AH! UNIVERSAL ANDROID ROOTING IS BACK

Wen `Memeda` Xu
@K33nTeam

KEEN TEAM
ABOUT ME

Wen Xu a.k.a Memeda @antlr7

- Security research intern at KeenTeam
- Android Rooting
- Software exploitation
- Senior student at Shanghai Jiao Tong University
- Member of LoCCS
- Vice-captain of CTF team 0ops
- Rank 2nd in the world on CTFTIME
AGENDA

• Present Situation of Android Rooting
• Awesome Bug (CVE-2015-3636)
  • Fuzzing
  • Analysis
• Awesome Exploitation Techniques
  • Object Re-filling in kernel UAF
  • Kernel Code Execution
  • Targeting 64bit Devices
• Future
PART I

Present Situation
PRESENT SITUATION

Root for what?

- Goal
  - uid=0(root) gid=0(root) groups=0(root)
- Kernel arbitrary read/write
  - Cleaning
  - SELinux
  - ...

PRESENT SITUATION

- SoC (Driver)
  - Missing argument sanitization (ioctl/mmap)
    - Qualcomm camera drivers bug
      CVE-2014-4321, CVE-2014-4324
      CVE-2014-0975, CVE-2014-0976
  - TOCTTOU
    - Direct dereference in user space
      CVE-2014-8299
- Chip-by-chip
A BIG DEAL

- Universal root solution
  - Universally applied bug
    - Confronting Linux kernel
  - Universally applied exploitation techniques
    - One exploit for hundreds of thousands of devices
    - Adaptability (Hardcode)
    - User-friendly (Stability)
- COMING BACK AGAIN!

PINGPONG ROOT
FUZZING

Open source kernel syscall fuzzer

- Trinity
  - https://github.com/kernelslacker/trinity
- Scalability
- Ported to ARM Linux
Let's take a look at our log when we wake up ;)

- Critical paging fault at 0x200200?!!

```
[ 3354.778717] Unable to handle kernel paging request at virtual address
[ 3354.778839] pgd = ea574000
[ 3354.778900] [00200200] *pgd=00000000
[ 3354.779052] Internal error: Oops: 805 [#1] PREEMPT SMP ARM
[ 3354.779144] Modules linked in:
[ 3354.779266] CPU: 1  Tainted: G   W   (3.4.0-Kali-g006dd6c #1)
[ 3354.779357] PC is at ping_unhash+0x50/0xd4
[ 3354.779479] LR is at _raw_write_lock_bh+0xc/0x8c
[ 3354.779541] pc : [<c08b18c>]  lr : [<c09f7d9>]  psr: 20010013
[ 3354.779541] sp : e99a5ee0  ip : c08a67ac  fp : 00000000
[ 3354.779724] r10 : 00000000  r9 : e99a4000  r8 : c000e928
[ 3354.779846] r7 : 00000000  r6 : 00000000  r5 : 00000000  r4 : eb3d200
[ 3354.779907] r3 : 00000000  r2 : 00200200  r1 : 00000000  r0 : c14ed98
[ 3354.780029] Flags: nzCv  IRqs on  FIqs on Mode SVC_32  ISA ARM  Segment user
[ 3354.780120] Control: 10c5787d  Table: ab974064  DAC: 00000015
```
```c
void ping_unhash(struct sock *sk)
{
    struct inet_sock *isk = inet_sk(sk);
    pr_debug("ping_unhash(isk=\%p, isk->num=\%u)\n",
            isk, isk->inet_num);
    if (sk_hashed(sk)) {
        write_lock_bh(&ping_table.lock);
        hlist_nulls_del(&sk->sk_nulls_node);
        sock_put(sk);
        isk->inet_num = 0;
        isk->inet_port = 0;
        sock_prot_inuse_add(ssock_net(sk), sk->sk_prot, -1);
        write_unlock_bh(&ping_table.lock);
    }
}
EXPORT_SYMBOL_GPL(ping_unhash);
```

**SK: PING SOCKET OBJECT IN KERNEL**

```c
user_sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```
LIST_POISON2 == 0x200200

ping_unhash
(__hlist_nulls_del)
TWO times

0x200200 not mapped

kernel crash
disconnect() in kernel

through

connect() in user program
Local denial of service? Not enough!

- **Avoid crash:** map 0x200200 in the user space
- Then hmm...
- **sock_put(sk)** is called twice ;)
- What’s **PUT**?
USE-AFTER-FREE

CVE-2015-3636

- `sock_put(sk)` twice → Ref count to 0 → `sk_free`!
- A dangling file descriptor left in the user program

```c
static inline void sock_put(struct sock *sk) {
    if (atomic_dec_and_test(&sk->sk_refcnt))
        sk_free(sk);
}
```
PART III

Exploitation
WHEN IT COMES TO UAF

ANNOYING PROBLEM RE-FILLING

- Our target: `struct sock object`
- `kmem_cache_alloc("PING", priority & ~__GFP_ZERO);`
- Custom use cache ;(
A specific area for the allocation of kernel objects of particular type

Here we meet the type called “PING” 😂
WE ARE IN THE KERNEL

RE-FILLING IS REALLY A TOUGH JOB

- **Slab allocator**
  - Hmm...Just like Isolated Heap
- **Multi-thread/core**
  - Hard to achieve fully predictable heap layout

- **Candidate kernel objects**
  - Lack of controllability
- **Controllable content?**
  - No BSTR in kernel lol
WHAT USED TO RE-FILL

Initial idea: `kmalloc()` buffer

- Common use slab cache
- Several choices on size
  - 32, 48, 64, 128, 256, 512, 1024...
- How to create: `sendmmsg()`
- Size control: length of control msg
- Content control: content of control msg
INTUITIVE IDEA

Basically, a completely free slab has large probability to be recycled for future allocation

• Why we always enjoy use-after-free bug?
• **Memory reuse** for efficiency and optimization
• No exception in kernel
• 1. Fill slabs with totally PING socket objects
• 2. Free all of them and spray kmalloc-x buffers
• Exactly possible, but ... **out of control**
Newly adopted SLUB allocator tries to put the objects of the same size together, which de-separates the kernel objects to some extent.

- Does our target object have a size of 32, 48, 64, 128, 256 or 512?
- Use `kmalloc()` buffers to re-occupy
- Much more stable and accurate
- **BUT** ping socket objects on different devices have different sizes
- Also the sizes may not be 32, 48, 64, ...
KERNEL MEMORY CONTROL

Universal Solution #1
RET2DIR

• ret2dir: Rethinking Kernel Isolation (USENIX 14’)
  • Vasileios P. Kemerlis Michalis Polychronakakis
    Angelos D. Keromytis
  • Physmap is originally used to bypass kernel protections
  • SMEP, SMAP, PXN, PEN ...
• Will it help to exploit kernel use-after-free as well?
Physmap, the direct-mapped memory, is memory in the kernel which would directly map the memory in the user space into the kernel space.
Again, based on the natural weakness of the system: **MEMORY REUSE**

- **How to create**: iteratively `mmap()` in user space
- **Data control**: fully user-controlled (fill `mmap()`’ed area with our payload)
- **Physmap** with payload grows by occupying the free memory in the kernel
**THE RETURN OF PHYSMAP**

With **physmap**, we are able to exploit UAF in the kernel regardless of what vulnerable object is...

- **Size control:**
- **Large enough to fill any freed memory in the theoretically**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>PHYS_OFFSET</th>
<th>Size</th>
<th>Prot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 (3G/1G)</td>
<td>0xC0000000</td>
<td>891MB</td>
<td>RW</td>
</tr>
<tr>
<td>x86 (2G/2G)</td>
<td>0x80000000</td>
<td>1915MB</td>
<td>RW</td>
</tr>
<tr>
<td>x86 (1G/3G)</td>
<td>0x40000000</td>
<td>2939MB</td>
<td>RW</td>
</tr>
<tr>
<td>AArch32 (3G/1G)</td>
<td>0xC0000000</td>
<td>760MB</td>
<td>RWX</td>
</tr>
<tr>
<td>AArch32 (2G/2G)</td>
<td>0x80000000</td>
<td>1784MB</td>
<td>RWX</td>
</tr>
<tr>
<td>AArch32 (1G/3G)</td>
<td>0x40000000</td>
<td>2808MB</td>
<td>RWX</td>
</tr>
<tr>
<td>x86-64</td>
<td>0xFFF880000000000</td>
<td>64TB</td>
<td>RW (X)</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFF0000000000</td>
<td>256GB</td>
<td>RWX</td>
</tr>
</tbody>
</table>

Table 1: **physmap** characteristics across different architectures (x86, x86-64, AArch32, AArch64).
INITIAL PLAN

**Achieve kernel spraying through user space spraying**

- 1. Allocate a large number of ping socket objects and then free all of them by triggering the bug.
- 2. Iteratively call `mmap()` in the user program and fill the area.
- Hope the memory collision will happen?
RELIABILITY

Universal Solution #2
RELIABLE MEMORY COLLISION

Make PING socket objects and physmap overlapped ;)

• Spray PING socket objects

• In each step, every 500 PADDING PING objects
  • Make our PING socket objects appear everywhere in kernel space

• 1 TARGET PING objects
  • Used to pwn
CREATE LEAK

Universal Solution #3
Find an info leak to know whether our targeting PING socket object has already been covered by physmap or not
Notice: certain adjustment and optimization in practical root tool

- Allocate hundreds of PING socket objects in group.
- Every 500 padding objects with 1 targeting object considered as a vulnerable one.
- Free padding PING socket objects normally by calling close()
- Free targeting PING socket objects by triggering the bug
- Such de-allocation generates large pieces of free memory for physmap
- Iteratively call mmap() in user space and fill the areas
- Payload + magic number for re-filling checking
- Iteratively call ioctl() on targeting PING socket objects
  - ioctl() returns magic number? Done.
  - Otherwise further physmap spraying is needed.
UNLEASH KERNEL UAF

- A generic memory collision model in Linux kernel
- Solve several difficulties when exploiting kernel use-after-frees
- Hard to mitigate due to kernel’s inherent property
Now we have **full control** of the content of a freed PING object with the corresponding dangling **fd** in our hand

- **User**: just close(fd)
- **Kernel**: inet_release called
  - ‘vftable’: sk->sk_prot
  - Set **sk_prot** as a mapped virtual address in user space
- **Return to user land shellcode**
WHAT DOES SHELLCODE DO

Geohot taught us again in Towelroot

- Leak kernel stack address
- Get thread_info address
- Change addr_limit to 0
- Achieve kernel arbitrary read/write through pipe

Original idea from Stackjacking Your Way to grsec/PaX Bypass
**WHAT ABOUT 64BIT DEVICES**

Don't count your chickens before they hatch.
Our goal is to root whatever devices of whatever brands.

- Bug existed? **YEAH**
- LIST_POISON2?
- Still **0x200200** which can be mapped **YEAH**
- Memory collision with **phsymap? **YEAH**
- Return to shellcode in user space? **NO!**
OOPS! PXN APPLIED.

PXN prevents userland execution from kernel

- Return to `physmap? NX ;(`
- ROP comes on stage
- Two steps
  - First step: leak kernel stack address
  - Second step: change `addr_limit` to 0
- Hardcoded addresses of gadgets ;(
In fact we prefer **JOP** (Jump-Oriented Programming)

- Avoid stack pivoting in kernel which brings uncertainty
- Make full use of current values of the registers
  - **X29** stores **SP** value on 64bit devices
  - **High 32bits of kernel addresses** are the same
  - Only need to read/write **low 32bits**
- Work hard to find cool gadgets
  - **One GOD gadget** does both leaking and overwriting in some ROMs
CONCLUSION

Victory!

- Root most popular Android devices on market
  - Android version $\geq 4.3$
  - First 64bit root case in the world as known
- S6 & S6 Edge root
- DEMO ;}
PART IV

Future
64bit devices could be more secure

- LIST_POISON2 in 64bit Android kernel
- 0x200200 Set as 0xDEAD000000000000
- Prevent memory collision with physmap
- Enough virtual address space there
- KASLR
- Days become harder for linux kernel pwners
- Where there is a will there is a way
In this paper, 3 pwnies nominations were leveraged.

- Wu Shi - Finding CVE-2015-3636
- Universal Root - Privilege escalation
- ret2dir - Exploitation technique
ACKNOWLEDGEMENT

Thanks for contributions and inspirations

- wushi
- jfang
- LeoC
- Liang Chen
- Slipper
- Peter

Pictures in the slide come from Google
THANK YOU

Wen Xu
@antlr7
hotdog3645@gmail.com