缓冲区溢出攻击

S4.1. 下列函数中，参数 a 和局部变量 x 的地址如何确定？换言之，程序在运行时如何定位它们？

```
void foo(int a)
{
   int x;
}
```

S4.2. 下列代码中的各个变量分别位于哪个内存区域？

```
int i = 0;
void func(char *str)
{
   char *ptr = malloc(sizeof(int));
   char buf[1024];
   int j;
   static int y;
}
```

S4.3. 请画出下列 C 函数的栈帧布局。

```
int bof(char *str)
{
   char buffer[24];
   strcpy(buffer,str);
   return 1;
}
```

S4.4. 一名学生建议改变栈的增长方向，让栈从低地址向高地址增长。这样，当前函数的局部缓冲区会位于其返回地址上方，向高地址溢出时便不会覆盖该返回地址。请评价这个方案。

S4.5. 在书中的易受攻击程序 stack.c 里，实际越界写发生于 strcpy()。因此有人认为程序会在 strcpy() 返回时跳到恶意代码，而不是等 foo() 返回。这个说法对吗？请解释。

S4.6. 缓冲区溢出示例已修复如下。这样安全吗？

```
int bof(char *str, int size)
{
    char *buffer = (char *) malloc(size);

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

S4.7. 一些学生已经正确构造了 `badfile`，并把 shellcode 放在文件末尾，但尝试不同返回地址时得到下列结果。为什么有些地址有效，有些无效？

```
buffer address : 0xbffff180
case 1 : retAddr = 0xbffff250 -> Able to get shell access
case 2 : retAddr = 0xbffff280 -> Able to get shell access
case 3 : retAddr = 0xbffff300 -> Cannot get shell access
case 4 : retAddr = 0xbffff310 -> Able to get shell access
case 5:  retAddr = 0xbffff400 -> Cannot get shell access
```

S4.8. 下列函数由特权程序调用。参数 `str` 完全由用户控制，最长 300 字节。调用时，`buffer` 起始地址为 `0xAABB0010`，保存返回地址的位置为 `0xAABB0050`。请构造输入字符串，使它复制到 `buffer` 后，`bof()` 返回时执行注入代码。无需写出 shellcode 的具体字节，但必须标明关键元素在字符串中的正确偏移。注意输入会经过 `strcpy()`，地址中的零字节是本题陷阱。

```
int bof(char *str)
{
   char buffer[24];
   strcpy(buffer,str);
   return 1;
}
```

S4.9. ★ ★ ★

本题研究如何利用堆缓冲区溢出执行恶意代码。下面给出一段执行序列，其中省略了部分代码。执行期间，堆上分配的 `buffer` 和节点 `p` 按图 1 布局。攻击者可控制最长 300 字节的 `user_input`，该输入会复制到 `buffer`。请溢出缓冲区，使程序执行到行 ❸ 返回时跳转到注入堆中的代码。假设堆可执行，保存返回地址的

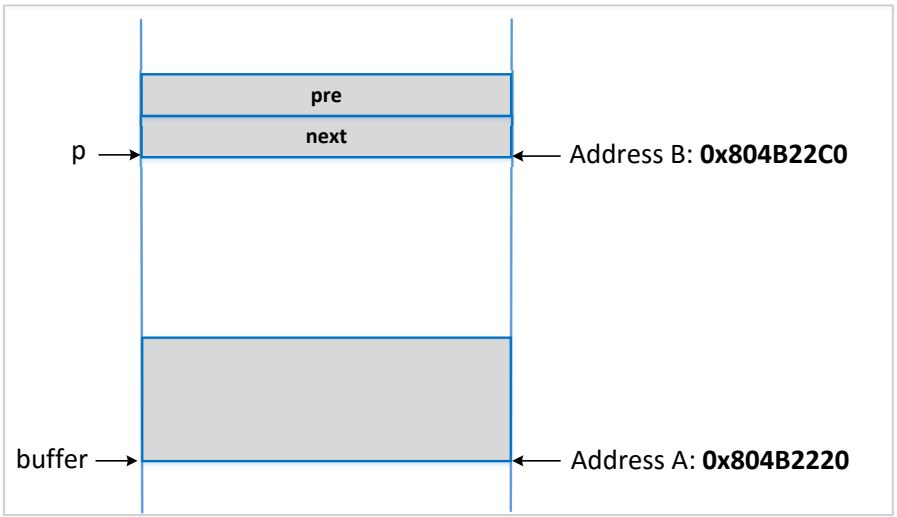

图 1: 问题图 S4.9.

位置是 `0xBBFFAACC`。

```
struct Node
{
  struct Node *next;
  struct Node *pre;
};


// The following is a snippet of a code execution sequence.

  struct Node *p = malloc(sizeof(struct Node));
  struct Node *q;
  char *buffer = malloc(100);

  /* Code omitted: Add Node p to a linked list */

  // There is a potential buffer overflow in the following
  strcpy(buffer, user_input);

  // remove Node p from the linked list
  q = p->pre;            ❶
  q->next = p->next;     ❷

  return;                ❸
```

提示：目标仍是把恶意代码起始地址写入 0xBBFFAACC 处的返回地址字段。与栈缓冲区溢出不同，堆上的连续越界写无法直接到达栈。应利用行 ❶ 和行 ❷ 对链表指针的操作，间接改写返回地址。

这是利用堆缓冲区溢出的一个简化示例。链表并不是易受攻击程序自身的数据结构，而是操作系统用于管理当前进程堆内存的元数据。由于链表也存放在堆上，攻击者可以通过溢出应用程序缓冲区篡改其中的值。操作系统随后操作被破坏的链表时，就可能改写函数返回地址并触发注入代码执行。

S4.10. 本题基于问题 S4.9.。若在 Node 开头新增一个整数字段，如下所示，请重新构造攻击。注意图 1 中，p 现在指向 next 字段之前 4 字节的位置。

```
struct Node
{
  int value;
  struct Node *next;
  struct Node *pre;
```

```
};
```

S4.11. 为什么 ASLR 会增加缓冲区溢出攻击的难度？

S4.12. ★

shellcode 需要取得字符串 `"/bin/sh"` 的地址。ASLR 使硬编码地址不再可靠。请说明 `exploit.c`（清单 4.2）中的 shellcode 如何动态取得该地址。

S4.13. 构造问题 S4.8. 的攻击字符串时，可选择多个值覆盖返回地址。可用的最小地址是多少？

S4.14. ★

下列函数运行在远程服务器中，参数 `str` 完全由用户控制，最长 300 字节。由于无法调试服务器，我们不知道缓冲区大小 X，只知道 $20 \leq X \leq 100$；同时已知 `buffer` 起始地址为 `0xAABBCC10`，缓冲区末尾与保存返回地址之间相隔 8 字节。

请构造一次输入，使字符串复制到 `buffer` 后，`bof()` 返回时执行你的代码。无需写出注入代码的具体字节，但必须给出关键元素的正确偏移，并保证在不知道 X 确切值时仍能成功。

```
int bof(char *str)
{
   char buffer[X];
   strcpy(buffer,str);
   return 1;
}
```