文章编号: 1003-2843(2008)05-1073-04

Linux 内核级 exploit 原理及应用

李晨曦[1], 周绍军[2]
(西南民族大学管理学院, 四川成都　610041；2.四川水利职业技术学院, 四川都江堰　611830)

摘要: 本文主要分析内核级 exploit 和应用层的 exploit 不同之处, 详细阐述了内核缓冲区溢出（kernel BOF）、内核格式化字符串漏洞（kernel format string vul）、内核整型溢出漏洞(kernel integer overflow)、内核 kfree()参数腐败(kernel kfree parameter corruption)、内核编程逻辑错误（kernel program logic error）的原理.

关键词: 内核; 漏洞; 开发
中图分类号: TP311　　**文献标识码**: **A**

1　引言

应用层 exploit 技术是内核级 exploit 技术的基础. 首先需要了解 linux 内核和 x86 的保护模式及中断函数的进入和返回过程, 在保护模式下 INT 指令对不同优先级之间的控制转移, 从 TSS（任务状态段）中获取高优先级的核心堆栈信息（SS 和 ESP）, 把低优先级堆栈信息（SS 和 ESP）保留到高优先级堆栈（即核心栈）中[1], 把 EFLAGS, 外层 CS, EIP 推入高优先级堆栈（核心栈）中, 通过 IDT 加载 CS, EIP（控制转移至中断处理函数）, 在 iret 返回时候假如 NT=1 的话, IRET 指令就反转 CALL 或者 INT 的操作, 这样引起一次任务切换. 这时候执行 IRET 的指令代码, 更新之后放在任务状态段内[2-4]. 然后需要掌握核心堆栈指针 ESP 和进程内核 task 的关系, 每个进程在内核中有个内核堆栈, 共占 2 页, 也就是 8192 个字节, 在这堆栈底部, 是该进程的 struct task_struct 结构. 所以在内核中使用 current 就可以访问到该进程的 struct task_struct. 接下来了解进程内核路径也是必要的, 一条内核控制路径由运行在内核态的指令序列组成, 这些指令处理一个中断或者一个异常. 当进程发出一个系统调用的请求时, 由应用态切换到内核态. 这样的内核控制路径被成为进程内核路径, 也叫进程上下文. 当 CPU 执行一个与中断有关的内核控制路径的时候, 被成为中断上下文. 最后需要掌握 CPU 的所处路径, CPU 路径包括处理硬件中断（此时不跟任何进程相关联）、处理软中断(softirqs), tasktets 和 BH,不跟任何进程相关联、运行在内核空间（进程内核路径）, 关联着一个进程、运行着用户空间的一个进程[1-6].

2　两种技术异同点

系统脆弱性: 内核级 exploit 和应用层的 exploit 有相同点, 应用层缺陷在内核层也是缺陷. 漏洞分类: 应用层漏洞与内核漏洞基本相同.

内核级 exploit 在 Ring0, 应用层作用于 Ring3. 可发挥的空间内核级 expoit 相对应用层 exploit（4G 内存空间）小很多. 内核级 exploit 研究难度相当大, 同时可参考资料很少.

3　内核级 exploit 原理

3.1 内核缓冲区溢出

首先找 retloc 的地址, 该地址用于程序的流程, 就缓冲区溢出来说, 就说找到函数的返回地址. 写程序

收稿日期：2008-07-25
作者简介：李晨曦(1980-), 男, 西南民族大学管理学院教师.

ker_bof 用于测试, 主要函数如下:

asmlinkage int test(unsigned int len,char * code) { char buf[256];memcpy(buf,code,len);};asmlinkage int new_function(unsigned int len, char * buf) { char * code = kmalloc(len, GFP_KERNEL);if (code ==NULL) goto out;if (copy_from_user(code, buf, len)) goto out; test(len,code);out:return 0;};int init_module(void) {old_function = sys_call_table[__NR_function]; sys_call_table[__NR_function] = new_function; printk("<1>kbof test loaded...\n"); return 0;}void cleanup_module(void) {sys_call_table[__NR_function] = old_function;printk("<1>kbof test unloaded...\n");}. 用溢出程序确定 RETLOC 的地址: static inline _syscall2(int, new_function, unsigned int ,len,char * ,code);int main(int argc,char **argv){ char code[1024]; unsigned int len; memset(code,'A',1024); len = 1024; new_function(len,code); system("/bin/sh");}运行 ker_bof,发生了段错误, 虽然非法指令出现在内核中, 但引起该非法操作的路径是进程内核路径, 未发生系统崩溃. 溢出点分析使用 objdump 查看 ker_bof 汇编代码 57push %edi；56 push %esi 从上面可以看到: 可以覆盖到 EIP, EBP. 却不能修改 ESP. 调用函数 new_function 栈帧, 被调用函数栈帧 ESI =-ebp+s+4, 找到 retloc: code[256+8+4], 修改测试代码查看得到 retloc 地址正确.

其次, 确定 retaddr 的地址, 知道保护模式下是开启了页表机制的, 一个地址要想能被访问到而不产生页错误的话, 那么需要由页表来映射该地址. 当由一个软中断陷入到内核中去的话, 原来应用层的页表映射是不会被清除的(其实内核自己也要使用那些页表来访问用户空间的数据, 还记得 copy_from_user 这些函数吧), 并且又加上了内核地址空间的映射. 所以, 可以用应用层的地址就可以了, 内核也可以正确寻址. 这样 shellcode 也就很容易得到了, shellcode 的地址可以从应用层得到.

Shellcode 功能: 因为漏洞发生是在内核进程路径上, 所以能够利用 ESP 和当前 TASK 的关系, 来使自己的进程的 UID 变为 0. 使得变成 ROOT 权限, 从而权限得到提升.

内核态返回到应用态得到了控制权后, 运行 SHELLCODE 代码, 必须让中断调用返回, 这样系统才不至于崩溃掉.

3.2 内核格式化字符串漏洞

首先看 kernel 的 printf()系列函数是否支持%n. 不支持的情况下, 即使系统内核中存在这样的漏洞的话, 最多能使内核的信息泄露, 或者也许可以把内核弄崩溃掉. 不过也许在特定的条件下还是值得去研究的. 确定 RETLOC 的地址, 写 ker_format 测试程序, 其中 snprintf(buffer,len,code)为字符串函数. 覆盖函数返回地址是一般应用层方法. 但是在 kernel 要想利用这种方法的有一定难度. 因为调试内核就比较困难, 无法预知内核中 ESP 的值, 也就没法覆盖函数的返回地址, 所以需要寻找另外的 RETLOC 地址. 在 head 中找 sys_call_table 符号又是在/porc/ksyms 的, 而且又是普通用户可读的, 所以就可以得到这个调用表的地址了[LCX@redhat73 alert7]$ cat /proc/ksyms |grep sys_call_table c02c209c sys_call_table_Rdfdb18bd 找一个在系统调用表里不使用的系统调用, 选择 241 这个系统调用. 然后把 sys_call_table+241*4 的地址上填上 SHELLCODE 的地址. 最后内核态返回到应用态. 在这里考虑 SHELLCODE 地址过大的问题, 从 mmap()系统调用 void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);看, 再修改 ld 的连接脚本, 默认的连接脚本是把地址分配到 0x08048000.

3.3 内核整型溢出漏洞

现在上面两种漏洞在内核中是很少见的. 整数溢出就是有符号数和无符号数的烂用, 导致可绕过一些检查. Kernel2.2 和几个 2.4.x 版本的 linux/kernel/sysctl.c: sysctl_string() 函数中就存在这样的错误. 同样构造 ker_integer 测试程序. 唯一需要注意的技术细节在于当传个负数 len 进来的话, 就可以绕过 if (len > 256) len = 256;的检查, 从而在 strncpy_from_user 的时候把它当成无符号处理的话, len 的值就会变的很大. 其他基本同以上的技术处理相同.

3.4 内核 kfree()参数腐败

内核中 kfree()参数腐败应用,首先看 kfree()函数中是如何用一个值得到另一个值, 变量之间的关系. static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp){#ifdef CONFIG_SMP；cpucache_t *cc = cc_data(cachep);CHECK_PAGE(virt_to_page(objp)); if (cc) {int batchcount;if (cc->avail < cc->limit) {STATS_INC_FREEHIT(cachep);cc_entry(cc)[cc->avail++] = objp; return;}分析加粗程序代码, 假如现在 kfree 的地址为 addr,那么: temp1= mem_map+(((unsigned long)addr- PAGE_OFFSET)>> PAGE_SHIFT)c =((kmem_cache_t *)(temp1)->list.next);cc = c->cpudata[0];

要确保(cc->avail < cc->limit)(((cpucache_t*) (cc))+1)[cc->avail++] = addr;来看看具体变量跟变量的数值关系 c= mem_map+(((unsigned long)addr- PAGE_OFFSET)>> PAGE_SHIFT)

cc=(c+116)要确保(cc)<(cc+4);(cc+8) = addr;从上面一些关系来看, C 的地址只能是 mem_map---mem_map+0x3ffffc, 也就是 0x c032e2f0 到 0x C072E2EC 之间. 很显然, 这些地址都不是所能控制的. 既然 C 是所控制不了的, 那么下面的几个变量（依靠 C）更是控制不了的. 得出 kfree()参数腐败在该原理分析下是不可用作攻击漏洞的.

3.5 内核编程逻辑错误

内核代码相当的多, 出现问题是难免的. 将逻辑错误划分到内核编程逻辑错误也是勉强可以的. kernel 2.4.9 之前的 ptrace 系统调用的漏洞属于这个类型. BUGTRAQ ID 为 6115 的漏洞报告, kernel 2.4.x 都存在的问题. “Linux 内核不正确处理系统调用的 TF 标记, 本地攻击者利用这个漏洞可以进行拒绝服务攻击. 如果系统调用 TF 标记设置的情况下, 可导致内核停止响应, 攻击者利用这个漏洞通过执行调用了 TF 标记系统调用的恶意应用程序, 可使 Linux 内核挂起, 需要重新启动获得正常功能.”[1]其实这是调用门处理代码处理不当导致的, linux 为了兼容, 保留了两个系统调用门. 一个 lcall7,一个 lcall27,在两个处理函数里处理 EFLAGS 的 NT 和 TF 标志不当, 导致系统崩溃. 报告同时也给出了一个 exploit

```
#define MSUX "mov $0x100,%eax\npushl %eax\nmov $0x1,%eax\npopfl\nlcall $7,$0"
```

在 redhat 7.3 默认安装的系统上没有使系统挂起. 修改为如下代码, 测试程序, 系统立刻崩溃掉了. //int NT_MASK = 0x00004000;int main(void){__asm__(" mov $0x00004000,%eax #设置 NT 标志 pushl %eax popfl lcall $7,$0 "); return 1;}

查看 KERNEL 的 SRC, 内核代码的处断门部分和处理调用门部分对 EFLAGS 的处理基本相同, 但为什么单是利用调用门的时候才会使系统崩溃呢? 这就需要对 x86 的保护模式有所了解. “一但特权级别及栈已被切换, 如果需要, 中断或异常处理的其余部分可继续进行. EFLAGS 寄存器压栈, 然后将 EFLAGS 寄存器的 NT 及 TF 位置 0. TF=0 表示处理程序不允许单步执行. NT=0 表示处理程序返回时, IRET 返回到同一任务而不是一个嵌套任务. 如果转移是通过一个中断门进行的, 则 EFLAGS 中的 IF 位也被置为 0, 使得进入处理程序后, INTR 中被屏蔽, 如果转移是通过一个陷阱门进行的, 则 IF 位保持不变”[1]而通过调用门进来的, EFLAGS 的 NT, TF 都不会改变. 明白了这些, 只要在应用层为 EFLAGS 设置 NT 位为 1, 那么又 lcall 调用门进入的内核后 NT 位还是为 1, 所以当由内核返回到应用态执行 IRET 指令时, 处理器会认为该任务是个嵌套任务, 所以将进行任务切换, 从当前 TSS 中取出要返回到的任务. 而实际上根本就不是嵌套的任务, 所以将会系统崩溃. 上面报告上说的设置 TF 为 1, 从原理上也应该会使系统挂起.

4　总结与推论

通过以上阐述了解内核级 exploit 和应用层的 exploit 不同之处, 内核缓冲区溢出（kernel BOF）、内核格式化字符串漏洞（kernel format string vul）、内核整型溢出漏洞(kernel integer overflow)、内核 kfree()参数腐败(kernel kfree parameter corruption)、内核编程逻辑错误（kernel program logic error）的原理. 同时发现有部分漏洞应用的

难度很大, 还有待继续研究. 特别是内核 kfree()参数腐败的应用.

参考文献:
[1] 周明德. 保护方式下的 80386 及其编程[M], 北京: 电子工业出版社, 2005.
[2] 李革梅. 嵌入式linux操作系统裁剪和定制研究[D]. 太原: 中北大学, 2005.
[3] 薛筱宇. 基于linux内核的操作系统实验系统[D]. 成都: 西南交通大学, 2003.
[4] 曹云鹏. 关于linux操作系统安全问题的研究[D]. 济南: 山东科技大学, 2005.
[5] 牛玲. 基于linux内核的嵌入式USB接口的加密研究[D]. 成都: 成都理工大学, 2005.
[6] 何昀峰. 一种通用的操作系统安全增强技术的设计与实现[D]. 北京: 中国科学院研究生院（软件研究所）. 2005.

The thoery and application of Linux kernel and its exploitation

LI Chen-xi[1], ZHOU Shao-jun[2]
（1. Southwest University for Nationalities, Chengdu 610041, P.R.C. ;
2. Sichuan Water Vocational and Technical College, Dujiangyan 611830, P.R.C.）

Abstract: The disquisition analyses the difference between the kernel exploitation and application expoitation, expatiates the theory about kernel BOF, kernel format string vul, kernel integer overflow, kernel kfree parameter corruption, and kernel program logic error detailedly.
Key words: kernel; hole; exploitation