|  |  |  |  |
| --- | --- | --- | --- |
| **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 | 2021.12.14 |
| 实验名称 | 期末大作业 | 实验周次 | 7 |
| 姓名 | 学号 | 专业 | 班级 |
| 庞远心 | 2019302050244 | 网安 | 2019级8班 |
|  |  |  |  |
| 1. 实验目的及实验内容 | | | |
| 1.实验目标  （1） 掌握如何创建shell并对shell添加拓展程序  （2） 掌握在shell中多任务并行的方法  （3） 掌握可执行文件的篡改以及缓冲区溢出的漏洞简易攻击  2.本次实验内容  （1）基本实现自己的os，能够实现前面章节所实现的内存分配与释放和多进程管理与调度，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。  （2）为shell扩展应用程序并编译生成存储在文件系统中，分析教材的shell代码  （4）在程序中查找OS中的可执行文件，向可执行文件中加入额外代码  （5）编写程序可对存在内存破坏地址的代码进行缓冲区溢出，控制返回地址到指定的位置。 | | | |
| 1. 实验环境及实验步骤 | | | |
| 1.实验环境  VMware 15.5, ubuntu-16.04.6-desktop-i386  2.实验步骤  （1）根据教材第十章，实现基本的shell并向shell中添加自己的应用程序。  Shell的基本原理是通过键盘键入字符串，将字符串按照一定格式分隔开，fork一个子进程，再将字符串参数传入子进程中，子进程根据字符串参数中的命令名字在文件系统中找到对应的可执行文件，执行可执行文件后返回到shell的页面。   1. fork一个子进程：   一个新的进程所需要的要素：   * 自己的代码， 数据和堆栈 * 在proc\_table[ ]中占用一个位置 * 在GDT中占用一个位置， 用以存放进程对应的LDT描述符   后两项工作容易完成，只需要在数据机构中加入相应的项就可以，但是代码、数据和堆栈的获取需要直接从已有的进程那里继承或者复制。如果新的进程C的代码、 数据和堆栈是从已有的进程P而来， 那么P被称为父进程（ parent） ， C被称为子进程（ child）。生成子进程的系统调用被称作fork（）, 操作系统接到一个fork请求后， 会将调用者复制一份， 这时就会有两个一模一样的进程同时运行。而查阅书籍，在linux中，所有的进程都有一个共同的祖先，init进程。代码如下：  Init( )中调用了我们即将实现的fork( )， 而且判断了返回值，返回值的作用是让进程判断自己是父进程还是子进程，如果返回值为0，则说明为子进程，反之为父进程。  上图为一个 进 程涉 及 的所 有 数据 结 构及 相 互 联系  在这里我们不光要增加一个Init进程， 还要增加一个MM进程——本章说的就是内存管理， 这个进程无疑是最重要的。 MM将负责从用户进程接收消息， 完成fork( )等操作。  在进行fork操作之前，我们需要进行一些**预处理**，首先在proc\_table[ ]中预留出一些空项， 这需要通过改变NR\_PROCS来实现：  这里我们不仅改变了NR\_PROCS， 还增加了一个宏：NR\_NATIVE\_PROCS， 它表示系统初启时共有多少个用户进程。 因为我们有TestA、 TestB、 TestC和Init， 所以它应该是4。 在表示进程数目的宏发生改变后， 我们还应该修改kernel\_main( )，代码做了一些调整， 主要是两点。   * 一是将暂时没有用到的proc\_table[ ]表项的p\_flags成员赋值为FREE\_SLOT，将来MM会根据此项值来判断一个表项是否为空 * 二是将Init进程单独区分开来， 其LDT单独赋值。   这样做的原因很简单， Init将来会fork出子进程， 那么Init的所有内存范围都将被复制到新的位置， 并在那里运行。 如果Init用与其他进程相同的处理方法， 使用0～4GB的扁平空间作为LDT描述符， 那么我们在内存中将找不到另一块空间来存放它——在32位系统中， 4GB是全部内存空间。那么Init进程的内存应该被限制在怎样的范围之内呢？ 最简单的思路就是， 内核有多大， Init进程的内存空间就有多大， 这样不但大大缩小了进程占用的内存， 而且可以保证对read( )、 write( )等所有库函数的调用都是合法的。 你可能会说， Init进程还可以更小， 没错， 因为内核中的大部分代码对Init是不会直接调用的， 但更小的Init将会增加代码的复杂程度， 所以我们还是偷个懒， 用整个内核占用的内存来当做Init的内存范围  **写一个fork函数：**   1. **MM：**   当MM接到FORK消息后， 调用do\_fork( )来处理，代码省略。  写完MM函数之后，我们还需要考虑**内存分配**等的内容  一个成熟的内存分配机制是比较复杂的， 但这并不意味着简单的机制就完全行不通。 在这里我们就使用了一种十分老土的方案， 那就是划定格子， 每个格子大小固定。 有新的进程需要内存， 就给它一个格子， 而且这个进程在整个生命周期中， 只能使用这个格子的内存。我们把格子的大小定为1MB， 这个数字在一定程度内可随意选取， 但至少应该大于内核的大小240KB， 这样才能盛得下Init。与此同时， 并不是所有的内存空间都能被Init的子进程使用。 0～1MB的空间被内核使用， 显然不能用。 另外别忘了， FS等进程还使用了一些内存作为缓冲区， 我们也要避开。综合各种因素， 我们定义了下面这几个宏：  确定了分配方案之后，我们需要定义**内存分配函数：**  我们这种分配方案， 其实就是建立了PID和进程内存空间之间的映射关系， 或者说， 内存空间是PID的一个函数。 这一方案的缺点非常明显， 对于小一点的程序， 1MB的内存太浪费， 而对于大一点的程序， 1MB又可能会不够。 不过我们先不管那么多， 内核我们写了这么久， 在内存中才只占约四分之一个1MB， 可见1MB也没有那么小； 至于浪费嘛， 那是属于优化的范畴， 我们以后再说。 总之， 跟设计一个简单的FS思路相同， 我们先不求好， 只求先有个能用的版本。   1. fork一个子进程还需要**FS**来协助， 其实FS主要是要增加两个计数器：   隶属于inode结构的i\_cnt这个计数器我们已经很熟悉了， 只要有进程使用这个inode， 其i\_cnt就应该加1。 属于file\_desc结构的fd\_cnt是我们新增加的成员， 它的道理很类似。 假设进程P生成了进程C， 那么P和C共享使用f\_desc\_table[ ]中的同一个file\_desc结构， 这时fd\_cnt为2， 表明有两个进程在使用这一结构。 等进程P或C退出时， fd\_cnt自减。 若P和C都已退出， fd\_cnt自减为零， 这时系统应将fd\_inode赋值为零， 这样这个f\_desc\_tabel[ ]条目就又可被使用了。   1. 生成子进程最重要的fork（）已经有了，但这还不够，因为进程有出生就有死亡，**exit和wait**负责了进程的死亡处理程序。Exit和wait的运行逻辑如下：   为配合exit( )和wait( )， 进程又多了两种状态： WAITING和HANGING。 如果一个进程X被置了HANGING位， 那么X的所有资源都已被释放， 只剩一个进程表项还占着。 为什么要占着进程表项不释放呢？ 因为这个进程表项里面有个新成员： exit\_status， 它记录了X的返回值。 只有当X的父进程通过调用wait( )取走了这个返回值， X的进程表项才被释放。如果一个进程Y被置了WAITING位， 意味着Y至少有一个子进程， 并且正在等待某个子进程退出 。   1. 在写完以上的函数之后，我们将会迎来重要的exec（）函数，该函数的作用是将当前的进程映像替换成另一个。 也就是说， 我们可以从硬盘上读取另一个可执行的文件， 用它替换掉刚刚被fork出来的子进程， 于是被替换的子进程摇身一边， 就成了彻头彻尾的新鲜进程了。以shell中常见的echo命令为例。 我们输入“ echo hello world” ， shell就会fork出一个子进程A， 这时A跟shell一模一样，fork结束后父进程和子进程分别判断自己的fork( )返回值， 如果是0则表明当前进程为子进程A， 这时A马上执行一个exec( )， 于是进程A的内存映像被echo替换， 它就变成echo了。   我们还需要将自己的应用程序加载到操作系统中，此时我们选择将其压缩至一个压缩包“cmd.tar”，内核加载时将该压缩包解压，应用程序加载到指定位置能够被shell读取即可。   1. 此时我们先在自己的操作系统中写入相应的程序2048和斗地主，再在makefile里修改，运行make install命令，再bochs即可使用该应用程序：     上面的截图为为了添加2048所对makefile进行的修改。  （2）shell多任务并行  前文我们介绍了shell的做事原理，我们如果想要通过输入多个命令从而让shell同时执行多个任务，我们需要考虑几个因素：   * 如何让命令行识别多个命令而不是认为是一个命令 * 如何fork多个子进程分别执行多个命令 * 父进程需不需要等待，需要等待多久   下面我将一个一个解释我的思路：  1） 由于shell键入的字符串会被完全存储在argv[]数组里，该数组是一个字符串指针数组，数组的每一项为一个word字符串的首字符地址，如键入“echo happy”，数组中会存放“echo”和“happy”两个字符串的首地址。**故我们需要将不同的命令首先通过一个标识符分隔开，存储在一个二维的字符串指针数组里。**  图为存储所有word字符串指针到字符串指针数组里  我的想法是键入的命令通过“$”分割开，如键入“echo happy $ pwd $ counter1”  将转换成三个命令，分别是“echo happy”,“pwd”,“counter1”。  图为将命令按照“$”隔开  设置一个首尾指针，首指针指向命令的开始，尾指向遇到的第一个“$”字符串，首尾指针（包括首不包括尾）拷贝到process二维指针数组。  2） 将命令拷贝到数组之后，我们需要关心如何fork多个子进程。在这个地方我的处理思路是先判断命令是否合法，若命令合法则fork子进程，否则不fork。为了保证fork多个子进程，**需要在每个子进程的时候return 0 ，否则将会导****致子进程继续fork子进程**。  图为fork子进程的代码  如图发现我并没有wait()，这是因为我发现如果调用了wait（），父进程等待时间过短，这导致所有的子进程将按照顺序执行，这显然是我们不想看到的，此处如果不等待的话就解决了这个问题，但存在的问题是结束时候会导致命令提示符无法在正确的地方出现。此时为了显示程序处在并行的状态，我定义了三个可执行程序counter1~counter3,分别打印1——200,200——400,400——600的数字，如果打印的数字交替出现，则说明此时多进程在并发的状态，其截图在后面的”实验结果”中。  （3）缓冲区溢出  缓冲区溢出的思路来自于大二上的计算机系统基础实验的思路，这个思路与数据存储在计算机内的位置有关，程序的动态数据（如动态变量）将会存储在堆栈中，此外，每调用一个函数，当前函数的返回地址和esp会被压入堆栈，方便被调用函数返回到调用函数。  如果我们向指定大小的缓冲区输入过长的数据，会导致缓冲区溢出，溢出的数据将会覆盖堆栈中的其他数据，当数据足够长时，就可能会覆盖前文提到的返回地址，原理如下图所示：  根据这个原理，我的思路是定义一个静态全局变量和动态变量，将溢出区的函数的首地址存放在静态变量的指定位置，然后将该变量值拷贝到动态变量中，这将导致返回地址被溢出区函数的首地址覆盖，当调用函数执行完时会返回到溢出区函数，执行该函数并无法返回到最初该返回的地址。    缓冲区溢出代码  函数的入口是main()函数，如果没有缓冲区溢出，main（）先调用overflow\_retaddr()，再返回到main（）并打印字符串“You are safe\n”，若调用overf\_retaddr（），testbuf数组中的值覆盖了main（）的返回地址，打印了“You are hacked\n”且没有返回，则说明发生了缓冲区溢出。实验截图在实验结果一栏。  （4） 查找os中的可执行文件并修改代码。  由于可执行文件都被cmd.tar解压出，此时我们如果想要寻找可执行文件，需要查看文件系统，但由于我们跳过了第九章，所以对于找到os中的可执行文件这个部分，我咨询了其他同学并参考了部分网上的代码，得到一个查看根目录下的可执行文件的方法，该方法的每16个字节是可执行文件的文件名，但并不是所有可执行文件都需要更改， 如tty、kernal.bin等文件，我们需要跳过这样的文件，在进行打印后可发现前6个文件不可以改变：  从echo（我们添加的可执行文件）可以改变，则跳过这些文件，代码如下：  从此时（echo）开始逐个读取可执行文件的名字，当遇到attack(攻击程序本身时)跳过，每4个字节读到缓冲区中，若读到elf头，则打开可以被更改的可执行文件，读到我们所需要的修改的字节，将字节改为我们所希望的代码，并写入，关闭该文件直到修改所有可执行文件完成。  我们所希望修改后的代码：    需要修改的字节：    （return 0的机器码）  修改之后函数无法return 0，我们选择一个testa函数显示修改的效果，预计效果为不断打印“This is testa”.  实验截图在实验效果一栏。 | | | |
| 1. 实验过程分析 | | | |
| 1. 实验分工：   我负责我组的parta3和partb的缓冲区与可执行文件的修改。   1. 实验过程遇到的困难：   最开始多任务并行的地方父进程wait（），发现执行的结果是多任务按照顺序执行；缓冲区溢出最开始静态变量的数组设置为12（2\*4+4），后来发现攻击失败，修改至20（2\*8+4）成功。 | | | |
| 1. 实验结果总结 | | | |
| 1. 扩展shell并在shell中加入应用程序   扩展1:创建自己的界面    拓展2：显示本os提供的应用程序    扩展3：加入应用程序斗地主和2048  斗地主：    2048：     1. 多任务并行（执行pwd $ echo happy $ counter1 $ counter2 $ counter3）   结果:    该图中的数字混杂，200以内的正整数和200~400，400~600内的数交替出现，说明多任务成功并行。   1. 缓冲区溢出攻击（调用命令 stackoverflow）      1. 修改可执行文件   被修改之前的testa：    Attack命令执行：    修改后执行testa： | | | |
| 1. 实验贡献体会 | | | |
| 该次试验最开始我毫无头绪觉得很难，因此走了很多弯路，最后及时询问同学，得到大致的思路之后自己再去看书，一点点debug写完了，掌握了多任务并行、缓冲区溢出、可执行文件的修改，最后发现实验没有想得那么难，做实验不是从0到1，而是站在巨人的肩膀上走的更远。 | | | |
| 1. 教师评语 | | | |
| 教师签名：  年 月 日 | | | |