内核后门实现及其检测

--Backdoor研究Linux系列

Xfocus
焦点峰会
2002

主要内容

常见的内核后门实现方式

常见的内核后门检测方法

实现方式

内核后门的安插方式

a.可加载内核模块方式

b./dev/kmem方式

c.静态内核补丁

实现方式

可加载内核模块方式

LKM（Loadable Kernel Module）后门顾名思义是以LKM方式实现的后门，程序被编译成具有init_module入口的object程序，然后通过系统命令insmod进行加载，这是最为普遍的内核后门的实现。

实现方式　可加载内核模块方式

根据一个简单例子了解module加载的原理

```
[root@linux-jbtzhm test]#cat hello.c
int init_module()
{
        char s[] = "hello world\n";
        printk("%s\n",s);
        return 0;
}
```

实现方式　可加载内核模块方式

编译后我们得到ELF标准的object文件，然后可以用标准的insmod系统命令进行加载。

[root@linux-jbtzhm test]#gcc -O2 -c hello.c

[root@linux-jbtzhm test]#insmod hello,o

实现方式 可加载内核模块方式

Objdump命令行可以打印出ELF格式的object文件

```
[root@linux-jbtzhm test]# objdump -x hello.o|more
..
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE          VALUE
00000004 R_386_32       .rodata
00000009 R_386_32       .rodata
0000000e R_386_PC32      printk
```

实现方式　可加载内核模块方式

```
[root@linux-jbtzhm test]# objdump -d hello.o
test.o:     file format elf32-i386
Disassembly of section .text:
00000000 <init_module>:
  0:   55                 push   %ebp
  1:   89 e5              mov    %esp,%ebp
  3:   68 00 00 00 00     push   $0x0
  8:   68 0d 00 00 00     push   $0xd
  d:   e8 fc ff ff ff     call   e <init_module+0xe>
 12:   31 c0              xor    %eax,%eax
 14:   c9                 leave
 15:   c3                 ret
```

实现方式　可加载内核模块方式

几个和module相关的系统调用

Sys_create_module

Sys_init_module

Sys_query_module

Sys_delete_module

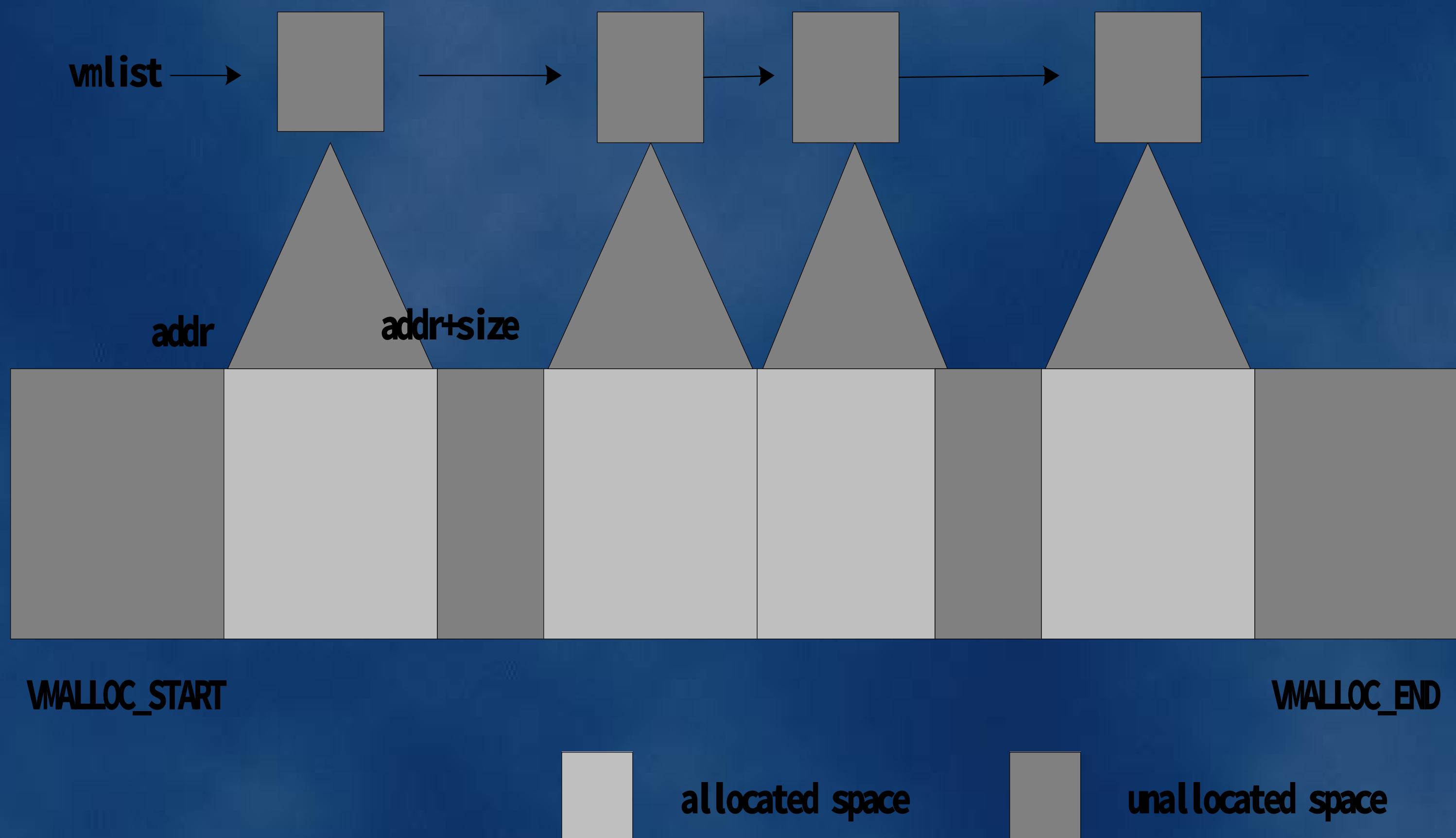
实现方式　可加载内核模块方式

内核提供了像用户空间malloc和free类似的两对独立的函数。

第一对是kmalloc和kfree，管理在内核段内分配的内存——这是真实地址已知的实际和物理内存块。

第二对是vmalloc和vfree，用于对内核使用的虚拟内存进行分配和释放

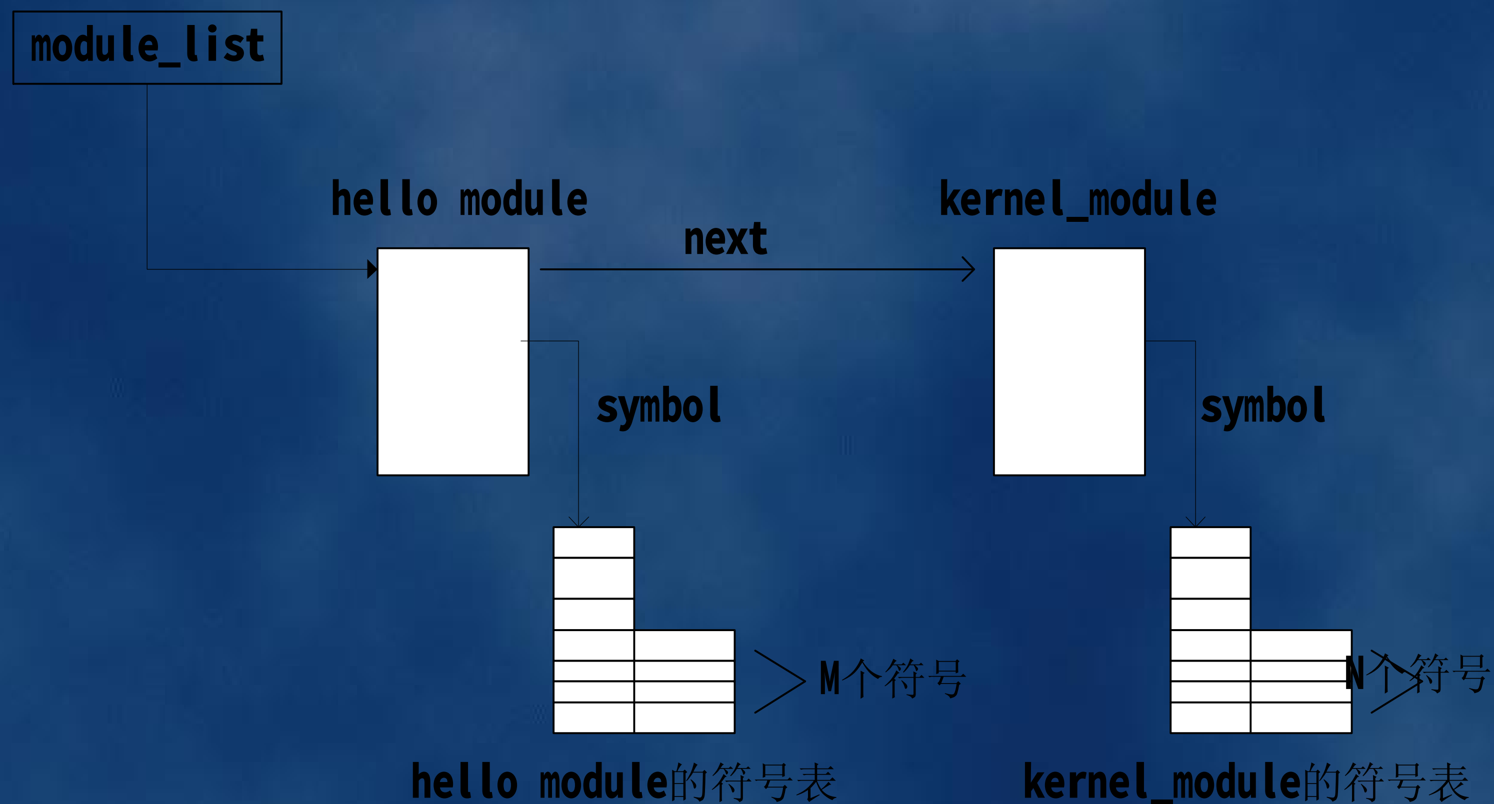
实现方式 可加载内核模块方式



实现方式　可加载内核模块方式

```
struct module
{
        unsigned long size_of_struct;   module结构的大小
        struct module *next;            下一个module
        const char *name;               名字
        unsigned long size;             整个module大小
        unsigned nsyms;                 symbol的个数
        unsigned ndeps;                 依赖module的个数
        struct module_symbol *syms;此module实现的对外输出的所有symbol
        struct module_ref *deps; 依赖的module数组
        struct module_ref *refs;  被引用的module的数组
        int (*init)(void);              用户实现的init_module函数的入口
        void (*cleanup)(void);          用户实现的clean_up函数的入口
};
```

实现方式　可加载内核模块方式



对hello.o执行sys_init_module之后

实现方式　可加载内核模块方式

加载一段代码到内核空间其实主要涉及三个重要的方面：

- 内核符号的定位（用于修复重定位代码）
- 内核空间的分配并复制代码（获取存活空间）
- 修改原有系统执行路径（获得执行权）

实现方式　/dev/kmem方式

系统提供kmem设备文件，使得在用户空间内对内核地址空间的读写成为可能，其实是寻求一种在内核不支持LKM方式时的内核后门安插方式。

获取内核符号地址

分配内核空间（巧妙的利用系统调用实行了kmalloc）

重定位代码

实现方式　静态内核补丁

首先内核符号的定位，它使用的方法可以和/dev/kmem的方法一样，我为了实现简单，只是采用了/boot/System.map这个文件，通过获取相应的kernel的符号地址，我们可以完成修复注入代码（为了通用和便于开发，我们就使用LKM的开发方式）的工作，当然为了完成重定位工作，我们首先需要知道代码的载入地址，也就是相应的第二个问题的，空间的获取。

实现方式　静态内核补丁

其实这对静态内核来讲非常简单，因为我们知道程序空间中有一段非初始化变量区bss，这段区域的数据不会存在静态文件中，但是程序运行时会在内存中留下bss的空间，因此我们只要将这段空间调大一些留给我们的代码使用就可以了。如下图所示

文本段	初始化变量段	**BSS**	注入代码	内核管理的内存 …

实现方式　静态内核补丁

对于获取获取执行权，简单的方法是将一个在启动是必然被调用的系统调用替换就可以了，然后回复这个值，不映像以后的使用。因此因此我们看到解决了上述的三个主要问题，其实剩下的只是一些技术细节去实现就可以了。

后门检测

基本原则

- 自己的存活空间
  - vmalloc,kmalloc,get_free_pages， BSS
- 改变原有系统流程
  - 系统提供的接口(dev_add_pack)
  - 文本段(函数劫持,…)
  - 数据段(中断表，系统调用表)

后门检测

lsmod是linux系统本身提供的命令行参数，但是其在查找lkm方式的backdoor方面基本退出了历史舞台，现有的后门基本没有直接就被发现的情况，但是试试不能绝对，在sunxkdoor的实现中，细心的管理员还是可以用lsmod发现一些蛛丝马迹的（sunx的实现见本系列《module隐藏》一文），由于其并不是将本身的module从module_list上删除，因此还是可以看出不同的，lsmod本身不能发现后门，大多数的用途只是判断是否是正常的module。

后门检测　vmlist链表查找

基本上不考虑**insmod**的发现方式，首先，我们考虑正常加载的LKM程序，考虑分析vmlist链表，根据module结构的特点，分析链表中可能的module节点。

缺点：

易被欺骗

对于kmalloc，get_free_pages分配的空间无法发现

扩展：

但是可以考虑物理内存范围，根据module的结构特点全局查找。

后门检测　验证系统调用表

我们知道linux kernel中export一个重要的全局变量sys_call_table，此表中存储了255（没有全用）系统调用的入口地址，从module的功能上说其一般都会对一些重要的系统调用进行替代。

缺点：

不能确定sys_call_table是否是系统使用的表

内核劫持方式不能被查出

扩展：

memmem (sc_asm,CALLOFF,“\xff\x14\x85”,3);方式取得系统调用表

后门检测　验证系统idr表

此想法和部分程序的实现原自文章《Execution path analysis: finding kernel based rootkits》，中断向量表中的入口地址既然是可以改变的，那么重要的想80的入口完全可以重新实现

后门检测 验证内核文本段

除了对内核数据段数据的检测，对系统代码段也可以进行检查，因为只要我们知道静态的内核映像文件，就可以将其和内存空间的代码段进行判断，通常的内核映像都是经过压缩的，但是从映像中可以找到gzip的magic,将其解压后就可以得到内核的文本段。

后门检测　验证执行路径

取得需要验证的函数地址，然后取得分析机器码，获取函数的执行路径，可以有效的防止内核劫持和其他潜在的后门，具有通用性。

缺点：

对触发性的后门，虚拟执行的流程有可能和系统真正的流程有一些不同。

扩展：

将所有条件跳转都认为是直接跳转，将所有流程都走。

考虑虚拟机的实现，创造相同的上下文

单步方式

后门检测　验证执行路径

单步方式：

思路来自《Execution path analysis》，文中为了计算系统调用的指令个数，在debug的中断函数中加了计数器，其实如果我们把EIP的检测部分加到这里，就可以完成对路径的检测，如果EIP的指向非常可疑，我们即可记录报警，起到检测目的

后话

分析了主要的内核后门实现方式，同时也讨论了一些检测方法，但是我们不得不承认，我们总是事后采取检测后门的存在，因此后门对于检测的工具具有先天的优越性，也就是说公开一种检测方法也就同时以为着我们可能失去一种检测方法，这其实也是矛与盾的不断较量，同时也是促进新技术产生的动力。

相关资料

* linux内核源代码情景分析》　　胡希明　毛德操
* Execution path analysis　　Jan K. Rutkowski
* 《RUNTIME KERNEL KMEM PATCHING》　　Silvio
* linux-kernel-source-2.4.7
* LKM backdoor研究linux系列-insmod源码分析篇
* LKM backdoor研究linux系列-module的隐藏
* LKM backdoor研究linux系列-静态内核补丁

谢谢！

jbtzhm jbtzhm@nsfocus.com

2002.12.27