缓冲区溢出

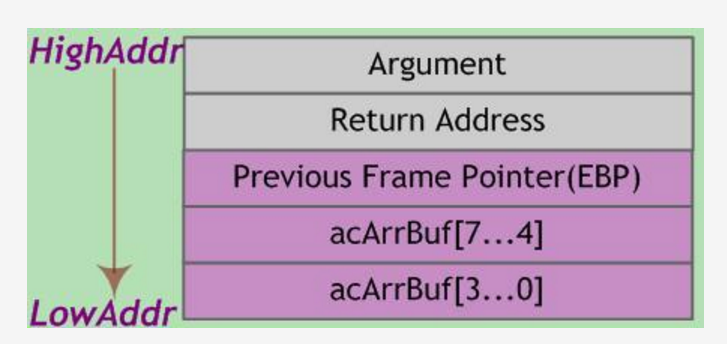
1.缓冲区溢出原理

缓冲区是一块连续的计算机内存区域，可保存相同数据类型的多个实例。缓冲区可以是栈(局部变量，函数的参数值)、堆(由new创建的对象)和静态数据区(全局或静态)。在C/C++语言中，通常使用字符数组和malloc/new之类内存分配函数实现缓冲区。溢出指数据被添加到分配给该缓冲区的内存块之外。缓冲区溢出是最常见的程序缺陷。

     栈帧结构的引入为高级语言中实现函数或过程调用提供直接的硬件支持，但由于将函数返回地址这样的重要数据保存在程序员可见的堆栈中，因此也给系统安全带来隐患。若将函数返回地址修改为指向一段精心安排的恶意代码，则可达到危害系统安全的目的。此外，堆栈的正确恢复依赖于压栈的EBP值的正确性，但EBP域邻近局部变量，若编程中有意无意地通过局部变量的地址偏移窜改EBP值(对于一个栈来说，寄存器EBP和ESP分别指向指向系统栈最上面一个[栈帧](http://baike.baidu.com/view/8128123.htm" \t "http://blog.csdn.net/ylyuanlu/article/details/_blank)的底部(高地址)和栈帧顶部（低地址）)，则程序的行为将变得非常危险。

     由于C/C++语言没有数组越界检查机制，当向局部数组缓冲区里写入的数据超过为其分配的大小时，就会发生缓冲区溢出。攻击者可利用缓冲区溢出来窜改进程运行时栈，从而改变程序正常流向，轻则导致程序崩溃，重则系统特权被窃取。

     例如，对于下图的栈结构：

IMG_256

     若攻击者用一个有意义的地址(否则会出现段错误)覆盖返回地址的内容，函数返回时就会去执行该地址处事先安排好的攻击代码。最常见的手段是通过制造缓冲区溢出使程序运行一个用户shell，再通过shell执行其它命令。若该程序有root或suid执行权限，则攻击者就获得一个有root权限的shell，进而可对系统进行任意操作。

     除通过使堆栈缓冲区溢出而更改返回地址外，还可改写局部变量(尤其函数指针)以利用缓冲区溢出缺陷。

2.缓冲区溢出实例（int = 4 byte）

 【示例1】改变函数的返回地址，使其返回后跳转到某个指定的指令位置，而不是函数调用后紧跟的位置。实现原理是在函数体中修改返回地址，即找到返回地址的位置并修改它。代码如下：

void foo(void){

int a, \*p;

p = (int \*)&a + 3;//让p指向main函数调用foo时入栈的返回地址

\*p += 12; //修改该地址的值，使其指向一条指令的起始地址

}

int main(void){

foo();

printf("First printf call\n");

printf("Second printf call\n");

return 0;

}

  编译运行，结果输出Second printf call，未输出First printf call。

栈

|  |  |  |
| --- | --- | --- |
| ESP=EBP | Return Address | 0x080483b8 |
| ESP=EBP-4 | EBP(main) | 0x080483a2 |
| EBP1=EBP-8;ESP=EBP1 | Params | 0x0804838a |
| ESP=EBP1-4 | Params | 0x08048390 |

其他

|  |  |
| --- | --- |
| 0x8048384 | Foo() |
| 0x804838a | p |
| 0x8048390 | a |
| 0x80483a2 | Main() |
| 0x80483b8 | printf("First printf call\n"); |
| 0x80483c4 | printf("Second printf call\n"); |

分析：

&a即为栈(ESP=EBP1-4=EBP-12)的指针，p为栈(ESP=EBP)的指针即为Return Address的地址

【示例2】暂存RunAway函数的返回地址后修改其值，使函数返回后跳转到Detour函数的地址；Detour函数内尝试通过之前保存的返回地址重回main函数内

int gPrevRet = 0; //保存函数的返回地址

void Detour(void){

int \*p = (int\*)&p + 2; //p指向函数的返回地址

\*p = gPrevRet;

printf("Run Away!\n");

}

int Runaway(void){

int \*p = (int\*)&p + 2;

gPrevRet = \*p;

\*p = (int)Detour;

return 0;

}

int main(void){

Runaway();

printf("Come Home!\n");

return 0;

}

编译运行后输出：

|  |
| --- |
| Run Away!  Come Home!  Run Away!  Come Home!  Segmentation fault |

栈

|  |  |  |
| --- | --- | --- |
| ESP=EBP | Return Address | 0x80483b7 |
| ESP=EBP-4; | EBP(main) | 0x80483b1 |
| EBP1=EBP-8;ESP=EBP1 | Param | 0x804838a |
| ESP=EBP1-4 | Retrun Address | 0x804839d |
| ESP=EBP1-8 | EBP(Runaway) | 0x8048391 |
| EBP2=EBP1-12;ESP=EBP2 | Param | 0x8048397 |

其他

|  |  |
| --- | --- |
| 0x8048384 | Detour() |
| 0x804838a | P1 |
| 0x8048391 | Runaway() |
| 0x8048397 | P2 |
| 0x804839d | Return 0; |
| 0x80483b1 | Main() |
| 0x80483b7 | printf("Come Home!\n") |

【示例3】越界访问造成死循环

void InfinteLoop(void){

unsigned char ucIdx, aucArr[10];

for(ucIdx = 0; ucIdx <= 10; ucIdx++)

aucArr[ucIdx] = 1;

}

在循环内部，当访问不存在的数组元素aucArr[10]时，实际上在访问数组aucArr所在地址之后的那个位置，而该位置存放着变量ucIdx。因此aucArr[10] = 1将ucIdx重置为1，然后继续循环的条件仍然成立，最终将导致死循环。

3.缓冲区溢出防范

 防范缓冲区溢出问题的准则是：确保做边界检查(通常不必担心影响程序效率)。不要为接收数据预留相对过小的缓冲区，大的数组应通过malloc/new分配堆空间来解决；在将数据读入或复制到目标缓冲区前，检查数据长度是否超过缓冲区空间。同样，检查以确保不会将过大的数据传递给别的程序，尤其是第三方COTS(Commercial-off-the-shelf)商用软件库——不要设想关于其他人软件行为的任何事情。

     若有可能，改用具备防止缓冲区溢出内置机制的高级语言(Java、C#等)。但许多语言依赖于C库，或具有关闭该保护特性的机制(为速度而牺牲安全性)。其次，可以借助某些底层系统机制或检测工具(如对C数组进行边界检查的编译器)。许多操作系统(包括Linux和Solaris)提供非可执行堆栈补丁，但该方式不适于这种情况：攻击者利用堆栈溢出使程序跳转到放置在堆上的执行代码。此外，存在一些侦测和去除缓冲区溢出漏洞的静态工具(检查代码但并不运行)和动态工具(执行代码以确定行为)，甚至采用grep命令自动搜索源代码中每个有问题函数的实例。

4.危险的库函数

 (1). gets

该函数从标准输入读入用户输入的一行文本，在遇到EOF字符或换行字符前，不会停止读入文本。即该函数不执行越界检查，故几乎总有可能使任何缓冲区溢出(应禁用)。

     gcc编译器下会对gets调用发出警告(the `gets' function is dangerous and should not be used)。

(2). Strcpy/strcat

该函数将源字符串复制到目标缓冲区，但并未指定要复制字符的数目。若源字符串来自用户输入且未限制其长度，则可能引发危险。规避的方法如下：

     1) 若知道目标缓冲区大小，则可添加明确的检查(不建议该法)：

if(strlen(szSrc) >= dwDstSize){

/\* Do something appropriate, such as throw an error. \*/

} else{

strcpy(szDst, szSrc);

}

     2) 改用strncpy函数：

strncpy(szDst, szSrc, dwDstSize-1);

szDst[dwDstSize-1] = '\0'; //Always do this to be safe!

     若szSrc比szDst大，则该函数不会返回错误；当达到指定长度(dwDstSize-1)时，停止复制字符。第二句将字符串结束符放在szDst数组的末尾。

3) 在源字符串上调用strlen()来为其分配足够的堆空间：

pszDst = (char \*)malloc(strlen(szSrc));

strcpy(pszDst, szSrc);

(3). strncpy/strncat

该对函数是strcpy/strcat调用的“安全”版本，但仍存在一些问题：

     1) strncpy和strncat要求程序员给出剩余的空间，而不是给出缓冲区的总大小。缓冲区大小一经分配就不再变化，但缓冲区中剩余的空间量会在每次添加或删除数据时发生变化。这意味着程序员需始终跟踪或重新计算剩余的空间，而这种跟踪或重新计算很容易出错。

     2) 在发生溢出(和数据丢失)时，strncpy和strncat返回结果字符串的起始地址(而不是其长度)。虽然这有利于链式表达，但却无法报告缓冲区溢出。

     3) 若源字符串长度至少和目标缓冲区相同，则strncpy不会使用NUL来结束字符串；这可能会在以后导致严重破坏。因此，在执行strncpy后通常需要手工终止目标字符串。

     4) strncpy还可复制源字符串的一部分到目标缓冲区，要复制的字符数目通常基于源字符串的相关信息来计算。这种操作也会产生未终止字符串。

     5) strncpy会在源字符串结束时使用NUL来填充整个目标缓冲区，这在源字符串较短时存在性能问题。

(4). sprintf

 该函数使用控制字符串来指定输出格式，该字符串通常包括"%s"(字符串输出)。若指定字符串输出的精确指定符，则可通过指定输出的最大长度来防止缓冲区溢出(如%.10s将复制不超过10个字符)。也可以使用"\*"作为精确指定符(如"%.\*s")，这样就可传入一个最大长度值。精确字段仅指定一个参数的最大长度，但缓冲区需要针对组合起来的数据的最大尺寸调整大小。

     注意，"字段宽度"(如"%10s"，无点号)仅指定最小长度——而非最大长度，从而留下缓冲区溢出隐患。

(5). scanf

     scanf系列函数具有一个最大宽度值，函数不能读取超过最大宽度的数据。但并非所有规范都规定了这点，也不确定是否所有实现都能正确执行这些限制。若要使用这一特性，建议在安装或初始化期间运行小[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/chdhust/article/details/_blank)来确保它能正确工作。

(6). streadd/strecpy

     这对函数可将含有不可读字符的字符串转换成可打印的表示。其原型包含在libgen.h头文件内，编译时需加-lgen [library ...]选项。

     char \*strecpy(char \*pszOut, const char \*pszIn, const char \*pszExcept);

     char \*streadd(char \*pszOut, const char \*pszIn, const char \*pszExcept);

     strecpy将输入字符串pszIn(连同结束符)拷贝到输出字符串pszOut中，并将非图形字符展开为[C语言](http://lib.csdn.net/base/c" \o "C语言知识库" \t "http://blog.csdn.net/chdhust/article/details/_blank)中相应的转义字符序列(如Control-A转为“\001”)。参数pszOut指向的缓冲区大小必须足够容纳结果字符串；输出缓冲区大小应为输入缓冲区大小的四倍(单个字符可能转换为\abc共四个字符)。出现在参数pszExcept字符串内的字符不被展开。该参数可设为空串，表示扩展所有非图形字符。strecpy函数返回指向pszOut字符串的指针。

     streadd函数与strecpy相同，只不过返回指向pszOut字符串结束符的指针。

     考虑以下代码：

int main(void){

char szBuf[20] = {0};

streadd(szBuf, "\t\n", "");

printf(%s\n", szBuf);

return 0;

}

     打印输出\t\n，而不是所有空白。

(7). strtrns

     该函数将pszStr字符串中的字符转换后复制到结果缓冲区pszResult。其原型包含在libgen.h头文件内：

     char \* strtrns(const char \*pszStr, const char \*pszOld, const char \*pszNew, char \*pszResult);

     出现在pszOld字符串中的字符被pszNew字符串中相同位置的字符替换。函数返回新的结果字符串。

     如下示例将小写字符转换成大写字符：

int main(int argc,char \*argv[]){

char szLower[] = "abcdefghijklmnopqrstuvwxyz";

char szUpper[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

if(argc < 2){

printf("USAGE: %s arg\n", argv[0]);

exit(0);

}

char \*pszBuf = (char \*)malloc(strlen(argv[1]));

strtrns(argv[1], szLower, szUpper, pszBuf);

printf("%s\n", pszBuf);12 return 0;

}

    以上代码使用malloc分配足够空间来复制argv[1]，因此不会引起缓冲区溢出