**缓存区溢出攻击实验（1）**

本实验预计分 3 个小实验来做，本文是第一个实验。

1. [缓存区溢出攻击实验（1）](http://blog.csdn.net/PTkin/article/details/50968063)
2. [缓存区溢出攻击实验（2）](http://blog.csdn.net/PTkin/article/details/50968066)
3. [缓存区溢出攻击实验（3）](http://blog.csdn.net/PTkin/article/details/50995130)

**背景介绍**

先看下百度百科里的 [缓冲区溢出攻击](http://baike.baidu.com/link?url=nnWSp3HwmPx8ju6m82fF4YJ5Icznaub92eaxvQ_DdA-unZ_sCtwrVXWwKEqAIJhJJB4EbzJS7v2ddpMLEFQGm_) ：

缓冲区溢出是指当计算机向缓冲区内填充数据位数时超过了缓冲区本身的容量，溢出的数据覆盖在合法数据上。理想的情况是：程序会检查数据长度，而且并不允许输入超过缓冲区长度的字符。但是绝大多数程序都会假设数据长度总是与所分配的储存空间相匹配，这就为缓冲区溢出埋下隐患。操作系统所使用的缓冲区，又被称为“堆栈”，在各个操作进程之间，指令会被临时储存在“堆栈”当中，“堆栈”也会出现缓冲区溢出。

大概了解一下缓冲区溢出的概念之后，需要知道一些关于堆栈的基础知识。

由于这些是计算机学科的基础，我在网上稍微搜了一下也没有找到讲得比较好的文章=。=考虑到对本实验有兴趣的基本都会是计算机相关学科的学习者，因此此处就不再补充关于堆栈的一些基础知识。

**实验环境**

* 系统

Windows 10 家庭中文版 (64 bit)

* IDE

[Code:：Blocks](http://www.codeblocks.org/)

* 编译器

[MinGW](http://www.mingw.org/)

下面先放源代码再来看实验内容。

**源代码**

#include <stdio.h>

#include<string.h>

void overflow(const char\* input) {

char buf[8];

printf("Virtual address of 'buf' = Ox%p\n", buf);

strcpy(buf,input);

}

void fun() {

printf("Function 'fun' has been called without an explicitly invocation.\n");

printf("Buffer Overflow attack succeeded!\n");

}

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

printf("Virtual address of 'overflow' = Ox%p\n",overflow);

printf("Virtual address of 'fun' = Ox%p\n",fun);

// Section 1 : normal

char input[]="AAAAAAAA";

// Section 2 : attcak

//char input[]="AAAAAAAAAAAAAAAAAAAA\xB9\x13\x40\x00";

overflow(input);

return 0;

}

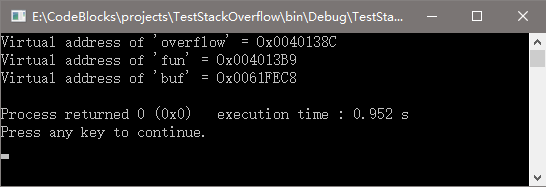
* 4

**实验目的**

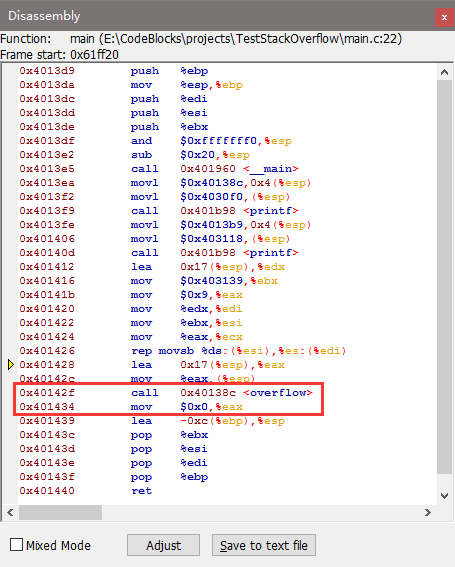
在 main 函数中调用 overflow 函数，在 overflow 函数的实现中有一段向数组写入数据的代码，在这段代码中填充过长的数据，令其缓冲区溢出，让我们的数据（实质是 fun 函数的入口地址）覆盖掉 overflow 执行完成之后的返回地址。这样做的结果就是，当 overflow 函数执行完之后本来应该返回 main 函数，结果返回到了本来没有被调用的 fun 函数的入口，执行了 fun 函数，缓冲区溢出攻击成功。

**实验步骤**

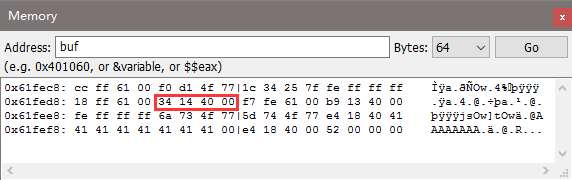
1. 安装好 Code::Blocks （以下简称 CB）与 MinGW，配置环境变量，打开 CB，设置 Compiler 为 GNU GCC Compiler，设置 Debugger 的 Executable path 为 ...\MinGW\bin\gdb.exe。
2. 新建 project，输入源代码，build，run，我们可以得到 fun 函数的入口地址，本例中为 0x004013B9，运行结果如下：



1. 在 main 函数中的 overflow(input); 与 overflow 函数中的 strcpy(buf,input); 处打上断点，重新运行程序。
2. 在 main 函数中的 overflow(input); 处暂停，打开菜单栏里的 Debug->Debugging windows->Disassembly，可以看到 main 函数执行时的汇编语言与逻辑地址，如下图所示：



1. 我们可以找到类似 call 0x40138c <overflow> 的语句，紧接着的下一句便是 overflow 函数执行结束之后返回 main 函数继续运行的语句，因此，在 overflow 执行时的堆栈中，函数执行完毕之后的返回地址应该为此处的地址。本例的地址为 0x401434。
2. 继续运行，在 overflow 函数中的 strcpy(buf,input); 处暂停，打开菜单栏里的 Debug->Debugging windows->Memory dump，在 Address 中输入 buf （overflow 中的缓冲区）进行搜索，得到此时栈空间中 buf 的地址与数据，如下图所示：



1. 观察该图，buf 存储的地址为 0x61fec8，而我们在其中寻找到在前几步得到的 overflow 执行完毕之后的返回地址 0x401434 如图中所示。距离 buf 的起始头一共 20 个字节。因此我们可以考虑向 buf 所在的地址写入 24 个字节，覆盖栈空间原有的数据，从而使 overflow 函数执行完毕之后返回我们写入的4个字节的地址，而不是返回 main 函数继续执行。
2. 将源代码中的语句

char input[]="AAAAAAAA";

* + 1

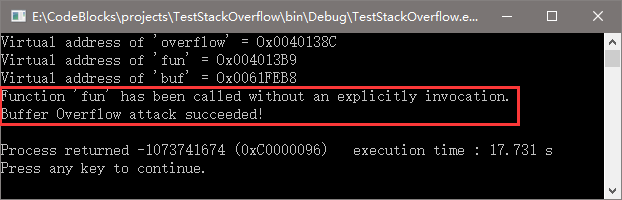
修改为

char input[]="AAAAAAAAAAAAAAAAAAAA\xB9\x13\x40\x00";

* + 1

其中修改后的字符串一共 24 个字节，最后 4 个字节是 fun 函数的入口地址。

1. 执行程序，运行结果如下所示：



攻击成功！