

# 《软件安全》2022 年秋季学期期末大作业

请同学们把答案写到 A4 空白纸上，完毕后，请扫描成一个 PDF 上传到云平台。上面需要写上自己的姓名、班级和学号，多页需要每页都需要写上姓名、班级和学号，并还需要写出页码。

请注意：

- 1、务必保证清晰，因为模糊导致无法批改，需要自行负责。
- 2、不可以互相讨论，互相抄袭。
- 3、提交时间为 2022 年 12 月 22 日晚 23 点 59 分，截止时间不可以补交。

## 一、填空题

- 1) 常见的字符串漏洞主要包括：\_\_\_\_\_。
- 2) 字符串“helloworld!”的长度为（ ）字节，存储该字符串需要（ ）字节。
- 3) 进程使用的内存可以按照功能分为四个部分：\_\_\_\_\_。
- 4) c和c++都支持几种不同类型的同步原语，包括\_\_\_\_\_。
- 5) 常见的内存管理错误有：\_\_\_\_\_。  
\_\_\_\_\_。

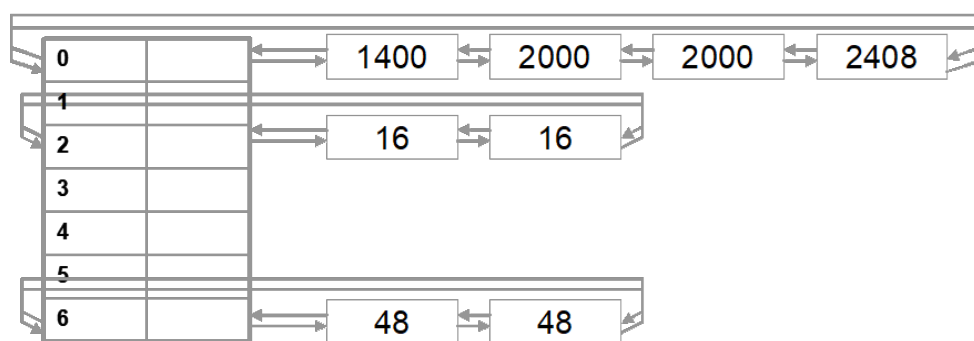
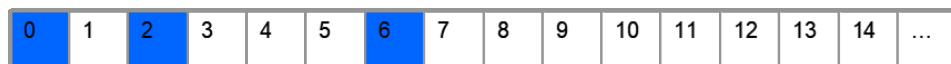
## 二、单项选择题

- 6) 以下哪项不存储在栈中？  
A. 函数参数   B. 全局变量   C. 局部（临时）变量   D. 调用函数的返回地址
- 7) 调用子函数 `function(4, 2)`时，以下语句顺序正确的是？  
① `call function`    ② `push 2`    ③ `push 4`  
①②③    B. ①③②    C. ③②①    D. ②③①
- 8) 缓冲区溢出覆写指针条件不包括：  
A. 缓冲区与目标指针必须分配在同一个段内  
B. 对于由上界限制的循环，缓冲区必须位于比目标指针更低的内存地址处  
C. 对于由上界限制的循环，缓冲区必须位于比目标指针更高的内存地址处

D. 该缓冲区必须是界限不充分的

9) 空闲链表结构如图所示，图中包含( )个空闲块，链表中的空闲块按照( )依次排列，对后备缓存链表的使用使得小块内存的分配速度( )

- A. 3, 从小到大, 加快                      B. 3, 从大到小, 变慢  
C. 8, 从小到大, 加快                      D. 8, 从大到小, 变慢



10) 要成功地利用双重释放漏洞，需满足的条件不包括：

- A. 被释放的内存块必须在内存中独立存在  
B. 被释放的内存块相邻的内存块必须是未分配的  
C. 被释放的内存块相邻的内存块必须是已分配的  
D. 被释放的内存块所被放入的筐（bin）必须为空

11) 关于整数错误缓解策略，下列说法错误的是（）

- A. 缓解策略还包括将每一个操作保护起来，实现起来代价较小  
B. 当一个整数值被赋给较小的整型时， Visual C++ NET 2003 编译器会生成一个警告  
C. 防止整数漏洞的第一条防线就是进行范围检查  
D. 使用强类型机制（strong typing）不能阻止溢出

12) 代码如下图所示，则该段代码输出为( )，是否发生整数截断？

```
int i = -3;
```

```

unsigned short u;

u = i;

printf("u = %hu\n", u);

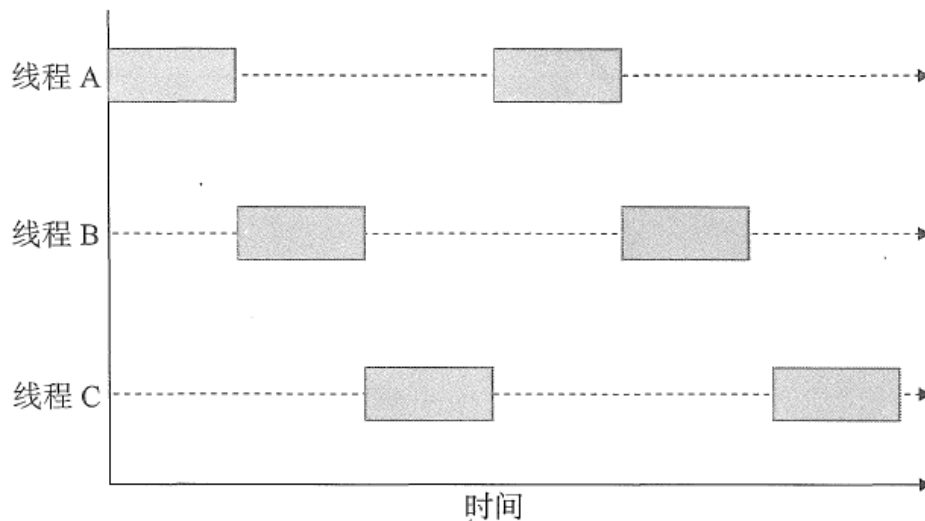
```

- A. -3; 是      B. -3; 否      C. 65533; 是      D. 65533; 否

13) 关于栈随机化, 下列说法错误的是 ( )

- A. 栈随机化使得很难预测栈上信息的位置, 包括返回地址和自动变量的位置  
 B. 栈随机化是通过向栈中插入随机的间隙实现的  
 C. 栈随机化加大了漏洞利用的难度  
 D. 栈随机化可以阻止漏洞利用

14) 并发和并行不等价, 所有的并行程序都是并发的, 但不是所有的并发程序都是并行的。则如图所示程序( )并发, ( )并行。



- A. 是; 是      B. 是; 不是      C. 不是; 是      D. 不是; 不是

15) 下列关于文件 I/O 的说法, 正确的是 ( )。

- A. 标准输入流[stdin]和标准输出流[stdout]的缓冲区类型不全是行缓冲。  
 B. fopen()函数的模式 ab+: 打开或创建二进制文件, 在文件的开始处写入。  
 C. fclose()函数关闭文件流时, 任何未写的缓存数据流将被丢弃。  
 D. C++使用 FILE 或 ifstream 来处理基于文件的输入输出流。

### 三、名词解释

- 16) ESP、EBP、EIP
- 17) 代码注入
- 18) 虚函数
- 19) 格式化输出
- 20) 并行度

### 四、简答题

21) 根据以下信息回答问题：

```

0xbffff9c0: 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36  1234567890123456
0xbffff9d0: 37 38 39 30 31 32 33 34 35 36 37 38 e0 f9 ff bf  789012345678....
0xbffff9e0: 31 c0 a3 23 fc ff bf b0 0b bb 27 fc ff bf b9 1f  1..#.....'.....
0xbffff9f0: fc ff bf 8b 15 23 fc ff bf cd 80 ff f9 ff bf 31  ....#.....'....1
0xbffffa00: 31 31 31 2f 75 73 72 2f 62 69 6e 2f 63 61 6c 0a  111/usr/bin/cal.
  
```

行号	地址	内容	描述
1	0xbffff9c0 - 0xbffff9cf	"1234567890123456"	用于密码（16 字节）和填充的存储区
2	0xbffff9d0 - 0xbffff9db	"789012345678"	额外的填充
3	0xbffff9dc	(0xbffff9e0)	新的返回地址
4	0xbffff9e0	xor %eax,%eax	将 eax 置零
5	0xbffff9e2	mov %eax,0xbffff9ff	用空指针终结指针数组
6	0xbffff9e7	mov \$0xb,%al	设置 execve() 函数的执行码
7	0xbffff9e9	mov \$0xbffffa03,%ebx	设置 ebx, 指向 execve() 的第 1 个参数
8	0xbffff9ee	mov \$0xbffff9fb,%ecx	设置 ecx, 指向 execve() 的第 2 个参数
9	0xbffff9f3	mov 0xbffff9ff,%edx	设置 edx, 指向 execve() 的第 3 个参数
10	0xbffff9f9	int \$80	执行 execve() 系统调用
11	0xbffff9fb	0xbffff9ff	被传给新程序的参数字符串数组
12	0xbffff9ff	"1111"	被变成 0x00000000, 用于终结指针数组, 同时用作第 3 个参数
13	0xbffffa03 - 0xbffffa0f	"/usr/bin/cal\0"	执行的命令

1) 函数新的返回地址是\_\_\_\_\_

2) xor %eax,%eax

mov %eax,0xbffff9ff

以上两行（第 4，5 行）的作用是\_\_\_\_\_

22) 根据以下代码回答问题:

```
void good_function(const char *str) {...}

void main(int argc, char **argv) {
    static char buff[BUFSIZE];

    static void (*funcPtr)(const char *str);

    funcPtr = &good_function;

    strncpy(buff, argv[1], strlen(argv[1]));

    (void)(*funcPtr)(argv[2]);
}
```

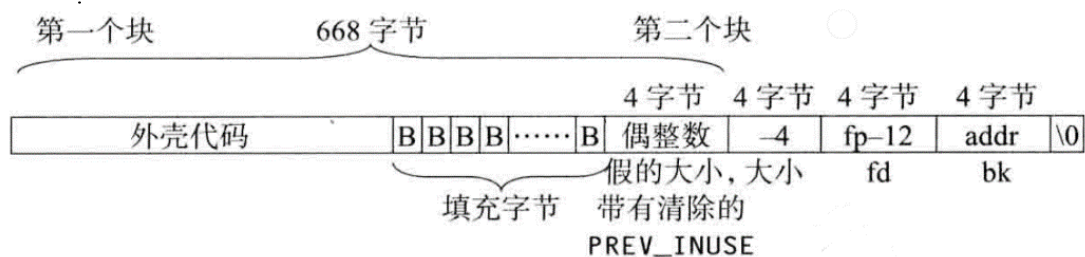
1) 当\_\_\_\_\_时, 就会发生缓冲区溢出。

2) 在以下代码的哪一行会发生任意内存写? ( )

1. void foo(void \* arg, size\_t len) {
2. char buff[100];
3. long val = ...;
4. long \*ptr = ...;
5. memcpy(buff, arg, len);
6. \*ptr = val;
7. ...
8. return;
9. }

A. 6    B. 5    C. 4    D. 3

23) 根据以下代码回答问题:



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

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

```

int main(int argc, char *argv[]) {
    char *first, *second, *third;
    first = malloc(666);
    second = malloc(12);
    third = malloc(12);
    strcpy(first, argv[1]);
    free(first);
    free(second);
    free(third);
    return(0);
}

```

- 1) 为了判断第二块内存是否处于空闲状态，free () 会检查\_\_\_\_\_
- 2) 第二块内存的大小域的值被修改为-4，此时 free() 确定的第三块内存的位置是\_\_\_\_\_
- 3) 在完成对 free()的调用后，将指针置为 NULL，无法阻止：( )
  - A. 大多数动态内存错误
  - B. 写已释放内存
  - C. 双重释放漏洞
  - D. 指针别名时出现的内存错误

24) 根据以下代码回答问题：

```

1  int *table = NULL;
2  int insert_in_table(int pos, int value){
3      if (!table) {
4          table = (int *)malloc(sizeof(int) * 100);
5      }
6      if (pos > 99) {
7          return -1;
8      }
9      table[pos] = value;
10     return 0;

```

11 }

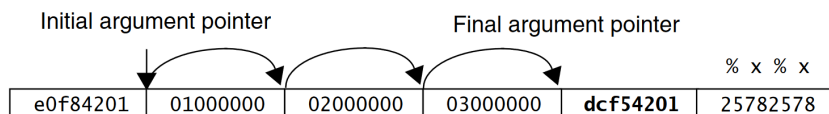
- 1) 代码中出现的漏洞是\_\_\_\_\_
- 2) 可以通过\_\_\_\_\_修复该漏洞。
- 3) 根据以下代码回答问题:

```
1 void getComment(unsigned int len, char *src) {  
2     unsigned int size;  
3     size = len - 2;  
4     char *comment = (char *)malloc(size + 1);  
5     memcpy(comment, src, size);  
6     return;  
7 }  
  
8 int _tmain(int argc, _TCHAR* argv[]) {  
9     getComment(len, "Comment ");  
10    return 0;  
11 }
```

当 getComment 的参数 len=1 时, 可能会发生\_\_\_\_\_错误。此时 size=\_\_\_\_\_。

- 25) 根据下图信息回答问题:

Memory:



\xdc - written to stdout      %x - advances argument pointer  
\xf5 - written to stdout      %x - advances argument pointer  
\x42 - written to stdout      %x - advances argument pointer  
\x01 - written to stdout      %s - outputs string at address specified  
   in next argument

- 1) 攻击者可以使用转换指示符%s 显示参数指针所指定的地址的内存, 将它作为一个 ASCII 字符串处理, 直至遇到一个空字符。参数指针可以使用转换指示符 %x 进行前向移动, 它所能移动的距离仅受\_\_\_\_\_所限制。
- 2) 当攻击者提供的格式字符串为\xdc\xf5\x42\x01%x%x%x%s, printf()将显

示\_\_\_\_\_，3 个转换指示符%x 应该使参数指针从格式字符串的开始位置\_\_\_\_\_

26) 根据代码回答下列问题:

```
01 mutex shared_lock;
02
03 void thread_function(int id) {
04     shared_lock.lock();
05     shared_data = id;
06     cout << "Thread " << id << " set shared value to "
07         << shared_data << endl;
08     usleep(id * 100);
09     cout << "Thread " << id << " has shared value as "
10         << shared_data << endl;
11     shared_lock.unlock();
12 }
```

- 1) 互斥量是一种锁机制，互斥量有两种可能的状态分别为\_\_\_\_\_和\_\_\_\_\_。
- 2) 当对已经锁定的互斥量执行 lock() 操作时，该函数会\_\_\_\_\_，从而确保一次只有一个线程可以运行花括号内的代码，使程序是线程安全的。
- 3) 以下说法中错误的是( )
  - A. 当两个或多个控制流以彼此都不可以继续执行的方式阻止对方时，就会发生死锁
  - B. 处理器速度不会影响死锁发生的可能性
  - C. 对于一个并发执行流的循环，如果其中在循环中的每个流都已经获得了导致在循环中随后的流悬停的同步对象，则会发生死锁
  - D. 死锁可能导致拒绝服务的安全漏洞

27) 分析下图回答下列问题:



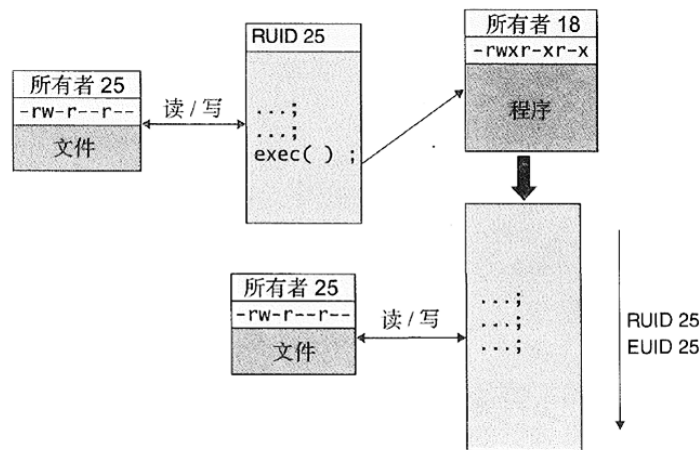


图 8.5 执行非 setuid 程序

- 1) 上图中运行的进程【RUID:25】执行 program 文件【所有者 UID:18】存储的进程映像，当程序执行时，他以的\_\_\_\_\_权限执行。
  - 2) 如果在 program 文件上设置了执行时设置用户 ID 模式，则程序执行时，其 RUID 为\_\_\_\_\_, EUID 为\_\_\_\_\_
  - 3) 设置用户 ID 模式的作用是: \_\_\_\_\_
  - 4) 若要永久放弃特权，则要在\_\_\_\_\_, 以使提升的特权不传递给新程序。
  - 5) 下列关于管理特权的说法，不正确的是（ ）。
- A. 非 root 的 setuid 和 setgid 程序通常用于执行有限或特定的任务。这些程序只能把 EUID 更改为 RUID 和 SSUID。
  - B. uid 程序不能以 root 身份运行，而是以更受限制的特权运行。
  - C. ping 程序是一个设置用户 ID 为 root 的程序。
  - D. 编写 setuid 程序时，要确保遵循最小特权原则。

## 五、分析题

- 28) 内存块结构如图所示，由“空闲块 1—已分配块—空闲块 2”组成，假设三个内存块大小相同，问：



- 1) 第 7 行和第 10 行的 P 标记位的值分别为\_\_\_和\_\_\_。
- 2) 按照块边界指示，已分配块范围是\_\_\_\_\_
- 3) 假设这三个内存块按如图顺序存在同一个筐内，并且筐内只有这三个块，当第二个块被分配给用户使用，所以 `unlink()`宏将从双链表中移除这个块，则解链前后第 3 行的 FD 分别指向\_\_\_\_\_
- 4) 根据以下代码回答问题：代码中所包含的 RTL 堆漏洞是\_\_\_\_\_。15 行对 `HeapAlloc()`的调用使得\_\_\_\_\_的地址被 shellcode 的地址所覆盖

---

```

1 typedef struct _unalloc {
2     PVOID fp;
3     PVOID bp;
4 } unalloc, *Punalloc;
5 char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
6 int _tmain(int argc, _TCHAR* argv[]) {
7     Punalloc h1;
8     HLOCAL h2 = 0;
9     HANDLE hp;
10    hp = HeapCreate(0, 0x1000, 0x10000);

```

---

---

```

11     h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12     HeapFree(hp, 0, h1);
13     h1->fp = (PVOID)(0x042B17C - 4);
14     h1->bp = shellcode;
15     h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16     HeapFree(hp, 0, h2);
17     return 0;
18 }

```

---

29) 根据以下代码回答问题

```

1  int main(int argc, char *const *argv) {
2      unsigned short int total;
3      total = strlen(argv[1]) + strlen(argv[2]) + 1;
4      char *buff = (char *) malloc(total);
5      strcpy(buff, argv[1]);
6      strcat(buff, argv[2]);
7  }

```

1) 请指出代码可能存在的问题：\_\_\_\_\_和

2) 代码如下所示，则当\_\_\_\_\_时，可以绕过范围检测导致缓冲区溢出错误，可以通过\_\_\_\_\_避免。

```

1  #define BUFF_SIZE 10
2  int main(int argc, char* argv[]){
3      int len;
4      char buf[BUFF_SIZE];
5      len = atoi(argv[1]);
6      if (len < BUFF_SIZE){
7          memcpy(buf, argv[2], len);
8      }
9  }

```

- 3) 整数范围检查形式如下: `if (off > len - sizeof(type-name))`。因为 `sizeof` 操作符返回的是一个无符号整型(`size_t`)，因此按照整数提升规则要求 `len - sizeof(类型名)` 应该按照无符号整型计算。当 `len` 小于 `sizeof` 的返回值时减法操作造成 ( ) (上/下) 溢并产生一个很大的 ( ) (正/负) 值，然后整数范围检查逻辑被绕过。
- 4) 代码如下图所示，在计算前会把比 `int` 小的整型提升为 `int`，则会发生整数截断的有\_\_\_\_\_

```
char cresult1, cresult2, c1, c2, c3;

c1 = 100;

c2 = 90;

c3 = -120;

cresult1 = c1 + c2 ;

cresult2 = c1 + c2 + c3;
```

- 5) 当计算一块内存区域的大小并调用 `calloc()`或其他内存分配函数来分配内存时可能会引起整数溢出，代码片段如下，如果函数返回一个 ( ) 需求的缓存，应用程序对分配的缓冲区的 ( ) 操作可能会越界，最后会导致基于堆的缓冲区溢出。

C: `p = calloc(sizeof(element_t), count);`

C++: `p = new ElementType[count];`

A.大于; 读

B.大于; 写

C.小于; 读

D.小于; 写