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
Some very interesting code
Some very interesting code

```c
int main(int argc, char *argv[])
{
    // Some interesting code...
    // ...

    // Other interesting code...
    // ...
}

void patch_syscall(unsigned long mirror_kaddr)
{
    // More interesting code...
    // ...
}
```
taimen/data/local/tmp $ ./exp_demo
 [+] pwned!
taijen/data/local/tmp $ getprop ro.product.model & getenforce
Pixel 2 XL
Enforcing

taijen/data/local/tmp $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_rw),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_hw_stats),3009(readproc),3011(uhid)
contextu::r:shell:s0
taijen/data/local/tmp $
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
• 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

- 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

Thread-1

- strlen(file_name)
- kmalloc(alloc_len, GFP_KERNEL);
- strcpy(event->name, file_name);

Thread-2

- kfree_rcu(old_name, u.head);
- add_key("spray", payload, 0, 0, 0);

Heap overflow
## Vulnerability analysis

### Kmalloc-256

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>freed</td>
<td>...</td>
<td>freed</td>
<td>freed</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>...</td>
<td>249</td>
<td>freed</td>
</tr>
<tr>
<td>freed</td>
<td>200</td>
<td>...</td>
<td>249</td>
<td>freed</td>
</tr>
<tr>
<td>256</td>
<td>200</td>
<td>...</td>
<td>249</td>
<td>freed</td>
</tr>
<tr>
<td>256</td>
<td>200</td>
<td>...</td>
<td>44</td>
<td>freed</td>
</tr>
</tbody>
</table>

- **T1:** trigger
- **T1:** kmalloc
- **T2:** rename
- **T2:** spray
- **T1:** strcpy
- **Non-zero**
- **Overflow**
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

```c
static ssize_t do_readv_writev(int type, struct file *file, const struct iovec_user *iuvector, unsigned long nr_segs, loff_t *pos)
{
    size_t tot_len;
    struct iovec iovstack[UIO_FASTIOV];
    struct iovec *iov = iovstack;
    struct iov_iter iter;
    ssize_t ret;
    io_fn_t fn;
    iter_fn_t iter_fn;

    ret = import_iovec(type, uvector, nr_segs,
                       ARRAY_SIZE(iovstack), &iov, &iter);

    // Further code...
}
```
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

```c
pipe_read(struct klock *lockb, struct iov_iter *io)
{
    size_t total_len = iov_iter_count(io);
    struct file *flp = lockb->filp;
    struct pipe_inode_info *pipe = flp->private_data;
    int do_wakeup;
    ssize_t ret;

    /* Null read succeeds. */
    if (unlikely(total_len == 0))
        return 0;
    do_wakeup = 0;
    ret = 0;
    _pipe_lock(pipe);
    for (;;) {
        int bufs = pipe->nrbufs;
        if (bufs) {
            int curbuf = pipe->curbuf;
            struct pipe_buffer *buf = pipe->bufs + curbuf;
            const struct pipe_buf_operations *ops = buf->ops;
            size_t chars = buf->len;
            size_t written;
            int error;

            if (chars > total_len)
                chars = total_len;
            error = ops->confirm(pipe, buf);
            if (error) {
                if (!ret)
                    ret = error;
                break;
            }
            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
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 iovcvs 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
  - 0xFFFFFC000D0E1CC
  - kernel data contains ideal callback pointers

```
thomaskingdeMacBook-Pro:sm thomasking$ cat System.map |grep "A _data"
ffffffffc0015f000 A _data
```

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

```
thomaskingdeMacBook-Pro:sm thomasking$ cat System.map |grep "A _end"
ffffffffc001a36000 A _end
```
## Ideal heap layout

### Kmalloc-256

<table>
<thead>
<tr>
<th></th>
<th>freed</th>
<th>freed</th>
<th>freed</th>
<th>...</th>
<th>freed</th>
<th>freed</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>iovs</td>
<td>iovs</td>
<td>iovs</td>
<td>...</td>
<td>iovs</td>
<td>iovs</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>freed-iov</td>
<td>iovs</td>
<td>freed-iov</td>
<td>...</td>
<td>freed-iov</td>
<td>iovs</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td>iovs</td>
<td>freed-iov</td>
<td>...</td>
<td>event</td>
<td>iovs</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>payload</td>
<td>iovs</td>
<td>name</td>
<td>...</td>
<td>event</td>
<td>iovs</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>payload</td>
<td>iovs</td>
<td>name</td>
<td>...</td>
<td></td>
<td>iovs</td>
<td></td>
</tr>
</tbody>
</table>
Shape the heap

• Many ‘hole’s in the heap

| freed | Obj-A | Obj-B | freed | freed | Obj-C | ...
|-------|-------|-------|-------|-------|-------|-------|

• Fill with events
  • Full list
    | Event-1 | Obj-A | Obj-B | Event-2 | Event-3 | Obj-C | ...
|-------|-------|-------|--------|---------|-------|-------|

• New empty list

| freed | freed | freed | freed | freed | freed | freed | ...
|-------|-------|-------|-------|-------|-------|-------|-------|
• Freed buffer holes
  • Trigger notifications with different actions
  • Not merge

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

User space

Kernel space

ptmx_fops

check_flags

0x40002000

0x40002028

0x40002048

kernel_sock_ioctl_end

---

**export kernel_sock_ioctl**

---

```assembly
var_10 = -0x10
var_s0 = 0

STP X20, X19, [SP,#-0x10+var_10]
STP X29, X30, [SP,#0x10+var_s0]
ADD X29, SP, #0x10
MRS X19, #0, c4, c1, #0
MOV X8, #0xFFFFFFFFFFFFFFFF
LDR X20, [X19,#8]
STR X8, [X19,#8]
LDR X8, [X0,#0x28]
LDR X8, [X8,#0x48]
BLR X8
STR X20, [X19,#8]
LDP X29, X30, [SP,#0x10+var_s0]
LDP X20, X19, [SP+0x10+var_10], #0x20
RET
```

---

End of function kernel_sock_ioctl
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 😞

```
struct external_name {
    union {
        atomic_t count;
        struct rcu_head head;
    } u;
    unsigned char name[];
};
```
Bypassing KASLR

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

Offset within Process PGD

Offset within PMD Page Frame

Offset within PTE Page Frame

Offset within Data Frame

pgd_offset() -> pgd_t

pmd_offset() -> pmd_t

pte_offset() -> pte_t

Page Frame
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
- 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.
• ARMv8-64 level 3 descriptor format

† Upper page attributes and Lower page attributes
‡ Field is RES0
General view of address translation

No level 0 table for Android

- D_Table is a Table descriptor
- D_Block is a Block descriptor
- D_Page is a Page descriptor

- a Indexed by IA[n:39], where IA width is (n+1) bits
- b Indexed by IA[38:30]
- c Indexed by IA[29:21]
- d Indexed by IA[20:12]
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,

<table>
<thead>
<tr>
<th>AP[2:1]</th>
<th>Access from EL1</th>
<th>Access from EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Read/write</td>
<td>None</td>
</tr>
<tr>
<td>01</td>
<td>Read/write</td>
<td>Read/write</td>
</tr>
<tr>
<td>10</td>
<td>Read-only</td>
<td>None</td>
</tr>
<tr>
<td>11</td>
<td>Read-only</td>
<td>Read-only</td>
</tr>
</tbody>
</table>
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

#### Level 1 table

<table>
<thead>
<tr>
<th>Start of 1GB region</th>
<th>Level 1 table</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFFFFFFC00000000</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFC240000000</td>
<td>0</td>
</tr>
<tr>
<td>FFFFFFFC200000000</td>
<td>0</td>
</tr>
<tr>
<td>FFFFFFFC1C0000000</td>
<td>0</td>
</tr>
<tr>
<td>FFFFFFFC180000000</td>
<td>0</td>
</tr>
<tr>
<td>FFFFFFFC140000000</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFC080000000</td>
<td>D_Table</td>
</tr>
<tr>
<td>FFFFFFFC040000000</td>
<td>D_Table</td>
</tr>
<tr>
<td>FFFFFFFC000000000</td>
<td>D_Table</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>FFFFFFF80000000000</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Physical memory (3GB)

- **Text and Data**
- **Freed Pages**
- **Freed Pages**

**D_Table** is a Table descriptor

**D_Block** is a Block descriptor
Principle of KSMA

Start of 1GB region

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFFFFFFC00000000</td>
<td>Text and Data</td>
</tr>
<tr>
<td>FFFFFFFC2400000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC2000000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC1C000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC18000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC14000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC08000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC04000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFFC00000000</td>
<td>Freed Pages</td>
</tr>
<tr>
<td>FFFFFFF8000000000</td>
<td>Freed Pages</td>
</tr>
</tbody>
</table>

Level 1 table

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D_Block</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>D_Table</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>D_Table</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

PA: 30002000

AP[2:1] = ‘01’

R/W from EL0 & EL1, break kernel isolation

Physical memory (3GB)

D_Table is a Table descriptor
D_Block is a Block descriptor
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:
    • Step 1: Prepare a special block descriptor
    • Step 2: Calculate the entry address (No ‘0’ bytes)
    • Step 3: Spawn race threads and win the race
    • Step 4: Disable SELinux
      • Write ‘0’ to the mirroring addresses of ‘selinux_enable’ and ‘selinux_enforcing’
    • Step 5: Patch a syscall
      • Write shellcode to the mirroring address
    • Step 6: 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
- \( \text{AP}[2:1] = '01' \)
Rooting Android 8 Demo
CPRooter Rooting Solution
• 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
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
• 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

- Virtual address
  - Kernel space
    - VDSO RW-
    - VDSO R-X
  - User space

- Physical Pages
  - TTBR0
  - TTBR1
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
• 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.
• 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
• ARM® Architecture Reference Manual(ARMv8, for ARMv8-A architecture profile)
• ret2dir: Rethinking Kernel Isolation (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
Thank you

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

ThomasKingNew@gmail.com