# [内存泄漏分析及常用内存泄漏检测工具分享](http://blog.csdn.net/panda_bear/article/details/8009421)

## **一.内存泄漏的定义**

　　一般我们常说的内存泄漏是指堆内存的泄漏。堆内存是指程序从堆中分配的，大小任意的（内存块的大小可以在程序运行期决定），使用完后必须显示释放的内存。应用程序一般使用malloc，realloc，new等函数从堆中分配到一块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了。以下这段小程序演示了堆内存发生泄漏的情形：

## **二．内存泄漏的发生方式**

1.    常发性内存泄漏。发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

2.    偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

3.    一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块且仅有一块内存发生泄漏。

4.    隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

## **三．内存泄漏的表现**

Private Bytes和Virtual Bytes至少有一个是一条斜向上曲线，大多数泄漏是这种情况

如果Private Byte和Virtual Bytes一起上升，但是后者比前者上升得快或者比例超过3:1，说明不仅仅有内存泄漏，而且泄漏导致了内存碎片

## **四．常用的两种检查内存泄漏的方式：**

1.     调试器和CRT调试堆函数

用MFC开发的应用程序，在DEBUG版模式下编译后，都会自动加入内存泄漏的检测代码。在程序结束后，如果发生了内存泄漏，在Debug窗口中会显示出所有发生泄漏的内存块的信息，以下两行显示了一块被泄漏的内存块的信息：

E:\TestMemLeak\TestDlg.cpp(70) : {59} normal block at 0x00881710, 200 bytes long.

Data: <abcdefghijklmnop> 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70

　　第一行显示该内存块由TestDlg.cpp文件，第70行代码分配，地址在0x00881710，大小为200字节，{59}是指调用内存分配函数的Request Order，关于它的详细信息可以参见MSDN中\_CrtSetBreakAlloc()的帮助。第二行显示该内存块前16个字节的内容，尖括号内是以ASCII方式显示，接着的是以16进制方式显示。

　　一般大家都误以为这些内存泄漏的检测功能是由MFC提供的，其实不然。MFC只是封装和利用了MS C-Runtime Library的Debug Function。非MFC程序也可以利用MS C-Runtime Library的Debug Function加入内存泄漏的检测功能。MS C-Runtime Library在实现malloc/free，strdup等函数时已经内建了内存泄漏的检测功能。

　　注意观察一下由MFC Application Wizard生成的项目，在每一个cpp文件的头部都有这样一段宏定义：

|  |
| --- |
| #ifdef \_DEBUG  #define new DEBUG\_NEW  #undef THIS\_FILE  static char THIS\_FILE[] = \_\_FILE\_\_;  #endif |

　　有了这样的定义，在编译DEBUG版时，出现在这个cpp文件中的所有new都被替换成DEBUG\_NEW了。那么DEBUG\_NEW是什么呢？DEBUG\_NEW也是一个宏，以下摘自afx.h，1632行

|  |
| --- |
| #define DEBUG\_NEW new(THIS\_FILE, \_\_LINE\_\_) |

　　所以如果有这样一行代码：

|  |
| --- |
| char\* p = new char[200]; |

　　经过宏替换就变成了：

|  |
| --- |
| char\* p = new( THIS\_FILE, \_\_LINE\_\_)char[200]; |

　　根据C++的标准，对于以上的new的使用方法，编译器会去找这样定义的operator new：

|  |
| --- |
| void\* operator new(size\_t, LPCSTR, int) |

　　我们在afxmem.cpp 63行找到了一个这样的operator new 的实现

|  |
| --- |
| void\* AFX\_CDECL operator new(size\_t nSize, LPCSTR lpszFileName, int nLine)  {  　return ::operator new(nSize, \_NORMAL\_BLOCK, lpszFileName, nLine);  }  void\* \_\_cdecl operator new(size\_t nSize, int nType, LPCSTR lpszFileName, int nLine)  {  　…  　pResult = \_malloc\_dbg(nSize, nType, lpszFileName, nLine);  　if (pResult != NULL)  　　return pResult;  　…  } |

　　第二个operator new函数比较长，为了简单期间，我只摘录了部分。很显然最后的内存分配还是通过\_malloc\_dbg函数实现的，这个函数属于MS C-Runtime Library 的Debug Function。这个函数不但要求传入内存的大小，另外还有文件名和行号两个参数。文件名和行号就是用来记录此次分配是由哪一段代码造成的。如果这块内存在程序结束之前没有被释放，那么这些信息就会输出到Debug窗口里。

 　要在非MFC程序中打开内存泄漏的检测功能非常容易，你只要在程序的入口处加入以下几行代码：

|  |
| --- |
| int tmpFlag = \_CrtSetDbgFlag( \_CRTDBG\_REPORT\_FLAG );  tmpFlag |= \_CRTDBG\_LEAK\_CHECK\_DF;  \_CrtSetDbgFlag( tmpFlag ); |

　　这样，在程序结束的时候，也就是winmain，main或dllmain函数返回之后，如果还有内存块没有释放，它们的信息会被打印到Debug窗口里。

　　如果你试着创建了一个非MFC应用程序，而且在程序的入口处加入了以上代码，并且故意在程序中不释放某些内存块，你会在Debug窗口里看到以下的信息：

|  |
| --- |
| {47} normal block at 0x00C91C90, 200 bytes long.  Data: < > 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F |

　　内存泄漏的确检测到了，但是和上面MFC程序的例子相比，缺少了文件名和行号。对于一个比较大的程序，没有这些信息，解决问题将变得十分困难。

　　为了能够知道泄漏的内存块是在哪里分配的，你需要实现类似MFC的映射功能，把new，maolloc等函数映射到\_malloc\_dbg函数上。

实现方式详见：<http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html>

2.利用第三方工具检

第三方监测工具有很多，这里选取几款比较好用的推荐给大家

**Visual Leak Detecter**

应用环境：Windows + VC

编程语言：C/C++

使用方法：只需包含头文件vld.h，并添加提供的lib

结果输出：输出到VC的调试窗口中

设计思路： 注册\_CrtSetAllocHook钩子函数，使用VC自带的CRT Debug Heap

优缺点：可以获得内存泄露点的调用堆栈，可以得到内存泄露的完整数据

如何获取：<http://www.codeproject.com/Articles/9815/Visual-Leak-Detector-Enhanced-Memory-Leak-Detectio>

**Bounds Checker**

应用环境：Windows + VC6.0

编程语言：C/C++

使用方法：安装使用，会自动在VC内创建右键菜单

结果输出：输出到VC的调试窗口中

设计思路： 未知

优缺点：可以检测内存泄露；资源泄漏；对指针的错误操作，内存读、写溢出；使用未初始化的内存

如何获取：<http://3ddown.com/soft/31594.htm>,安装licence时，需要将日期调整为2008年，然后安装licence。有一个licence安装后显示是8.3的，但是可以使用。

**mtrace**

应用环境：Linux GLIBC

编程语言：C

使用方法： 包含头文件mcheck.h，定义环境变量MALLOC\_TRACE为输出文件名，程序开始时调用mtrace()即可。

结果输出：用户指定的文件

设计思路： 为malloc,realloc,free函数添加钩子函数,记录每一对malloc-free的执行

优缺点：只能检查使用malloc/realloc/free造成的的内存泄露

如何获取：GLIBC自带，可直接使用

**memwatch**

应用环境：Linux

编程语言：C

使用方法：  加入memwatch.h，编译时加上-DMEMWATCH -DMW\_STDIO及memwatch.c

结果输出：输出文件名称为memwatch.log，在程序执行期间，错误提示都会显示在stdout上

设计思路：将malloc/realloc/calloc/strdup/free等重定义为mwMalloc(sz, \_\_FILE\_\_, \_\_LINE\_\_)等，内部维护一个操作链表

优缺点：能检测双重释放（double-free）、错误释放（erroneous free）、内存泄漏（unfreed memory）、溢出(Overflow)、下溢(Underflow)等等

如何获取：<http://memwatch.sourceforge.net/>

**valgrind**

应用环境：Linux

编程语言：C/C++

使用方法：  加入memwatch.h，编译时加上-DMEMWATCH -DMW\_STDIO及memwatch.c

结果输出：输出文件名称为memwatch.log，在程序执行期间，错误提示都会显示在stdout上

设计思路：根据软件的内存操作维护一个有效地址空间表和无效地址空间表（进程的地址空间）

优缺点：能够检测：

使用未初始化的内存 (Use of uninitialised memory)

使用已经释放了的内存 (Reading/writing memory after it has been free’d)

使用超过 malloc分配的内存空间(Reading/writing off the end of malloc’d blocks)

对堆栈的非法访问 (Reading/writing inappropriate areas on the stack)

申请的空间是否有释放 (Memory leaks – where pointers to malloc’d blocks are lost forever)

如何获取：<http://valgrind.org/>

**debug\_new**

应用环境：Linux/Windows

编程语言：C++

使用方法： 包含头文件debug\_new.h，链接debug\_new.cpp

结果输出：控制台console

设计思路： 通过重载new和delete操作符来捕获内存申请/释放请求，并在程序内部维护一个全局静态变量的哈希链表。在new操作符中，不仅仅分配用户所要求的内存，而是在为每次分配的内存都添加一个头部，存储着此次分配的位置信息和链表指针，new返回的是分配的这块内存加上头部偏移后的值，而在之前已经将此返回值作了HASH计算并添加到HASH链表中了。delete的时候先根据要释放的指针地址做HASH计算，然后再遍历数组HASH值处的链表进行查找，如果找到则将该节点移除，未找到就abort。这样在程序结束之后，通过检查此数组中是否还有未释放的内存块来确定是否有内存泄露。

优缺点：跨平台，仅用于C++程序，

如何获取：<http://www.ibm.com/developerworks/cn/linux/l-mleak2/index.html>

工具原理分析

以上的这些分析工具，所使用的方法大致分为以下几种：

1、注册内存分配/释放钩子函数(hook)。在Linux下可以malloc\_hook, free\_hook等5个钩子函数，在Windows下可以注册\_CrtSetAllocHook钩子函数，这样在分配内存的时候就可以捕获这一请求并加以处理。Visual Leak Detecter和mtrace使用此方式。

2、使用宏定义替换。将用户代码中的malloc, free 替换为宏定义的 mwMalloc(sz, \_\_FILE\_\_, \_\_LINE\_\_)等自定义函数，从而跟踪内存请求，memwatch即使用此方式。

3、操作符重载。此方法仅用于C++语言中，通过重载new、delete操作符来实现跟踪内存请求，重载后的操作符类似于钩子函数意义。debug\_new采用此方式。

这些工具的输出方式也分以下几种：

1、Windows VC环境下一般输出到调试窗口中，因此VC本身就提供了一个理想的输出场所，并且GUI应用程序输出到标准输出时不可见的。Visual Leak Detecter采用此法。

2、输出到标准输出或标准错误输出：控制台应用程序可以输出到屏幕，如memwatch, valgrind, debug\_new都是采用这种方法。

3、输出到日志文件：将结果输出到用户指定或默认的日志文件中，如mtrace和memwatch。

此外，这些工具的内存检测方式无非也分为两种：

1、维护一个内存操作链表，当有内存申请操作时，将其加入此链表中，当有释放操作时，从申请操作从链表中移除。如果到程序结束后此链表中还有内容，说明有内存泄露了；如果要释放的内存操作没有在链表中找到对应操作，则说明是释放了多次。使用此方法的有VC内置的调试工具，Visual Leak Detecter，mtrace, memwatch, debug\_new。

2、模拟进程的地址空间。仿照操作系统对进程内存操作的处理，在用户态下维护一个地址空间映射，此方法要求对进程地址空间的处理有较深的理解。因为Windows的进程地址空间分布不是开源的，所以模拟起来很困难，因此只支持Linux。采用此方法的是valgrind。

**总结：**

检测内存泄漏的关键是要能截获住对分配内存和释放内存的函数的调用。截获住这两个函数，我们就能跟踪每一块内存的生命周期，比如，每当成功的分配一块内存后，就把它的指针加入一个全局的list中；每当释放一块内存，再把它的指针从list中删除。这样，当程序结束的时候，list中剩余的指针就是指向那些没有被释放的内存。如果要检测堆内存的泄漏，那么需要截获住malloc/realloc/free和new/delete就可以了（其实new/delete最终也是用malloc/free的，所以只要截获前面一组即可）。对于其他的泄漏，可以采用类似的方法，截获住相应的分配和释放函数。比如，要检测BSTR的泄漏，就需要截获SysAllocString/SysFreeString；要检测HMENU的泄漏，就需要截获CreateMenu/ DestroyMenu。（有的资源的分配函数有多个，释放函数只有一个，比如，SysAllocStringLen也可以用来分配BSTR，这时就需要截获多个分配函数）

# [堆和栈的区别（转过无数次的文章）](http://blog.csdn.net/hairetz/article/details/4141043)

 一、预备知识—程序的内存分配

  一个由C/C++编译的程序占用的内存分为以下几个部分

  1、栈区（stack）—   由编译器自动分配释放   ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

  2、堆区（heap）   —   一般由程序员分配释放，   若程序员不释放，程序结束时可能由OS回收   。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

  3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，   未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。   -   程序结束后由系统释放。

  4、文字常量区   —常量字符串就是放在这里的。   程序结束后由系统释放

  5、程序代码区—存放函数体的二进制代码。

    二、例子程序

  这是一个前辈写的，非常详细

  //main.cpp

  int   a   =   0;   全局初始化区

  char   \*p1;   全局未初始化区

  main()

  {

  int   b;   栈

  char   s[]   =   "abc";   栈

  char   \*p2;   栈

  char   \*p3   =   "123456";   123456/0在常量区，p3在栈上。

  static   int   c   =0；   全局（静态）初始化区

  p1   =   (char   \*)malloc(10);

  p2   =   (char   \*)malloc(20);

  分配得来得10和20字节的区域就在堆区。

  strcpy(p1,   "123456");   123456/0放在常量区，编译器可能会将它与p3所指向的"123456"  优化成一个地方。

  }

     二、堆和栈的理论知识

  2.1申请方式

  stack:

  由系统自动分配。   例如，声明在函数中一个局部变量   int   b;   系统自动在栈中为b开辟空间

  heap:

  需要程序员自己申请，并指明大小，在c中malloc函数

  如p1   =   (char   \*)malloc(10);

  在C++中用new运算符

  如p2   =   new   char[10];

  但是注意p1、p2本身是在栈中的。

  2.2

  申请后系统的响应

  栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

  堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部 分重新放入空闲链表中。

  2.3申请大小的限制

  栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

  堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

  2.4申请效率的比较：

  栈由系统自动分配，速度较快。但程序员是无法控制的。

  堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便. 另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈,是直接在进程的地址空间中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活。

  2.5堆和栈中的存储内容

  栈：   在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

  堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

  2.6存取效率的比较

  char   s1[]   =   "aaaaaaaaaaaaaaa";

  char   \*s2   =   "bbbbbbbbbbbbbbbbb";

  aaaaaaaaaaa是在运行时刻赋值的；

  而bbbbbbbbbbb是在编译时就确定的；

  但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

  比如：

  #include

  void   main()

  {

  char   a   =   1;

  char   c[]   =   "1234567890";

  char   \*p   ="1234567890";

  a   =   c[1];

  a   =   p[1];

  return;

  }

  对应的汇编代码

  10:   a   =   c[1];

  00401067   8A   4D   F1   mov   cl,byte   ptr   [ebp-0Fh]

  0040106A   88   4D   FC   mov   byte   ptr   [ebp-4],cl

  11:   a   =   p[1];

  0040106D   8B   55   EC   mov   edx,dword   ptr   [ebp-14h]

  00401070   8A   42   01   mov   al,byte   ptr   [edx+1]

  00401073   88   45   FC   mov   byte   ptr   [ebp-4],al

  第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到

  edx中，再根据edx读取字符，显然慢了。

     2.7小结：

  堆和栈的区别可以用如下的比喻来看出：

  使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。   (经典！)

# [C/C++内存泄漏及检测](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

2011-02-20 17:51 by 吴秦, 72449 阅读, 19 评论, [收藏](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html), [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=1959162)

“该死系统存在内存泄漏问题”，项目中由于各方面因素，总是有人抱怨存在内存泄漏，系统长时间运行之后，可用内存越来越少，甚至导致了某些服务失败。内存泄漏是最难发现的常见错误之一，因为除非用完内存或调用malloc失败，否则都不会导致任何问题。实际上，使用C/C++这类没有垃圾回收机制的语言时，你很多时间都花在处理如何正确释放内存上。如果程序运行时间足够长，如后台进程运行在服务器上，只要服务器不宕机就一直运行，一个小小的失误也会对程序造成重大的影响，如造成某些关键服务失败。

对于内存泄漏，本人深有体会！实习的时候，公司一个项目中就存在内存泄漏问题，项目的代码两非常大，后台进程也比较多，造成内存泄漏的地方比较难找。这次机会是我对如何查找内存泄漏问题，有了一定的经验，后面自己的做了相关实验，在此我分享一下内存泄漏如何调试查找，主要内容如下：

* 1、内存泄漏简介
* 2、Windows平台下的内存泄漏检测
  + 2.1、检测是否存在内存泄漏问题
  + 2.2、定位具体的内存泄漏地方
* 3、Linux平台下的内存泄漏检测
* 4、总结

其实Windows、Linux下面的内存检测都可以单独开篇详细介绍，方法和工具也远远不止文中介绍到的，我的方法也不是最优的，如果您有更好的方法，也请您告诉我和大家。

**1、内存泄漏简介及后果**

wikipedia中这样定义内存泄漏：在计算机科学中，内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

最难捉摸也最难检测到的错误之一是内存泄漏，即未能正确释放以前分配的内存的 bug。 只发生一次的小的内存泄漏可能不会被注意，但泄漏大量内存的程序或泄漏日益增多的程序可能会表现出各种征兆：从性能不良（并且逐渐降低）到内存完全用尽。 更糟的是，泄漏的程序可能会用掉太多内存，以致另一个程序失败，而使用户无从查找问题的真正根源。 此外，即使无害的内存泄漏也可能是其他问题的征兆。

内存泄漏会因为减少可用内存的数量从而降低计算机的性能。最终，在最糟糕的情况下，过多的可用内存被分配掉导致全部或部分设备停止正常工作，或者应用程序崩溃。内存泄漏可能不严重，甚至能够被常规的手段检测出来。在现代操作系统中，一个应用程序使用的常规内存在程序终止时被释放。这表示一个短暂运行的应用程序中的内存泄漏不会导致严重后果。

在以下情況，内存泄漏导致较严重的后果：

* 程序运行后置之不理，并且随着时间的流失消耗越来越多的内存（比如服务器上的后台任务，尤其是[嵌入式系统](http://zh.wikipedia.org/wiki/%E5%B5%8C%E5%85%A5%E5%BC%8F%E7%B3%BB%E7%BB%9F)中的后台任务，这些任务可能被运行后很多年内都置之不理）；
* 新的内存被频繁地分配，比如当显示电脑游戏或动画视频画面时；
* 程序能够请求未被释放的内存（比如[共享内存](http://zh.wikipedia.org/wiki/%E5%85%B1%E4%BA%AB%E5%86%85%E5%AD%98)），甚至是在程序终止的时候；
* 泄漏在操作系统内部发生；
* 泄漏在系统关键驱动中发生；
* 内存非常有限，比如在[嵌入式系统](http://zh.wikipedia.org/wiki/%E5%B5%8C%E5%85%A5%E5%BC%8F%E7%B3%BB%E7%BB%9F)或便携设备中；
* 当运行于一个终止时内存并不自动释放的操作系统（比如[AmigaOS](http://zh.wikipedia.org/w/index.php?title=AmigaOS&action=edit&redlink=1)）之上，而且一旦丢失只能通过重启来恢复。

下面我们通过以下例子来介绍如何检测内存泄漏问题：

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdlib.h>  #include <iostream>  using namespace std;    void GetMemory(char \*p, int num)  {      p = (char\*)malloc(sizeof(char) \* num);//使用new也能够检测出来  }    int main(int argc,char\*\* argv)  {      char \*str = NULL;      GetMemory(str, 100);      cout<<"Memory leak test!"<<endl;      //如果main中存在while循环调用GetMemory      //那么问题将变得很严重      //while(1){GetMemory(...);}      return 0;  } |

实际中不可能这么简单，如果这么简单也用不着别的方法，程序员一眼就可以看出问题，此程序只用于测试。

**2、Windows平台下的内存泄漏检测**

**2.1、检测是否存在内存泄漏问题**

Windows平台下面Visual Studio 调试器和 C 运行时 (CRT) 库为我们提供了检测和识别内存泄漏的有效方法，原理大致如下：内存分配要通过CRT在运行时实现，只要在分配内存和释放内存时分别做好记录，程序结束时对比分配内存和释放内存的记录就可以确定是不是有内存泄漏。在vs中启用内存检测的方法如下：

* STEP1，在程序中包括以下语句： （#include 语句必须采用上文所示顺序。 如果更改了顺序，所使用的函数可能无法正常工作。）

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1  2  3 | #define \_CRTDBG\_MAP\_ALLOC  #include <stdlib.h>  #include <crtdbg.h> |

通过包括 crtdbg.h，将 [malloc](http://msdn.microsoft.com/zh-cn/library/6ewkz86d.aspx) 和 [free](http://msdn.microsoft.com/zh-cn/library/we1whae7.aspx) 函数映射到它们的调试版本，即 [\_malloc\_dbg](http://msdn.microsoft.com/zh-cn/library/faz3a37z.aspx) 和 [\_free\_dbg](http://msdn.microsoft.com/zh-cn/library/16swbsbc.aspx)，这两个函数将跟踪内存分配和释放。 此映射只在调试版本（在其中定义了**\_DEBUG**）中发生。 发布版本使用普通的 **malloc** 和 **free** 函数。

#define 语句将 CRT 堆函数的基版本映射到对应的“Debug”版本。 并非绝对需要该语句；但如果没有该语句，内存泄漏转储包含的有用信息将较少。

* STEP2， 在添加了上述语句之后，可以通过在程序中包括以下语句（通常应恰好放在程序退出位置之前）来转储内存泄漏信息：

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

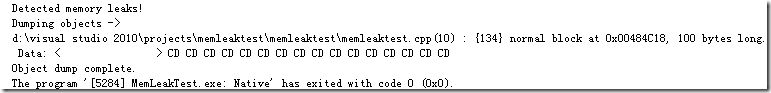
|  |  |
| --- | --- |
| 1 | \_CrtDumpMemoryLeaks(); |

此时，完整的代码如下：

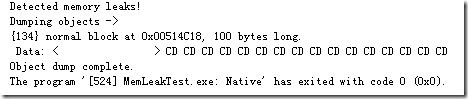
[+ View Code?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #define \_CRTDBG\_MAP\_ALLOC  #include <stdlib.h>  #include <crtdbg.h>    #include <iostream>  using namespace std;    void GetMemory(char \*p, int num)  {      p = (char\*)malloc(sizeof(char) \* num);  }    int main(int argc,char\*\* argv)  {      char \*str = NULL;      GetMemory(str, 100);      cout<<"Memory leak test!"<<endl;      \_CrtDumpMemoryLeaks();      return 0;  } |

当在调试器下运行程序时，[\_CrtDumpMemoryLeaks](http://msdn.microsoft.com/zh-cn/library/d41t22sb.aspx) 将在[“输出”窗口](http://msdn.microsoft.com/zh-cn/library/3hk6fby3.aspx)中显示内存泄漏信息。 内存泄漏信息如下所示：

[](http://images.cnblogs.com/cnblogs_com/skynet/201102/20110220175032138.png)

如果没有使用 #define \_CRTDBG\_MAP\_ALLOC 语句，内存泄漏转储将如下所示：

[](http://images.cnblogs.com/cnblogs_com/skynet/201102/201102201750331467.png)

未定义 \_CRTDBG\_MAP\_ALLOC 时，所显示的会是：

* 内存分配编号（在大括号内）。
* [块类型](http://msdn.microsoft.com/zh-cn/library/htdyz80k.aspx)（普通、客户端或 CRT）。
* “普通块”是由程序分配的普通内存。
* “客户端块”是由 MFC 程序用于需要析构函数的对象的特殊类型内存块。 MFC new 操作根据正在创建的对象的需要创建普通块或客户端块。
* “CRT 块”是由 CRT 库为自己使用而分配的内存块。 CRT 库处理这些块的释放，因此您不大可能在内存泄漏报告中看到这些块，除非出现严重错误（例如 CRT 库损坏）。

从不会在内存泄漏信息中看到下面两种块类型：

* “可用块”是已释放的内存块。
* “忽略块”是您已特别标记的块，因而不出现在内存泄漏报告中。
* 十六进制形式的内存位置。
* 以字节为单位的块大小。
* 前 16 字节的内容（亦为十六进制）。

定义了 \_CRTDBG\_MAP\_ALLOC 时，还会显示在其中分配泄漏的内存的文件。 文件名后括号中的数字（本示例中为 10）是该文件中的行号。

注意：如果程序总是在同一位置退出，调用 [\_CrtDumpMemoryLeaks](http://msdn.microsoft.com/zh-cn/library/d41t22sb.aspx) 将非常容易。 如果程序从多个位置退出，则无需在每个可能退出的位置放置对 **\_CrtDumpMemoryLeaks** 的调用，而可以在程序开始处包含以下调用：

[+ View Code?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1 | \_CrtSetDbgFlag ( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF ); |

该语句在程序退出时自动调用 **\_CrtDumpMemoryLeaks**。 必须同时设置 **\_CRTDBG\_ALLOC\_MEM\_DF** 和 **\_CRTDBG\_LEAK\_CHECK\_DF** 两个位域，如前面所示。

**2.2、定位具体的内存泄漏地方**

通过上面的方法，我们几乎可以定位到是哪个地方调用内存分配函数malloc和new等，如上例中的GetMemory函数中，即第10行！但是不能定位到，在哪个地方调用GetMemory()导致的内存泄漏，而且在大型项目中可能有很多处调用GetMemory。如何要定位到在哪个地方调用GetMemory导致的内存泄漏？

定位内存泄漏的另一种技术涉及在关键点对应用程序的内存状态拍快照。 CRT 库提供一种结构类型 **\_CrtMemState**，您可用它存储内存状态的快照：

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1 | \_CrtMemState s1, s2, s3; |

若要在给定点对内存状态拍快照，请向 [\_CrtMemCheckpoint](http://msdn.microsoft.com/zh-cn/library/h3z85t43.aspx) 函数传递 **\_CrtMemState** 结构。 该函数用当前内存状态的快照填充此结构：

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1 | \_CrtMemCheckpoint( &s1 ); |

通过向 [\_CrtMemDumpStatistics](http://msdn.microsoft.com/zh-cn/library/swh3417y.aspx) 函数传递 **\_CrtMemState** 结构，可以在任意点转储该结构的内容：

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1 | \_CrtMemDumpStatistics( &s1 ); |

若要确定代码中某一部分是否发生了内存泄漏，可以在该部分之前和之后对内存状态拍快照，然后使用 [\_CrtMemDifference](http://msdn.microsoft.com/zh-cn/library/k4htzb06.aspx) 比较这两个状态：

[?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \_CrtMemCheckpoint( &s1 );  // memory allocations take place here  \_CrtMemCheckpoint( &s2 );    if ( \_CrtMemDifference( &s3, &s1, &s2) )     \_CrtMemDumpStatistics( &s3 ); |

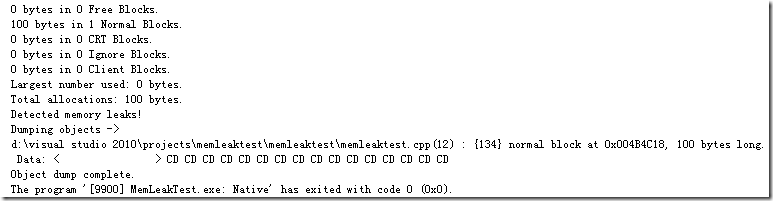
顾名思义，**\_CrtMemDifference** 比较两个内存状态（s1 和 s2），生成这两个状态之间差异的结果（s3）。 在程序的开始和结尾放置 **\_CrtMemCheckpoint** 调用，并使用**\_CrtMemDifference** 比较结果，是检查内存泄漏的另一种方法。 如果检测到泄漏，则可以使用 **\_CrtMemCheckpoint** 调用通过二进制搜索技术来划分程序和定位泄漏。

如上面的例子程序我们可以这样来定位确切的调用GetMemory的地方：

[+ View Code?](http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #define \_CRTDBG\_MAP\_ALLOC  #include <stdlib.h>  #include <crtdbg.h>    #include <iostream>  using namespace std;    \_CrtMemState s1, s2, s3;    void GetMemory(char \*p, int num)  {      p = (char\*)malloc(sizeof(char) \* num);  }    int main(int argc,char\*\* argv)  {      \_CrtMemCheckpoint( &s1 );      char \*str = NULL;      GetMemory(str, 100);      \_CrtMemCheckpoint( &s2 );      if ( \_CrtMemDifference( &s3, &s1, &s2) )          \_CrtMemDumpStatistics( &s3 );      cout<<"Memory leak test!"<<endl;      \_CrtDumpMemoryLeaks();      return 0;  } |

调试时，程序输出如下结果：

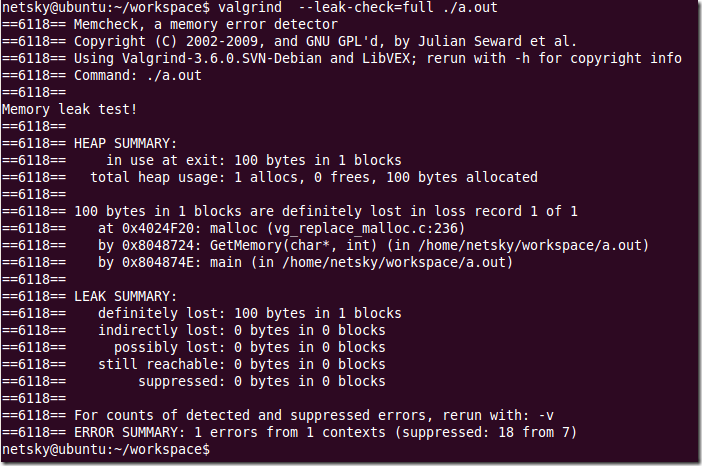
[](http://images.cnblogs.com/cnblogs_com/skynet/201102/201102201750335022.png)

这说明在s1和s2之间存在内存泄漏！！！如果GetMemory不是在s1和s2之间调用，那么就不会有信息输出。

**3、Linux平台下的内存泄漏检测**

在上面我们介绍了，vs中在代码中“包含crtdbg.h，将 [malloc](http://msdn.microsoft.com/zh-cn/library/6ewkz86d.aspx) 和 [free](http://msdn.microsoft.com/zh-cn/library/we1whae7.aspx) 函数映射到它们的调试版本，即 [\_malloc\_dbg](http://msdn.microsoft.com/zh-cn/library/faz3a37z.aspx) 和 [\_free\_dbg](http://msdn.microsoft.com/zh-cn/library/16swbsbc.aspx)，这两个函数将跟踪内存分配和释放。 此映射只在调试版本（在其中定义了**\_DEBUG**）中发生。 发布版本使用普通的 **malloc** 和 **free** 函数。”即为malloc和free做了钩子，用于记录内存分配信息。

Linux下面也有原理相同的方法——mtrace，<http://en.wikipedia.org/wiki/Mtrace>。方法类似，我这就不具体描述，参加给出的链接。这节我主要介绍一个非常强大的工具valgrind。如下图所示：

[](http://images.cnblogs.com/cnblogs_com/skynet/201102/201102201750352698.png)

如上图所示知道：

==6118== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1   
==6118==    at 0x4024F20: malloc (vg\_replace\_malloc.c:236)   
==6118==    by 0x8048724: GetMemory(char\*, int) (in /home/netsky/workspace/a.out)   
==6118==    by 0x804874E: main (in /home/netsky/workspace/a.out)

是在main中调用了GetMemory导致的内存泄漏，GetMemory中是调用了malloc导致泄漏了100字节的内存。

Things to notice:   
• There is a lot of information in each error message; read it carefully.   
• The 6118 is the process ID; it’s usually unimportant.   
• The ﬁrst line ("Heap Summary") tells you what kind of error it is.   
• Below the ﬁrst line is a stack trace telling you where the problem occurred. Stack traces can get quite large, and be   
confusing, especially if you are using the C++ STL. Reading them from the bottom up can help.

• The code addresses (eg. 0x4024F20) are usually unimportant, but occasionally crucial for tracking down weirder   
bugs.

The stack trace tells you where the leaked memory was allocated. Memcheck cannot tell you why the memory leaked,   
unfortunately. (Ignore the "vg\_replace\_malloc.c", that’s an implementation detail.)   
There are several kinds of leaks; the two most important categories are:   
• "deﬁnitely lost": your program is leaking memory -- ﬁx it!   
• "probably lost": your program is leaking memory, unless you’re doing funny things with pointers (such as moving   
them to point to the middle of a heap block)

Valgrind的使用请见手册<http://valgrind.org/docs/manual/manual.html>。

**4、总结**

其实内存泄漏的原因可以概括为：调用了malloc/new等内存申请的操作，但缺少了对应的free/delete，总之就是，malloc/new比free/delete的数量多。我们在编程时需要注意这点，保证每个malloc都有对应的free，每个new都有对应的deleted！！！平时要养成这样一个好的习惯。

要避免内存泄漏可以总结为以下几点：

* 程序员要养成良好习惯，保证malloc/new和free/delete匹配；
* 检测内存泄漏的关键原理就是，检查malloc/new和free/delete是否匹配，一些工具也就是这个原理。要做到这点，就是利用宏或者钩子，在用户程序与运行库之间加了一层，用于记录内存分配情况。

# DEP—百度百科

DEP -[数据执行保护](http://baike.baidu.com/view/187961.htm)的缩写，Data Execution Prevention。

中文名

数据执行保护

外文名

Data Execution Prevention

优    点

帮助防止数据页执行代码

分    类

硬件技术

工作状态

四种

作    用

帮助防止在系统上运行恶意代码

## **目录**

## DEP简介

[编辑](javascript:;)

[数据执行保护](http://baike.baidu.com/view/187961.htm)(DEP) 是一套软硬件技术，能够在内存上执行额外检查以帮助防止在系统上运行恶意代码。在 Microsoft Windows XP Service Pack 2、 Microsoft Windows Server 2003 Service Pack 1 、Microsoft Windows XP Tablet PC Edition 2005 、 Microsoft Windows Vista 和 Microsoft windows7 中，由硬件和软件一起强制实施 DEP。[1]

## DEP主要优点

[编辑](javascript:;)

可帮助防止数据页当作代码执行，从而有效分离数据与代码。通常情况下，不执行默认堆和[堆栈](http://baike.baidu.com/view/93201.htm)中的代码。硬件实施 DEP 检测从这些位置运行的代码，并在发现执行情况时引发异常。软件实施 DEP 可帮助阻止恶意代码利用 Windows 中的[异常处理](http://baike.baidu.com/view/1072586.htm)机制进行破坏。

硬件DEP 是某些 DEP 兼容处理器的功能，可以防止在已标记为数据存储区的内存区域中的代码执行。 此功能也称为非执行和执行保护。 Windows XP SP2 还包括软件实施 DEP，其目的在于减少利用 Windows 中的例外处理机制的情况。

与防病毒程序不同，硬件和软件实施 DEP 技术的目的并不是防止在计算机上安装有害程序。 而是监视您的已安装程序，帮助确定它们是否正在安全地使用[系统内存](http://baike.baidu.com/view/404470.htm)空间。 为监视您的程序，硬件实施 DEP 将跟踪已指定为“不可执行”的内存区域。 如果已将内存指定为“不可执行”，但是某个程序试图通过内存执行代码，Windows 将关闭该程序以防止恶意代码。 无论代码是不是恶意，都会执行此操作。从而使代码内存空间存放可执行的代码，数据内存空间只存储数据，数据空间中如果有任何代码都视作数据，不给予代码执行权限。[1]

## DEP注意事项

[编辑](javascript:;)

注:基于软件的 DEP 是 Windows XP SP2 的一部分并默认启用，不考虑处理器的硬件实施 DEP 功能。 默认情况下，软件实施 DEP 应用于核心操作系统组件和服务。

DEP 默认配置的目的在于保护您的计算机，并尽量减小对应用程序兼容性的影响。 但是某些程序也可能无法正确运行，视您的 DEP 配置而定。

在运行 Microsoft Windows XP 64 位版本并附带 DEP 兼容处理器的计算机上，硬件实施 DEP 已默认启用。 64 位应用程序将不会通过内存的“不可执行”区域来运行。不能禁用硬件启用的 DEP。

Windows XP SP2 上的软件启用 DEP 以及在任何处理器上运行的 32 位应用程序可以配置为使用内存的“可执行”或“不可执行”区域。

为了配合微软的DEP功能，Intel为自己的CPU开发了“Execute Disable Bit”(EDB)内存保护技术。目前Intel P4 Prescott(mPGA478与LGA775封装)为C0或D0[步进](http://baike.baidu.com/view/16839.htm)核心，最新的J系列P4 Prescott采用E0步进核心。其中只有J系列P4 Prescott具备防病毒功能，只有它才真正支持EDP技术，能够配合SP2的DEP防毒功能，让针对[缓冲区溢出](http://baike.baidu.com/view/36638.htm" \t "_blank)(buffer overrun)漏洞设计的病毒失效，预防它们复制并散播到其他系统。

AMD[64位处理器](http://baike.baidu.com/view/838140.htm)最先支持微软的DEP技术。为了配合DEP，AMD与微软一起设计研发了AMD的新芯片功能“Enhanced Virus Protection”(EVP增强病毒保护)。AMD64位处理器(包括Athlon 64/Athlon 64 FX/Athlon 64移动版本/Sempron移动版本等)都将具有EVP功能。EVP功能可以和SP2的DEP技术配合，防范“[缓存溢出](http://baike.baidu.com/view/973698.htm)”这一常见攻击手段，打击一些病毒和[蠕虫](http://baike.baidu.com/view/2596.htm)，对收发电子邮件、下载文件等日常工作进行更好的保护。[1]

## DEP安全机制

[编辑](javascript:;)

DEP(Data Execution Prevention)即“[数据执行保护](http://baike.baidu.com/view/187961.htm)”，这是Windows的一项安全机制，主要用来防止病毒和其他安全威胁对系统造成破坏。微软从Windows XP SP2引入了该技术，并一直延续到此后的Windows Server 2003、Windows Server 2008中。毫无例外，在Windows 7中DEP也作为一项安全机制被引入进来。本文将对Windows 7下的DEP进行一番解析。

### DEP防止溢出

可以说，溢出是操作系统(应用软件)永远的痛，Windows 7自然也不例外。所谓溢出主要指缓冲区溢出，就是利用系统(应用软件)漏洞从只有Windows和其他程序可以使用的内存位置执行恶意代码从而达到控制系统的目的。如前所述，缓冲区溢出攻击经常在其它程序的内存缓冲区写入可执行的恶意代码，然后诱骗程序执行恶意代码。使用DEP的目的是阻止恶意插入代码的执行，其运行机制是，Windows利用DEP标记只包含数据的内存位置为非可执行(NX)，当应用程序试图从标记为NX的内存位置执行代码时，Windows的DEP逻辑将阻止应用程序这样做，从而达到保护系统防止溢出。

### DEP实现方式

微软DEP实现采用了两种方式，即硬件强制DEP和软件强制DEP。硬件强制DEP，这需要处理器的支持，不过现在大多数处理器是支持DEP的。软件强制DEP，这是由Windows操作系统在[系统内存](http://baike.baidu.com/view/404470.htm)中为保存的数据对象自动添加的一组特殊指针提供。如何知道自己的处理器是否支持DEP呢？右键单击桌面上的“计算机”图标，选择“属性”，在打开的“系统”窗口中点击“高级系统设置”链接打开“系统属性”面板。在“高级”选项卡页面中的“性能”下点击“设置”打开“性能选项”面板。点击“[数据执行保护](http://baike.baidu.com/view/187961.htm)”选项卡，在该页面中我们可确认自己计算机的处理器是否支持DEP。如果支持会在底部的一行显示“你的计算机处理器支持基于硬件的DEP。”，反之会显示“你的计算机处理器不支持基于硬件的DEP，不过，Windows能使用DEP软件防止一些类型的攻击。”(图1)

[1]

### DEP运行级别

根据启动参数的不同，DEP工作状态可以分为四种。

（1）Optin：默认仅将DEP保护应用于Windows系统组件和服务，对于其他程序不予保护，但用户可以通过应用程序兼容性工具(ACT，Application Compatibility Toolkit)为选定的程序启用DEP，在Vista下边经过/NXcompat选项编译过的程序将自动应用DEP。这种模式可以被应用程序动态关闭，它多用于普通用户版的操作系统，如Windows XP、Windows Vista、Windows7。

（2）Optout：为排除列表程序外的所有程序和服务启用DEP，用户可以手动在排除列表中指定不启用DEP保护的程序和服务。这种模式可以被应用程序动态关闭，它多用于服务器版的操作系统，如 Windows 2003、Windows 2008。

（3）AlwaysOn：对所有进程启用DEP 的保护，不存在排序列表，在这种模式下，DEP不可以被关闭，目前只有在64位的操作系统上才工作在AlwaysOn模式。

（4）AlwaysOff：对所有进程都禁用DEP，这种模式下，DEP也不能被动态开启，这种模式一般只有在某种特定场合才使用，如DEP干扰到程序的正常运行。[1]

在Windows 7中，DEP默认是激活的。不过，DEP不能保护系统中所有运行的应用程序，实际DEP能够保护的程序列表由DEP的保护级别定义。DEP支持两种保护级别：级别1，只保护Windows系统代码和可执行文件，不保护系统中运行的其它微软或第三方应用程序；级别2，保护系统中运行的所有可执行代码，包括Windows系统代码和微软或第三方应用程序。默认情况下，Windows 7的DEP运行在级别1的保护状态下。在“[数据执行保护](http://baike.baidu.com/view/187961.htm)”配置面板中，我们能够设置DEP的保护级别。如图所示笔者的Windows 7默认“只为基本的Windows程序和服务激活了DEP”，即DEP保护级别为1。当然，我们也可选择“除了以下所选择的，为所有程序和服务打开DEP” 切换到DEP保护级别2。

在保护级别Level 2可以选择特定的应用程序不受DEP保护。在实际应用中，这个功能非常重要，因为一些老的应用程序在激活DEP时无法正常运行。 例如，我我们在使用Word进行文本编辑时，它会自动被排除在DEP保护之外。需要注意的是，在将DEP保护切换到级别2之前，必须运行应用程序兼容性测试，确保所有的应用程序在DEP激活时能正常运行。从DEP中排除应用程序， 需要在DEP配置页面使用“添加”按钮，将应用程序的可执行文件加入到排除列表中。

### DEP如何关闭

需要提醒的是，当DEP运行在保护级别为2时，由于需要在处理器和[系统内存](http://baike.baidu.com/view/404470.htm)运行所有的DEP检查，会影响系统性能，使得系统运行将会变慢一些，所以在某些情况下我们可以考虑完全关闭DEP保护。我们知道在DEP设置面板中是不提供关闭DEP选项的，那如何关闭呢？如果是Vista以前的系统，我们可以通过修改Boot.ini文件，在其中添加NoExecute=Always0ff语句来关闭。而在Windows Vista、Windows Server 2008和Windows 7中，boot.ini文件已经被启动配置数据(Boot Configuration Data)即BCD文件所代替，不过我们可以使用微软提供的命令行工具bcdedit.exe来编辑BCD文件。

我们在命令提示符下运行不带有任何参数的bcedit命令，可以看到当前的启动配置，如图所示显示了在Windows 7下运行bcdedit的结果，其中最后一行显示nx OptIn，表示当前的DEP保护级别为1，如果显示为OptOut则表示当前的EDP保护级别为2。如果我们要关闭DEP，只需将nx设置为Always0ff即可。在命令行下执行命令“bcdedit /set nx alwaysoff”，重启系统后Windows 7的DEP就关闭了。反之，如果要开启所有服务和应用程序的DEP，执行命令“bcdedit /set nx alwayson”就可以了。

## DEP局限性

[编辑](javascript:;)

如同前面介绍的安全机制一样，DEP也有着自身的局限性。

首先，硬件DEP需要CPU的支持，但并不是所有的CPU都提供了硬件DEP的支持，在一些比较老的CPU上边DEP是无法发挥作用的。

其次，由于兼容性的原因Windows不能对所有进程开启DEP保护，否则可能会出现异常。例如一些第三方的插件DLL，由于无法确认其是否支持DEP，对涉及这些DLL的程序不敢贸然开启DEP保护。再有就是使用ATL 7.1或者以前版本的程序需要在数据页面上产生可以执行代码，这种情况就不能开启DEP保护，否则程序会出现异常。

再次，/NXCOMPAT编译选项，或者是IMAGE\_DLLCHARACTERISTICS\_NX\_COMPAT的设置，只对Windows Vista 以上的系统有效。在以前的系统上，如Windows XP SP3等，这个设置会被忽略。也就是说，即使采用了该链接选项的程序在一些操作系统上也不会自动启用DEP保护。

最后，当DEP工作在最主要的两种状态Optin和Optout下时，DEP是可以被动态关闭和开启的，这就说明操作系统提供了某些API函数来控制DEP的状态。同样很不幸的是早期的操作系统中对这些API函数的调用没有任何限制，所有的进程都可以调用这些API函数，这就埋下了很大的安全隐患，也为我们突破DEP提供了一条道路。[1]

参考资料

* 1.  [DEP机制的保护原理](http://baike.baidu.com/redirect/cab92CwXOcy5-Dky-Zt9-wNENH91zzkqYZPJ9W3W43ahgCMFQSn7OSK1EDdpvNduNTzK_uTLGPwc6spY7MUVIHdW2NEYH_edUF2xhI2_RmV8) ．51CTO[引用日期2012-08-28]

# Data Execution Prevention

* Data Execution Prevention (DEP) is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003. DEP enables the system to mark one or more pages of memory as non-executable. Marking memory regions as non-executable means that code cannot be run from that region of memory, which makes it harder for the exploitation of buffer overruns.
* DEP prevents code from being run from data pages such as the default heap, stacks, and memory pools. If an application attempts to run code from a data page that is protected, a memory access violation exception occurs, and if the exception is not handled, the calling process is terminated.
* DEP is not intended to be a comprehensive defense against all exploits; it is intended to be another tool that you can use to secure your application.

## **How Data Execution Prevention Works**

* If an application attempts to run code from a protected page, the application receives an exception with the status code STATUS\_ACCESS\_VIOLATION. If your application must run code from a memory page, it must allocate and set the proper virtual [memory protection](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/memory_protection.htm) attributes. The allocated memory must be marked PAGE\_EXECUTE, PAGE\_EXECUTE\_READ, PAGE\_EXECUTE\_READWRITE, or PAGE\_EXECUTE\_WRITECOPY when allocating memory. Heap allocations made by calling the malloc and HeapAlloc functions are non-executable.
* Applications cannot run code from the default process heap or the stack.
* DEP is configured at system boot according to the no-execute page protection policy setting in the boot configuration data. An application can get the current policy setting by calling the [**GetSystemDEPPolicy**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/getsystemdeppolicy.htm) function. Depending on the policy setting, an application can change the DEP setting for the current process by calling the [**SetProcessDEPPolicy**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/setprocessdeppolicy.htm) function.

## **Programming Considerations**

* An application can use the [**VirtualAlloc**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/virtualalloc.htm) function to allocate executable memory with the appropriate memory protection options. It is suggested that an application set, at a minimum, the PAGE\_EXECUTE memory protection option. After the executable code is generated, it is recommended that the application set memory protections to disallow write access to the allocated memory. Applications can disallow write access to allocated memory by using the [**VirtualProtect**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/virtualprotect.htm) function. Disallowing write access ensures maximum protection for executable regions of process address space. You should attempt to create applications that use the smallest executable address space possible, which minimizes the amount of memory that is exposed to memory exploitation.
* You should also attempt to control the layout of your application's virtual memory and create executable regions. These executable regions should be located in a lower memory space than non-executable regions. By locating executable regions below non-executable regions, you can help prevent a buffer overflow from overflowing into the executable area of memory.

## **Application Compatibility**

* Some application functionality is incompatible with DEP. Applications that perform dynamic code generation (such as Just-In-Time code generation) and do not explicitly mark generated code with execute permission may have compatibility issues on computers that are using DEP. Applications written to the Active Template Library (ATL) version 7.1 and earlier can attempt to execute code on pages marked as non-executable, which triggers an NX fault and terminates the application; for more information, see **SetProcessDEPPolicy**. Most applications that perform actions incompatible with DEP must be updated to function properly.
* A small number of executable files and libraries may contain executable code in the data section of an image file. In some cases, applications may place small segments of code (commonly referred to as thunks) in the data sections. However, DEP marks sections of the image file that is loaded in memory as non-executable unless the section has the executable attribute applied.
* Therefore, executable code in data sections should be migrated to a code section, or the data section that contains the executable code should be explicitly marked as executable. The executable attribute, IMAGE\_SCN\_MEM\_EXECUTE, should be added to the **Characteristics** field of the corresponding section header for sections that contain executable code. For more information about adding attributes to a section, see the documentation included with your linker.

## **See Also**

* [Data Execution Prevention (TechNet)](Http://go.microsoft.com/FWLink/?LinkId=84122)  
  [How to Configure Memory Protection](Http://go.microsoft.com/FWLink/?LinkId=84124)  
  [KB Article 875352](Http://go.microsoft.com/FWLink/?LinkId=83993)

# GetProcessDEPPolicy Function---MSDN

* [This documentation is preliminary and is subject to change.]
* Gets the data execution prevention (DEP) and DEP-ATL thunk emulation settings for the specified 32-bit process.
* BOOL WINAPI GetProcessDEPPolicy(
* \_\_in HANDLE *hProcess*,
* \_\_out LPDWORD *lpFlags*,
* \_\_out PBOOL *lpPermanent*
* );
* **Parameters**
* *hProcess*
* A handle to the process. PROCESS\_QUERY\_INFORMATION privilege is required to get the DEP policy of a process.
* *lpFlags*
* A **DWORD** that receives one or more of the following flags.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| 0 | DEP is disabled for the specified process. |
| PROCESS\_DEP\_ENABLE 0x00000001 | DEP is enabled for the specified process. |
| PROCESS\_DEP\_DISABLE\_ATL\_THUNK\_EMULATION 0x00000002 | DEP-ATL thunk emulation is disabled for the specified process. For information about DEP-ATL thunk emulation, see [**SetProcessDEPPolicy**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/setprocessdeppolicy.htm). |

* *lpPermanent*
* TRUE if DEP is enabled or disabled permanently for the specified process; otherwise FALSE. If *lpPermanent* is TRUE, the current DEP setting persists for the life of the process and cannot be changed by calling **SetProcessDEPPolicy**.
* **Return Value**
* If the function succeeds, it returns TRUE.
* If the function fails, it returns FALSE. To retrieve error values defined for this function, call **GetLastError**.
* **Remarks**
* **GetProcessDEPPolicy** is supported for 32-bit processes only. If this function is called on a 64-bit process, it fails with STATUS\_NOT\_SUPPORTED.
* **Requirements**

|  |  |
| --- | --- |
| **Client** | Requires Windows Vista SP1. |
| **Server** | Requires Windows Server 2008. |
| **Header** | Declared in Winbase.h; include Windows.h. |
| **Library** | Use Kernel32.lib. |
| **DLL** | Requires Kernel32.dll. |

# SetProcessDEPPolicy Function---MSDN

[This documentation is preliminary and is subject to change.]

Changes data execution prevention (DEP) and DEP-ATL thunk emulation settings for a 32-bit process.

BOOL WINAPI SetProcessDEPPolicy(

\_\_in DWORD *dwFlags*

);

**Parameters**

*dwFlags*

A **DWORD** that can be one or more of the following values.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| 0 | If the DEP system policy is OptIn or OptOut and DEP is enabled for the process, setting *dwFlags* to 0 disables DEP for the process. |
| PROCESS\_DEP\_ENABLE 0x00000001 | Enables DEP permanently on the current process. After DEP has been enabled for the process by setting PROCESS\_DEP\_ENABLE, it cannot be disabled for the life of the process. |
| PROCESS\_DEP\_DISABLE\_ATL\_THUNK\_EMULATION 0x00000002 | Disables DEP-ATL thunk emulation for the current process, which prevents the system from intercepting NX faults that originate from the Active Template Library (ATL) thunk layer. For more information, see the Remarks section. This flag can be specified only with PROCESS\_DEP\_ENABLE. |

**Return Value**

If the function succeeds, it returns TRUE.

If the function fails, it returns FALSE. To retrieve error values defined for this function, call **GetLastError**.

**Remarks**

**SetProcessDEPPolicy** overrides the system DEP policy for the current process. The system DEP policy setting must be OptIn or OptOut. If the system DEP policy is AlwaysOff or AlwaysOn, **SetProcessDEPPolicy** returns an error. After DEP is enabled for a process, subsequent calls to **SetProcessDEPPolicy** are ignored.

**SetProcessDEPPolicy** is supported for 32-bit processes only. If this function is called on a 64-bit process, it fails with STATUS\_NOT\_SUPPORTED.

Applications written to ATL 7.1 and earlier can attempt to execute code on pages marked as non-executable, which triggers an NX fault and terminates the application. DEP-ATL thunk emulation allows an application that would otherwise trigger an NX fault to run with DEP enabled. For information about ATL versions, see [ATL and MFC Version Numbers](http://go.microsoft.com/fwlink/?LinkId=95266).

If DEP-ATL thunk emulation is enabled, the system intercepts NX faults, emulates the instructions, and handles the exceptions so the application can continue to run. If DEP-ATL thunk emulation is disabled by setting PROCESS\_DEP\_DISABLE\_ATL\_THUNK\_EMULATION for the process, NX faults are not intercepted, which is useful when testing applications for compatibility with DEP.

The following table summarizes the interactions between system DEP policy, DEP-ATL thunk emulation, and **SetProcessDEPPolicy**. To get the system DEP policy setting, use the [**GetSystemDEPPolicy**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/memory/base/getsystemdeppolicy.htm) function.

|  |  |  |  |
| --- | --- | --- | --- |
| **System DEP policy** | **DEP behavior** | **DEP\_ATL thunk emulation behavior** | **SetProcessDEPPolicy behavior** |
| AlwaysOff  0 | Disabled for the operating system and all processes. | Not applicable. | Returns an error. |
| AlwaysOn  1 | Enabled for the operating system and all processes. | Disabled. | Returns an error. |
| OptIn  2  Default configuration for Windows client versions. | Enabled for the operating system and disabled for nonsystem processes. Administrators can explicitly enable DEP for selected executable files. | Not applicable. | DEP can be enabled for the current process.  If DEP is enabled for the current process, DEP-ATL thunk emulation can be disabled for that process. |
| OptOut  3  Default configuration for Windows Server versions. | Enabled for the operating system and all processes. Administrators can explicitly disable DEP for selected executable files. | Enabled. | DEP can be disabled for the current process.  If DEP is disabled for the current process, DEP-ATL thunk emulation is automatically disabled for that process. |

**Requirements**

|  |  |
| --- | --- |
| **Client** | Requires Windows Vista SP1. |
| **Server** | Requires Windows Server 2008. |
| **Header** | Declared in Winbase.h; include Windows.h. |
| **Library** | Use Kernel32.lib. |
| **DLL** | Requires Kernel32.dll. |

# SafeSEH原理及绕过技术浅析

 作者：magictong

时间：2012年3月16日星期五

**摘要：**主要介绍SafeSEH的基本原理和SafeSEH的绕过技术，重点在原理介绍。

**关键词：**SafeSEH；绕过技术；异常处理

**目录**

## **前言**

设计SafeSEH保护机制的目的，以为了防止那种攻击者通过覆盖堆栈上的异常处理函数句柄，从而控制程序执行流程的攻击。

自Windwos XP SP2之后，微软就已经引入了SafeSEH技术。不过由于SafeSEH需要编译器在编译PE文件时进行特殊支持才能发挥作用，而xpsp2下的系统文件基本都是不支持SafeSEH的编译器编译的，因此在xpsp2下，SafeSEH还没有发挥作用（VS2003及更高版本的编译器中已经开始支持）。

从Vista开始，由于系统PE文件基本都是由支持SafeSEH的编译器编译的，因此从Vista开始，SafeSEH开始发挥他强大的作用，对于以前那种简单的通过覆盖异常处理句柄的漏洞利用技术，也就基本失效了。

## **SafeSEH的保护原理**

SafeSEH的基本原理很简单，即在调用异常处理函数之前，对要调用的异常处理函数进行一系列的有效性校验，如果发现异常处理函数不可靠（被覆盖了，被篡改了），立即终止异常处理函数的调用。不过SafeSEH需要编译器和系统双重支持，缺少一个则保护能力基本就丧失了。下面从两个方面来阐述怎样来实现SafeSEH。

### **（1）二进制层面**

首先我们先看看编译器做了些什么事情（通过启用链接选项/SafeSEH即可使编译出来的二进制文件具备SafeSEH功能，微软VS2003及以后的编译器已经默认支持）。在编译器生成二进制IMAGE的时候，把所有合法的SEH函数的地址解析出来，在IMAGE里生成一张合法的SEH函数表，用于异常处理时候进行严格的匹配检查。可以使用VC下面的dumpbin工具查看一个二进制文件的config信息，这样调用dumpbin /loadconfig file\_all\_path\_filename。

http://my.csdn.net/uploads/201204/27/1335513174_3174.jpg

输出的可能是下面这样（注：tttt.exe使用vs2005编译）：

Dump of file H:\Prj\_N\tttt\Release\tttt.exe

File Type: EXECUTABLE IMAGE

  Section contains the following load config:

            00000048 size

                   0 time date stamp

                0.00 Version

                   0 GlobalFlags Clear

                   0 GlobalFlags Set

                   0 Critical Section Default Timeout

                   0 Decommit Free Block Threshold

                   0 Decommit Total Free Threshold

            00000000 Lock Prefix Table

                   0 Maximum Allocation Size

                   0 Virtual Memory Threshold

                   0 Process Heap Flags

                   0 Process Affinity Mask

                   0 CSD Version

                0000 Reserved

            00000000 Edit list

            00403018 Security Cookie

            00402360 Safe Exception Handler Table

                   1 Safe Exception Handler Count

**Safe Exception Handler Table**

**Address**

**--------**

**004018A1  \_\_except\_handler4**

  Summary

        1000 .data

        1000 .rdata

        1000 .rsrc

        1000 .text

注意里面加粗标红的部分，这就是该二进制文件里面的SEH异常处理函数地址表。上面的输出实际上涉及如下的一个结构，是保存在二进制文件里面的一份配置表：

#include <windows.h>

extern DWORD\_PTR \_\_security\_cookie;  /\* /GS security cookie \*/

/\*

 \* The following two names are automatically created by the linker for any

 \* image that has the safe exception table present.

\*/

extern PVOID \_\_safe\_se\_handler\_table[]; /\* base of safe handler entry table \*/

extern BYTE  \_\_safe\_se\_handler\_count;  /\* absolute symbol whose address is

                                           the count of table entries \*/

typedef struct {

    DWORD       Size;

    DWORD       TimeDateStamp;

    WORD        MajorVersion;

    WORD        MinorVersion;

    DWORD       GlobalFlagsClear;

    DWORD       GlobalFlagsSet;

    DWORD       CriticalSectionDefaultTimeout;

    DWORD       DeCommitFreeBlockThreshold;

    DWORD       DeCommitTotalFreeThreshold;

    DWORD       LockPrefixTable;            // VA

    DWORD       MaximumAllocationSize;

    DWORD       VirtualMemoryThreshold;

    DWORD       ProcessHeapFlags;

    DWORD       ProcessAffinityMask;

    WORD        CSDVersion;

    WORD        Reserved1;

    DWORD       EditList;                   // VA

    DWORD\_PTR   \*SecurityCookie;

    PVOID       \*SEHandlerTable;

    DWORD       SEHandlerCount;

} IMAGE\_LOAD\_CONFIG\_DIRECTORY32\_2;

const IMAGE\_LOAD\_CONFIG\_DIRECTORY32\_2 \_load\_config\_used = {

    sizeof(IMAGE\_LOAD\_CONFIG\_DIRECTORY32\_2),

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    0,

    &\_\_security\_cookie,

    \_\_safe\_se\_handler\_table,

    (DWORD)(DWORD\_PTR) &\_\_safe\_se\_handler\_count

};

### **（2）系统层面**

基本过程如下（XP SP2和VISTA一样）。

加载准备过程：

加载PE文件时，定位和读出合法SEH函数表的地址（如果该IMAGE是不支持SafeSEH的，则这个SEH函数表的地址为0），并使用共享内存中的一个随机数加密。将加密后的SEH函数表地址，IMAGE的开始地址，IMAGE的长度，合法SEH函数的个数，作为一条记录放入ntdll（ntdll模块是进行异常分发的模块）的加载模块数据内存中。

异常发生后，异常处理过程如下（RtlDispatchException框架伪码）：

void RtlDispatchException(...)

{

if (exception record is not on the stack)

goto corruption;

if (handler is on the stack)

goto corruption;

if (RtlIsValidHandler(handler, process\_flags) == FALSE)

goto corruption;

// execute handler

RtlpExecuteHandlerForException(handler, ...)

...

}

RtlDispatchException()这个函数的检测主要分三步，首先检查异常处理节点是否在栈上，如果不在栈上程序将终止异常处理，其次检查异常处理句柄是否在栈上，如果在栈上程序将止异常处理，这两个检测可以防止那种在堆上伪造异常链和把shellcode放置在栈上的情况。最后检测handler的有效性，这才是SafeSEH的重点。

下面看一下RtlIsValidHandler的伪码（vista sp1）：

BOOL **RtlIsValidHandler**(handler)

{

if (handler is in an image)

{

**// 在加载模块的进程空间**

if (image has the **IMAGE\_DLLCHARACTERISTICS\_NO\_SEH** flag set)

return FALSE; **// 该标志设置，忽略异常处理，直接返回FALSE**

if (image has a SafeSEH table) **// 是否含有SEH表**

if (handler found in the table)

return TRUE; **// 异常处理handle在表中，返回TRUE**

else

return FALSE; **// 异常处理handle不在表中，返回FALSE**

if (image is a .NET assembly with the ILonly flag set)

return FALSE; **// .NET 返回FALSE**

// fall through

}

if (handler is on a non-executable page)

{

**// handle在不可执行页上面**

if (ExecuteDispatchEnable bit set in the process flags)

return TRUE; **// DEP关闭，返回TRUE；否则抛出异常**

else

raise ACCESS\_VIOLATION; // enforce DEP even if we have no hardware NX

}

if (handler is not in an image)

{

**// 在加载模块内存之外，并且是可执行页**

if (ImageDispatchEnable bit set in the process flags)

return TRUE; **// 允许在加载模块内存空间外执行，返回验证成功**

else

return FALSE; **// don't allow handlers outside of images**

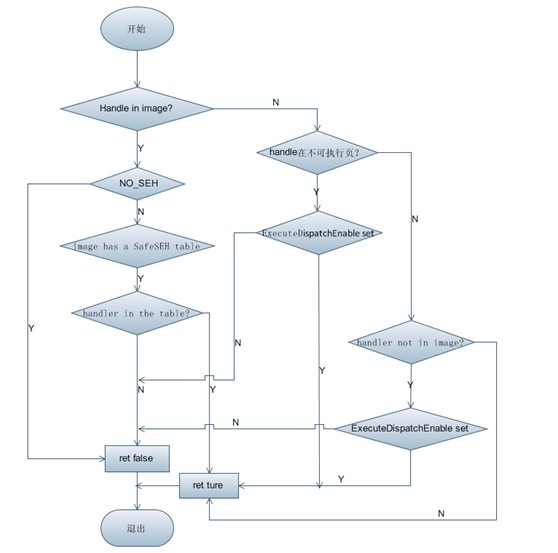
}

// everything else is allowed

return TRUE;

}

对上面的伪码的理解，请看代码注释和流程图：



伪码里面的ExecuteDispatchEnable和ImageDispatchEnable位标志是内核KPROCESS结构的一部分，这两个位用来控制当异常处理函数在不可以执行内存或者不在异常模块的映像（IMAGE）内时，是否执行异常处理函数。这两个位的值可以在运行时修改，不过默认情况下如果进程的DEP被关闭，则这两个位置1，如果进程的DEP是开启状态，则这两个位被置0。

在进程的DEP是开启的情况，有两种异常处理函数被异常分发器认为是有效的：

（a）异常处理函数在进程映像的SafeSEH表中，并且没有NO\_SEH标志。

（b）异常处理函数在进程映像的可执行页，并且没有NO\_SEH标志，没有SafeSEH表，没有.NET的ILonly标志。

在进程的DEP关闭的情况下，有三种情况异常处理函数被异常分发器认为是有效的：

（a）异常处理函数在进程映像的SafeSEH表中，并且没有NO\_SEH标志。

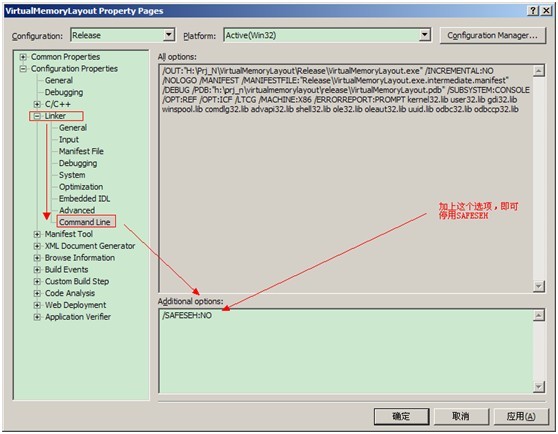
（b）异常处理函数在进程映像的可执行页，并且没有NO\_SEH标志，没有SafeSEH表，没有.NET的ILonly标志。

（c）异常处理函数不在当前进程的映像里面，但是不在当前线程的堆栈上。

这里的伪码是非常简单的，还有很多值得探讨的问题，譬如ntdll里面，当前异常处理句柄是怎么和SEH表进行对比的等等，不过这暂时不列入讨论。

## **怎么关掉编译器的SafeSEH支持**

虽然我不知道你为什么要这么做，而且我觉得很疯狂，但是方法还是有的，在编译器的属性框Liker|CommandLine的Additional options 加入/SAFESEH:NO即可，见下图。



## **怎样检测一个PE文件是否启用了SafeSEH**

前面介绍过数据目录里面的一个结构（IMAGE\_LOAD\_CONFIG\_DIRECTORY），该结构的成员SEHandlerTable是指向合法SEH处理程序地址列表的指针，成员SEHandlerCount是数目。而IMAGE\_LOAD\_CONFIG\_DIRECTORY这个结构体只有/SAFESEH选项设置了才存在，因此，就可以根据它来判断PE文件是否加了/SAFESEH链接选项。这个结构在PE中的偏移由PE附加头IMAGE\_DATA\_DIRECTORY 数组的第11项指定。

#define **IMAGE\_DIRECTORY\_ENTRY\_LOAD\_CONFIG**  10  // Load Configuration Directory

## **绕过方法简介**

（1）利用堆地址覆盖SEH结构绕过SafeSEH

上面讲过，在禁用DEP的进程中，异常分发器允许SEH handler位于除栈空间之外的非映像页面。也就是说我们可以把shellcode放置在堆中，然后通过覆盖SEH跳至堆空间以执行shellcode，这样即可绕过SafeSEH保护。

（2）利用没有启用SafeSEH保护的模块绕过SafeSEH

在介绍原理时讲过，在国内，目前大部分的PC都是安装的Windows XP，也就是说对于大部分Windows操作系统，其系统模块都没有受到SafeSEH保护，可以选用未开启SafeSEH保护的模块来利用，另外，现在还有很多VC6编译的软件，这些软件本身和自带的dll文件，都是可能没有SafeSEH保护的。这时就可以使用它里面的指令作为跳板来绕过SafeSEH。

（3）利用加载模块之外的地址绕过SafeSEH

同样是根据SafeSEH的原理可知，对于加载模块之外的地址，SafeSEH同样是不进行有效性检测的（当然假设是DEP是关闭的，或者DEP已经被绕过）。

注：绕过方法这里没有细讲，原因是没有找到很好的例子，在《0day安全：软件漏洞分析技术》上面有自己书籍作者自己写的例子。以后这块再详说。

## **参考文献**

[1] Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP

<http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx>（可以列入翻译计划）

[2] SafeSEH笔记<http://pstgroup.blogspot.com/2007/08/tipssafeseh.html>

[3] /SAFESEH (Image has Safe Exception Handlers)

<http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx>

[4] 0day安全：软件漏洞分析技术（第二版）

[5] Bypassing Browser Memory Protections

[6] pecoff\_v8

# [为什么使用剪切板时都用GlobalAlloc分配内存](http://blog.csdn.net/wenzhou1219/article/details/17693241)

我在使用剪切板时，发现通用的都是使用GlobalAlloc来分配内存，我就想不是说在Win32中GlobalAlloc和LocalAlloc是一样的那为什么不用LocalAlloc呢，原谅我的好奇心吧，对此我研究了一番，如果你也有此疑问，可以看一看。

先看一看关于GlobalAlloc和LocalAlloc的由来，这里有前人做了比较好的解释，这里为了保证博客连贯性，直接截取过来([原文链接](http://www.cnblogs.com/tekkaman/archive/2011/04/22/2025042.html))

16位windows用一个全局堆和局部堆来管理内存，每一个应用程序或dll装入内存时，**代码段被装入全局堆**，而系统又为每个实例从全局堆中分配了一个64kb的数据段作为该**实例的局部堆**，用来存放应用程序的堆栈和所有全局或静态变量。而LocalAlloc/GlobalAlloc就是分别用于在局部堆或全局堆中分配内存。

由于每个进程的局部堆很小，所以在局部堆中分配内存会受到空间的限制。但**这个堆是每个进程私有的**，相对而言分配数据较安全，数据访问出错不至于影响到整个系统。  而**在全局堆中分配的内存是为各个进程共享的**，每个进程只要拥有这个内存块的句柄都可以访问这块内存，但是每个全局内存空间需要额外的内存开销，造成分配浪费。而且一旦发生严重错误，可能会影响到整个系统的稳定。

不过在Win32中，每个进程都只拥有一个**缺省的私有堆**，它只能被当前进程访问。应用程序也不可能直接访问系统内存。所以在Win32中全局堆和局部堆都指向进程的缺省堆。用LocalAlloc/GlobalAlloc分配内存没有任何区别。甚至LocalAlloc分配的内存可以被GlobalFree释放掉。所以在Win32下编程，无需注意Local和Global的区别。

结合上面的解释可知道，在Windows早期中的跨进程共享数据是通过将数据内存分配在共享内存(全局堆)中，然后使Clipboard拥有它来保证跨进程通信的。

那么刚刚说道GlobalAlloc和LocalAlloc分配的内存都是在进程的堆上，那么他们是如何来完成进程间共享数据的呢。这个只能依靠Clipboard函数了。事实上这个涉及到虚拟内存的内容(如果你对虚拟内存不了解的话，可以看一下我的[这篇博文](http://blog.csdn.net/wenzhou1219/article/details/17659485)中的第一部分内容)，在《Windows核心编程》书中讲到进程的地址空间划分，其中提到进程的内核模式分区是[**操作系统**](http://lib.csdn.net/base/operatingsystem)代码的驻地，其中的所有东西为所有进程所有，再结合MSDN中SetClipboardData中描述"After**SetClipboardData** is called, the system owns the object identified by the hMem parameter. The application can read the data, but must not free the handle or leave it locked.If the hMem parameter identifies a memory object, the object must have been allocated using the **GlobalAlloc** function with the **GMEM\_MOVEABLE** and **GMEM\_DDESHARE** flags. "那么我们很自然会想到当调用SetClipboardData后，系统在虚拟内存中移动要共享的内存到进程的内核模式分区中托管，这也是为什么一定要指明GMEM\_MOVABLE的原因。

聪明的小伙伴你一定会问那为什么一定要指定GMEM\_DDESHARE 参数呢，我们查看MSDN中GlobalAlloc中关于GMEM\_SHARE和GMEM\_DDESHARE参数的描述，“This flag is provided primarily for compatibility with 16-bit Windows. However, this flag may be used by some applications to enhance the performance of DDE operations and therefore can be specified if the memory is to be used for DDE. .”显然这里提供GMEM\_SHARE和GMEM\_DDESHARE参数只是为了兼容以前的程序编译，事实上你尽管把SHARE参数去掉吧，没什么问题，一切运行如初。

那么，既然这样了，使用LocalAlloc应该没有什么问题吧。

事实上在如下的程序中[**测试**](http://lib.csdn.net/base/softwaretest)GlobalAlloc和LocalAlloc完全正常

**剪切和复制**

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/17693241) [copy](http://blog.csdn.net/wenzhou1219/article/details/17693241)

[print?](http://blog.csdn.net/wenzhou1219/article/details/17693241)[在CODE上查看代码片](https://code.csdn.net/snippets/138495)

1. case IDM\_EDIT\_CUT:
2. case IDM\_EDIT\_COPY:
3. if (!pText)
4. {
5. return (0);
6. }
7. #ifdef GLOBALFUNC
8. //将Text数据复制到Global分配的数据区
9. hGlobal = GlobalAlloc(GMEM\_MOVEABLE | GMEM\_ZEROINIT, (lstrlen(pText) + 1) \* sizeof(TCHAR));
10. pGlobal = GlobalLock(hGlobal);
11. lstrcpy(pGlobal, pText);
12. GlobalUnlock(hGlobal);
14. //将Global数据设为剪切板所有以实现跨进程使用
15. OpenClipboard(hwnd);
16. EmptyClipboard();
17. SetClipboardData(CF\_TCHAR, hGlobal);
18. CloseClipboard();
20. #else
21. //将Text数据复制到Global分配的数据区
22. hGlobal = LocalAlloc(LMEM\_MOVEABLE | LMEM\_ZEROINIT, (lstrlen(pText) + 1) \* sizeof(TCHAR));
23. pGlobal = LocalLock(hGlobal);
24. lstrcpy(pGlobal, pText);
25. LocalUnlock(hGlobal);
27. //将Global数据设为剪切板所有以实现跨进程使用
28. OpenClipboard(hwnd);
29. EmptyClipboard();
30. SetClipboardData(CF\_TCHAR, hGlobal);
31. CloseClipboard();
32. #endif
34. if(LOWORD(wParam) == IDM\_EDIT\_COPY)
35. {
36. return (0);
37. }
38. //对于剪切还要向下执行以清除Text数据
40. case IDM\_EDIT\_CLEAR:
41. if (pText)
42. {
43. free(pText);
44. pText = NULL;
45. }
46. InvalidateRect(hwnd, NULL, TRUE);
47. return (0);
48. case IDM\_EDIT\_PASTE:
49. #ifdef GLOBALFUNC
50. OpenClipboard(hwnd);
52. if (hGlobal = GetClipboardData(CF\_TCHAR))
53. {
54. pGlobal = GlobalLock(hGlobal);
56. if (pText)
57. {
58. free(pText);
59. pText = NULL;
60. }
61. pText = malloc(GlobalSize(hGlobal));
62. lstrcpy(pText, pGlobal);
63. InvalidateRect(hwnd, NULL, TRUE);
65. GlobalUnlock(hGlobal);
66. }
68. CloseClipboard();
69. #else
70. OpenClipboard(hwnd);
72. if (hGlobal = GetClipboardData(CF\_TCHAR))
73. {
74. pGlobal = LocalLock(hGlobal);
76. if (pText)
77. {
78. free(pText);
79. pText = NULL;
80. }
81. pText = malloc(LocalSize(hGlobal));
82. lstrcpy(pText, pGlobal);
83. InvalidateRect(hwnd, NULL, TRUE);
85. LocalUnlock(hGlobal);
86. }
88. CloseClipboard();
89. #endif
90. return (0);

那么，很明显这个问题的答案令人感到可笑，所谓剪切板必须使用GlobalAlloc的说法不过是前人遗留的写法而已，在现在的操作系统(至少的得是XP了吧)中使用GlobalAlloc和LocalAlloc什么差别也没有。

如果你问我要使用哪个函数，那还是使用GlobalAlloc吧，毕竟你拿不准是否有人还会在年代久远的windows系统上编译你的程序呢。但是有一点需要说明的是，千万不要再说必须使用GlobalAlloc了。

其实再大胆一点不用GMEM\_MOVABLE而使用GMEM\_FIXED参数分配内存，你会发现一切还是正常的，原来SetClipboardData对这个分配的内存类型都没有了检查,直接强制性的移动了。尽管是个古老的东西，在现在的Windows操作系统中剪切板还是非常的重要，但是微软将它的标准放的这么低来保证程序的兼容性，说不定这个里面藏有好几个漏洞呢，这个黑客们慢慢探讨吧。