栈缓冲区溢出的理解与实践

使用

本文档以通关方式撰写，完成一关进入下一关，请将需要填写的内容写在空白处。

条件

这个练习用来帮助大家理解栈溢出原理。这需要 Linux 操作系统，32 位。

有两个方式：

第一，**安装虚拟机**（如 VirtualBox 或 VMWare），并安装 CentOS 6.10 操作系统，下载地址：

http://mirror.bit.edu.cn/centos/6.10/isos/i386/

文件：CentOS-6.10-i386-minimal.iso

如果安装时出现“unable to boot use a kernel appropriate for your CPU”错误，请配置虚拟处理器（CPU）启用 PAE/NX 特性。

系统安装后，以 root 用户执行：

```
yum install gcc
yum install gdb
```

GATE 1

用编辑器（vim 或其他）输入如下代码，命名为 stack_overflow.c 。
建议用虚拟机的同学将该程序在 Windows 平台下保存为 stack_overflow.c，拷贝到虚拟机中。

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

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

  int value = 5;
  char buffer_one[8], buffer_two[8];

  strcpy(buffer_one, "one");
  strcpy(buffer_two, "two");

  printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n",
buffer_two, buffer_two);
  printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n",
buffer_one, buffer_one);
  printf("[BEFORE] value is at %p and is %d (0x%08x)\n\n", &value,
value, value);

  printf("[STRCPY] copying %d bytes into buffer_two\n\n",
strlen(argv[1]));
  strcpy(buffer_two, argv[1]);

  printf("[AFTER] buffer_two is at %p and contains \'%s\'\n",
buffer_two, buffer_two);
  printf("[AFTER] buffer_one is at %p and contains \'%s\'\n",
buffer_one, buffer_one);
  printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value,
value);

}
```

理解以上代码，在空白处简要介绍上述代码功能。

此处是空白处：

程序设置了两个字符数组，并获取命令行参数 argv[1]。在对应的内存空间内进行字符串的拷贝，并比较拷贝前后、栈溢出前后地址的变化和其他变量的值的变化。

使用 debug 选项编译代码：

gcc -g -o so stack_overflow.c

按照下面命令执行程序，多执行几次该程序，比较输出结果，将某一次输出拷贝在空白处。（本文中需要拷贝输出的地方可以直接将输出粘贴在空白处，也可以截取屏幕中输出部分的图片粘贴到空白处。注意：如果采用截屏方式，请仅把输出部分截取出来，不要粘贴整个屏幕。）

./so hi

此处是空白处：

```
[xiabee@localhost Desktop]$ ./so hi
[BEFORE] buffer_two is at 0xbfcbbefc and contains 'two'
[BEFORE] buffer_one is at 0xbfcbbf04 and contains 'one'
[BEFORE] value is at 0xbfcbbf0c and is 5 (0x00000005)

[STRCPY] copying 2 bytes into buffer_two

[AFTER] buffer_two is at 0xbfcbbefc and contains 'hi'
[AFTER] buffer_one is at 0xbfcbbf04 and contains 'one'
[AFTER] value is at 0xbfcbbf0c and is 5 (0x00000005)
[xiabee@localhost Desktop]$
```

执行如下命令，将输出拷贝到空白处。

./so hiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii

此处是空白处：

```
[xiabee@localhost Desktop]$ ./so hiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiii
[BEFORE] buffer_two is at 0xbf91c01c and contains 'two'
[BEFORE] buffer_one is at 0xbf91c024 and contains 'one'
[BEFORE] value is at 0xbf91c02c and is 5 (0x00000005)

[STRCPY] copying 62 bytes into buffer_two

[AFTER] buffer_two is at 0xbf91c01c and contains 'hiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii'
[AFTER] buffer_one is at 0xbf91c024 and contains 'iiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiii'
[AFTER] value is at 0xbf91c02c and is 1768515945 (0x69696969)
Segmentation fault (core dumped)
[xiabee@localhost Desktop]$
```

通过这个程序，将你得到的结论写在空白处。

此处是空白处：

程序并没有修改 value 的值，但由于用户可控输入，构造特定的字符串造成栈溢出，可以改变 value 等变量的值，甚至使得晨曦崩溃。

GATE 2

用编辑器（vim 或其他）输入如下代码，命名为 ans_check.c 。

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

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[16];

  strcpy(ans_buf, ans);

  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;

  return ans_flag;

}

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

  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }

}
```

理解代码，在空白处简要介绍上述代码功能。

此处是空白处：

检测输入的 argv[1]是否正确，若 argv[1]的值等于"forty-two"，则输出"Right answer!\n"，否则输出"Wrong answer!\n"

编译代码：

```
gcc -g -o ans_check ans_check.c
```

按如下方式执行该程序，将控制台信息（含输入和输出）复制在空白处。

```
./ans_check
./ans_check yes
./ans_check forty-two
./ans_check 1111111111              #注:10 个 1   (32 位计算机)
./ans_check 1111111111111111       #注:16 个 1   (32 位计算机)
./ans_check 11111111111111111      #注:17 个 1   (32 位计算机)
```

此处是空白处：

```
[xiabee@localhost Desktop]$ ./ans_check
Usage: ./ans_check <answer>
[xiabee@localhost Desktop]$ ./ans_check yes
Wrong answer!
[xiabee@localhost Desktop]$ ./ans_check forty-two
Right answer!
[xiabee@localhost Desktop]$ ./ans_check 1111111111
Wrong answer!
[xiabee@localhost Desktop]$ ./ans_check 1111111111111111
Wrong answer!
[xiabee@localhost Desktop]$ ./ans_check 11111111111111111
Right answer!
[xiabee@localhost Desktop]$
```

静静地思考 2 分钟，**简要**解释一下最后一个命令的结果，填写在空白处。

此处是空白处：

ans_buf 长度为 16，程序在 strcmp(ans_buf, "forty-two")时，若给定的字符串长度大于 16，ans_buf 将溢出，覆盖 ans_flag 的值。即当我们输入 17 个 1 时，函数返回的 ans_flag 被我们覆盖为”1”，在 main 函数中 if 条件成立，输出”Right answer!”

GATE 3

为了具体分析发生结果的细节，使用 gdb 进行检查，输入如下命令：

```
gdb ans_check -q
list 1
list
list
break 10   #注：断点放在 strcpy 所在行，根据程序行号调整
break 15   #注：断点放在该函数 return 语句所在行，根据程序行号调整
run 1111111111111111 #注:17 个 1
```

将最后一行命令的结果填写到空白处。

此处是空白处：

Starting program: /home/xiabee/Desktop/ans_check 1111111111111111
Breakpoint 1, check_answer (ans=0xbffff635 '1' <repeats 17 times>)
    at ans_check.c:10
10   strcpy(ans_buf, ans);

大家熟知，断点位置是程序尚未执行的位置。因此，程序当前在 strcpy 所在行，但并未执行该行指令。检查下面两个变量的值，将结果粘贴到空白处。

```
x/s ans_buf
x/x &ans_flag
```

此处是空白处：

0xbffff4fc: "@\203\004\b\026"
0xbffff50c: 0x00

在 gdb 中输入命令 `c <enter>`，继续执行程序。

在这个断点，strcpy 已经执行完毕，再次检查变量并将输出结果拷贝在空白处。

```
x/s ans_buf
x/x &ans_flag
```

此处是空白处：

0xbffff4fc: '1' <repeats 17 times>
0xbffff50c: 0x31

由此可见，栈中的缓冲区溢出改变了条件变量的值。C 语言中，任何非零值都是 true，因此，覆盖后的字符'1'（0x31）表示 true。
（输入 quit 退出 gdb）

GATE 4

简单修改代码，将 ans_flag 和 ans_buf 两个变量声明交换位置，编译后运行：

```
./ans_check 1111111111111111      #注:17 个 1
```

将输出结果和解释写到空白处。

此处是空白处：

Right answer!
输入的 ans 长度大于 16，使得 ans_buf 溢出，覆盖了其后的变量 ans_flag，使得 ans_flag 值非零，而后续的程序仅在为真的时候修改 flag 的值，为假时并未改回，故此时 flag 依然非零，输出 Right。而交换变量声明位置，结果不变，说明编译器并没有按照申明顺序放置变量，进行了编译优化。

这类栈溢出属于“**溢出**(后覆盖其他)**变量**”类型。

思考 1 分钟：这类栈溢是否与程序设计有关？这类栈溢出的利用是否是通用方法？

此处是空白处：

这类栈溢出与程序设计有关，但编译器重新排列了变量顺序。这类栈溢出不是通用方法，常见的通用方法为覆盖返回地址，如 NSR 模式：即填充数据+shellcode+返回地址，或 RSN 模式：大量返回地址+shellcode+填充数据，或 AR 模式：将 shellcode 写入环境变量等

GATE 5

下面，我们将介绍一种普遍存在于所有程序中的栈溢出问题和利用，这种方法通过**修改函数返回地址**实现漏洞的利用。（“溢出地址”类型）

重新启动 gdb：

```
gdb ans_check -q
```

在第 26 行（调用 check_answer()函数的地方）设断点。

反编译 main 函数。

```
disass main
```

在 main 函数的反编译代码中，找到断点所在位置。将断点所在指令到 main 函数结尾的汇编代码复制在下面。

此处是空白处：

```
0x080484f7 <+49>:    mov    0xc(%ebp),%eax
0x080484fa <+52>:    add    $0x4,%eax
0x080484fd <+55>:    mov    (%eax),%eax
0x080484ff <+57>:    mov    %eax,(%esp)
0x08048502 <+60>:    call   0x8048484 <check_answer>
0x08048507 <+65>:    test   %eax,%eax
0x08048509 <+67>:    je     0x8048519 <main+83>
0x0804850b <+69>:    movl   $0x8048612,(%esp)
0x08048512 <+76>:    call   0x8048394 <puts@plt>
0x08048517 <+81>:    jmp    0x8048525 <main+95>
0x08048519 <+83>:    movl   $0x8048620,(%esp)
0x08048520 <+90>:    call   0x8048394 <puts@plt>
0x08048525 <+95>:    leave
0x08048526 <+96>:    ret
```

再将第 10 行和 15 行设为断点（`strcpy` 所在行和所在函数 `return`）。这样，该程序共有 3 个断点： 26 行 (pre-call), 10 行 (pre-strcpy)和 15 行 (pre-return).

在 gdb 中执行如下命令

run 11111111111111111　　#注:17 个 1

用以下指令检查栈寄存器和栈内容：

i r esp
x/32xw $esp

将输出粘贴在空白处。

此处是空白处：

esp　　0xbffff520　　0xbffff520

0xbffff520:　0x08048540　0x080483d0　0x0804854b　0x0060fff4
0xbffff530:　0x08048540　0x00000000　0xbffff5b8　0x00493d28
0xbffff540:　0x00000002　0xbffff5e4　0xbffff5f0　0xb7fff3d0
0xbffff550:　0x080483d0　0xffffffff　0x00475fc4　0x08048284
0xbffff560:　0x00000001　0xbffff5a0　0x00464e85　0x00476ab8
0xbffff570:　0xb7fff6b0　0x0060fff4　0x00000000　0x00000000
0xbffff580:　0xbffff5b8　0x78a852c4　0x153b05bb　0x00000000
0xbffff590:　0x00000000　0x00000000　0x00000002　0x080483d0

上述输出是栈在调用 `check_answer` 之前的内容和位置

输入 `c <enter>` 到下一个断点

使用下面指令再次检查栈信息，并粘贴输出到空白处。

```
i r esp
x/32xw $esp
```

此处是空白处：

esp　　0xbffff4e0　　0xbffff4e0

0xbffff4e0:　0x000000c2　0x0000fff4　0x0060e1d8　0x00000000
0xbffff4f0:　0xbffff590　0x080497b0　0xbffff508　0x08048340
0xbffff500:　0x00000016　0x080497b0　0xbffff538　0x00000000
0xbffff510:　0x0060e1d8　0x08048284　0xbffff538　0x08048507
0xbffff520:　0xbffff748　0x080483d0　0x0804854b　0x0060fff4
0xbffff530:　0x08048540　0x00000000　0xbffff5b8　0x00493d28
0xbffff540:　0x00000002　0xbffff5e4　0xbffff5f0　0xb7fff3d0
0xbffff550:　0x080483d0　0xffffffff　0x00475fc4　0x08048284

检查如下两个静态变量的地址和内容

```
x/s ans_buf
x/x &ans_flag
```

在空白处 B 记录的栈中找到上面两个变量内容，将对应地址的字体加粗。同时，将返回地址的字体加粗。

最后，继续执行程序到下一个断点 `c <enter>`，使用如下命令检查堆栈，将内容粘贴到空白处，将 `ans_buf`、`ans_flag` 和返回地址的字体加粗。

```
i r esp
x/32xw $esp
```

此处是空白处：

0xbffff4fc: "@\203\004\b\026"
0xbffff50c: 0x00

0xbffff4fc: '1' <repeats 17 times>
0xbffff50c: 0x31
这里可以看到， ans_flag 变量被改写了，但返回地址没有变化。

GATE 6

重复 Gate 5 步骤，但这次找到覆盖返回地址的最小字符串长度。在 gdb 中使用下面命令输出你覆盖后的堆栈，并用加粗你覆盖的返回地址。

```
i r esp
x/32xw $esp
```

此处是空白处：

esp 0xbffff4c4 0xbffff4c4

0xbffff4c4: 0x00000000 0xbffff508 0x080484a3 0xbffff4ec
0xbffff4d4: 0xbffff739 0x0060e1d8 0x00000000 0xbffff580
0xbffff4e4: 0x080497b0 0xbffff4f8 0x31313131 0x31313131
0xbffff4f4: 0x32323131 0x32323232 0x00000032 0x33333333
0xbffff504: 0x33333333 0x34343333 **0x08048507** 0xbffff739
0xbffff514: 0x080483d0 0x0804854b 0x0060fff4 0x08048540
0xbffff524: 0x00000000 0xbffff5a8 0x00493d28 0x00000002
0xbffff534: 0xbffff5d4 0xbffff5e0 0xb7fff3d0 0x080483d0

GATE 7

下课了！