第 3 章　堆溢出漏洞分析

3.1　堆溢出简史

以前，很多Exploit公布站点上关于堆溢出的Exploit明显要比栈溢出少，一方面是由于堆漏洞比栈漏洞更难发现，另一方面是因为其利用难度较栈溢出大，而且受漏洞场景影响较大，很多以往的堆溢出利用方式只能在Windows XP下运行，并且受SP版本影响，特别是XP SP2之后对堆结构做了较大改动，这也大大地增加了黑客及安全研究人员在利用堆溢出漏洞上投入的成本。

1999年，w00w00安全小组的Matt Conver编写了*w00w00 on Heap Overflows*一文，中文版译名为《Heap/BSS溢出机理分析》，该文是最早将堆溢出原理公布的佳作。当时，Matt只有十几岁，无疑是安全界的传奇少年，其在堆溢出方面具有相当深入的研究。

2002年，在BlackHat大会上，Halvar Flake发表题为*Third Generation Exploitation*的演讲，主要揭露Windows 2000平台上的堆数据结构和算法，并提出堆溢出利用方法。

2003年9月，Dave Aitel针对MS03-026:RPC DCOM堆溢出漏洞写的*Exploiting the MSRPC Heap Overflow*一文详细描述了针对实际堆溢出漏洞的Exploit技巧——Arbitrary DWORD Reset（在《0day安全：软件漏洞分析技术》一书中称为“DWORD SHOOT”）。著名的“冲击波病毒”正是利用此漏洞发动攻击的，“冲击波”的源码在网上可下载。

2004年，David Litchfield在BlackHat大会上发表的演讲*Windows Heap Overflows*较为全面地讲述了Windows 2000/XP 平台下的堆溢出利用技术，覆盖VEH、UEF、PEB和TEB等方法，这些利用方法在《0day安全：软件漏洞分析技术》的第8章“高级内存攻击技术”中有详细讲解。

2004年，Matt Conver在CanSecWest黑客大会上发表的演讲*Windows Heap Exploitation*全面地揭露了从Windows 2K SP0到Windows XP SP2平台上的堆溢出利用技术，包括如何突破Windows XP SP2上的堆保护机制。

2007年，Nicolas Waisman发表*Understanding and bypassing Windows Heap Protection*，主要讲解Windows XP SP2及Windows Vista的堆保护机制及绕过方法，同时提供多款Immunity Debugger（ImmDeb）插件用于调试堆漏洞，也正是因为Immunity Debugger提供了许多漏洞分析辅助插件，所以很多漏洞分析者都习惯使用ImmDbg调试漏洞，尤其是国外。

2008年，Ben Hawkes在Ruxcon大会上演讲的*Attacking the Vista Heap*首次公开了Windows Vista平台下的堆内部结构，以及绕过Vista堆保护的方法。

2009年，John McDonald和Chris Valasek在BlackHat USA大会上发表题为*Practical Windows XP/2003 Heap Exploitation*的演讲，详细地介绍了Windows XP SP3和Server 2003的堆结构及核心算法，同时较为全面地总结出各种堆溢出利用技术。

2010年，在Pwn2Own 2010黑客大赛上Peter vreugdenhil利用内存泄露和覆盖虚表指针的方法攻下Windows 7的IE8浏览器，此处利用的是CVE-2011-0027：MDAC ADO记录堆溢出漏洞。首先，利用heap spray将Shellcode传递到可预测的地址，再利用堆溢出覆盖虚表指针劫持EIP，同时借助内存泄露定位对象所属DLL（msado15.dll）的基址来绕过ASLR，又使用ROP技术绕过DEP保护，从而实现Exploit的稳定利用。在随后的几年里，很多信息泄露漏洞被拿来绕过ASLR，而其也被很多大厂商当作独立的漏洞来处理，比如Google、微软等。

现在，随着栈溢出的减少，堆溢出的漏洞反而越来越多了，针对此类型的漏洞利用技术也变多了，利用技术已经逐步趋于成熟。很多主流软件的栈溢出漏洞已经很少见了，比如IE浏览器、Adobe软件，反而是堆上的漏洞成为主流。

3.2　堆溢出原理

以下列代码来演示堆溢出漏洞：

```
// heapoverflow.c
#include <windows.h>
#include <stdio.h>

int main ()
{
        HANDLE hHeap;
        char *heap;
        char str[] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";    // 0x20大小

        hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x1000, 0xffff);
        getchar();    // 用于暂停程序，便于调试器附加进程

        heap = HeapAlloc(hHeap, 0, 0x10);
        printf("heap addr:0x%08x\n",heap);

        strcpy(heap,str);    // 导致堆溢出
        HeapFree(hHeap, 0, heap);    // 触发崩溃

        HeapDestroy(hHeap);
```

```
    return 0;
}
```

分析环境如表3-1所示。

表 3-1 分析环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文旗舰版
编译器	Microsoft Visual C++	版本号：6.0
调试器	WinDbg	版本号：6.1 内部版本 7600

说明：实验过程中的一些地址可能会出现变化，读者需要在实际调试中确定。

由于调试堆与常态堆的结构不同，因此在演示代码中加入getchar函数，主要用于暂停进程，方便运行heapoverflow.exe后用调试器附加进程。Debug版的程序与Release版实际运行的进程中各内存结构和分配过程也是不同的，比如堆分配过程等，因此在测试时，应将heapoverflow.exe编译成RELEASE版。

运行heapoverflow.exe，然后使用WinDbg附加运行，在命令行下输入g命令按回车键后，进程出现异常：

```
0:001> g
(1dec.138c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00660bb8 ebx=00000000 ecx=41414141 edx=00660ba0 esi=00660b98 edi=00660000
eip=77582eef esp=0012fdd8 ebp=0012feb8 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010283
ntdll!RtlpFreeHeap+0x4d6:
77582eef 8b19            mov     ebx,dword ptr [ecx]  ds:0023:41414141=????????
0:000> kb
ChildEBP RetAddr  Args to Child
0012feb8 77582bf0 00660b98 00660ba0 0012ff49 ntdll!RtlpFreeHeap+0x4d6
0012fed8 59ac9ed9 00660000 00000000 00660ba0 ntdll!RtlFreeHeap+0x142
0012fef0 75d1f14c 00660000 00000000 00660ba0
AcXtrnal!NS_FaultTolerantHeap::APIHook_RtlFreeHeap+0x61
0012ff04 00401094 00660000 00000000 00660ba0 kernel32!HeapFree+0x14
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff48 00401327 00000001 004f1980 004f1a08 heapoverflow+0x1094
0012ff88 75d21114 7ffff000 0012ffd4 7758b299 heapoverflow+0x1327
0012ff94 7758b299 7ffff000 74cb5c1c 00000000 kernel32!BaseThreadInitThunk+0xe
0012ffd4 7758b26c 00401273 7ffff000 00000000 ntdll!__RtlUserThreadStart+0x70
0012ffec 00000000 00401273 7ffff000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

上面的ecx已经被我们的“AAAA”字符串覆盖掉了，最后导致在引用该地址时程序崩溃。由于这里使用Release版程序，所以编译时没有生成pdb符号文件，笔者主要通过前面的栈回溯定位到main

函数入口，然后找到复制字符串的地址并对其下断点：

```
0:000> bp 00401084
0:001> g
Breakpoint 0 hit
eax=00000021 ebx=00530ba0 ecx=00000008 edx=77576194 esi=0012ff28 edi=00530ba0
eip=00401084 esp=0012ff10 ebp=00530000 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
heapoverflow+0x1084:
00401084 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> dd edi
00530ba0  005300c4 005300c4 00000000 00000000
00530bb0  00000086 00000003 005300c4 005300c4
00530bc0  00000000 00000000 00000000 00000000
00530bd0  00000000 00000000 00000000 00000000
00530be0  00000000 00000000 00000000 00000000
00530bf0  00000000 00000000 00000000 00000000
00530c00  00000000 00000000 00000000 00000000
00530c10  00000000 00000000 00000000 00000000
0:000> dd esi
0012ff28  41414141 41414141 41414141 41414141
0012ff38  41414141 41414141 41414141 41414141
0012ff48  00407000 00401227 00000001 005e1980
0012ff58  005e1a08 00000000 00000000 7ffff000
0012ff68  00000000 00000000 0012ff5c 00000000
0012ff78  0012ffc4 00402c90 00000000 00000000
0012ff88  0012ff94 75d41114 7ffff000 0012ffd4
0012ff98  77580209 7ffff000 74b4c39a 00000000
```

此时堆块已经分配完成，其对应的分配地址位于0x00530ba0，如图3-1所示。0x00530ba0是堆块数据的起始地址，并非堆头信息的起始地址。对于已分配的堆块，其开头均有8字节的HEAP_ENTRY结构，而对于空闲堆块，它的开头就是HEAP_FREE_ENTRY结构，因此heap的HEAP_ENTRY结构位于0x00530ba0−8=0x00530B98。

图3-1　heapoverflow.exe的输出信息

在WinDbg上查看两个堆块的信息，这两个堆块目前处于占用状态，共有0x10大小的数据空间：

```
0:000> !heap -p -a 0x00530B98
    address 00530b98 found in
    _HEAP @ 530000
```

```
    HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
      00530b98 0003 0000  [00]   00530ba0    00010 - (busy)
```

在WinDbg中使用!heap命令查看HeapCreate创建的整个堆信息，可以发现在堆块heap之后还有个空闲堆块0x00530bb0：

```
0:000> !heap
NtGlobalFlag enables following debugging aids for new heaps:    heap tagging
Index   Address  Name      Debugging options enabled
  1:   00180000  Process
  2:   00010000
  3:   00020000
  4:   003e0000
  5:   005e0000
  6:   00530000
0:000> !heap -a 00530000
……省略部分内容……
    Heap entries for Segment00 in Heap 00530000
        00530000: 00000 . 00b98 [01] - busy (197
        00530b98: 00b98 . 00018 [01] - busy (19) // 占用堆，即示例中的 heap 堆块
        00530bb0: 00018 . 00430 [00]   // 空闲堆
        00530fe0: 00430 . 00020 [11] - busy (1d)
        00531000:     0000f000      - uncommitted bytes.
```

在复制字符串时，原本只有0x10大小的堆，当填充过多的字符串时就会覆盖到下方的空闲堆块0x00530bb0。在复制前0x00530bb0空闲块的HEAP_FREE_ENTRY结构数据如下：

```
0:000> dt _HEAP_FREE_ENTRY 0x00530bb0
ntdll!_HEAP_FREE_ENTRY
   +0x000 Size             : 0x86
   +0x002 Flags            : 0 ''
   +0x003 SmallTagIndex    : 0 ''
   +0x000 SubSegmentCode   : 0x00000086
   +0x004 PreviousSize     : 3
   +0x006 SegmentOffset    : 0 ''
   +0x006 LFHFlags         : 0 ''
   +0x007 UnusedBytes      : 0 ''
   +0x000 FunctionIndex    : 0x86
   +0x002 ContextValue     : 0
   +0x000 InterceptorValue : 0x86
   +0x004 UnusedBytesLength : 3
   +0x006 EntryOffset      : 0 ''
   +0x007 ExtendedBlockSignature : 0 ''
```

```
   +0x000 Code1           : 0x86
   +0x004 Code2           : 3
   +0x006 Code3           : 0 ''
   +0x007 Code4           : 0 ''
   +0x000 AgregateCode    : 0x3`00000086
   +0x008 FreeList        : _LIST_ENTRY [ 0x5300c4 - 0x5300c4 ]
0:000> dt _LIST_ENTRY 0x00530bb0+8
ntdll!_LIST_ENTRY
 [ 0x5300c4 - 0x5300c4 ]
   +0x000 Flink           : 0x005300c4 _LIST_ENTRY [ 0x530bb8 - 0x530bb8 ]
   +0x004 Blink           : 0x005300c4 _LIST_ENTRY [ 0x530bb8 - 0x530bb8 ]
```

覆盖后，0x00530bb0空闲块的HEAP_FREE_ENTRY结构数据如下：

```
0:000> dt _HEAP_FREE_ENTRY 0x00530bb0
ntdll!_HEAP_FREE_ENTRY
   +0x000 Size            : 0x4141
   +0x002 Flags           : 0x41 'A'
   +0x003 SmallTagIndex   : 0x41 'A'
   +0x000 SubSegmentCode  : 0x41414141
   +0x004 PreviousSize    : 0x4141
   +0x006 SegmentOffset   : 0x41 'A'
   +0x006 LFHFlags        : 0x41 'A'
   +0x007 UnusedBytes     : 0x41 'A'
   +0x000 FunctionIndex   : 0x4141
   +0x002 ContextValue    : 0x4141
   +0x000 InterceptorValue : 0x41414141
   +0x004 UnusedBytesLength : 0x4141
   +0x006 EntryOffset     : 0x41 'A'
   +0x007 ExtendedBlockSignature : 0x41 'A'
   +0x000 Code1           : 0x41414141
   +0x004 Code2           : 0x4141
   +0x006 Code3           : 0x41 'A'
   +0x007 Code4           : 0x41 'A'
   +0x000 AgregateCode    : 0x41414141`41414141
   +0x008 FreeList        : _LIST_ENTRY [ 0x41414141 - 0x41414141 ]
```

整个空闲堆头信息都被覆写，包括最后空闲链表中的前后向指针都被覆写为0x41414141。当后面调用HeapFree去释放堆块时，就会将heap与后面的空闲堆0x00530bb0进行合并，修改两个合并堆块的前后向指针，此时就会引用到0x41414141，最后造成异常崩溃。如果将上面释放堆块的操作换成分配堆块HeapAlloc(hHeap, 0, 0x10) 也会导致崩溃，因为在分配堆块时会去遍历空闲链表指针，造成地址引用异常，示例程序的堆溢出原理如图3-2所示。当内存中已分配多个堆块时，可能覆盖到的

就是已分配的堆块，此时可能就是覆盖到HEAP_ENTRY结构，而非HEAP_FREE_ENTRY结构，图3-2是根据上文中的示例程序而画的。如果是反方向溢出就可能覆盖到堆管理结构HEAP和HEAP_SEGMENT结构，这种漏洞称为堆下溢，相对比较少见而且利用难度较大，本书主要讨论堆上溢，即平常所说的堆溢出。

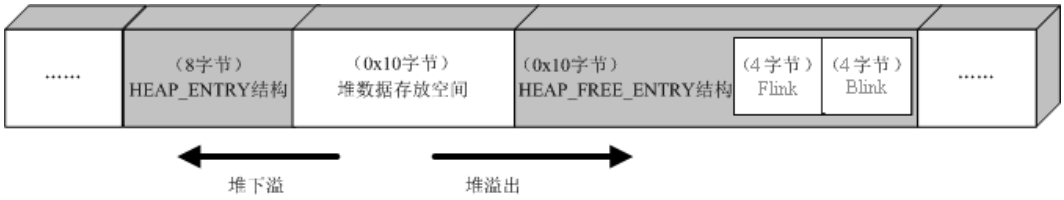

图3-2 堆溢出原理

在Windows XP SP2上可利用调整链表指针的方式，向固定的地址写入可控数据（为Flink数据写入Blink指向的地址），比如，将Shellcode地址写到返回地址，或者覆写PEB的函数指针、SEH结构等，但在后续的系统版本中，逐渐加强对堆溢出的保护，比如，堆cookie、PEB地址随机化、safe-unlink等，这使得以往的很多方法失效。上面运行在Windows 7平台上的示例程序，崩溃时写入地址ebx固定为0，而不再像在Windows XP下那样可控，这也是Windows 7上一些安全机制保护的结果，关于Windows 7平台上堆溢出的利用在实例漏洞章节中我们会提到，到时再讨论。

3.3 堆调试技巧

微软为了帮助程序员快速找到内存错误导致的BUG，在堆管理器中提供了一些调试选项用于辅助堆调试，下面是一些常见的调试选项，可通过WinDbg提供的gflags.exe[默认位于C:\Program Files\Debugging Tools for Windows (x86)目录下]或者!gflag命令来设置：

```
htc - 堆尾检查，在堆块末尾附加额外的标记信息（通常为 8 字节），用于检查堆块是否发生溢出。
hfc - 堆释放检查，在释放堆块时对堆进行各种检查，防止多次释放同一个堆块。
hpc - 堆参数检查，对传递给堆管理的参数进行更多的检查。
ust - 用户态栈回溯，即将每次调用堆函数的函数调用信息记录到一个数据库中。
htg - 堆标志，为堆块增加附加标记，以记录堆块的使用情况或其他信息。
hvc - 调用时验证，即每次调用堆函数时都对整个堆进行验证和检查。
hpa - 启用页堆，在堆块后增加专门用于检测溢出的栅栏页，若发生堆溢出触及栅栏页便会立刻触发异常。
```

比如，要针对app.exe程序添加堆尾检查功能和页堆，去掉堆标志，可以执行以下命令：

```
0:003> !gflag -i app.exe +htc +hpa -htg
```

或者

```
gflags.exe -i app.exe +htc +hpa -htg
```

在堆漏洞调试中，较为常用的是htc、hpc、hfc和hpa。我们以heapoverflow.exe为例，分别采用堆尾检查和页堆进行调试。

3.3.1 堆尾检查

堆尾检查主要是在每个堆块的尾部，即用户数据之后添加8字节内容，通常为连续的2个0xabababab，如果该段数据被破坏就说明可能存在堆溢出。若需要检测堆尾的附加数据是否被覆盖，还需要开启堆参数检查hpc或者堆释放检查hfc，在下面的heapoverflow.exe例子中我们开启hpc和htc就足够了。首先，用WinDbg加载heapoverflow.exe（如果先运行再用WinDbg附加进程开启，就无法在堆尾添加额外的标记信息，也就无法进行堆尾检查），执行以下命令开启堆尾检查和堆参数检查：

```
0:000> !gflag +htc +hpc
New NtGlobalFlag contents: 0x00000050
    htc - Enable heap tail checking
    hpc - Enable heap parameter checking
```

执行g命令后，在cmd运行窗口中按下回车键后断下（程序中使用getchar函数的缘故）：

```
0:000> g
ModLoad: 758a0000 758eb000   C:\Windows\System32\apphelp.dll
ModLoad: 502b0000 50509000   C:\Windows\AppPatch\AcXtrnal.DLL
FTH: (5116): *** Fault tolerant heap shim applied to current process. This is usually due
to previous crashes. ***
HEAP[heapoverflow.exe]: Heap block at 005E0588 modified at 005E05A0 past requested size of
10
(13fc.19c8): Break instruction exception - code 80000003 (first chance)
eax=005e0588 ebx=005e05a0 ecx=778e0535 edx=0012fafd esi=005e0588 edi=00000010
eip=77985204 esp=0012fd44 ebp=0012fd44 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!RtlpBreakPointHeap+0x23:
77985204 cc              int     3
0:000> kb
ChildEBP RetAddr  Args to Child
0012fd44 7796da8f 005e0588 005e0000 005e0588 ntdll!RtlpBreakPointHeap+0x23
0012fd5c 7794fb7e 00000000 005e0588 005e0588 ntdll!RtlpCheckBusyBlockTail+0x171
0012fd7c 7798625f 005e0000 005e0588 7790f81a ntdll!RtlpValidateHeapEntry+0x116
0012fdc4 77947a12 005e0000 50000067 005e0590 ntdll!RtlDebugFreeHeap+0x9a
0012feb8 77912bf0 005e0588 005e0590 0012ff49 ntdll!RtlpFreeHeap+0x5d
0012fed8 502b9ed9 005e0000 00000000 005e0590 ntdll!RtlFreeHeap+0x142
0012fef0 7718f14c 005e0000 00000000 005e0590
AcXtrnal!NS_FaultTolerantHeap::APIHook_RtlFreeHeap+0x61
```