缓冲区溢出炸弹

实验目标

- 掌握函数调用时的栈帧结构
- 利用输入缓冲区的溢出漏洞，将攻击代码嵌入当前程序的栈帧中，使得程序执行我们所期望的过程

实验代码

- 解压文件，得到三个文件
  - makecookie：生成cookie
    - 例：./makecookie SA18225155 生成cookie
  - Bufbomb：可执行程序-攻击对象
  - Sendstring：字符格式转换

bufbomb程序

- Bufbomb中包含一个getbuf函数，该函数实现如下

```
1 int getbuf()
2 {
3         char buf[12];
4         Gets(buf);
5         return 1;
6 }
```

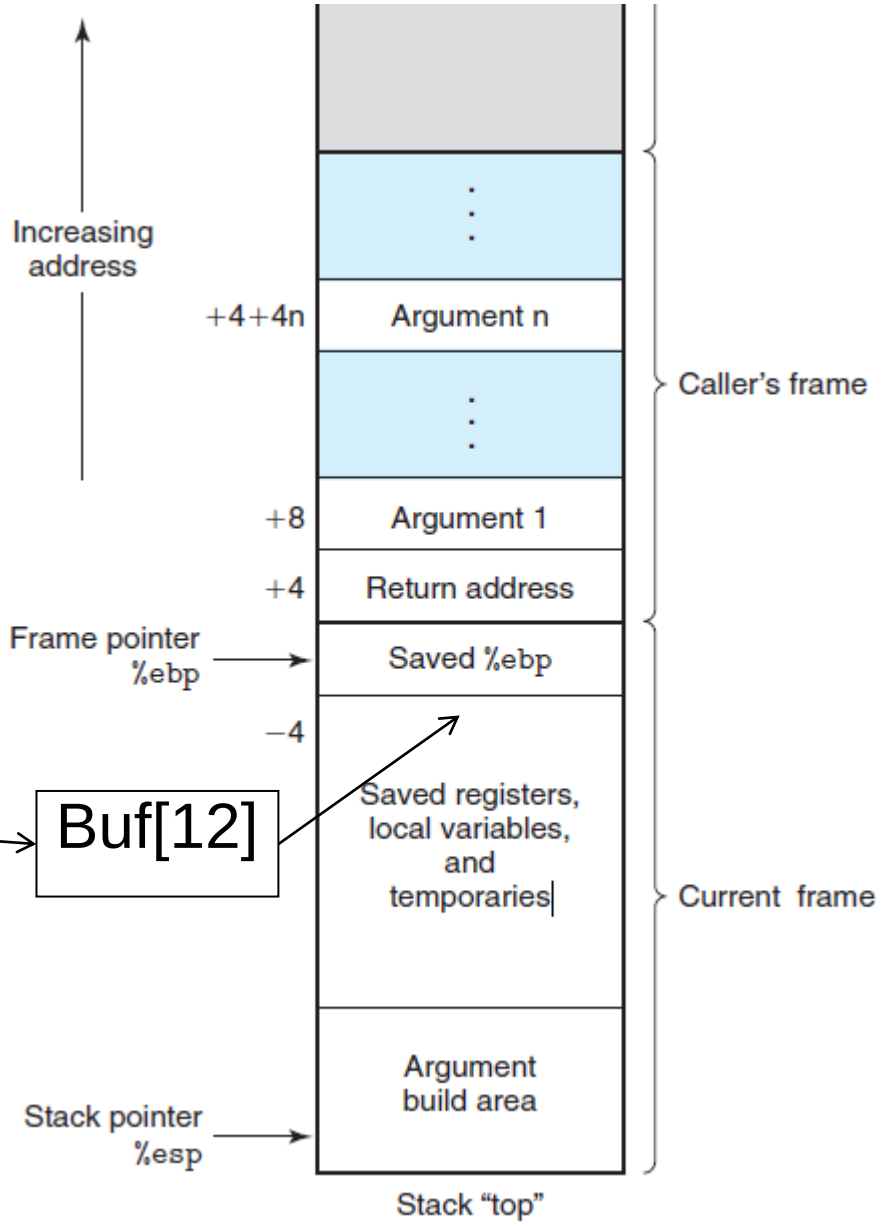
- 对buf没有越界检查（常见c编程错误）
- 超过11个字符将溢出

溢出

- 溢出的字符将覆盖栈帧上的数据
  - 特别的，会覆盖程序调用的返回地址
  - 赋予我们控制程序流程的能力
- 通过构造溢出字符串，程序将“返回”至我们想要的代码上

```
1 int getbuf()
2 {
3  char buf[12];
4  Gets(buf);
5  return 1;
6 }
```

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
lea    -0xc(%ebp),%eax
mov    %eax,(%esp)
call   0x8048e60 <Gets>
mov    $0x1,%eax
leave
ret
```



栈帧结构-可参考教材相关章节

字符构造

- 计算机系统中，字符以ASCII码表示 /存储
  - 例如，输入’1’，存储为’0x31’
  - 本实验也需要扩展的ASCII码（128~255）
- 为了构造所需要的地址或其他数据，我们需要逆反“字符->ASCII码”的过程
  - 采用代码包给出的　　sendstring工具。
  - 方法：./sendstring < exploit.txt > exploit-raw.txt
  - 其中exploit.txt保存目标数据（即空格分隔的ASCII码），exploit-raw.txt为逆向出的字符串

字符串输入

- 前述方法构造出的字符串按如下方式输入：
  - ./bufbomb -t SA18225155 < exploit-raw.txt
- 从标准输入设备输入，方式如下：
  - ALT+ASC码的十进制数（小键盘输入）
    - 注意，最后一个数字按下后与ALT键同时放开
    - 例，输入字符“1”为ALT+49
- 实验完成后提交exploit.txt文件

Level 0: Candle

- 主体函数

```
1 void test()
2 {
3  int val;
4  volatile int local = 0xdeadbeef;
5  entry_check(3); /* Make sure entered this function properly */
6  val = getbuf();
7  /* Check for corrupted stack */
8  if (local != 0xdeadbeef) {
9  printf("Sabotaged!: the stack has been corrupted\n");
10 }
11   else if (val == cookie) {
       ……..
18 }
```

- getbuf函数在test中被调用，当getbuf返回时继续执行第八行

Level 0: Candle

- Bufbomb中一个正常情况下不会被执行的函数：

```
void smoke()
{
    entry_check(0); /* Make sure entered this function properly */
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

- 我们的目标：在getbuf返回时跳到smoke函数执行

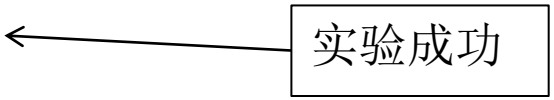
Level 0: Candle

- 思路
  - 通过调试得到我们输入的字符串首地址,并打印出该字符串作验证x/s $ebp-0xc
  - 找到函数smoke的地址 p/x &smoke
  - 用smoke函数的地址覆盖getbuf的返回地址

Level 1: Sparkler

- 另一函数

```
void fizz(int val)
{
    entry_check(1); /* Make sure entered this function properly */
    if (val == cookie) {
    printf("Fizz!: You called fizz(0x%x)\n", val);
    validate(1);
} else
    printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```



- 目标：“返回”到该函数并传送参数cookie
  - Cookie必须为自己学号生成
    - 格式示例如下：
    - *SA18225155使用“./makecookie SA08225155”生成*

Level 2: Firecracker（选做）

- 第三个函数

```
int global_value = 0;
void bang(int val)
{
        entry_check(2); /* Make sure entered this function properly */
        if (global_value == cookie) {
                printf("Bang!: You set global_value to 0x%x\n", global_value);
                validate(2);
        } else
            printf("Misfire: global_value = 0x%x\n", global_value);
        exit(0);
}
```

实验成功

- 目标：构造若干条指令，修改全局变量global_val，然后跳转到上述函数
  - 具体指令构造方法参加实验说明文档
  - 可通过execstack工具解除栈执行限制

Level 3: Dynamite（选做）

- void test()

```
2 {
3  int val;
4  volatile int local = 0xdeadbeef;
5  entry_check(3); /* Make sure entered this function properly */
6  val = getbuf();
7  /* Check for corrupted stack */
8  if (local != 0xdeadbeef) {
9  printf("Sabotaged!: the stack has been corrupted\n");
10 }
11  else if (val == cookie) {
12  printf("Boom!: getbuf returned 0x%x\n", val);
13  validate(3);
14 }
15  else {
16  printf("Dud: getbuf returned 0x%x\n", val);
17 }
18 }
```

- 目标：函数正常返回时执行 第15行，我们要让函数执行第12行

一些说明

- Call 地址：返回地址入栈（等价于“Push %eip，mov 地址，%eip”；注意eip指向下一条尚未执行的指令）
- ret：从栈中弹出地址，并跳到那个地址（pop %eip）
- leave：使栈做好返回准备，等价于
  - mov %ebp，%esp
  - pop %ebp
- push： R[%esp]<--R[%esp]-4; M[R[%esp]]<--S
- pop: D<--M[R[%esp]]; R[%esp]<--R[%esp]+4;

指令构造方法示例

- pushl $0x89abcdef # Push value onto stack
- addl $17,%eax # Add 17 to %eax
- .align 4 # Following will be aligned on multiple of 4
- .long 0xfedcba98 # A 4-byte constant
- .long 0x00000000 # Padding
- 保存成example.s
- 然后
- unix> gcc -c example.s
- unix> objdump -d example.o > example.d

实验提交

- exploit.txt （ASCII码文件）