看雪软件安全精选：二进制各种漏洞原理实战分析总结

转载

鸿渐之翼 于 2021-08-23 16:33:40 发布 168 收藏

分类专栏： 漏洞挖掘 PWN 文章标签： 软件安全 二进制安全 看雪安全

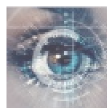
原文链接：https://bbs.pediy.com/thread-252569.htm

版权

漏洞挖掘 同时被 2 个专栏收录

32 篇文章 3 订阅

订阅专栏

 PWN

18 篇文章 1 订阅

订阅专栏

看雪安全二进制漏洞专区0xbird原创

本部分将对常见的二进制漏洞做系统分析，方便在漏洞挖掘过程中定位识别是什么类型漏洞，工欲善其事，必先利其器。

0x01栈溢出漏洞原理

栈溢出漏洞属于缓冲区漏洞的一种，实例如下：

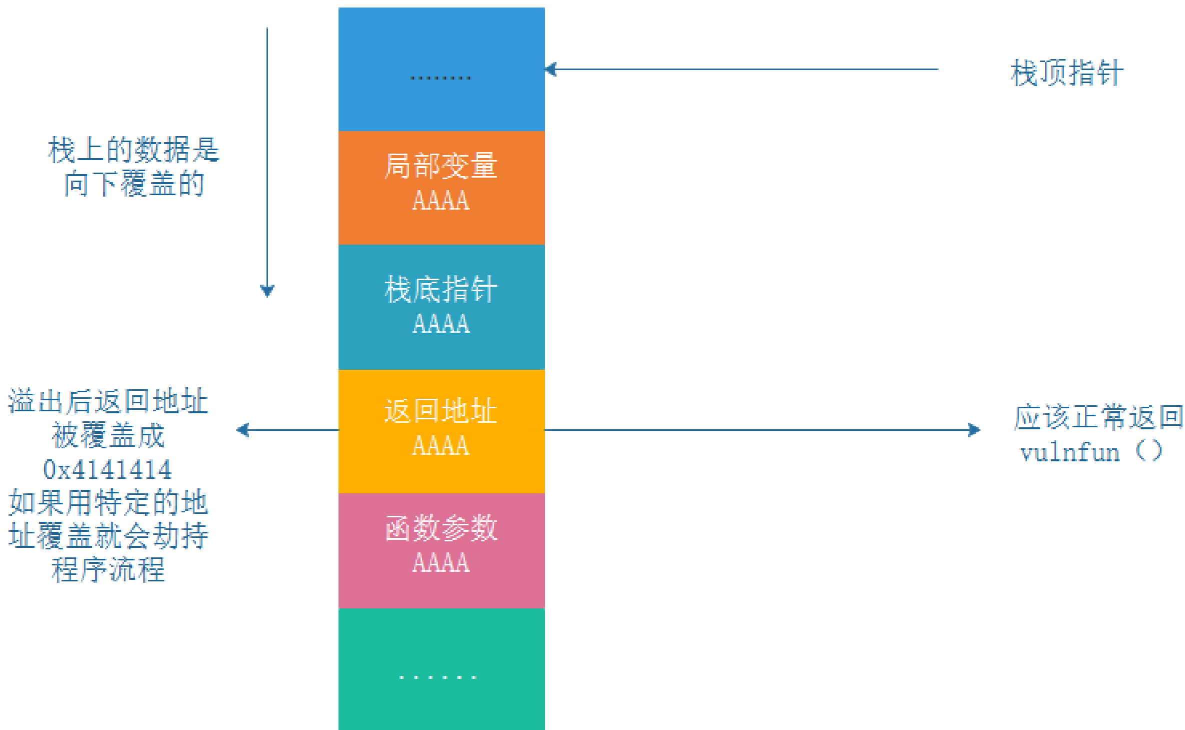
```
#include <stdio.h>
#include <string.h>

int main()
{
    char *str = "AAAAAAAAAAAAAAAAAAAAAAAA";
    vulnfun(str);
    return;
}

int vulnfun(char *str)
{
    char stack[10];
    strcpy(stack,str);        // 这里造成溢出!
}
```

编译后使用windbg运行

```
0:000> g
ModLoad: 00000000`00430000 00000000`004dc000   WOW64_IMAGE_SECTION
ModLoad: 00000000`75320000 00000000`75400000   WOW64_IMAGE_SECTION
ModLoad: 00000000`00430000 00000000`004dc000   NOT_AN_IMAGE
ModLoad: 00000000`00510000 00000000`00675000   NOT_AN_IMAGE
ModLoad: 00000000`54380000 00000000`5438a000   C:\Windows\System32\wow64cpu.dll
ModLoad: 00000000`75320000 00000000`75400000   C:\Windows\SysWOW64\KERNEL32.DLL
ModLoad: 00000000`74ec0000 00000000`75061000   C:\Windows\SysWOW64\KERNELBASE.dll
(296c.4c2c): WOW64 breakpoint - code 4000001f (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll_77a70000!LdrpDoDebuggerBreak+0x2b:
77b174bc cc              int     3
0:000:x86> g
(296c.4c2c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for stackoverflow1.exe
41414141 ??              ???
```

https://blog.csdn.net/qq_43332010

直接运行到了地址0x41414141，这个就是字符串AAAA，就是变量str里面的字符串通过strcpy拷贝到栈空间时，没有对字符串长度做限制，导致了栈溢出，最后覆盖到了返回地址，造成程序崩溃。
溢出后的栈空间布局如下：



栈溢出原理图

0x02 堆溢出漏洞原理

使用以下代码演示堆溢出漏洞

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

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

    hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x1000, 0xffff);
    getchar();    // 用于暂停程序，便于调试器加载

    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;
}
```

由于调试堆和常态堆的结构不同，在演示代码中加入getchar函数，用于暂停进程，方便运行heapoverflowexe后用调试器附加进程。debug版本和Release版本实际运行的进程中各个内存结构和分配过程也不同，因此测试的时候应该编译成release版本。
运行程序，使用windbg附加调试（一定要附加调试），g运行后程序崩溃

```
0:000> g
(2acc.2ac4): Access violation - code c0000005 (!!! second chance !!!)
eax=008104a0 ebx=00810498 ecx=41414141 edx=00810260 esi=008104b8 edi=00810000
eip=7775919d esp=0019fdb0 ebp=0019fea8 iopl=0         nv up ei ng nz na po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010283
ntdll!RtlpFreeHeap+0x5bd:
7775919d 8b11            mov     edx,dword ptr [ecx]  ds:002b:41414141=????????
0:000> kb
 # ChildEBP RetAddr  Args to Child
00 0019fea8 77758b98 00810498 008104a0 008104c1 ntdll!RtlpFreeHeap+0x5bd
*** WARNING: Unable to verify checksum for F:\vulns\Release\heapoverflow.exe
01 0019fefc 00401094 00810000 00000000 008104a0 ntdll!RtlFreeHeap+0x758
WARNING: Stack unwind information not available. Following frames may be wrong.
02 0019ff40 00401327 00000001 00720ea8 00720ee0 heapoverflow+0x1094
03 0019ff80 75c262c4 002c0000 75c262a0 cf955f3b heapoverflow+0x1327
04 0019ff94 77770fa9 002c0000 dad453c0 00000000 KERNEL32!BaseThreadInitThunk+0x24
05 0019ffdc 77770f74 ffffffff 77792eed 00000000 ntdll!__RtlUserThreadStart+0x2f
06 0019ffec 00000000 0040b000 002c0000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

https://blog.csdn.net/qq_43332010

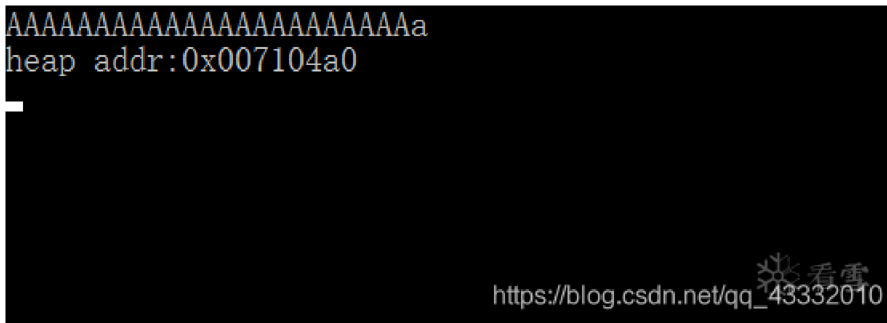
上面的ecx已经被AAAA字符串覆盖掉了，最后在引用该地址的时候导致崩溃，通过前面的栈回溯定位到了main函数入口，找到复制字符串的函数下断点

```
0:000> bp 00401084
*** WARNING: Unable to verify checksum for F:\vulns\Release\heapoverflow.exe
0:000> bl
     0 e Disable Clear  00401084     0001 (0001)  0:**** heapoverflow+0x1084
0:000> g
Breakpoint 0 hit
eax=00000021 ebx=007104a0 ecx=00000008 edx=0019fa54 esi=0019ff20 edi=007104a0
eip=00401084 esp=0019ff08 ebp=00710000 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
heapoverflow+0x1084:
00401084 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> dd edi
007104a0  baadf00d baadf00d baadf00d baadf00d
007104b0  abababab abababab 00000000 00000000
```

```
007104c0  ac166298 0000cfb9 007100c0 007100c0
007104d0  feeefeee feeefeee feeefeee feeefeee
007104e0  feeefeee feeefeee feeefeee feeefeee
007104f0  feeefeee feeefeee feeefeee feeefeee
00710500  feeefeee feeefeee feeefeee feeefeee
00710510  feeefeee feeefeee feeefeee feeefeee
0:000> dd esi
0019ff20  41414141 41414141 41414141 41414141
0019ff30  41414141 41414141 41414141 41414141
0019ff40  00407000 00401327 00000001 00800f48
0019ff50  00800f90 0040b000 0040b000 0028d000
0019ff60  75c10000 0040b000 0019ff54 0019ff94
0019ff70  0019ffcc 00402c50 004060b8 00000000
0019ff80  0019ff94 75c262c4 0028d000 75c262a0
0019ff90  e0b33895 0019ffdc 77770fa9 0028d000
```

https://blog.csdn.net/qq_43332010

此时堆块已经分配完毕，对应的分配地址位于0x007104a0，0x007104a0是堆块数据的起始地址，并非堆头信息的起始地址，对于已经分配的堆块，开头有8字节的HEAP_ENTRY结构，因此heap的HEAP_ENTRY结构位于0x007104a0-8=0x710498。



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大小空间

```
0:000> !heap -p -a 0x710498
    address 00710498 found in
    _HEAP @ 710000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        00710498 0005 0000  [00]   007104a0    00010 - (busy)
```

在windbg中，使用！heap查看HeapCreate创建的整个堆块信息，可以发现堆块heap后面还有一个空闲堆块0x007104c0：

```
 Heap Address      NT/Segment Heap

             560000              NT Heap
             800000              NT Heap
             710000              NT Heap
0:000> !heap -a 710000
HEAPEXT: Unable to get address of ntdll!RtlpHeapInvalidBadAddress.
Index   Address  Name      Debugging options enabled
  3:   00710000
    Segment at 00710000 to 00720000 (00001000 bytes committed)
    Flags:                40001064
    ForceFlags:           40000064
    Granularity:          8 bytes
    Segment Reserve:      00100000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size:      00000164
    Max. Allocation Size: 7ffdefff
    Lock Variable at:     00710248
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   0071009c
    Uncommitted ranges:   0071008c
            00711000: 0000f000  (61440 bytes)
    FreeList[ 00 ] at 007100c0: 007104c8 . 007104c8
        007104c0: 00028 . 00b20 [104] - free

    Segment00 at 00710000:
        Flags:           00000000
        Base:            00710000
        First Entry:     00710498
        Last Entry:      00720000
        Total Pages:     00000010
        Total UnCommit:  0000000f
        Largest UnCommit:00000000
        UnCommitted Ranges: (1)

    Heap entries for Segment00 in Heap 00710000
         address: psize . size  flags   state (requested size)
        00710000: 00000 . 00498 [101] - busy (497)
        00710498: 00498 . 00028 [107] - busy (10), tail fill //heap的占用堆块
        007104c0: 00028 . 00b20 [104] free fill  //空闲堆块
        00710fe0: 00b20 . 00020 [111] - busy (1d)
        00711000:      0000f000      - uncommitted bytes.
```

在复制字符串的时候，原本只有0x10大小的堆块，填充过多的字符串的时候就会覆盖到下方的空闲堆块007104c0，在复制前007104c0空闲堆块的HEAP_FREE_ENTRY结构数据如下：

0:000> dt _HEAP_FREE_ENTRY 0x007104c0

```
ntdll!_HEAP_FREE_ENTRY
   +0x000 HeapEntry        : _HEAP_ENTRY
   +0x000 UnpackedEntry    : _HEAP_UNPACKED_ENTRY
   +0x000 Size             : 0x6298
   +0x002 Flags            : 0x16 ''
   +0x003 SmallTagIndex    : 0xac ''
   +0x000 SubSegmentCode   : 0xac166298
   +0x004 PreviousSize     : 0xcfb9
   +0x006 SegmentOffset    : 0 ''
   +0x006 LFHFlags         : 0 ''
   +0x007 UnusedBytes      : 0 ''
   +0x000 ExtendedEntry    : _HEAP_EXTENDED_ENTRY
   +0x000 FunctionIndex    : 0x6298
   +0x002 ContextValue     : 0xac16
   +0x000 InterceptorValue : 0xac166298
   +0x004 UnusedBytesLength : 0xcfb9
   +0x006 EntryOffset      : 0 ''
   +0x007 ExtendedBlockSignature : 0 ''
   +0x000 Code1            : 0xac166298
   +0x004 Code2            : 0xcfb9
   +0x006 Code3            : 0 ''
   +0x007 Code4            : 0 ''
   +0x004 Code234          : 0xcfb9
   +0x000 AgregateCode     : 0x0000cfb9`ac166298
   +0x008 FreeList         : _LIST_ENTRY [ 0x7100c0 - 0x7100c0 ]
```

```
 dt _LIST_ENTRY 0x007104c0+8
ntdll!_LIST_ENTRY
 [ 0x7100c0 - 0x7100c0 ]
   +0x000 Flink            : 0x007100c0 _LIST_ENTRY [ 0x7104c8 - 0x7104c8 ]
   +0x004 Blink            : 0x007100c0 _LIST_ENTRY [ 0x7104c8 - 0x7104c8 ]
```

覆盖后0x007104c0空闲块的HEAP_FREE_ENTRY结构数据如下：

```
g
(2c08.234): Access violation - code c0000005 (!!! second chance !!!)
eax=007e04a0 ebx=007e0498 ecx=41414141 edx=007e0260 esi=007e04b8 edi=007e0000
eip=7775919d esp=0019fdb0 ebp=0019fea8 iopl=0         nv up ei ng nz na po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010283
ntdll!RtlpFreeHeap+0x5bd:
7775919d 8b11            mov     edx,dword ptr [ecx]  ds:002b:41414141=????????
0:000> dt _HEAP_FREE_ENTRY 0x007104c0
ntdll!_HEAP_FREE_ENTRY
   +0x000 HeapEntry        : _HEAP_ENTRY
   +0x000 UnpackedEntry    : _HEAP_UNPACKED_ENTRY
   +0x000 Size             : ??
   +0x002 Flags            : ??
   +0x003 SmallTagIndex    : ??
   +0x000 SubSegmentCode   : ??
   +0x004 PreviousSize     : ??
   +0x006 SegmentOffset    : ??
   +0x006 LFHFlags         : ??
   +0x007 UnusedBytes      : ??
   +0x000 ExtendedEntry    : _HEAP_EXTENDED_ENTRY
   +0x000 FunctionIndex    : ??
   +0x002 ContextValue     : ??
   +0x000 InterceptorValue : ??
   +0x004 UnusedBytesLength : ??
   +0x006 EntryOffset      : ??
   +0x007 ExtendedBlockSignature : ??
   +0x000 Code1            : ??
   +0x004 Code2            : ??
   +0x006 Code3            : ??
   +0x007 Code4            : ??
   +0x004 Code234          : ??
   +0x000 AgregateCode     : ??
   +0x008 FreeList         : _LIST_ENTRY
Memory read error 007104c0
```

整个空闲堆头信息都被覆盖了，包括最后的空闲链表中的前后向指针都被成了0x41414141，后面调用HeapFree释放堆块的时候，就会将buf2和后面的空闲堆块0x007104c0合并，修改两个空闲堆块的前后向指针就会引用0x41414141，最后造成崩溃。

如果把上面释放堆块的操作换成分配堆块HeapAlloc（），也会导致崩溃，因为在分配堆块的时候会去遍历空闲链表指针，会造成地址引用异常，当内存中已经分配多个堆块的时候，可能覆盖到的就是已经分配到的堆块，此时可能就是覆盖HEAP_ENTRY结构，而不是HEAP_FREE_ENTRY结构。

https://blog.csdn.net/qq_43332010

堆溢出原理图

0x03 堆调试技巧

微软提供了一些调试选项用于辅助堆调试，可以通过windbg提供的gflag.exe或者！gflag命令来设置：
htc：堆尾检查，是否发生溢出
hfc：堆释放检查
hpc：堆参数检查
hpa：启用页堆
htg：堆标志
ust：用户态栈回溯
对heapoverflow.exe添加堆尾检查和页堆，去掉堆标志：
gflags.exe -i F:\vulns\Release\heapoverflow +htc +hpa +htg
堆尾检查
主要是在每个堆块的尾部，用户数据之后添加8字节，通常是连续的2个0xabababab，该数据段被破坏就可能发生了溢出。

对heapoverflow.exe开启hpc和htc，用windbg加载对heapoverflow程序，附加进程无法在堆尾添加额外标志，使用以下命令开启堆尾检查和堆参数检查：

```
0:000> !gflag +htc +hpc
Current NtGlobalFlag contents: 0x00000070
    htc - Enable heap tail checking
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
```

```
0:000:x86> g
HEAP[heapoverflow.exe]: Heap block at 001E0498 modified at 001E04B0 past requested size of 10
(13d0.3c9c): WOW64 breakpoint - code 4000001f (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll_77710000!RtlpBreakPointHeap+0x13:
777f07c7 cc              int     3
```

执行命令g后，按下回车键程序会断下来

```
0:000:x86> kb
 # ChildEBP RetAddr  Args to Child
00 0019fd18 777dd85b 00000000 001e0000 001e0498 ntdll_77710000!RtlpBreakPointHeap+0x13
01 0019fd30 77793e9c 001e0000 00000000 77786780 ntdll_77710000!RtlpCheckBusyBlockTail+0x1a2
02 0019fd4c 777ef9f1 7772e4dc 9ceeef49 001e0000 ntdll_77710000!RtlpValidateHeapEntry+0x633d9
03 0019fda4 7775991d 001e04a0 9ceeec45 001e0498 ntdll_77710000!RtlDebugFreeHeap+0xbf
04 0019fea8 77758b98 001e0498 001e04a0 001e04c1 ntdll_77710000!RtlpFreeHeap+0xd3d
*** WARNING: Unable to verify checksum for F:\vulns\Release\heapoverflow.exe
05 0019fefc 00401094 001e0000 00000000 001e04a0 ntdll_77710000!RtlFreeHeap+0x758
WARNING: Stack unwind information not available. Following frames may be wrong.
06 001e0000 01000709 ffeeffee 00000000 001e00a4 heapoverflow+0x1094
07 001e0004 ffeeffee 00000000 001e00a4 001e00a4 0x1000709
08 001e0008 00000000 001e00a4 001e00a4 001e0000 0xffeeffee
```

```
HEAP[heapoverflow.exe]: Heap block at 001E0498 modified at 001E04B0 past requested size of 10
```

上面一句调试输出信息的意思是，在大小为0x10的堆块0x001E0498的0x001E04B0覆盖破坏了，0x10大小的空间加上堆头的8字节一共0x18字节，0x001E04B0-0x001E0498=0x18，也就是说0x001E04B0位于堆块数据的最后一个字节上，基于上面的信息，可以分析出程序主要是因为向0x10的堆块中复制过多数据导致的堆溢出。

页堆

在调试漏洞的时候，经常需要定位导致漏洞的代码和函数，比如导致堆溢出的字节复制指令rep movsz等，前面的堆尾检查方式主要是堆被破坏的场景，不利于定位导致漏洞的代码。为此。引入了页堆的概念，开启后，会在堆块中增加不可访问的栅栏页，溢出覆盖到栅栏页就会触发异常。

开启页堆：

```
gflags.exe -i F:\vulns\Release\heapoverflow  +hpa
```

```
C:\Program Files\Windows Kits\8.1\Debuggers\x86>gflags /i C:\Users\15pb-win7\Desktop\heapoverflow\Release\heapoverflow.exe +hpa
Current Registry Settings for heapoverflow.exe executable are: 02000000
    hpa - Enable page heap

C:\Program Files\Windows Kits\8.1\Debuggers\x86>_
```

用windbg加载heapoverflow，运行！gflag命令开启了页堆，然后g运行后在cmd按下回车键断下

```
0:000> g
(46c.b74): Access violation - code c0000005 (!!! second chance !!!)
eax=00000021 ebx=01795ff0 ecx=00000004 edx=77d364f4 esi=0012ff38 edi=01796000
eip=00401084 esp=0012ff10 ebp=01790000 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
image00400000+0x1084:
00401084 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> dd esi
0012ff38  41414141 41414141 41414141 41414141
0012ff48  00407000 00401327 00000001 01699fb0
0012ff58  0169bf70 00000000 00000000 7ffdd000
0012ff68  c0000005 00000000 0012ff5c 0012fb1c
0012ff78  0012ffc4 00402c50 004060b8 00000000
0012ff88  0012ff94 76281174 7ffdd000 0012ffd4
0012ff98  77d4b3f5 7ffdd000 77cb48a4 00000000
0012ffa8  00000000 7ffdd000 c0000005 76292b35
0:000> dc edi
01796000  ???????? ???????? ???????? ????????  ????????????????
01796010  ???????? ???????? ???????? ????????  ????????????????
01796020  ???????? ???????? ???????? ????????  ????????????????
01796030  ???????? ???????? ???????? ????????  ????????????????
01796040  ???????? ???????? ???????? ????????  ????????????????
01796050  ???????? ???????? ???????? ????????  ????????????????
01796060  ???????? ???????? ???????? ????????  ????????????????
01796070  ???????? ???????? ???????? ????????  ????????????????
```

可以发现程序在复制A字符串的时候触发了异常，程序复制到0x11字节的时候被断下，此时异常还未破坏到堆块，直接定位导致溢出的复制指令rep movs

```
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff48 00401327 00000001 01699fb0 0169bf70 image00400000+0x1084
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Windows\system32\kernel32.dll -
0012ff88 76281174 7ffdd000 0012ffd4 77d4b3f5 image00400000+0x1327
0012ff94 77d4b3f5 7ffdd000 77cb48a4 00000000 kernel32!BaseThreadInitThunk+0x12
0012ffd4 77d4b3c8 0040b000 7ffdd000 00000000 ntdll!RtlInitializeExceptionChain+0x63
0012ffec 00000000 0040b000 7ffdd000 00000000 ntdll!RtlInitializeExceptionChain+0x36
0:000> ub image00400000+0x1327
image00400000+0x1301:
00401301 e847120000      call    image00400000+0x254d (0040254d)
00401306 e8af0e0000      call    image00400000+0x21ba (004021ba)
0040130b a150994000      mov     eax,dword ptr [image00400000+0x9950 (00409950)]
00401310 a354994000      mov     dword ptr [image00400000+0x9954 (00409954)],eax
00401315 50              push    eax
00401316 ff3548994000    push    dword ptr [image00400000+0x9948 (00409948)]
0040131c ff3544994000    push    dword ptr [image00400000+0x9944 (00409944)]
00401322 e8d9fcffff      call    image00400000+0x1000 (00401000)
```

根据栈回溯，调用rep movs的上一层函数位于image00400000+0x1084的上一条指令，也就是00401322，此处调用了00401000函数，很容易发现这是主入口函数：

```
0:000> uf 00401000
image00400000+0x1000:
00401000 83ec24          sub     esp,24h
00401003 b908000000      mov     ecx,8
00401008 53              push    ebx
00401009 55              push    ebp
0040100a 56              push    esi
```

```
0040100b 57              push    edi
0040100c be44704000      mov     esi,offset image00400000+0x7044 (00407044)
00401011 8d7c2410        lea     edi,[esp+10h]
00401015 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
00401017 68ffff0000      push    0FFFFh
0040101c 6800100000      push    1000h    //堆块大小压入
00401021 6a04            push    4
00401023 a4              movs    byte ptr es:[edi],byte ptr [esi]
00401024 ff150c604000    call    dword ptr [image00400000+0x600c (0040600c)]//调用HeapCreate创建堆块
0040102a 8be8            mov     ebp,eax
0040102c a16c704000      mov     eax,dword ptr [image00400000+0x706c (0040706c)]
00401031 48              dec     eax
00401032 a36c704000      mov     dword ptr [image00400000+0x706c (0040706c)],eax
00401037 7808            js      image00400000+0x1041 (00401041)

image00400000+0x1039:
00401039 ff0568704000    inc     dword ptr [image00400000+0x7068 (00407068)]
0040103f eb0d            jmp     image00400000+0x104e (0040104e)

image00400000+0x1041:
00401041 6868704000      push    offset image00400000+0x7068 (00407068)
00401046 e896000000      call    image00400000+0x10e1 (004010e1)
0040104b 83c404          add     esp,4

image00400000+0x104e:
0040104e 6a10            push    10h
00401050 6a00            push    0
00401052 55              push    ebp
00401053 ff1508604000    call    dword ptr [image00400000+0x6008 (00406008)]  //调用HeapAlloc分配0x10的堆块
00401059 8bd8            mov     ebx,eax   //分配的堆块地址
0040105b 53              push    ebx
0040105c 6830704000      push    offset image00400000+0x7030 (00407030)
00401061 e84a000000      call    image00400000+0x10b0 (004010b0)
00401066 8d7c2418        lea     edi,[esp+18h]
0040106a 83c9ff          or      ecx,0FFFFFFFFh
0040106d 33c0            xor     eax,eax
0040106f 83c408          add     esp,8
00401072 f2ae            repne scas byte ptr es:[edi]
00401074 f7d1            not     ecx   //获取str长度
00401076 2bf9            sub     edi,ecx
00401078 53              push    ebx
00401079 8bc1            mov     eax,ecx
0040107b 8bf7            mov     esi,edi  //str = 0x20
0040107d 8bfb            mov     edi,ebx  //分配的堆块只有0x10
0040107f 6a00            push    0
00401081 c1e902          shr     ecx,2
00401084 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
00401086 8bc8            mov     ecx,eax
00401088 55              push    ebp
00401089 83e103          and     ecx,3
0040108c f3a4            rep movs byte ptr es:[edi],byte ptr [esi]  //0x20 < 0x10 循环复制导致溢出
0040108e ff1504604000    call    dword ptr [image00400000+0x6004 (00406004)]
00401094 55              push    ebp
00401095 ff1500604000    call    dword ptr [image00400000+0x6000 (00406000)]
0040109b 5f              pop     edi
0040109c 5e              pop     esi
0040109d 5d              pop     ebp
0040109e 33c0            xor     eax,eax
004010a0 5b              pop     ebx
004010a1 83c424          add     esp,24h
```

```
004010a1 83c424          add     esp,24h
004010a4 c3              ret
```

0x04整数溢出漏洞原理

整数分为有符号和无符号两类，有符号数以最高位作为符号位，正整数最高位为1，负整数最高位为0，不同类型的整数在内存中有不同的取值范围，unsigned int = 4字节，int = 4字节，当存储的数值超过该类型整数的最大值就会发生溢出。

在一些有符号和无符号转换的过程中最有可能发生整数溢出漏洞。

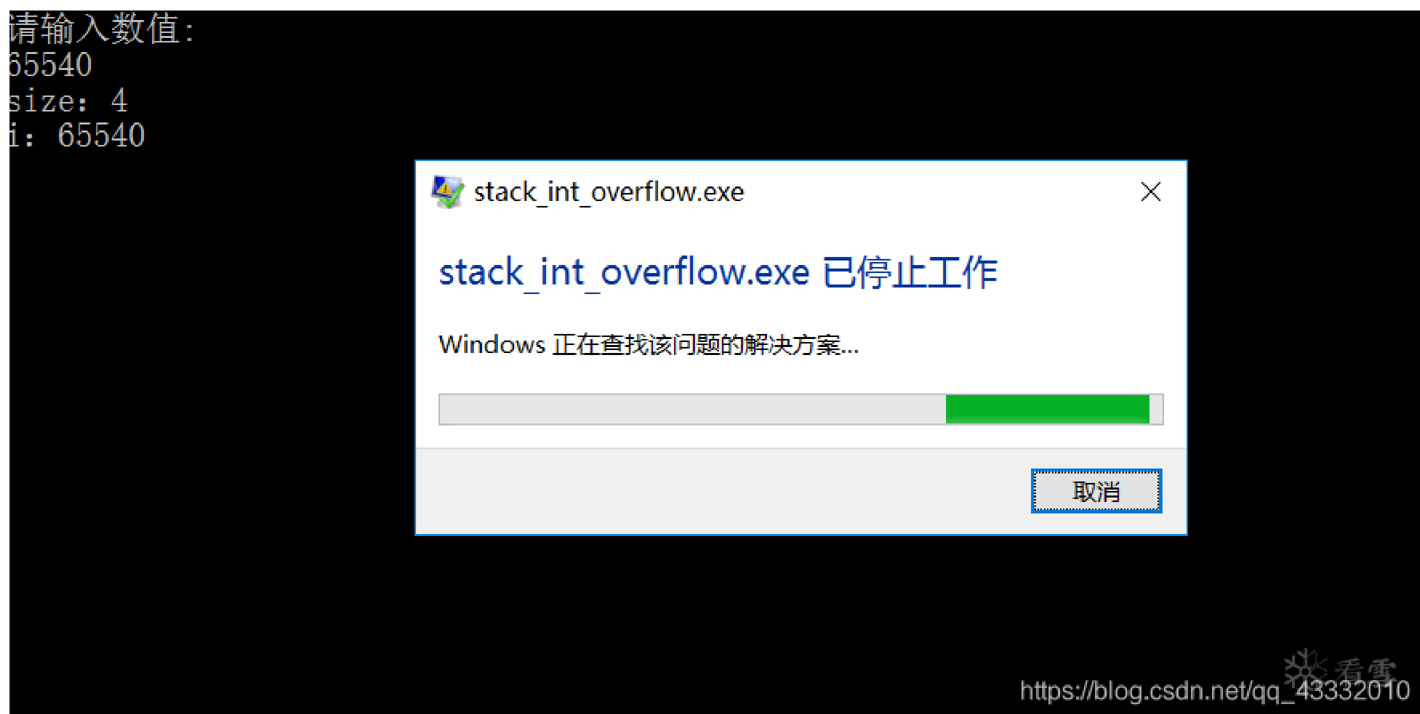
数据类型	占用字节数	范围
字符类型 char	1	-128~127
无符号字符类型 unsigned char	1	0~255
布尔类型 bool	1	0~1
短整型 short	2	-32768~32767

数据类型	占用字节数	范围
整型 int	4	-2147483648 ~2147483647
无符号整型 unsigned int	4	0~4294967295
长整型 long	4	-2147483648 ~2147483647
无符号长整型 unsigned long	4	0~4294967295

基于栈的整数溢出
基于栈的整数溢出的例子：

```
#include "stdio.h"
#include "string.h"

int main(int argc, char *argv){

    int i;
    char buf[8];     // 栈缓冲区
    unsigned short int size;    // 无符号短整数取值范围: 0 ~ 65535
    char overflow[65550];

    memset(overflow,65,sizeof(overflow));    // 填充为"A"字符

    printf("请输入数值:\n");
    scanf("%d",&i);        //输入65540

    size = i;
    printf("size: %d\n",size);    // 输出系统识别出来的size数值  4
    printf("i: %d\n",i);    // 输出系统识别出来的i数据   65540

    if (size > 8)  //边界检查
        return -1;
    memcpy(buf,overflow,i);        // 栈溢出

    return 0;

}
```

代码中size变量是无符号短整型，取值范围是0~65535，输入的值大于65535就会发生溢出，最后得到size为4，这样会通过边界检查，但是用memcpy复制数据的时候，使用的是int类型的参数i，这个值是输入的65540，就会发生栈溢出：



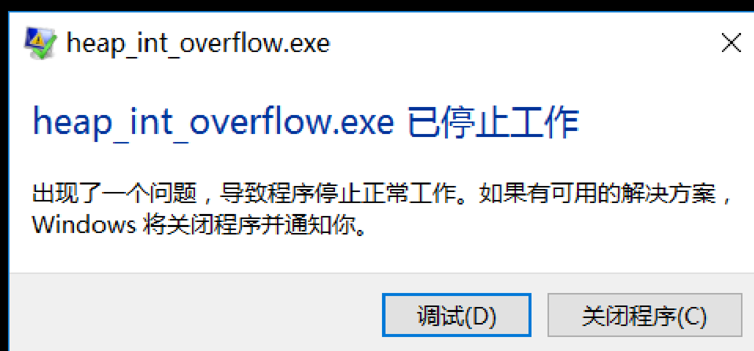
基于堆的整数溢出

基于堆的整数溢出的例子：

```c
#include "stdio.h"
#include "windows.h"


int main(int argc, char * argv)
{
    int* heap;
    unsigned short int size;  // 无符号短整数取值范围：0 ~ 65535
    char *pheap1, *pheap2;
    HANDLE hHeap;

    printf("输入size数值：\n");
    scanf("%d",&size);

    hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x100, 0xfff);       //创建一个堆块

    if (size <= 0x50)
    {
        size -= 5;             //输入2，size=-3=65533，
        printf("size：%d\n",size);
        pheap1 = HeapAlloc(hHeap, 0, size);        //    pheap1会分配过大的堆块，导致溢出！
        pheap2 = HeapAlloc(hHeap, 0, 0x50);
    }

    HeapFree(hHeap, 0, pheap1);
    HeapFree(hHeap, 0, pheap2);

    return 0;
}
```

代码中的size是unsigned short int类型，当输入小于5，size减去5会得到负数，但由于unsigned short int取值范围的限制无法识别负数，得到正数65533，最后分配得到过大的堆块，溢出覆盖了后面的堆管理结构：

```
输入size数值：
2
size：65533
```



0x05 格式化字符串漏洞原理

格式化漏洞产生的原因主要是对用户输入的内容没有做过滤，有些输入数据都是作为参数传递给某些执行格式化操作的函数的，比如：printf，fprintf，vprintf，sprintf。

恶意用户可以使用%s和%x等格式符，从堆栈和其他内存位置输出数据，也可以使用格式符%n向任意地址写入数据，配合printf（）函数就可以向任意地址写入被格式化的字节数，可能导致任意代码执行，或者读取敏感数据。

以下面的代码为例讲解格式化字符串漏洞原理：

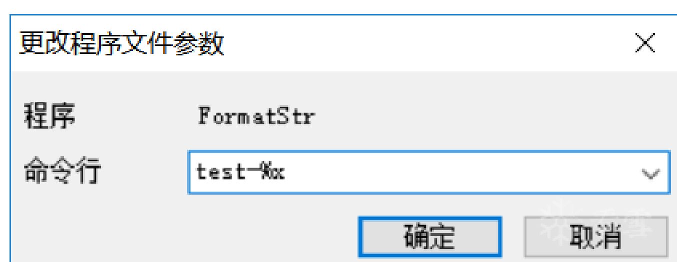
```c
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[])
{
        char buff[1024];    // 设置栈空间

        strncpy(buff,argv[1],sizeof(buff)-1);
        printf(buff); //触发漏洞

        return 0;
}
```

可以发现当输入数据包含%s和%x格式符的时候，会意外输出其他数据：

```
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test
test
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test-%x
test-19fec4
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test-%s
test-test-%s
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test-%x-%x-%x-%x
test-19fec4-7e0f2d-7f-74736574
F:\vulns\FormatStr\Release>_
```

https://blog.csdn.net/qq_43332010

用ollydbg附加调试程序，执行前需要先设置命令行参数，调试-参数-命令行：test-%x



在运行程序后，传递给printf的参数只有test-%x，但是他把输入参数test-%x之后的另一个栈上数据当做参数传给了printf函数，因为printf基本类型是：

```
printf（“格式化控制符”，变量列表）；
```

传递给printf的参数只有一个，但是程序默认将栈上的下一个数据作为参数传递给了printf函数，刚好下一个数据是strcpy（）函数的目标地址，就是buff变量，buff刚好指向test-%x的地址0x0019fec4，所以程序会输出0x0019fec4，如果后面再加一个%x就会将src参数的值也输出了，这样就可以遍历整个栈上数据了。

```
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test-%x
test-19fec4
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test-%s
test-test-%s
F:\vulns\FormatStr\Release>F:\vulns\FormatStr\Release\FormatStr.exe test-%x-%x-%x-%x
test-19fec4-7e0f2d-7f-74736574
F:\vulns\FormatStr\Release>
```

除了利用%x读取栈上数据，还可以用%n写入数据修改返回地址来实现漏洞利用。

0x06 双重释放漏洞原理

Double Free漏洞是由于对同一块内存进行二次释放导致的，利用漏洞可以执行任意代码，编译成release示例代码如下：

```
#include <stdio.h>
#include "windows.h"

int main (int argc, char *argv[])
{
    void *p1,*p2,*p3;

    p1 = malloc(100);
    printf("Alloc p1: %p\n",p1);
    p2 = malloc(100);
    printf("Alloc p2: %p\n",p2);
    p3 = malloc(100);
    printf("Alloc p3: %p\n",p3);

    printf("Free p1\n");
    free(p1);
    printf("Free p3\n");
    free(p3);
    printf("Free p2\n");
    free(p2);

    printf("Double Free p2\n"); //二次释放
    free(p2);

    return 0;
}
```

在二次释放p2的时候就会发生程序崩溃，但是并不是每次出现Double Free都会发生崩溃，要有堆块合并的动作发生才会发生崩溃

```c
#include <stdio.h>
#include "windows.h"

int main (int argc, char *argv[])
{
    void *p1,*p2,*p3;

    p1 = malloc(100);
    printf("Alloc p1: %p\n",p1);
    p2 = malloc(100);
    printf("Alloc p2: %p\n",p2);
    p3 = malloc(100);
    printf("Alloc p3: %p\n",p3);

    printf("Free p2\n");
    free(p2);
    printf("Double Free p2\n");
    free(p2);
    printf("Free p1\n");
    free(p1);
    printf("Free p3\n");
    free(p3);

    return 0;
}
```

```
printf("Free p2\n");
free(p2);
printf("Double Free p2\n");
free(p2);
printf("Free p1\n");
free(p1);
printf("Free p3\n");
free(p3);
```

```
se\double_free.exe"
Alloc p1: 00030D98
Alloc p2: 00030E10
Alloc p3: 00030E88
Free p2
Double Free p2
Free p1
Free p3
```

https://blog.csdn.net/qq_43332010

释放p2后，p1，p2，p3进行合并

对p2进行第二次释放，找不到地址，导致访问异常

https://blog.csdn.net/qq_43332010

双重释放原理图

在释放过程中，邻近的已经释放的堆块存在合并操作，这会改变原有堆头信息，之后再对其地址引用释放就会发生访问异常。

0x07释放后重引用漏洞原理

通过以下代码理解UAF漏洞原理：

```c
#include <stdio.h>

#define size 32

int main(int argc, char **argv) {

    char *buf1;
    char *buf2;

    buf1 = (char *) malloc(size);
    printf("buf1: 0x%p\n", buf1);
    free(buf1);

    // 分配 buf2 去“占坑”buf1 的内存位置
    buf2 = (char *) malloc(size);
    printf("buf2: 0x%p\n\n", buf2);

    // 对buf2进行内存清零
    memset(buf2, 0, size);
    printf("buf2: %d\n", *buf2);

    // 重引用已释放的buf1指针，但却导致buf2值被篡改
    printf("==== Use After Free ===\n");
    strncpy(buf1, "hack", 5);
    printf("buf2: %s\n\n", buf2);

    free(buf2);
}
```

```
F:\vulns\uaf\Debug>F:\vulns\uaf\Debug\uaf.exe
buf1: 0x00570E18
buf2: 0x00570E18

buf2: 0
==== Use After Free ===
buf2: hack
```

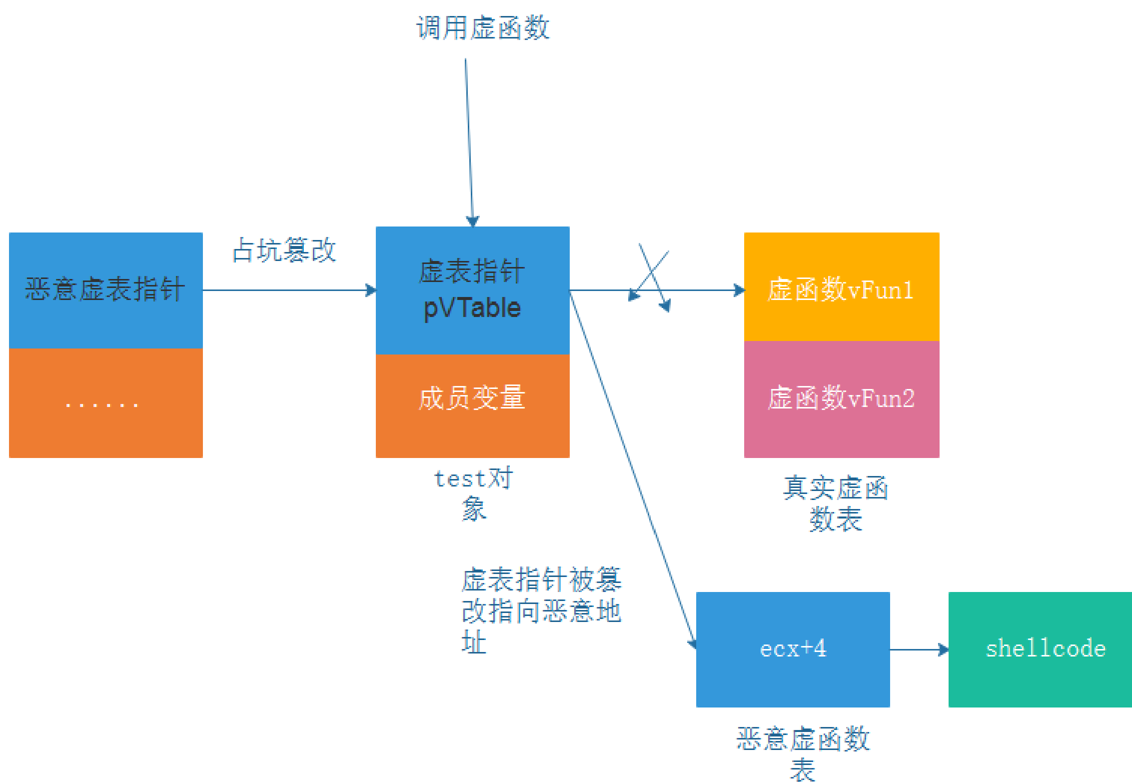
https://blog.csdn.net/qq_43332010

buf2 “占坑”了buf1 的内存位置，经过UAF后，buf2被成功篡改了

程序通过分配和buf1大小相同的堆块buf2实现占坑，似的buf2分配到已经释放的buf1内存位置，但由于buf1指针依然有效，并且指向的内存数据是不可预测的，可能被堆管理器回收，也可能被其他数据占用填充，buf1指针称为悬挂指针，借助悬挂指针buf1将内存赋值为hack，导致buf2也被篡改为hack。

如果原有的漏洞程序引用到悬挂指针指向的数据用于执行指令，就会导致任意代码执行。

在通常的浏览器UAF漏洞中，都是某个C++对象被释放后重引用，假设程序存在UAF的漏洞，有个悬挂指针指向test对象，要实现漏洞利用，通过占坑方式覆盖test对象的虚表指针，虚表指针指向虚函数存放地址，现在让其指向恶意构造的shellcode，当程序再次引用到test对象就会导致任意代码执行。



UAF漏洞利用原理图