COMP2700 实验室 6 - 软件安全

在本实验中，我们将研究软件中的几种漏洞模式以及如何利用这些模式。为了更加具体，我们将研究一些包含这些漏洞模式的玩具 C 程序，但这些漏洞模式本身确实会出现在真实的（更复杂的）软件中。

标有 (*) 的练习是扩展练习，不包括在实验室中。

实验室设置

在本实验中，我们将再次使用实验 1 中的实验虚拟机。对于某些需要探测特定内存地址内容的练习，每次运行程序时，实验室虚拟机的默认设置可能会产生不同的变量内存位置。这是因为 Linux 中有一种称为 "**地址空间布局随机化**"（**ASLR**）的防御机制，它会随机化可执行文件的内存位置。你需要先禁用 ASLR，如下所示：

-以用户 admin2700 的身份　　登录并运行

```
sudo sysctl -w kernel.randomize_va_space=0
```

请注意，这会暂时禁用 ASLR，直到下次重启。因此，如果重新启动系统，需要再次重复上述步骤禁用 ASLR。

本实验室的 C 语言程序包含在 lab6.tar.gz 文件中，您可以使用以下命令下载并解压该文件：

```
wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab6.tar.gz
extract-lab lab6.tar.gz
```

如果上述链接无效，请使用以下替代链接：

```
wget https://cloudstor.aarnet.edu.au/plus/s/IgSC2mVQOS5BWr0/download -O lab6.tar.gz
extract-lab lab6.tar.gz
```

这将在 admin2700 的主目录下创建一个名为 lab6 的目录。

C 代码编译

下面的每个练习都附有 C 代码，在测试前需要先编译。我们将使用已安装在实验室虚拟机中的 gcc 编译器。在本实验中，我们将使用以下命令编译 C 程序：假设输入的 C 程序名为 "myprogram.c

```
gcc myprogram.c -o myprogram -m32 -fno-stack-protector
```

该命令将编译 myprogram.c，并生成输出文件 myprogram，即 myprogram.c 的可执行二进制文件。选项 -m32 将指示编译器生成 32 位架构的二进制文件（当前实验室虚拟机假定为 64 位架构）。这样可以更容易地重现某些攻击，尤其是与格式字符串漏洞相关的攻击。选项 -fno-stack-protector 会禁用堆栈保护，因此我们可以可靠地发起缓冲区溢出攻击。

缓冲区超限

练习 1.请看下面的程序，它是堆栈缓冲区溢出讲座中讨论过的一个程序的变体。

文件： ex1.c

```
void Win()
{
    printf("Well done!\n");
}
int main(int argc, char *argv[])
{
    int a = 0;
    char buffer[16];
    if(argc < 2){
        printf("Usage: %s <string>\n", argv[0]);
        exit(1);
    }
    printf("Address of a: %p\n", &a);
    printf("Address of buffer: %p\n", buffer);
    strcpy(buffer, argv[1]);
    printf("Value of a: %08x\n", a);
    if (a == 0x41424344) Win();
    else printf("Try again\n");
    return 0;
}
```

正如讲座中所讨论的，攻击者可以溢出 "缓冲区 "变量的值，从而覆盖变量 "a "的值。尝试以调用 Win() 函数的方式覆盖 a 的值。*提示：注意整数 a 在堆栈中的字节序，确保以正确的顺序覆盖其底层字节。*

练习 2.请看下面的代码

文件： ex2.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char a = 'a';
    char *buffer;

    if(argc < 2){
        printf("Usage: %s <string>\n", argv[0]);
        exit(1);
    }

    buffer=(char *) malloc(16);

    if(buffer == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }

    printf("Address of a: %p\n", &a);
    printf("Address of buffer: %p\n", buffer);

    strcpy(buffer, argv[1]);
    printf("Value of a: %c\n", a);

    free(buffer); 返

    回 0;
}
```

这与练习 1 中的 C 代码非常相似。不过，你会发现在练习 1 中有效的缓冲区溢出攻击在本练习中不起作用。这是为什么呢?

练习 3.请看下面的代码

文件： ex3.c

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

void print_message(char * msg)
{
  char secret[8] = "SECRET";

  char buffer[15]; strncpy（缓冲

  区，msg，15）；

  printf("Message: %s\n", buffer);
}

int main(int argc, char* argv[])
{
  if(argc < 2) {
     printf("Usage: %s <message>\n", argv[0]);

     return 0;

  }

  print_message(argv[1]);
}
```

函数 print_message 首先将参数 msg 复制到本地缓冲区，然后打印缓冲区的内容。请注意，该函数使用了一个更安全的函数 (strncpy) 将字符串复制到缓冲区。这将防止缓冲区溢出覆盖存储在堆栈中的返回地址。但是，仍然有可能利用这个函数来显示变量 "secret "的值。请演示这种攻击是如何工作的，并解释其工作原理。

练习 4.考虑以下程序：

ex4.c

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    int secret_int = 0x41424344;

    if(argc < 2) {
        printf("Usage: %s <message>\n", argv[0]);
        return 0;
    }

    printf(argv[1]); 返回
    0;
}
```

该程序中的 printf 命令打印一个由用户（通过 argv[1]）控制的格式字符串。为使程序能打印 secret_int 的值？打印变量 secret_int 需要多少个格式指定符？您可以使用格式说明符 %x 来打印一个十六进制整数，以帮助您在函数 main 的堆栈中识别与 secret_int 变量相对应的正确值。

练习 5.考虑以下程序：

文件： ex5.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int n = 0;
    int guess;
    int helper=0x12345678; // 使用该值查找猜测的位置 char str[64]="What is your
    guess?";
    char answer[16];

    printf("Address of n: %p\n", &n);
    printf("Address of guess: %p\n", &guess);
    printf("Address of helper: %p\n", &helper);
    srand(time(0));
    guess = rand();
    do {
        printf(str);
        fgets(answer,16,stdin);
        printf("Your answer: ");
        printf(answer);
        n = atoi(answer);
        if(n == guess) {
            printf(" is correct!\n");
            break;
        }
        printf(" is wrong!\n");
        printf("Guess again (y/n)?");
        fgets(answer,3,stdin);
    }
    while(answer[0] == 'y');
    return 0;
}
```

该程序包含一个格式字符串漏洞，即攻击者可以影响 "printf "命令中使用的格式字符串。找出存在漏洞的 printf，并说明攻击者如何利用这一漏洞打印堆栈帧的内容，以获取变量

"guess "中随机生成的值。

提示：由于在这种情况下可以利用的缓冲区（"answer"）只有 16 个字节，您可能无法插入足够的格式指定符（例如 %x）来打印整个堆栈，因此可能会错过目标变量。幸运的是，printf 允许在格式说明符中插入 "n$"（其中 n 是相对位置），通过指定参数相对于格式字符串的相对位置来直接处理参数。因此，"%n$x "表示 "将 printf 的第 n 个参数以十六进制整数格式打印出来"。例如

```
printf("%3$x", 1, 2, 3)
```

将打印 "3"（格式字符串后的第三个参数）。使用此格式指定符逐个打印堆栈地址，当发现一个值与 "helper "变量的值（0x12345678）相匹配时，就可以知道变量 "guess "一定是 "helper "之后的下一个 "参数"。你需要编写一个 shell 脚本来自动完成这一过程。有关 shell 脚本的一些提示，请参阅下面的附录。

*请注意，这种技巧也适用于其他类型的格式指定符（不只是整数），例如，"%10s "会将 printf 的第 10 个参数（相对于格式字符串）视为字符串（char *）指针，并打印该参数指向的内容。*

伸展运动

下面的练习是扩展练习，实验室不会涉及。

练习 6 (*)。在上一个练习中，我们看到了如何使用 printf 格式字符串漏洞来探测堆栈帧元素，并使用 "锚点（辅助变量中的值 0x12345678）"来确定堆栈中目标变量的位置。在本练习中，我们将研究如何使用格式字符串本身作为锚，如果目标的内存地址已知，我们就可以进一步利用它来取消引用该地址（使用格式指定符 %s -- 即把该地址当作指向 char * 的指针）。让我们看看如何利用这两个阶段的漏洞来打印下面代码中的秘密字符串：

ex6.c

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

int main()
{
    char fmtstr[64];
    char secret[8] = "findme";
    char junk[100] = "填充堆栈的随机内容";

    printf("Address of fmtstr: %p\n", fmtstr);
    printf("Address of secret: %p\n", secret);

    printf("Input a format string: ");
    fgets(fmtstr, 64, stdin);
    printf(fmtstr);

    返回 0;
}
```

请注意，在这里我们对秘密字符串（"findme"）进行了硬编码，因此我们实际上可以逐个查看堆栈元素，直到在堆栈中找到该模式。但是，如果秘密的值未知，我们可能无法知道何时真正找到了秘密。相反，我们将分两个阶段来攻击这个程序：

第 1 阶段。找出格式字符串本身距离 "printf(ffmtstr) "的堆栈帧有多远。也就是说，如果我们将每个内存地址（以 4 字节为增量）视为 printf(fmstr) 的整数参数，那么为了打印变量

"fmtstr "本身的地址，我们需要在 fmtstr 中放入多少个格式指定符？

提示：由于我们可以控制fmtstr 的值，因此可以在fmtstr 中注入一个区分模式，并将该模式作为"锚点"来确定何时到达fmtstr 的地址。例如，如果fmtstr 以"AAAA "开头，那么在堆栈中将编码为十六进制数0x41414141（因为0x41 是"A "的ASCII 码）。因此，如果我们逐个打印堆栈中的元素，最终遇到这种模式，我们就知道找到了fmstr 在堆栈中的位置。

第 2 阶段。一旦找出到达 fmtstr 所需的格式说明符数量，就可以将目标地址注入 fmtstr（而不是模式 "AAAA"），然后取消引用该地址（使用 %s）来打印目标字符串。

提示：您需要在fmstr 的前四个字节中注入目标地址。例如，如果目标地址是0xffff9c1b，那么fmstr 的前4 个字节应包含这4 个字节：0x1b、0x 9c、0xff、0xff（注意顺序相反）。您可能想了解如何使用"echo "生成这个字节数组，将目标地址注入格式字符串fmtstr。请参阅附录中的示例。

练习 7 (*)。考虑下面的程序：

ex7.c

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

int n = 0;

void Win1()
{
    printf("Well done! 你能运行 Win2() 吗? \n");
}

void Win2()
{
    printf("You did it!\n");
}

int main(int argc, char * argv[])
{
    char buf[64];

    printf("Address of n = %p\n",
    &n); printf("Enter a string: ");
    fgets(buf, 64, stdin); printf(buf)
    ;
    printf("Value of n = %08x\n", n);
    if(n == 100) Win1();
    if(n == 3) Win2(); 返
    回 0;
}
```

使用格式指定符 %n 覆盖变量 n 的内存位置，执行 Win1() 和 Win2()（分两次运行）。格式指定符 %n 用于将迄今打印的字符数写入变量。在正常使用中，例如以下代码

Int x;
printf("hello%n", &x)
;

将导致值 5（字符串 `hello` 的长度）被存储到变量 x 中。

附录。一些 shell 脚本技巧

以下是一些有用的 shell 脚本命令/功能，它们可以帮助你自动执行一些任务，这些任务需要你在上述练习中操作输入到程序中的内容。

存储命令结果

shell 脚本的一个有用功能是将程序的输出（打印到标准输出/显示屏的输出）存储到变量中。具体操作语法如下

```
var=$(command)
```

其中 var 是变量名，command 是要执行的命令。例如

```
x=$(echo "hello")
```

将字符串 "hello "存储到变量 x 中。

用 echo "打印 "字节

在 shell 脚本中，有时你会发现需要生成一个字节流作为程序的输入。如果要输入的字节属于 ASCII 编码中 "可打印字符 "的范畴（如大/小写字母、数字或标点符号），则只需直接键入该字符即可。但是，如果字节值与可打印字符不对应，一种输入方法是使用 "echo "命令（带选项-e）生成该字节。例如

```
echo -e "\x1b"
```

输出字节 1b（十六进制），对应于 Esc（转义）键按下时的 ASCII 码。

在测试一个从标准输入（键盘）接收输入的程序时，如果想以编程方式向该程序输入字节流，这将非常有用。

测试字符串中的子串

要测试一个字符串是否是另一个字符串的子串，我们可以在字符串比较中使用 "通配符 "符号 "*"。例如

```
if [[ $x == *"test "* ]]
then
```

```
    echo "test 是 $x 的子串"
fi
```

检查变量 x 是否包含子串检验。