security-assessment.com

White Paper

Exploiting Freelist[0] On XP Service Pack 2

Prepared by: Brett Moore
Network Intrusion Specialist, CTO
Security-Assessment.com
brett.moore@security-assessment.com

Date: December 2005

Abstract

Windows XP Service pack 2 introduced some new security measures in an attempt to prevent the use of overwritten heap headers to do arbitrary byte writing. This method of exploiting heap overflows, and the protection offered by service pack 2, is widely known and has been well documented in the past.

What this paper will attempt to explain is how other functionality of the heap management code can be used to gain execution control after a chunk header has been overwritten.

In particular this paper takes a look at exploiting freelist[0] overwrites.

This document does not cover the basics of heap overflows, and we suggest the reader has studied and fully understands the following material;

- Reliable Windows Heap Exploits, Matt Conover & Oded Horovitz http://www.cybertech.net/~sh0ksh0k/heap/CSW04%20-%20Reliable%20Windows%20Heap%20Exploits.ppt
- XPSP2 Heap Exploitation, Matt Conover http://www.cybertech.net/~sh0ksh0k/heap/XPSP2%20Heap%20Exploitation.ppt

Introduction

Firstly the methods explained in this document are not generic. They are specific to the vulnerable application and heap overflow situation. However, it is highly likely that these methods could be used under real world scenarios.

To demonstrate code execution conditions a sample vulnerable application has been constructed that contains a function lookup table. The sample exploits either overwrite the function table or a pointer to it which is then used in a call instruction.

The later two examples demonstrate how the atexit() pointers and CRT termination routines can be overwritten to execute arbitrary code.

XP SP 2 Heap Protection

Windows XP service pack 2 contains code to prevent the abuse of the unlinking functionality of the heap management routines. It does this by validating the forward and back links before doing the unlinking.

If this validation fails, a debug message is displayed (if a debugger is enabled) BUT execution will still continue as long as an exception is not caused.

It is due to this continued execution that the methods explained in this document are successful; in fact one of the methods exploits the fact that the heap management will return a known invalid heap chunk to the requesting process.

The Methods

Two new methods of exploitation are explained in this paper. The first allows for the address of user supplied data to be written to a semi arbitrary location. The other allows for a semi arbitrary address to be returned to a HeapAlloc call.

The Heap In Use

Initially one free chunk is created, that is then split up as required. Chunks are freed to either the lookaside or freelists linked lists. Chunks on the freelists are coalesced when one or more marked free chunk are grouped together.

The Heap Chunks

The heap allocates memory in blocks that are referred to as chunks. A heap chunk consists of both the chunk header and the chunk data.

A Used Chunk

WORD	WORD	BYTE	BYTE	BYTE	BYTE
Self Size	Prev Size	CK	FL	UN	SI
User Data					
User Data					

A Chunk On Lookaside

WORD	WORD	BYTE	BYTE	BYTE	BYTE
Self Size	Prev Size	CK	FL	UN	SI
FLINK		User Data			
User Data					

A Chunk On FreeLists

WORD	WORD	BYTE	BYTE	BYTE	BYTE
Self Size	Prev Size	CK	FL	UN	SI
FLINK		BLINK			
User Data					

Chunk Fields

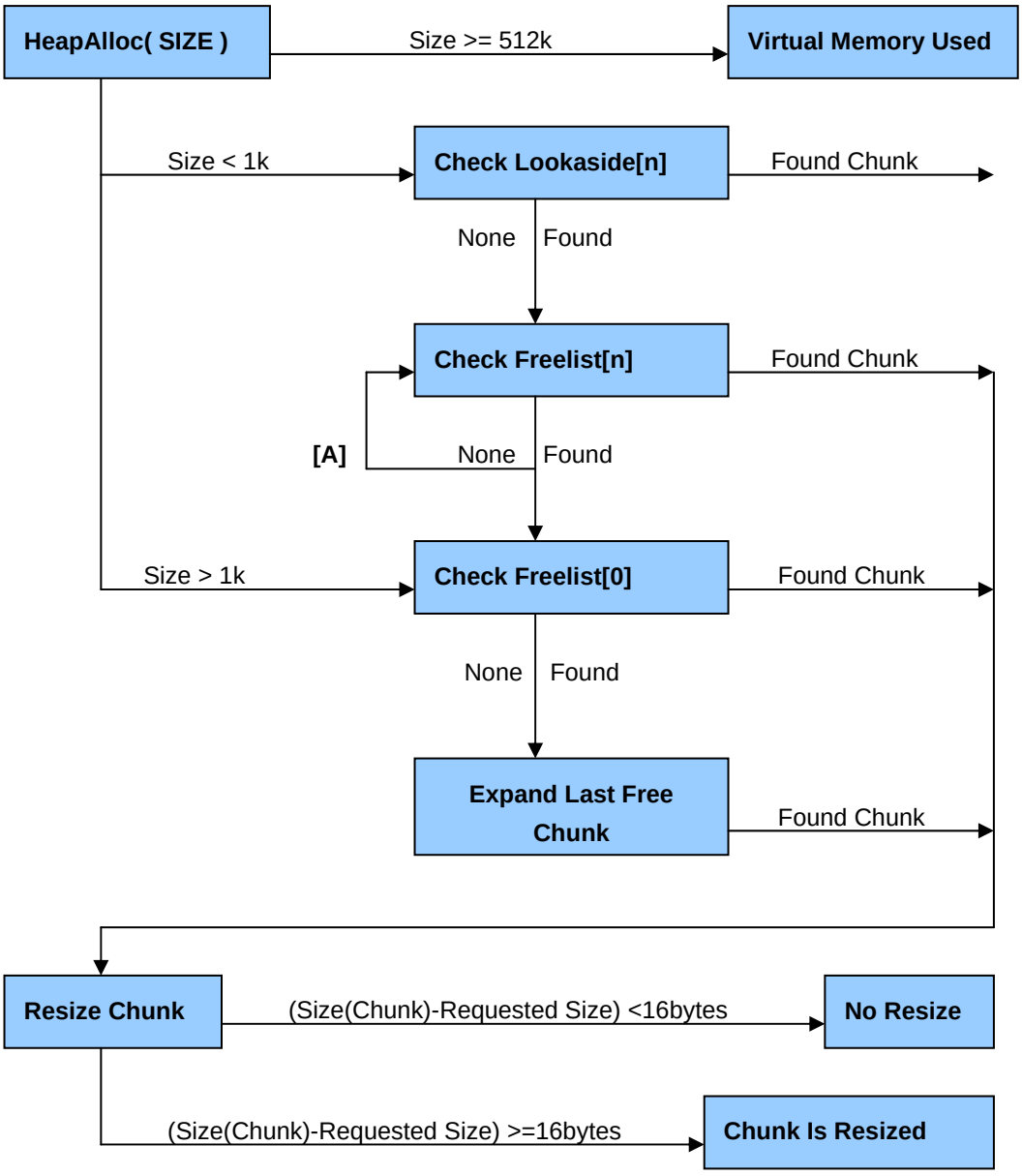
Self Size
Size of this chunk
Prev Size
Size of previous chunk
CK
Chunk cookie
FL
Chunk flags
UN
Unused
SI
Segment Index
FLINK
Forward Link
BLINK
Back Link

Flags

00	Free
01	Busy
02	Extra Present
04	Fill Pattern
08	Virtual Alloc
10	Last Entry
20	FFU1
40	FFU2
80	No Coalesce

Allocation Process

When a request to allocate heap memory is received the following process is followed.



If the found chunk needs to be resized then the following happens;

- Chunk->Size is set to Requested Size
- A new chunk is created at Chunk+Requested Size
- This new chunk is returned to the applicable freelist

[A] If a chunk doesn't exist in the applicable freelist[n] it looks for a freelist holding larger chunks. If none exist then it falls back to checking freelists[0]. It does not check the lookaside for larger chunks.

Alloc From Freelist[0]

When an allocation is done from freelist[0] the heap manager first checks to make sure that there is at least one chunk large enough for the request. It does this by checking the size of the last chunk in the list.

If the last chunk is large enough, then searching starts at the first chunk in the list. The heap manager moves through the list by loading the FLINK of each chunk and checking the size, until it finds a chunk large enough.

```
7C9113E2   movzx       eax,word ptr [eax]           ; Check size of last chunk
7C9113E5   cmp         eax,edi                      ; Compare against requested
7C9113E7   jb          7C911C6B                     ; Jump if too small (need to alloc more)
7C9113ED   mov         eax,dword ptr [ebp-28h]      ; Get first chunk in freelist[0]

7C9113F0   mov         eax,dword ptr [eax]          ; Load FLINK
7C9113F2   mov         dword ptr [ebp-6Ch],eax
7C9113F5   cmp         dword ptr [ebp-28h],eax
7C9113F8   je          7C911C6B
7C9113FE   lea         esi,[eax-8]                  ; Move to size
7C911401   mov         dword ptr [ebp-38h],esi
7C911404   movzx       ecx,word ptr [esi]           ; Load size
7C911407   cmp         ecx,edi                      ; Check size
7C911409   jb          7C9113F0                     ; To small get next (JMP above)
```

Once a suitable chunk is found, it is unlinked as per normal freelist[] unlinking.

```
7C91142E   mov         edi,dword ptr [ecx]          ; Do the security check
7C911430   cmp         edi,dword ptr [eax+4]
7C911433   jne         7C934380                     ; Jump if corrupt
7C911439   cmp         edi,edx
7C91143B   jne         7C934380                     ; Jump if corrupt

7C911441   mov         dword ptr [ecx],eax          ; Do the unlink
7C911443   mov         dword ptr [eax+4],ecx        ;

. Return to here from corrupt chunk message
7C911446   mov         al,byte ptr [esi+5]          ; Get flag
```

If the size of the chunk to use is larger than the requested size by more than 1 block, then it needs to be split and a new chunk header created.

```
7C911496   test        ebx,ebx                      ; Check if diff = 0 (No resize)
7C911498   je          7C911566
7C91149E   cmp         ebx,1                        ; Check if difference is 1 block
7C9114A1   je          7C911158                     ; Jump if no resize required

7C9114A7   mov         eax,dword ptr [ebp-64h]      ; Start Creation of new chunk
7C9114AA   lea         edi,[esi+eax*8]              ; Move to place new header

7C9114AD   mov         dword ptr [ebp-144h],edi     ;
7C9114B3   mov         cl,byte ptr [ebp-1Dh]        ;
7C9114B6   mov         byte ptr [edi+5],cl          ;
7C9114B9   mov         word ptr [edi+2],ax          ; CREATE NEW HEADER
7C9114BD   mov         al,byte ptr [esi+7]          ;
7C9114C0   mov         byte ptr [edi+7],al          ;
7C9114C3   mov         word ptr [edi],bx            ; Set Size of new chunk
7C9114C6   test        cl,10h                       ;
7C9114C9   je          7C911633                     ; Test if last chunk

7C9114CF   xor         eax,eax
7C9114D1   mov         al,byte ptr [edi+5]
7C9114D4   and         eax,10h
7C9114D7   mov         byte ptr [edi+5],al
```

Then the chunk needs to be inserted back in the freelists[]. If the size of the new chunk is >= 80 blocks then this is done by checking all the chunks in freelist[0] to find one larger than the newly created chunk. Once this is done the new chunk is linked back in to the freelist

```
7C9114DA   cmp         bx,80h                      ; Check size of new chunk
7C9114DF   jb          7C911815                    ; Jump if < 80

                                                   ; New chunk will be in freelist[0]
7C9114E5   mov         eax,dword ptr [ebp-1Ch]
7C9114E8   lea         esi,[eax+178h]              ; Ptr to freelist[0]
7C9114EE   mov         dword ptr [ebp-0E0h],esi
7C9114F4   cmp         dword ptr [eax+170h],0
7C9114FB   jne         7C9122DC

7C911501   mov         eax,dword ptr [esi]         ; Load ptr from freelist
7C911503   mov         ecx,eax
7C911505   mov         dword ptr [ebp-90h],ecx
7C91150B   cmp         esi,ecx                     ; Compare to see if last Chunk
7C91150D   jne         7C910FCA                    ; If not last chunk (below)

7C911513   lea         eax,[edi+8]                 ; Move to FLINK of new chunk
7C911516   mov         dword ptr [ebp-0F0h],eax    ;
7C91151C   mov         edx,dword ptr [ecx+4]       ; Load BLINK From FLINK
7C91151F   mov         dword ptr [ebp-0F8h],edx    ;

7C911525   mov         dword ptr [eax],ecx         ; Store Flink IN NEW CHUNK
7C911527   mov         dword ptr [eax+4],edx       ; Store BLINK in new chunk

7C91152A   mov         dword ptr [edx],eax         ; Store Blinks->FLINK
7C91152C   mov         dword ptr [ecx+4],eax       ; Store Flinks->BLINK
...

7C910FCA   lea         eax,[ecx-8]                 ; Move to beginning of current chunk
7C910FCD   mov         dword ptr [ebp-0E8h],eax    ;
7C910FD3   cmp         bx,word ptr [eax]           ; Check size
7C910FD6   jbe         7C911513                    ; Large enough

7C910FDC   mov         ecx,dword ptr [ecx]         ; Otherwise Load FLINK
7C910FDE   jmp         7C911505                    ; Jump above and try again
```

Valid Allocation

The following shows the process in a valid allocation.

FreeList[0]

0x00340178h	0x00341E90h	0x003430C0h

Chunk A

0x00341E88	0x0241h	0x0301h	CK	FL	UN	SI
0x00341E90h	0x003430C0h		0x00340178h			

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00341E90h			

The process does an allocation.

g = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,0x30)

Once a suitable chunk (Chunk A) is found, it is unlinked as per normal freelist[] unlinking.

FreeList[0]

0x00340178h	0x003430C0h	0x003430C0h

Chunk A

0x00341E88	0x0241h	0x0301h	CK	FL	UN	SI
0x00341E90h	0x003430C0h		0x00340178h			

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00340178h			

Since Chunk A is larger than the amount requested, a resize occurs.

FreeList[0]

0x00340178h	0x003430C0h	0x003430C0h

Chunk A

0x00341E88	0x0007h	0x0301h	CK	FL	UN	SI
0x00341E90h	0x003430C0h		0x00340178h			

Chunk A(2)

0x00341EC0	0x023Ah	0x0007h	CK	FL	UN	SI
0x00341EC8h						

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00340178h			

The heap manager starts searching at the FLINK of freelist[0]. In this case Freelist[0]->FLINK points to the last chunk, so Chunk A(2) is inserted before Chunk B.

ChunkA(2)->FLINK = Chunk B
ChunkA(2)->BLINK = Chunk B->BLINK
ChunkB->BLINK->FLINK = Chunk A(2)
ChunkB->BLINK – ChunkA(2)

Exploiting Freelist[0] ReLinking

The following shows the process when the header of any chunk sitting in freelist[0] is overwritten. It exploits the fact that our overwritten header is used to calculate the linked list position to insert a resized block.

FreeList[0] with overflowed chunk A header

0x00340178h	0x00341E90h	0x003430C0h

Chunk A

0x00341E88	0x0202h	0x0202h	0x58585858h			
0x00341E90h	0xAAAAAAAAh		0xAAAAAAAAh			

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00341E90h			

The process does an allocation.

g = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,0x30)

Once a suitable chunk (Chunk A) is found, it is unlinked as per normal freelist[] unlinking. EXCEPT, the security check will fail. So Freelists[0] will not be updated

FreeList[0]

0x00340178h	0x00341E90h	0x003430C0h

Chunk A

0x00341E88	0x0202h	0x0202h	0x58585858h
0x00341E90h	0xAAAAAAAAh		0xAAAAAAAAh

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00341E90h			

Since Chunk A is larger than the amount requested, a resize occurs.

FreeList[0]

0x00340178h	0x00341E90h	0x003430C0h

Chunk A

0x00341E88	0x0007h	0x0202h	0x58585858h
0x00341E90h	0xAAAAAAAAh		0xAAAAAAAAh

Chunk A(2)

0x00341EC0	0x01FBh	0x0007h	CK	FL	UN	SI
0x00341EC8h						

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00341E90h			

The heap manager starts searching at the FLINK of freelist[0]. In this case Freelist[0]->FLINK still points to chunk A. Chunk A has been updated with the requested size, so is smaller than the new chunks size. The heap manager will load the FLINK from chunk A as the address of the next chunk in the list.

If ChunkA->FLINK points to a ‘fake chunk’ with a layout as shown below

Fake Chunk

-8	> newsize	????	????????
0xAAAAAAAAh	????????		0xWRITEABLE

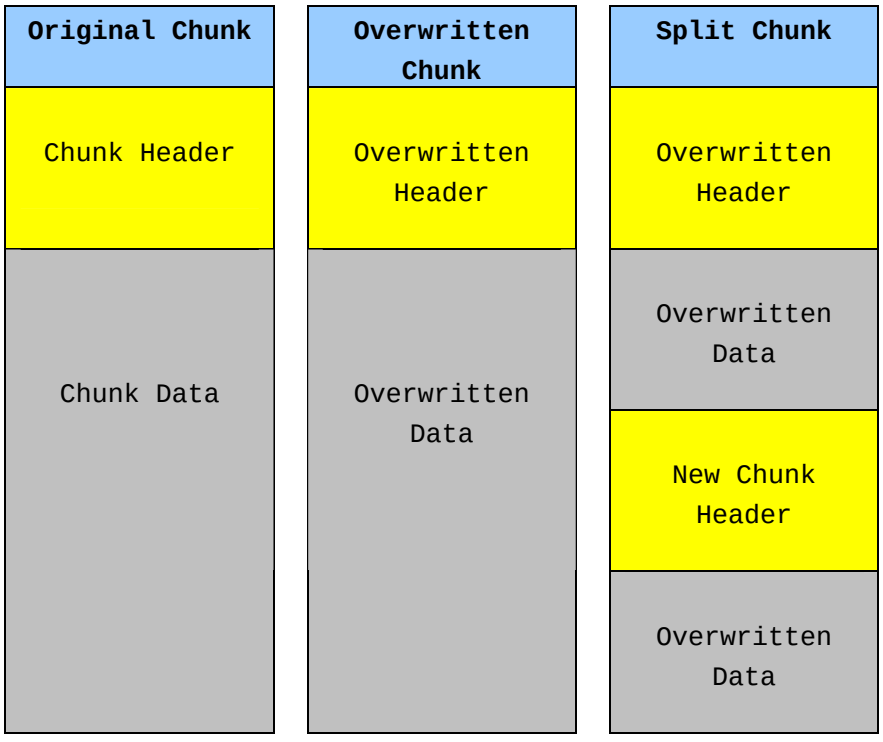
Then Chunk A(2) will be inserted before our new chunk.
Thus

ChunkA(2)->FLINK = Chunk B
ChunkA(2)->BLINK = Chunk B->BLINK
ChunkB->BLINK->FLINK = Chunk A(2)
ChunkB->BLINK – ChunkA(2)

Which will cause;

ChunkA(2)->FLINK = 0xAAAAAAAAh
ChunkA(2)->BLINK = 0xWRITEABLE
[0xWRITEABLE] = Chunk A(2)
[0xAAAAAAAAh +4]= ChunkA(2)

As can be seen the address of the new chunk is written to an arbitrary location.
So what good is this?



The address that overwrites the location at our fake chunks FLINK points to the FLINK of the new chunk header. So it can be used to overwrite a pointer to a lookup table list, or if the FLINK/BLINK is executable opcodes it can be used to overwrite a function pointer.

See Sample1.c for an example of this method

Exploiting Freelist[0] Searching

The following shows the process when the header of any chunk, except the last, sitting in freelist[0] is overwritten. It exploits the fact that the overwritten header is used to search for a chunk to return to the process.

The size field of Chunk A is overwritten with a smaller size than is requested, and the FLINK is overwritten with the address we want to force the allocation to return. The BLINK of the chunk is not overwritten.

FreeList[0] with overflowed chunk A header

0x00340178h	0x00341E90h	0x003430C0h

Chunk A

0x00341E88	0x0001h	0x0001h	0x58585858h
0x00341E90h	0xAAAAAAAAh		0x00390178h

Chunk B

0x003430B8h	0x03E9h	0x0005h	CK	FL	UN	SI
0x003430C0h	0x00340178h		0x00341E90h			

The process does an allocation.

g = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,0x30)

If the overwritten chunk is queried during the search, then the FLINK will be loaded and used as the address of the next chunk in the list.

ChunkA->FLINK needs to point to a 'fake chunk' with a layout as shown below.

#size needs to <= the requested number of blocks +1, to prevent chunk resizing.

Fake Chunk

-8	#newsize	????	????????
0xAAAAAAAAh	0xREADABLE		0xREADABLE

The heap manager will attempt to use the fake chunk as if it were real. This means that it will attempt to unlink the chunk from freelist[0] which is why the two addresses need to be readable. The unlinking will fail but the heap manager will still return the address of the fake chunk to the process.

This will allow for the overwriting of arbitrary memory with any data the process stores in the returned chunk.

See Sample2.c for an example of this method

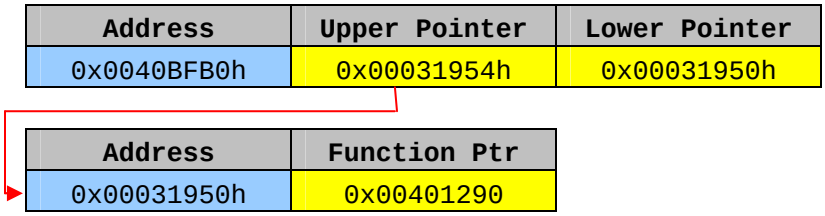
Exploiting atexit() Pointers

If the vulnerable process has installed an atexit pointer through the use of the atexit() function, it is possible to overwrite the pointer table.

When the process exits, the pointer table is referenced using code similar to that shown below.

```
0040141C   mov        ecx,dword ptr ds:[40BFB0h]    ; Load upper pointer
00401422   push       esi
00401423   lea        esi,[ecx-4]                   ; Adjust
00401426   cmp        esi,eax                       ; Check list not empty
00401428   jb         0040143D
0040142A   mov        eax,dword ptr [esi]           ; Load the function pointer
0040142C   test       eax,eax                       ; Check it is not NULL
0040142E   je         00401432
00401430   call       eax                           ; Call the function
00401432   sub        esi,4                         ; Adjust upper pointer
00401435   cmp        esi,dword ptr ds:[40BFB4h]    ; Check against lower pointer
0040143B   jae        0040142A                      ; Jump back and call next function
```

The referenced table looks like the diagram below.



This sample is included as an example only and would probably be difficult to carry out under real world conditions. The exploit will overwrite the freelists[n] with pointers to the atexit pointer table.

To overwrite the atexit() ptr table we will;

1. overwrite a chunk in freelist[0]
2. exploit the freelist[0] searching routine to return a pointer to the heap freelists
3. overwrite the freelists with a pointer to the atexit() pointer table
4. overwrite the atexit() ptr table to point to a portion of itself containing the code to execute.

See Sample3.c for this example

Exploiting CRT Termination Pointers

If the vulnerable process has been dynamically linked with MSVCRT.DLL then it is possible to overwrite the CRT termination pointer table. The termination routine has changed slightly since I talked about it at Black Hat 2004 but is still exploitable under the right situation.

When the process exits, the pointer table is referenced using code similar to that shown below.

```
77C39E06   mov        ecx,dword ptr ds:[77C627A0h]  ; Load lower pointer
77C39E0C   test       ecx,ecx                       ; Check it is not NULL
77C39E0E   je         77C39E39
77C39E10   mov        eax,[77C6279C]                ; Load upper pointer
77C39E15   sub        eax,4                         ; Adjust
77C39E18   cmp        eax,ecx                       ; Check not finished
77C39E1A   jmp        77C39E32
77C39E1C   mov        eax,dword ptr [eax]           ; Load the function pointer
77C39E1E   test       eax,eax                       ; Check it is not NULL
77C39E20   je         77C39E24
77C39E22   call       eax                           ; Call the termination routine
77C39E24   mov        eax,[77C6279C]                ; Load the upper pointer
77C39E29   sub        eax,4                         ; Adjust it
77C39E2C   cmp        eax,dword ptr ds:[77C627A0h]  ; Check not finished
77C39E32   mov        [77C6279C],eax                ; Store the new upper pointer
77C39E37   jae        77C39E1C                      ; Jump back to call the function
```

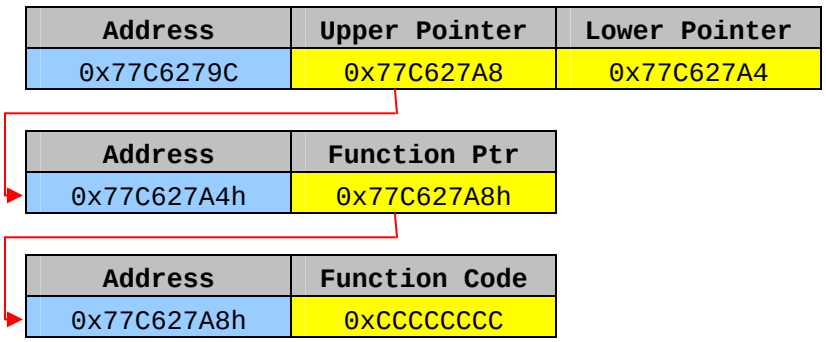
Again, this sample is included as an example only and would probably be difficult to carry out under real world conditions. The exploit will overwrite the heaps lookaside list pointer, with a pointer to the overflow data. The overflow data will contain pointers to the CRT termination pointer table.

To overwrite the termination ptr table we will;

1. overwrite a chunk in freelist[0]
2. exploit the freelist[0] relinking routine to overwrite the heap pointer to the lookaside list @ [base+0x580h]
3. This will cause the lookaside list to contain our pointers to the termination() pointer table
4. overwrite the termination() pointer table to point to a portion of itself containing the code to execute.

See Sample4.c for this example

After overwriting the termination pointer table it will look like this.



Final Summary

The age of the standard 4byte arbitrary write method of heap overflow exploitation is over. Microsoft has appeared to fix the problem with a quick and simple fix.

The only obvious flaw with the FLINK/BLINK fix is that after a heap corruption has been detected, process execution will continue. Perhaps there could be a per process setting, so that important services could restart after a heap corruption instead of blindly using user supplied input for further operations.

Hopefully this paper has shown that heap overflows are still dangerous and in some situations can still be exploited reliably and easily. This paper has demonstrated two new methods of exploiting heap overflows, and there are sure to be more.

References

Reliable Windows Heap Exploits, Matt Conover & Oded Horovitz
http://www.cybertech.net/~sh0ksh0k/heap/CSW04%20-%20Reliable%20Windows%20Heap%20Exploits.ppt
XPSP2 Heap Exploitation, Matt Conover
http://www.cybertech.net/~sh0ksh0k/heap/XPSP2%20Heap%20Exploitation.ppt

Security-Assessment.com
www.security-assessment.com

About Security-Assessment.com

Security-Assessment.com is an established team of Information Security consultants specialising in providing high quality Information Security Assurance services to clients throughout Australasia. We provide independent advice, in-depth knowledge and high level technical expertise to clients who range from small businesses to some of the worlds largest companies

Using proven security principles and practices combined with leading software and proprietary solutions we work with our clients to provide simple and appropriate assurance solutions to Information security challenges that are easy to understand and use for their clients.

Security-Assessment.com provides security solutions that enable developers, government and enterprises to add strong security to their businesses, devices, networks and applications.

Copyright Information