|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
| 课程名称 | 操作系统设计与实践 | | 实验日期 | |  |
| 实验名称 | 操作系统实验 | | 实验周次 | |  |
|  |  | |  | |  |
| 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） | | | | | |
| 1. **OS综合装配**   功能综合：  1.硬盘/软盘启动  2.实现内存分配与释放  3.实现多进程管理与调度  4.目录树结构管理、MAKEFILE、文档  扩展SHELL：   1. 利用当前Orange’s提供的系统调用和API，编写两个以上可执行程序 2. 分析SHELL，画出流程图，调入所编写的可执行程序 3. 进程结束后返回SHELL   SHELL进阶（自选）：  实现同一个SHELL多任务执行  内存管理进阶（自选）：  支持基于分页的虚拟内存管理   1. **安全性分析、可信防御**   安全分析：   1. 可执行文件篡改（ELF注入） 2. 内存破坏（栈溢出劫持） 3. PRINTF安全检查（泄露内存内容） 4. …   可信防护（静态）   1. 加载程序时检查可执行文件是否被篡改 2. 相关探讨   可信防护（动态）   1. 编写一个自动化触发程序，触发时解析当前进程内存、堆栈结构，检查是否合法 2. 相关探讨（效果、运行效率、额外问题等） | | | | | |
| 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） | | | | | |
| 本OS采用软盘启动，运行、编译环境和相关参数请参考仓库中的README.MD。 综合装配 **一、多级队列反馈调度**  相关代码在kernel/proc.c。  要求将在进程调度章节中实现的多级队列反馈调度算法整合入当前OS。这里采用的队列实现和算法核心部分与之前相同，重点在于增加进程阻塞机制，这里新增了两个队列：Q\_block和Q\_new。  在开始原调度算法前，**先执行wash**函数（wash(Q, &Q\_block, &Q\_new);）。wash主要将进程按照其p\_flags将其放入对应的队列。其中，标记为FREE\_SLOT的进程将进入Q\_new，标记为0的将进入Q（调度队列），其它进入Q\_block。  由Q\_new进入Q，将进入Q[0]，即最高优先级队列，这符合多级队列反馈的定义，即新进程进入Q[0]。  由Q\_block进入Q，将进入Q[2]，即最低优先级队列。这是因为，考虑到，一旦进程通过系统调用、IPC等发生阻塞，若放入较高优先级，则可能导致其它进程**饥饿**。设想两个进程不断相互通信、轮流被阻塞，那它们将一直处在最高优先级，其他进程将得不到调度。  多级队列反馈调度原理和效果图可参考进程调度章节实验报告，这里不再赘述。    **二、编写该OS下的可执行程序**  相关代码见command/目录下cat.c ticker.c mod.c stack.c target.c kill.c ls.c (echo.c pwd.c lseektest.c payload.c payload.asm)  **cat.c**  调用open和read读取指定文件（文件名由命令行给出）前0x100个字节并回显其十六进制内容：    **ls.c**  显示当前文件系统中的所有文件。    为实现此功能，新加了一个系统调用，名为readdir，见fs/readdir.c： fs\_readdir()。    对应的接口在lib/readdir.c。该接口告知文件系统要读取的目录（其实当前只有根目录），以及结果存放的位置。    对应的fs\_readdir()则在取得目录条目，拼接好字符串后，将之复制到调用者内存空间中的指定位置    其它可执行文件多用于测试或演示，将在相应部分详细解释。  **三、shell流程图**  循环开始 =>  -> 打印提示符  -> 读stdin  -> 解析stdin  -> 尝试打开指定的应用程序  -> 若有此程序，则进行fork（若无此程序，则加上花括号，回显指令,进入下次循环）  -> fork后父进程进行等待，子进程执行execv加载指定应用程序  -> 子进程执行完毕退出，父进程等待结束并进入下次循环  **四、shell支持多任务执行**  预期语法：仿照Linux命令行的默认语法，在命令的最后加一个“&”符号表示执行进程时不阻塞shell。  解析实现：strcmp检查最后一个参数是否为“&”，如果是，则记block\_shell = 0，接下来argc自减，并将argv中原本存“&”处改为0（NULL）。  执行实现：  fork()时，当前操作系统没有考虑到父进程不退出且不等待当前子进程的情况。父进程若退出，则子进程过继给INIT，父进程若不退出且不wait，子进程会一直处于HANGING态（或者没运行完），此后父进程只要调用wait，就会变成等待这个HANGING的子进程。  故若直接通过取消wait，即if (block\_shell) {wait}来实现，则会出现父进程在下一循环（接受到下一个命令，记为进程B）中调用wait时，本意等待这次产生的子进程B，实际却变成了等待上一轮的进程（记为A）。效果上若A已经运行完，则此次shell将不等待B，其wait状态会由HANGING的A取消，若A未运行完，则shell会一直不响应，直至A结束。  **总之，一般解法会导致shell等待的进程错位。**为解决这个简单的问题去扩展脆弱的Orange’s显然是不划算的，所以这里采用的方法是令fork后的父进程再fork一次，让原shell进行等待与退出，新shell继续运行，这样虽然多创建了一个进程，但好处是方便、快捷、可行。    效果/测试：  写了一个应用程序**ticker.c**，该程序内有一个不断自增的计数器，不断循环自增（循环中有delay()），并且每次自增后会将该计数器值写入指定文件（文件名由命令行给出，默认文件名为“record”）。令该程序运行且不阻塞shell，**在其运行过程中执行cat查看文件变化**。  如图，两次cat显示的内容（第一个字节，即写入的计数器值）不同，分别为14和1F。    **五、内存分配与释放（alloc\_pages）**  要求整合之前章节中实现的页分配函数，这里照搬过来即可，见kernel/page.asm。代码片段如下：    在kernel.asm中加入测试代码：    运行效果图参考之前章节的截图，这里不再赘述。  由于Orange’s现在已经实现类似基于位图的简单内存分配，即通过p\_flags是否为FRESS\_SLOT来判断内存空闲与否，虽然它没有加上虚拟内存映射，但我们还是决定不把alloc\_pages也整合到进程空间分配中。 安全分析与可信防御  1. **ELF（可执行文件）注入**   原OS存在可执行文件篡改的问题，这里通过成功的ELF注入，演示如下。  攻击者功能：  编写可执行文件**mod.c**，该文件会调用之前实现的readdir来遍历当前目录下（其实就是整个文件系统中）的所有可执行文件并对其进行注入。  这里出于便于测试和便捷的角度考虑，不希望出现注入后所有程序都不能执行原本功能的情况，故对除**target.c**生成的**target**程序以外的ELF文件，只改变其.text节内有效代码之后的内容（即永远不会被执行到的地方）。  对target，则令其在**被注入后变为打印-Infected-，而后正常退出**。  target:  原功能为两条打印，代码如下：    攻击者实现：  首先要定位到ELF文件中代码节的位置和有效代码节大小。先找到代码节头的位置，elf文件节头表中sh\_name字段存储的是该节头的名称在节头名称表中的索引，而节头名称表的偏移可以在elf文件头的shstrdx字段得到，因此定位代码节程序的逻辑如下：  1. 在elf文件头中获取节头名称表的偏移  2. 在elf文件头中获取节头的偏移  3. 遍历节头表，通过每个节头的sh\_name字段在节头名称表中找到对应的字符串，如果该字符串为.text,则说明该节头为代码节节头  4.通过代码节头的sh\_offset字段定位到代码节  具体代码参考**mod.c: process()**。  然后是payload的构造。这里由于gcc编译时会加上一些奇怪的函数导致注入后的代码跳转到错误的地址而失效，最终选择手写汇编，见**payload.c**、**payload.asm**（最终编译生成二进制文件作payload的是payload.asm）。其中C代码如下，这里直接暴力使用objdump获得函数地址，以该常量作函数指针调用。        make payload后再用objdump获取机器码：    运行效果：  如图，依次运行target、mod、target，可以看见target的输出发生了变化。    **二、栈溢出漏洞**  见**command/stack.c**。该程序内vulnerability()函数接收一个数组并将之复制到其局部变量buf中，这里通过构造payload传给vulnerability()使其写buf溢出并覆盖函数返回地址，新返回地址为dest()函数，该函数在程序中没有被调用。    这里为便于测试，不采用复杂的方法排查找出栈溢出点位，而是程序内直接计算得出（通过内联汇编获取ebp得知栈起始位置）。    运行效果：  可见函数流程因为栈溢出被跳转到了dest()函数内。    **三、其它漏洞**  如printf参数未检查等简单漏洞，这里不再一一复现。  **四、静态度量防护**  总体思路   1. INIT untar出每个文件时计算checksum并存入对应文件。 2. execv时会比对文件计算所得checksum是否与存的一样。 3. 进行访问控制，禁止用户进程open存checksum的文件。   具体实现   1. 存checksum较简单，这里不再赘述。 2. 修改**mm/exec.c: do\_exec()**进行检查。     若发现文件被修改则不进行原execv任务，而是打印一条警告信息并返回-1。     1. **fs/open.c: do\_open()**访问控制，用户进程调用open时会直接打开失败。     运行效果  可见：①：mod无法打开存有checksum的文件（“chk-”开头）；②target在被篡改后，再次尝试执行target，没有任何效果（target没被加载），且在主控制台看到了警告信息。      该校验算法的问题：目前采用的checksum计算是直接异或文件每一位，这个其实是很弱的，正常应该使用密码学安全的哈希函数，确保对方无法实现碰撞，这里出于实验便捷性，没有加用。  **五、动态度量**  参见**kernel/proc.c: stack\_chk(), schedule()**。    总体思路   1. 增加stack\_chk函数，其参数为struct proc\*类型，检测到栈异常即返回0，否则返回1。 2. 在调度算法内调用该函数。①当前进程若检查异常则禁止其继续运行；②一切检查异常的进程都不enqueue，即永久禁止参与调度。   思考、实现与更改   1. stack\_chk。原本的实现方式是通过**检查ebp链进行验证**。因为ebp总是指向当前程序栈帧顶部，故可以追溯到栈的起点，检查时如此追溯，若链中任一ebp不指向栈内，则判定为异常。 2. 验收时和老师探讨发现由时钟触发检测其实是“半动态”检测，无法避免在检测空隙内被提权。此后我又稍微思考了一下，发现栈空间其实都是完全由“攻击者”掌握的，即**只通过栈空间检测的方式必然是不可信**的。故干脆删除了对整个栈内ebp链的检查，只保留了对链首的检查，即当前ebp指向的ebp是否合法。这样也能防住之前的栈溢出漏洞利用程序，但实际上无论是检查整条栈链还是只查链首，都防不住经过精心设计的栈布局，除非再增加一些攻击者无法控制/得知的动态检查量。   **以下是目前版本的检测**，注意检查ebp链时需进行地址转换，以及只对用户进程进行检查：     1. 验收之前其实只做了动态漏洞检测，测出后没有采取任何行动。后为了验证老师说的“**不能保证在间隙内不会被提权**”，后又在shcedule内增加了对应措施。   事实证明**stack**运行时虽然会被警告，但stack仍能打印出“In dest now”。即检查空隙过大，没能防住其攻击。  如图，stack在进入dest()后，**delay()前的消息成功被打印**，**delay()后的消息没能打印出来**，**被检测出异常并被阻塞了**。      编写动态检测时实现的其它功能  按道理，检测出异常应该立即终止（**Kill**）该进程，但Orange’s没有提供这个系统调用，于是我们在exit的基础上稍加修改得到了一个kill，见**command/kill.c**、**lib/kill.c**、**mm/forkexit.c: do\_kill()**。  但实现kill后发现Oranges’s在schedule中**无法调用kill**(后续测试发现好像也没法调用其它系统调用)，会引发系统异常。猜测主要原因是schedule通常在中断时被触发，不是在进程中，无法调用IPC。其它原因可能有操作原子性等等。总之最后只得放弃采用kill。  **mm/forkexit.c: do\_kill()**实现的代码片段如下，这里禁止杀死INIT及其之前的进程。    编写了**command/kill.c**进行测试。    测试①：Orange’s启动完成后共11个进程（PID<=10），再运行新进程PID就是11。故这里实际上是kill自己，可以看到只打印出killing，没有end。    测试②：kill 10，即kill第二个shell，可见kill后再到TTY #2输入ls，没输出了，因为shell已经不在了。 | | | | | |
| 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） | | | | | |
| **分工**  小组内共同探讨  **Orange’s BUG汇总**  这里汇总开发过程中发现的主要BUG。   1. BUG：进程初始化时采用strcmp( …name,“INIT”)识别是否是INIT进程，这里在本实验环境下运行出现BUG，调查发现貌似字符串或其指针发生了错位。   SOLVE：没有选择解决字符串问题，直接用整数，…pid == INIT即可、   1. BUG：该OS下的应用程序main函数接受的argv[1]出现神奇的“一次性”性质。即printf(argv[1])后其字符串内容即被修改，无法再次使用。   SOLVE：没有选择解决argv的问题，先将其复制到进程内后再使用即可。   1. BUG：无法untar，在untar时一直出现assertion\_failure等错误。   SOLVE：**kernel/hd.c：hd\_rdwt()**一处加个delay后即解决。猜测是因为没有等待硬盘读写完成？     1. BUG：引入多级队列调度后，系统直接崩溃。   REASON：定位到崩溃前调用了informt\_int，调查后得知因为有进程在等硬件中断，被阻塞，进入调度算法，但中断很快就到了，打断调度，把进程的阻塞解除了。这样导致调度丢失了原子性，后续出现assertion\_failure（在**kernel/proc：msg\_receive()**的结尾处）。  SOLVE：发现这几个assert可以直接删去，故删除。     1. BUG：不能短间隔连续fork，导致实现shell多进程支持受阻。   SOLVE：推测是原子性等问题，不好解决，故选择增加delay。   1. BUG：disp\_str这个汇编函数在本实验环境下出现换行错误。   SOLVE：重写换行逻辑后解决。 | | | | | |
| 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） | | | | | |
| **总结**  较好地完成了各项实验任务，达到了实验基本要求，并在进阶要求上有所扩展。  **思考**  详细的思考已于“二、实验环境及实验步骤”中给出。 | | | | | |
| 1. 各人实验贡献与体会（每人各自撰写） | | | | | |
|  | | | | | |
| 1. 教师评语 | | | | | |
|  | | | | | |
| **教师评分（请填写好姓名、学号）** | | | | | |
| 姓名 | | 学号 | | 分数 | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| 教师签名：  年 月 日 | | | | | |