have, so I use the space bar to flip the direction.

The idea is that you load up the raw pixels into the window, and then resize the window until you see something that isn't garbage. I add the `CS_ HREDRAW` window class flag to say that I want to perform a full repaint any time the width changes.

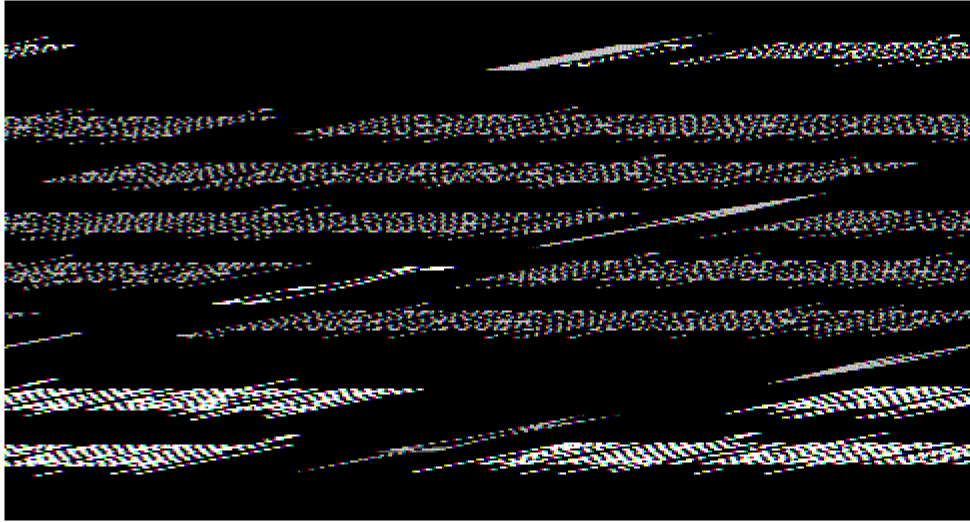
Okay, now to get the raw pixels. We can do that from the debugger.

```
0:000> .writemem c:\temp\mystery 00000000`546a0080 L 7bc00
```

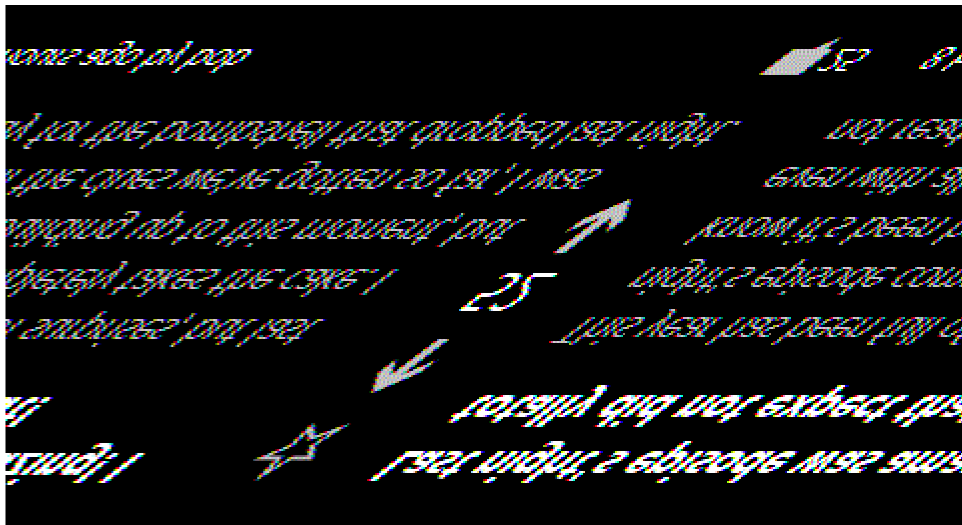
Feeding the file into the scratch program shows this:



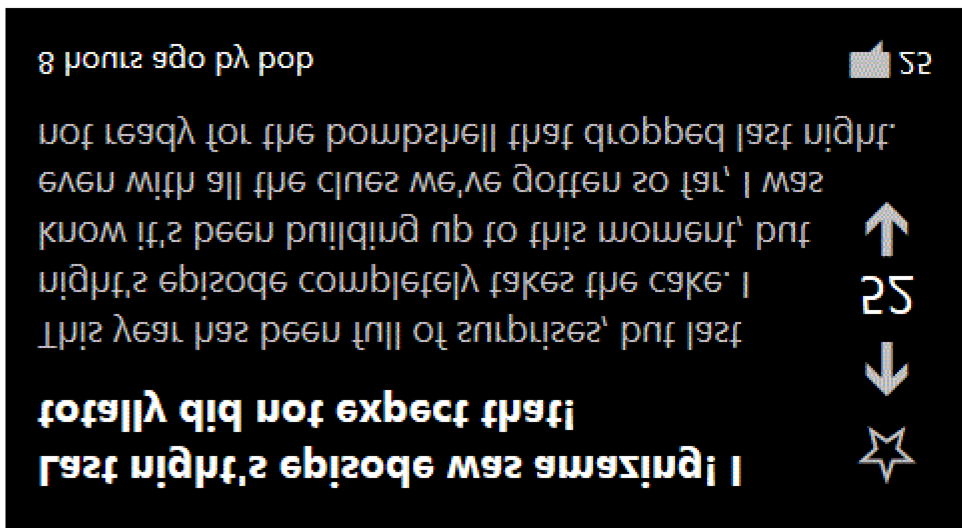
Now I start making the window more narrow in an attempt to find the correct stride by trial and error.



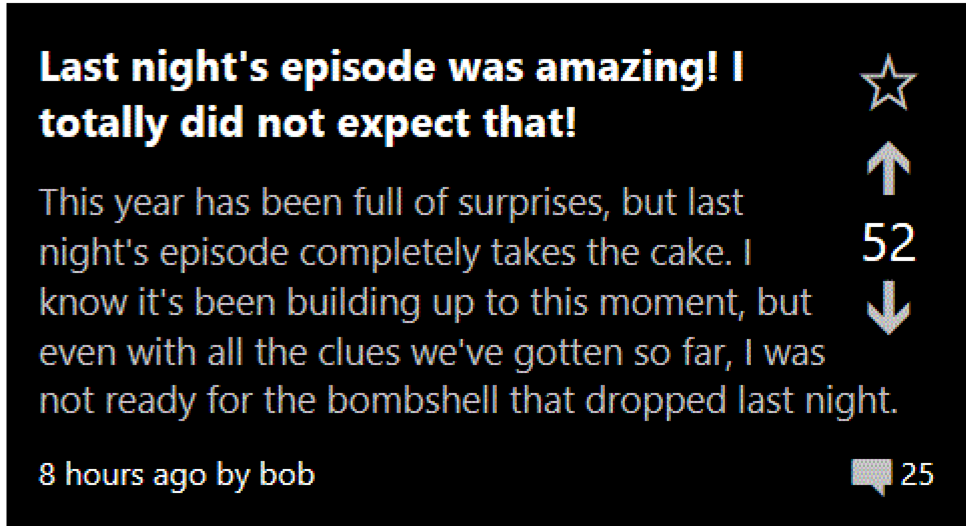
This is exactly like adjusting the horizontal hold on your analog TV set, for those of you old enough to remember analog TV sets.



Almost there.



Success. Except it’s upside-down. So tap the space bar to flip the image.



Bingo. It’s some sort of information tile.

Okay, what about the next memory block?

```
0:000> db 00000000546a0080+7bc00
00000000`5471bc80  00 00 00 00 00 00 00 00-07 88 15 3f c1 f9 08 10  ...........?....
00000000`5471bc90  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`5471bca0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`5471bcb0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`5471bcc0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`5471bcd0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`5471bce0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`5471bcf0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0:000> !heap -p -a 00000000`5471bc90
    address 000000005471bc90 found in
    _HEAP @ 490000
              HEAP_ENTRY Size Prev Flags            UserPtr UserSize - state
        000000005471bc80 7bc1 0000  [00]   000000005471bc90    7bc00 - (busy)
0:000> .writemem c:\temp\mystery2 000000005471bc90 L 7bc00
```

This is another block of exactly the same size. Feeding this to the scratch program confirms that it too is an information tile.

The next memory block is

```
0:000> db 00000000`5471bc80+7bc00+10
00000000`54797890  00 00 00 00 00 00 00 00-07 88 15 3f c1 f9 10 10  ...........?....
00000000`547978a0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`547978b0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`547978c0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`547978d0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`547978e0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`547978f0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`54797900  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0:000> !heap -p -a 00000000`547978a0
    address 00000000547978a0 found in
    _HEAP @ 490000
              HEAP_ENTRY Size Prev Flags            UserPtr UserSize - state
        0000000054797890 7bc1 0000  [00]   00000000547978a0    7bc00 - (busy)
0:000> .writemem c:\temp\mystery3 00000000547978a0 L 7bc00
```

This also decodes as an information tile.

So three for three on the leaked memory blocks being bitmaps of information tiles. Hopefully this is enough of a hint to give the developers a clue where to look for their leak.

Bonus chatter: Debugging the problem took much less time than writing this up. The most time-consuming part was creating a fake information tile!

Raymond Chen

Follow