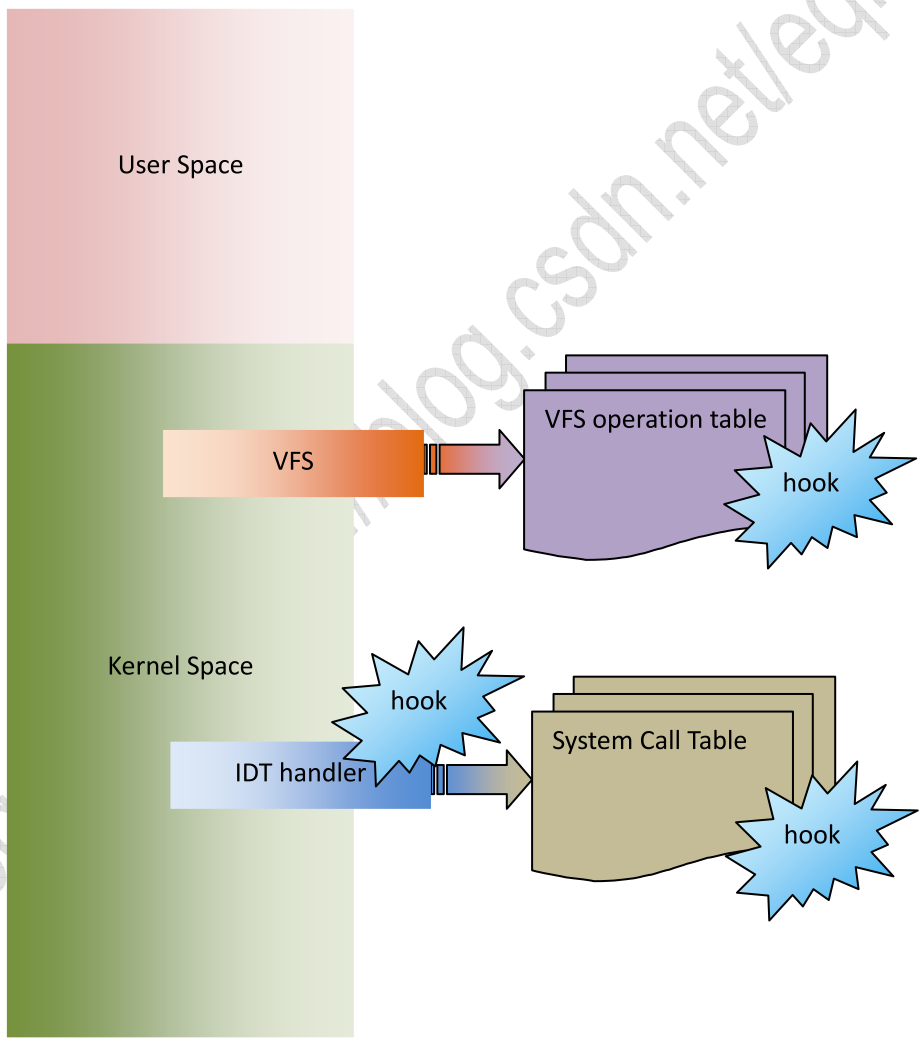
detect rootkit using zeppoo

by eqmcc

Rootkit

there are certain pattern on rootkit's performance: hooking stuff, i.e.: replace the value in certain address in memory. So the best way to detect a rootkit is find these differences in memory space. zeppoo in this case will dump and find out memory difference in following places:

1. IDT handler
2. system call table
3. kernel symbols
4. processes list
5. loaded libraries



zeppoo test run

in this test:

1. fingerprint is dumped
2. wnps is run to hook the IDT handler and hind some process
3. run check against fingerprint and find out the rootkit's hooks and hidden process

test env

os

```
user@ubuntu:~$ uname -a
Linux ubuntu 2.6.10-5-386 #1 Tue Apr 5 12:12:40 UTC 2005 i686 GNU/Linux
```

build dir exist

```
user@ubuntu:~$ ls /lib/modules/2.6.10-5-386/build -l
lrwxrwxrwx    1 root root 35 2013-05-12 05:09 /lib/modules/2.6.10-5-386/build -> /usr/src/linux-headers-2.6.10-5-386
```

compile and install

compile and install

```
user@ubuntu:~/zeppoo-0.0.4$ grep DEBUG Makefile
DEBUG=yes # enable debug
make
```

```
user@ubuntu:~/zeppoo-0.0.4$ make
make[1]: Entering directory `/home/user/zeppoo-0.0.4/libzeppoo'
gcc -Wall -D_DEBUG_  -c -fPIC zeppoo.c
gcc -Wall -D_DEBUG_  -c -fPIC hash.c
gcc -Wall -D_DEBUG_  -c -fPIC md5.c
gcc -Wall -D_DEBUG_  -c -fPIC memory.c
gcc -Wall -D_DEBUG_  -c -fPIC mem.c
gcc -Wall -D_DEBUG_  -c -fPIC kmem.c
gcc -Wall -D_DEBUG_  -c -fPIC version.c
gcc -Wall -D_DEBUG_  -c -fPIC symbols.c
gcc -Wall -D_DEBUG_  -c -fPIC tasks.c
tasks.c: In function `init_taskInfo_k26':
tasks.c:397: warning: long unsigned int format, pointer arg (arg 2)
gcc -Wall -D_DEBUG_  -c -fPIC syscalls.c
gcc -Wall -D_DEBUG_  -c -fPIC idt.c
gcc -Wall -D_DEBUG_  -c -fPIC vfs.c
gcc -Wall -D_DEBUG_  -c -fPIC binaries.c
gcc -o libzeppoo.so -shared zeppoo.o hash.o md5.o memory.o mem.o kmem.o version.o symbols.o tasks.o syscalls.o idt.o vfs.o binaries.o
make[1]: Leaving directory `/home/user/zeppoo-0.0.4/libzeppoo'
gcc -Wall -D_DEBUG_  -c main.c
gcc -Wall -D_DEBUG_  -c tasks.c
gcc -Wall -D_DEBUG_  -c syscalls.c
gcc -Wall -D_DEBUG_  -c idt.c
gcc -Wall -D_DEBUG_  -c symbols.c
gcc -Wall -D_DEBUG_  -c vfs.c
gcc -Wall -D_DEBUG_  -c binaries.c
gcc -Wall -D_DEBUG_  -c fingerprints.c
gcc -o zeppoo main.o tasks.o syscalls.o idt.o symbols.o vfs.o binaries.o fingerprints.o libzeppoo/libzeppoo.so
user@ubuntu:~/zeppoo-0.0.4$
```

dump fingerprint

the cmd and output

```
user@ubuntu:~/zeppoo-0.0.4$ sudo ./zeppoo  -h
HELP :
-z FP           check the system !
-p              display tasks in memory
-s              display syscalls
-i              display idt
-f FILE  generate a fingerprint(syscalls, idt)
-c OPTIONS      check tasks, networks, fingerprints
-d DEVICE       use device(/dev/mem, /dev/kmem)
-m              more quick/portable
-k              use zepprotect
-r              patch the kernel(must be used for REDHAT, UBUNTU and the option -d /dev/mem -m)
-t SYSTEMMAP    specify System.map
-V              version
```

```
user@ubuntu:/tmp/zeppoo-0.0.4$ sudo ./zeppoo -f test_dump
Kernel : 2.6.10-5-386
Kernel : 2.610000
proc : i386
KERNEL_START 0xc0000000
KERNEL_END 0xc1000000
PAGE_OFFSET 0xc0000000
PAGE_MAX 0xffffffff
Memory : /dev/kmem
[+] Begin Generating Fingerprints in test_dump

IDTR BASE 0xc031a000 LIMIT 0x7ff
idt80: flags = 239 flags=EF sel=60 off1=2fc4 off2=C010
SYSTEM_CALL : 0xc0102fc4
Sys Call Table 0xc02a8880
        [+] Begin : Generating Syscalls Fingerprints
        [+] End : Generating Syscalls Fingerprints

IDT TABLE : 0xc031a000 , SIZE : 2047
        [+] Begin : Generating IDT Fingerprints
        [+] End : Generating IDT Fingerprints

        [+] Begin : Generating Symbols Fingerprints
        [+] End : Generating Symbols Fingerprints

[+] proc_root @ 0xc02ae360
[+] proc_root_operations @ 0xc02ae280
[+] proc_root_readdir @ 0xc016f6cc
        [+] 0xe9 => proc_pid_readdir 0xc017127c
        [+] 0xe8 => get_tgid_list 0xc0171173
        [+] 0x81 => init_task 0xc02a5b00
[+] GETTASKS INIT TASK @ 0xc02a5b00
[+] OFFSET NAME 430
[+] OFFSET LIST 88
LIST_ADDR 0xdfba5a38
DIFFADDR 16
JMPFIVEADDR 20
[+] POS = 0x5c
[+] OFFSET NEXT 92
CURRENT_ADDR 0xdfba59e0
[+] OFFSET BINFMT 120
[+] OFFSET PID 148
EGAL 4 0
EGAL 12 1
EGAL 28 2
EGAL 52 3
EGAL 72 4
EGAL 120 5
COUNT 120
[+] OFFSET UID 300
        [+] Begin : Generating Binaries Fingerprints
        [+] End : Generating Binaries Fingerprints

[+] End Generating Fingerprints in test_dump
```

run detect after running WNPS

```
user@ubuntu:/tmp/zeppoo-0.0.4$ sudo ./zeppoo -z test_dump
Kernel : 2.6.10-5-386
Kernel : 2.610000
proc : i386
KERNEL_START 0xc0000000
KERNEL_END 0xc1000000
PAGE_OFFSET 0xc0000000
PAGE_MAX 0xffffffff
Memory : /dev/kmem
[+] proc_root @ 0xc02ae360
[+] proc_root_operations @ 0xc02ae280
[+] proc_root_readdir @ 0xc016f6cc
        [+] 0xe9 => proc_pid_readdir 0xc017127c
        [+] 0xe8 => get_tgid_list 0xc0171173
        [+] 0x81 => init_task 0xc02a5b00
[+] GETTASKS INIT TASK @ 0xc02a5b00
[+] OFFSET NAME 430
[+] OFFSET LIST 88
LIST_ADDR 0xdfba5a38
DIFFADDR 16
JMPFIVEADDR 20
[+] POS = 0x5c
[+] OFFSET NEXT 92
CURRENT_ADDR 0xdfba59e0
[+] OFFSET BINFMT 120
[+] OFFSET PID 148
EGAL 4 0
EGAL 12 1
EGAL 28 2
EGAL 52 3
EGAL 72 4
EGAL 120 5
COUNT 120
[+] OFFSET UID 300

------------------------------------------------------------------------------
[+] Begin : Task

LIST OF HIDDEN TASKS
PID         UID             GID                 NAME          ADDR
11268       1000            1000                bash @ 0x00000000

[+] End : Task
------------------------------------------------------------------------------

[+] Begin Checking Fingerprints in test_dump

[+] [DATE Tuesday 11 June 2013, 09:29 PM]
[+] [INFO Linux i686 2.6.10-5-386]

IDTR BASE 0xc031a000 LIMIT 0x7ff
idt80: flags = 239 flags=EF sel=60 off1=2fc4 off2=C010
SYSTEM_CALL : 0xc0102fc4
Sys Call Table 0xc02a8880
------------------------------------------------------------------------------
[+] Begin : Syscall

LIST OF HIJACK SYSCALLS
POS         MEM                             NAME
289     0x00000018                          UNKNOWN
290     0xffffffff                          UNKNOWN
296     0x0025216f                          UNKNOWN
297     0x00000001                          UNKNOWN
314     0x656d6974                          UNKNOWN
315     0x00000072                          UNKNOWN
```

```
[+] End : Syscall
-------------------------------------------------------------------------------

IDT TABLE : 0xc031a000 , SIZE : 2047
-------------------------------------------------------------------------------
[+] Begin : IDT

LIST OF HIJACK IDT
POS                 MEM                                              NAME
128         0xc0102fc4                                        system_call

[+] End : IDT
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
[+] Begin : Symbols

NO HIJACK SYMBOLS

[+] End : Symbols
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
[+] Begin : Binaries Format

[+] proc_root @ 0xc02ae360
[+] proc_root_operations @ 0xc02ae280
[+] proc_root_readdir @ 0xc016f6cc
        [+] 0xe9 => proc_pid_readdir 0xc017127c
        [+] 0xe8 => get_tgid_list 0xc0171173
        [+] 0x81 => init_task 0xc02a5b00
[+] GETTASKS INIT TASK @ 0xc02a5b00
[+] OFFSET NAME 430
[+] OFFSET LIST 88
LIST_ADDR 0xdfba5a38
DIFFADDR 16
JMPFIVEADDR 20
[+] POS = 0x5c
[+] OFFSET NEXT 92
CURRENT_ADDR 0xdfba59e0
[+] OFFSET BINFMT 120
[+] OFFSET PID 148
EGAL 4 0
EGAL 12 1
EGAL 28 2
EGAL 52 3
EGAL 72 4
EGAL 120 5
COUNT 120
[+] OFFSET UID 300
NO HIJACK BINFMT

[+] End : Binaries Format
-------------------------------------------------------------------------------

[+] End Checking Fingerprints in test_dump
```

the results shows that IDT entry of system call is hijacked and one process is hiding which is exactly what wnps has done.

zeppoo explained

zeppoo features

1. detect hook in IDT entries(IDT handler)
2. detect hook in system call table
3. detect hook in kernel symbols
4. detect hidden process
5. detect hook in loaded libaraies
6. resolve symbols using /boot/System.map

zeppoo in general

When confirmed that the system is clean, zeppoo will make fingerprint by dumping memory space of system call table, IDT entry, kernel symbol list, system task double link chain and loaded libraries of each process. Later, detect of kernel hook can be achieved by compare the values of the check points against the fingerprint.

dump fingerprint in zeppoo

in doFingerprints, following functions will be called to dump various information:

```
writeSyscallsMemory
writeIdtMemory
writeSymbols
writeBinfmt
```

dump system call table

call flow

```
doFingerprints
    writeSyscallsMemory
        getSyscallsMemory
            zeppoo_get_syscalls
                zeppoo_get_syscalltable
                    zepsyscalls.vGetSyscallTable
                        get_syscalltable_i386
                zepsyscalls.vGetSyscalls
                    get_syscalls_kgeneric
            zeppoo_resolve_syscalls // according to /boot/System.map-2.6.10-5-386
        zeppoo_get_syscall_md5sum // get md5 sum
```

zeppoo_get_syscalls

```
asm("sidt %0" : "=m" (idtr)); //read IDTR
//Get the address of system_call from the 0x80th entry of the IDT
zeppoo_read_memory(idtr.base+(2*LENADDR)*0x80, &idt, sizeof(idt));
zepsyscalls.system_call = (idt.off2 << 16) | idt.off1;
```

get_syscalltable_i386

```
zeppoo_read_memory(zepsyscalls.system_call, buffer, 255);
p = (char *)memmem(buffer,255,"\xff\x14\x85",3); // find match
zepsyscalls.sys_call_table = *(unsigned long *)(p + 3);
```

get_syscalls_kgeneric

```
for(i=0;i<_NR_syscalls;i++) // read in each system call's address
   zeppoo_read_memory(zepsyscalls.sys_call_table + (LENADDR*tmp_syscall->pos), &tmp_syscall->addr, LENADDR);
   hash_insert(mysyscalls, key, KEYSIZE, tmp_syscall);
```

dump IDT

call flow

```
doFingerprints
    writeIdtMemory
       getIdtMemory
           zeppoo_get_idt
              zepidt.vGetIdt
                  get_idt_kgeneric
                      idt_table = get_addr_idt();
                      idt_size = get_size_idt();
           zeppoo_resolve_idt // according to /boot/System.map-2.6.10-5-386
       zeppoo_get_idt_md5sum // get md5 sum
```

idt_table = get_addr_idt() and idt_size = get_size_idt()

```
   asm("sidt %0" : "=m" (idtr));
   asm("sidt %0" : "=m" (idtr));
```

get_idt_kgeneric

```
for(i = 0;i < (idt_size + 1)/(LENADDR*2); i++) // read in all idt entry
   zeppoo_read_memory(idt_table+LENADDR*2*i, &idt, sizeof(idt));
   tmpdidt->stub_addr = (unsigned long)(idt.off2 << 16) + idt.off1;
```

dump kernel symbols

call flow

```
doFingerprints
    writeSymbols
       zeppoo_get_symbols // read /proc/kallsyms
```

dump processes

all tasks in the system are maintained in a double linked the chain, so what zeppoo do is find start of the chain(i.e.: init task) and read out any other items in the chain which will not get impacted by any hook in /proc file syetem.

call flow

```
doFingerprints
    writeBinfmt
        getBinfmt
            zeppoo_get_binfmts
                zepbin.vGetBinfmts
                    get_binfmts_k26
                        zeppoo_init_taskInfo
                            zeptasks.vInitTaskInfo
                                init_taskInfo_k26 // determine the offset of interested fields for future use
                                                  // name, binfmt, pid, uidgid, next task in the chain
                                    zeppoo_find_init_task//get address of init_task(swapper with pid 0 ) and it's sub-structure
                                        zepsymb.vFindInitTask
                                            find_init_task_k26
                                                zeppoo_lookup_root
                                                    zepsymb.vLookupRoot
                                                        lookup_root_k26
                                                zeppoo_walk_tree
                        zeppoo_get_task
            zeppoo_get_binfmts_md5sum
            zeppoo_resolve_binfmts
```

find_init_task_k26

find init process through following chain:

proc_root => proc_root_operations => proc_root_readdir => proc_pid_readdir => get_tgid_list => init_task

zeppoo_lookup_root will find directory structure for /proc

```
while (t < zepglob.kernel_end)
     for (i = 0; i < 4096; i++)
            // this is the signature for /proc
            if(buffer[i] == PROC_ROOT_INO && buffer[i+2] == PROC_ROOT_NOTHING && buffer[i+4] == PROC_ROOT_NAMELEN && buffer[i+12] == PROC_ROOT_MODE)
                 zeppoo_read_memory(t+i, proc_root, sizeof(proc_root));
   if(proc_root[16] == 0 && proc_root[20] == 0)
                 return t+i;
```

and proc is inited as bellow in fs/proc/root.c:

```
extern int __init proc_init_inodecache(void);
void __init proc_root_init(void)
{
	int err = proc_init_inodecache();
	if (err)
		return;
	err = register_filesystem(&proc_fs_type);
	if (err)
		return;
	proc_mnt = kern_mount(&proc_fs_type);
	err = PTR_ERR(proc_mnt);
	if (IS_ERR(proc_mnt)) {
		unregister_filesystem(&proc_fs_type);
		return;
	}
	proc_misc_init();
	proc_net = proc_mkdir("net", NULL);
	proc_net_stat = proc_mkdir("net/stat", NULL);

#ifdef CONFIG_SYSVIPC
	proc_mkdir("sysvipc", NULL);
#endif
#ifdef CONFIG_SYSCTL
	proc_sys_root = proc_mkdir("sys", NULL);
#endif
#if defined(CONFIG_BINFMT_MISC) || defined(CONFIG_BINFMT_MISC_MODULE)
	proc_mkdir("sys/fs", NULL);
	proc_mkdir("sys/fs/binfmt_misc", NULL);
#endif
	proc_root_fs = proc_mkdir("fs", NULL);
	proc_root_driver = proc_mkdir("driver", NULL);
	proc_mkdir("fs/nfsd", NULL); /* somewhere for the nfsd filesystem to be mounted */
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
	/* just give it a mountpoint */
	proc_mkdir("openprom", NULL);
#endif
	proc_tty_init();
#ifdef CONFIG_PROC_DEVICETREE
	proc_device_tree_init();
#endif
	proc_bus = proc_mkdir("bus", NULL);
}
```

get structure of proc_root_operations

```
zeppoo_read_memory(proc_root+zepsymb.proc_root_operations, &proc_root_operations, 4);
```

and the init of proc_root_operations is at fs/proc/root.c:

```
/*
 * The root /proc directory is special, as it has the
 * <pid> directories. Thus we don't use the generic
 * directory handling functions for that..
 */
static struct file_operations proc_root_operations = {
	.read		 = generic_read_dir,
	.readdir	 = proc_root_readdir,
};
```

get function of proc_root_readdir

```
zeppoo_read_memory(proc_root_operations+zepsymb.proc_root_readdir, &proc_root_readdir, 4);
```

and the definition of proc_root_readdir is at fs/proc/root.c as well:

```
static int proc_root_readdir(struct file * filp,
	void * dirent, filldir_t filldir)
{
	unsigned int nr = filp->f_pos;
	int ret;

	lock_kernel();

	if (nr < FIRST_PROCESS_ENTRY) {
		int error = proc_readdir(filp, dirent, filldir);
		if (error <= 0) {
			unlock_kernel();
			return error;
		}
		filp->f_pos = FIRST_PROCESS_ENTRY;
	}
	unlock_kernel();

	ret = proc_pid_readdir(filp, dirent, filldir);
	return ret;
}
```

finally in get_tgid_list will find the address of init_task at fs/proc/base.c:

```
/*
 * Get a few tgid's to return for filldir - we need to hold the
```

```
 * tasklist lock while doing this, and we must release it before
 * we actually do the filldir itself, so we use a temp buffer..
 */
static int get_tgid_list(int index, unsigned long version, unsigned int *tgids)
{
	struct task_struct *p;
	int nr_tgids = 0;

	index--;
	read_lock(&tasklist_lock);
	p = NULL;
	if (version) {
		p = find_task_by_pid(version);
		if (p && !thread_group_leader(p))
			p = NULL;
	}

	if (p)
		index = 0;
	else
		p = next_task(&init_task);

	for ( ; p != &init_task; p = next_task(p)) {
		int tgid = p->pid;
		if (!pid_alive(p))
			continue;
		if (--index >= 0)
			continue;
		tgids[nr_tgids] = tgid;
		nr_tgids++;
		if (nr_tgids >= PROC_MAXPIDS)
			break;
	}
	read_unlock(&tasklist_lock);
	return nr_tgids;
}
```

zeppoo_walk_tree will finally identify the the init stask process by checking certain signatures stored at:

```
pKernelSym inittask_sym[] = {
        { "proc_root_readdir", "proc_pid_readdir", 1, 1, 0xe9, 0, 0, 0 },
        { "proc_pid_readdir", "get_tgid_list", 2, 1, 0xe8, 0, 0, 0 },
        { "get_tgid_list", "init_task", 3, 1, 0x81, 0, 0, 0 },
        { NULL, NULL, 0, 0, 0, 0, 0, 0 }
};
```

and also updated as:

```
inittask_sym[0].start = proc_root_readdir;
inittask_sym[2].prefix = zepsymb.get_tgid_list;
```

the checked signatures are:

```
case 0xe9 :
case 0xe8 :
case 0x81 :
case 0x3d :
case 0xc7 :
```

the init taks is inited as bellow in include/linux/init_task.h:

```
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1fffff (=2MB)
 */
#define INIT_TASK(tsk)  \
{                                                                    \
        .state          = 0,                                         \
        .thread_info    = &init_thread_info,                         \
        .usage          = ATOMIC_INIT(2),                            \
        .flags          = 0,                                         \
        .lock_depth     = -1,                                        \
        .prio           = MAX_PRIO-20,                               \
        .static_prio    = MAX_PRIO-20,                               \
        .policy         = SCHED_NORMAL,                              \
        .cpus_allowed   = CPU_MASK_ALL,                              \
        .mm             = NULL,                                      \
        .active_mm      = &init_mm,                                  \
        .run_list       = LIST_HEAD_INIT(tsk.run_list),              \
        .time_slice     = HZ,                                        \
        .tasks          = LIST_HEAD_INIT(tsk.tasks),                 \
        .ptrace_children= LIST_HEAD_INIT(tsk.ptrace_children),       \
        .ptrace_list    = LIST_HEAD_INIT(tsk.ptrace_list),           \
        .real_parent    = &tsk,                                      \
        .parent         = &tsk,                                      \
        .children       = LIST_HEAD_INIT(tsk.children),              \
        .sibling        = LIST_HEAD_INIT(tsk.sibling),               \
        .group_leader   = &tsk,                                      \
        .wait_chldexit  = __WAIT_QUEUE_HEAD_INITIALIZER(tsk.wait_chldexit),\
        .real_timer     = {                                          \
                .function       = it_real_fn                         \
        },                                                           \
        .group_info     = &init_groups,                              \
        .cap_effective  = CAP_INIT_EFF_SET,                          \
        .cap_inheritable = CAP_INIT_INH_SET,                         \
        .cap_permitted  = CAP_FULL_SET,                              \
        .keep_capabilities = 0,                                      \
        .user           = INIT_USER,                                 \
        .comm           = "swapper",                                 \
        .thread         = INIT_THREAD,                               \
        .fs             = &init_fs,                                  \
        .files          = &init_files,                               \
        .signal         = &init_signals,                             \
        .sighand        = &init_sighand,                             \
        .pending        = {                                          \
                .list = LIST_HEAD_INIT(tsk.pending.list),            \
                .signal = {{0}}},                                    \
        .blocked        = {{0}},                                     \
        .alloc_lock     = SPIN_LOCK_UNLOCKED,                        \
        .proc_lock      = SPIN_LOCK_UNLOCKED,                        \
        .switch_lock    = SPIN_LOCK_UNLOCKED,                        \
        .journal_info   = NULL,                                      \
}
```

the task_struct is defined at include/linux/sched.h as bellow:

```
struct task_struct {
        volatile long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        struct thread_info *thread_info;
        atomic_t usage;
        unsigned long flags;     /* per process flags, defined below */
        unsigned long ptrace;

        int lock_depth;          /* Lock depth */

        int prio, static_prio;
        struct list_head run_list;
        prio_array_t *array;

        unsigned long sleep_avg;
        long interactive_credit;
        unsigned long long timestamp, last_ran;
        int activated;

        unsigned long policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
        struct sched_info sched_info;
#endif

        struct list_head tasks;
        /*
```

```
         * ptrace_list/ptrace_children forms the list of my children
         * that were stolen by a ptracer.
         */
        struct list_head ptrace_children;
        struct list_head ptrace_list;

        struct mm_struct *mm, *active_mm;

/* task state */
        struct linux_binfmt *binfmt;
        long exit_state;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        /* ??? */
        unsigned long personality;
        unsigned did_exec:1;
        pid_t pid;
        pid_t tgid;
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->parent->pid)
         */
        struct task_struct *real_parent; /* real parent process (when being debugged) */
        struct task_struct *parent;     /* parent process */
        /*
         * children/sibling forms the list of my children plus the
         * tasks I'm ptracing.
         */
        struct list_head children;      /* list of my children */
        struct list_head sibling;       /* linkage in my parent's children list */
        struct task_struct *group_leader;       /* threadgroup leader */

        /* PID/PID hash table linkage. */
        struct pid pids[PIDTYPE_MAX];

        wait_queue_head_t wait_chldexit;        /* for wait4() */
        struct completion *vfork_done;          /* for vfork() */
        int __user *set_child_tid;              /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;            /* CLONE_CHILD_CLEARTID */

        unsigned long rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        unsigned long utime, stime;
        unsigned long nvcsw, nivcsw; /* context switch counts */
        struct timespec start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt;
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        unsigned keep_capabilities:1;
        struct user_struct *user;
#ifdef CONFIG_KEYS
        struct key *session_keyring;    /* keyring inherited over fork */
        struct key *process_keyring;    /* keyring private to this process (CLONE_THREAD) */
        struct key *thread_keyring;     /* keyring private to this thread */
#endif
        unsigned short used_math;
        char comm[16];
/* file system info */
        int link_count, total_link_count;
/* ipc stuff */
        struct sysv_sem sysvsem;
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
```

```
            struct fs_struct *fs;
/* open file information */
            struct files_struct *files;
/* namespace */
            struct namespace *namespace;
/* signal handlers */
            struct signal_struct *signal;
            struct sighand_struct *sighand;

            sigset_t blocked, real_blocked;
            struct sigpending pending;

            unsigned long sas_ss_sp;
            size_t sas_ss_size;
            int (*notifier)(void *priv);
            void *notifier_data;
            sigset_t *notifier_mask;

            void *security;
            struct audit_context *audit_context;

/* Thread group tracking */
            u32 parent_exec_id;
            u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
            spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
            spinlock_t proc_lock;
/* context-switch lock */
            spinlock_t switch_lock;

/* journalling filesystem info */
            void *journal_info;

/* VM state */
            struct reclaim_state *reclaim_state;

            struct dentry *proc_dentry;
            struct backing_dev_info *backing_dev_info;

            struct io_context *io_context;

            unsigned long ptrace_message;
            siginfo_t *last_siginfo; /* For ptrace use.    */
/*
 * current io wait handle: wait queue entry to use for io waits
 * If this thread is processing aio, this points at the waitqueue
 * inside the currently handled kiocb. It may be NULL (i.e. default
 * to a stack based synchronous wait) if its doing sync IO.
 */
            wait_queue_t *io_wait;
#ifdef CONFIG_NUMA
            struct mempolicy *mempolicy;
            short il_next;                  /* could be shared with used_math */
#endif
};
```

after finding the address of init task, above yellow line will be read out from the memory:

1. name of the process, "\x73\x77\x61\x70\x70\x65\x72" aka ‘swapper’ with PID 0
2. list pointer in the task chain
3. next task in the chain(which is the task init with PID 1)
4. binfmt of next task
5. pid of next task
6. uid of next task

the purpose of read the information out is to calculate the offset of these fields against start of the task_struct which will be used in future to read out task information for other processes.

zeppoo_get_task -> get_tasks_k26

get all tasks in the chain starting from pid 1 which is init task:

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 17:00 ?        00:00:01 init [2]
```

in init_taskInfo_k26 zeptaskinfo.first_addr is inited as current_addr and now in get_tasks_k26, the loop in the chain will started from zeptaskinfo.first_addr:

```
current_addr = zeptaskinfo.first_addr;
do{
    // read task information according to the offset determined in init_taskInfo_k26
    zeppoo_fread_memory(current_addr + zeptaskinfo.offset_name, name, 16);
    zeppoo_fread_memory(current_addr + zeptaskinfo.offset_binfmt, &bin_fmt, LENADDR);
    zeppoo_fread_memory(current_addr + zeptaskinfo.offset_pid, &pid, 4);
    zeppoo_fread_memory(current_addr + zeptaskinfo.offset_uid, &uid, 4);
    zeppoo_fread_memory(current_addr + zeptaskinfo.offset_uid+16, &gid, 4);

    // fill in structure will information retrived
    memcpy(current_task->name, name, sizeof(current_task->name));
    current_task->mybin_fmt.format = bin_fmt;
    current_task->pid = pid;
    current_task->uid = uid;
    current_task->gid = gid;
    current_task->addr = current_addr;
    current_task->mybin_fmt.md5sum_loadbinary = NULL;
    current_task->mybin_fmt.md5sum_loadshlib = NULL;
    current_task->mybin_fmt.md5sum_coredump = NULL;
    current_task->mybin_fmt.name = NULL;

    // get binfmt
    zeppoo_get_binfmt(current_task);

    // next in the chain
    zeppoo_fread_memory(current_addr + zeptaskinfo.offset_list, &list_addr, LENADDR);
    zeppoo_fread_memory(list_addr + zeptaskinfo.offset_next, &current_addr, LENADDR);

}while(current_addr != zeptaskinfo.init_task);
```

following fields are filled by reading offset from task_struct in get_binfmt_k26:

```
    zeppoo_read_memory(mytask->mybin_fmt.format, &mytask->mybin_fmt.next, LENADDR);
    zeppoo_read_memory(mytask->mybin_fmt.format+LENADDR, &mytask->mybin_fmt.module, LENADDR);
    zeppoo_read_memory(mytask->mybin_fmt.format+LENADDR*2, &mytask->mybin_fmt.load_binary, LENADDR);
    zeppoo_read_memory(mytask->mybin_fmt.format+LENADDR*3, &mytask->mybin_fmt.load_shlib, LENADDR);
    zeppoo_read_memory(mytask->mybin_fmt.format+LENADDR*4, &mytask->mybin_fmt.core_dump, LENADDR);
```

linux support different kind of executable format, and each format will be describe by linux_binfmt which is defined at include/linux/binfmts.h as bellow:

```
    struct linux_binfmt {
                // the chain which will be linked into formats chain in the kernel
            struct list_head lh;
            struct module *module;
                // read in executable
            int (*load_binary)(struct linux_binprm *, struct   pt_regs * regs);
                // dynamicly load a shared lib by uselib system call
            int (*load_shlib)(struct file *);
                // core dump file
            int (*core_dump)(struct coredump_params *cprm);
            unsigned long min_coredump;      /* minimal dump size */
    };
```

detect rootkit in zeppoo

during detect process, in function checkFingerprints, all the steps of dump fingerprint will be repeated again with addition information checked as well. Following functions will be called to achieve this:

```
viewHijackSyscalls
viewHijackIdt
viewHijackSymbols
viewHijackBinfmt
```

detect hijacked system call in viewHijackSyscalls

in this procedure, current system call table will be read out from memory again via function getSyscallsMemory and later to be checked against previous dumped fingerprint in following section in the dump file:

```
[BEGIN SYSCALLS]
… …
[END SYSCALLS]
```

any entry has change in md5sum will be marked as hijacked

detect hijacked IDT in viewHijackIdt

in this procedure, current IDT will be read out from memory again via function getIdtMemory and later to be checked against previous dumped fingerprint in following section in the dump file:

```
[BEGIN IDT]
… …
[END IDT]
```

any entry has change in md5sum will be marked as hijacked

detect hijacked kernel symbol in viewHijackSymbols

in this procedure, current kernel symbol list will be read out from memory again via function getSymbolsFingerprints and later to be checked against previous dumped fingerprint in following section in the dump file:

```
[BEGIN SYMBOLS]
… …
[END SYMBOLS]
```

for the hijacking of kernel symbols, most possible way is put a JMP OP code in the symbol’s place, so function zeppoo_search_jmp will be called to check if there’s JMP instruction in original kernel symbol's space.

detect hijacked process in viewHijackBinfmt

in this function, getBinfmtFingerprints will read in all the dumped fingerprint for processes in following section in the dump file:

```
[BEGIN BINFMT]
… …
[END BINFMT]
```

and later checkBinfmt function will check all the fingerprint entry against current living kernel, so it will dump all the processes' info again via zeppoo_init_taskInfo and zeppoo_get_tasks to a temp data structure and compare against the fingerprint loaded.

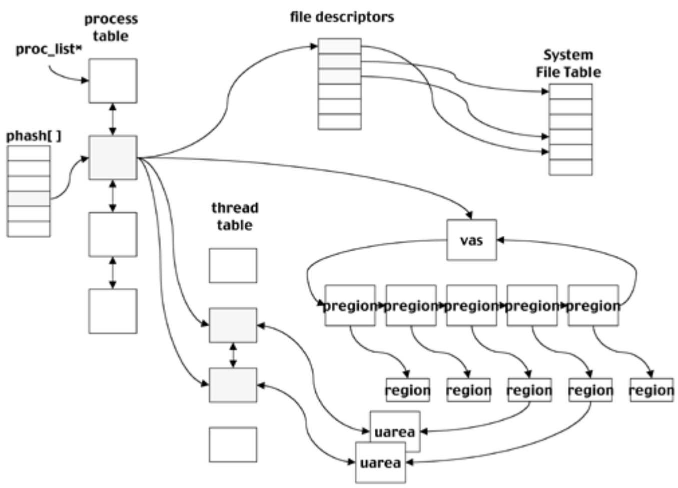
detect hidden process in viewHiddenTasks

in this procedure, will get process information from several places and compared against each other, any difference between will help identify that particular hidden process, so process information in memory, /proc file system, ps command result and kill test will be retrieved in following functions:

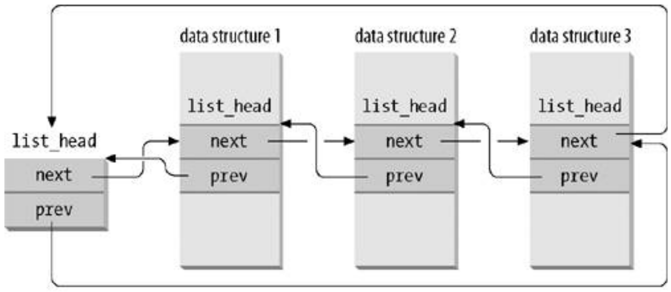
```
getTasksMemory(tasksmemory);
getTasksProc(tasksproc);
getTasksProcForce(tasksprocforce);
getTasksPS(tasksps);
getTasksKill(taskskill);
```

getTasksMemory

as did in dump fingerprint before, will call zeppoo_init_taskInfo and zeppoo_get_tasks to read out task list from kernel which will provide exact list of processes running in this machine.
the processes and threads are organized in following form:



using a double linked list:



getTasksProc

this function will read out all the directories matching "/proc/PID/status"(process) and "/proc/PID/task/LWPID"(threads) to find out all processes and threads inside a process, the result here can be hijacked already.

getTasksProcForce

his function will blindly check all PIDs from 1 to 65535 and trying to read out "/proc/PID/status" and "/proc/PID/task/LWPID".

getTasksPS

read from output of ps command:

```
output = popen("/bin/ps -eo user,pid,uid,gid,state,fname", "r")
```

getTasksKill

kill(PID,0) function will test and return valid pid, and this function is called here to again blindly check all PIDs from 1 to 65535 to get alive PID in the system.

after the dump, following results will be checked against each other:

Source of dump	Memory	/proc/PID	/proc/1-65565	PS command	Kill(PID,0)
Memory		Yes			
/proc/PID				Yes	
/proc/1-65565		Yes			
PS command					
Kill(PID,0)		Yes			

with bellow functions:

```
checkTasks(tasksproc, tasksps, taskscheck);
checkTasks(tasksprocforce, tasksproc, taskscheck);
checkTasks(tasksmemory, tasksproc, taskscheck);
checkTasks(taskskill, tasksproc, taskscheck);
```

view all processes in memory via viewTaskMemory

read processes information from kernel via zeppoo_init_taskInfo and zeppoo_get_task and show it.