



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
 - Main focus on Android rooting and software exploitation
 - Pwn2Own 2015, Adobe Reader category
- Senior student at Shanghai Jiao Tong University
 - Member of LoCCS
- Vice-captain of CTF team Oops
 - CodeGate Champion
 - 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**
- **TOCTOU**
 - **Direct dereference in user space CVE-2014-8299**
- **~~Chip by chip ?~~**



A BIG DEAL

- Universal root solution
 - Once Towelroot(futex bug) by Geohot(Pinkie Pie)
 - Universally applied bug
 - Confronting Linux kernel
 - Universally applied exploitation techniques
 - One exploit for hundreds of thousands of devices
 - Adaptability (Hardcode)
 - User-friendly (Stability)



PINGPONG ROOT

- **COMING BACK AGAIN!**



PART II

Bug Hunting



FUZZING

Open source kernel syscall fuzzer

Trinity

- <https://github.com/kernelslacker/trinity>
 - Easy to use
 - Scalability
- Ported to ARM Linux





FUZZING

Let's take a look at our log when we wake up ;)

- Critical paging fault at 0x200200?!!

```
2301 [ 3354.778717] Unable to handle kernel paging request at virtual address 00200200
2302 [ 3354.778839] pgd = ea574000
2303 [ 3354.778900] [00200200] *pgd=00000000
2304 [ 3354.779052] Internal error: Oops: 805 [#1] PREEMPT SMP ARM
2305 [ 3354.779144] Modules linked in:
2306 [ 3354.779266] CPU: 1 Tainted: G W (3.4.0-Kali-g006dd6c #1)
2307 [ 3354.779357] PC is at ping_unhash+0x50/0xd4
2308 [ 3354.779479] LR is at _raw_write_lock_bh+0xc/0x8c
2309 [ 3354.779541] pc : [<c08b18b>] lr : [<c09f7d9>] psr: 20010013
2310 [ 3354.779541] sp : e99a5ee0 ip : c08a67ac fp : 00000000
2311 [ 3354.779724] r10: 00000000 r9 : e99a4000 r8 : c000e928
2312 [ 3354.779846] r7 : 0000011b r6 : 00000053 r5 : 00000000 r4 : eb3cd200
2313 [ 3354.779907] r3 : 00000003 r2 : 00200200 r1 : 00000000 r0 : c144ed98
2314 [ 3354.780029] Flags: nzCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
2315 [ 3354.780120] Control: 10c5787d Table: ab97406a DAC: 00000015
```

```
34 void ping_unhash(struct sock *sk)
35 {
36     struct inet_sock *isk = inet_sk(sk);
37     pr_debug("ping_unhash(isk=%p,isk->num=%u)\n",
38             isk, isk->inet_num);
39     if (sk_hashed(sk)) {
40         write_lock_bh(&ping_table.lock);
41         hlist_nulls_del(&sk->sk_nulls_node);
42         sock_put(sk);
43         isk->inet_num = 0;
44         isk->inet_sport = 0;
45         sock_prot_inuse_add(sock_net(sk), sk->sk_prot, -1);
46         write_unlock_bh(&ping_table.lock);
47     }
48 }
49 EXPORT_SYMBOL_GPL(ping_unhash);
```



SK: PING SOCKET OBJECT IN KERNEL

```
user_sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

```
71 static inline void __hlist_nulls_del(struct hlist_nulls_node *n)
72 {
73     struct hlist_nulls_node *next = n->next;
74     struct hlist_nulls_node **pprev = n->pprev; 2
75     *pprev = next; 3
76     if (!is_a_nulls(next))
77         next->pprev = pprev;
78 }
79
80 static inline void hlist_nulls_del(struct hlist_nulls_node *n)
81 {
82     __hlist_nulls_del(n);
83     n->pprev = LIST_POISON2; 1
84 }
```



LIST_POISON2 == 0X200200

ping_unhash(__hlist_nulls_del)

TWO times



0x200200 not mapped

kernel crash


```
1 int inet_dgram_connect(struct socket *sock,  
2                       struct sockaddr * uaddr,  
3                       int addr_len, int flags)  
4 {  
5     struct sock *sk = sock->sk;  
6  
7     if (addr_len < sizeof(uaddr->sa_family))  
8         return -EINVAL;  
9     if (uaddr->sa_family == AF_UNSPEC)  
10        return sk->sk_prot->disconnect(sk, flags);  
11  
12     if (!inet_sk(sk)->inet_num && inet_autobind(sk))  
13         return -EAGAIN;  
14     [...]  
15 }  
16 EXPORT_SYMBOL(inet_dgram_connect);
```

```
19 int udp_disconnect(struct sock *sk, int flags)  
20 {  
21     struct inet_sock *inet = inet_sk(sk);  
22  
23     sk->sk_state = TCP_CLOSE;  
24     [...]  
25     if (!(sk->sk_userlocks & SOCK_BINDPORT_LOCK)) {  
26         sk->sk_prot->unhash(sk);  
27         inet->inet_sport = 0;  
28     }  
29     sk_dst_reset(sk);  
30     return 0;  
31 }  
32 EXPORT_SYMBOL(udp_disconnect);
```



ROAD TO PING_UNHASH

disconnect() in kernel through **connect()** in user program



USE-AFTER-FREE

Local denial of service? Not enough!

- Avoid crash: map 0x200200 in the user space

- Then hmm...

- `sock_put(sk)` can be called twice ;)

- What's **PUT**?

```
34 void ping_unhash(struct sock *sk)
35 {
36     struct inet_sock *isk = inet_sk(sk);
37     pr_debug("ping_unhash(isk=%p,isk->num=%u)\n",
38             isk, isk->inet_num);
39     if (sk_hashed(sk)) {
40         write_lock_bh(&ping_table.lock);
41         hlist_nulls_del(&sk->sk_nulls_node);
42         sock_put(sk);
43         isk->inet_num = 0;
44         isk->inet_sport = 0;
45         sock_prot_inuse_add(sock_net(sk), sk->sk_prot, -1);
46         write_unlock_bh(&ping_table.lock);
47     }
48 }
49 EXPORT_SYMBOL_GPL(ping_unhash);
```



USE-AFTER-FREE

CVE-2015-3636

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

```
52  static inline void sock_put(struct sock *sk)
53  {
54      if (atomic_dec_and_test(&sk->sk_refcnt))
55          sk_free(sk);
56  }
```



PART III

Exploitation

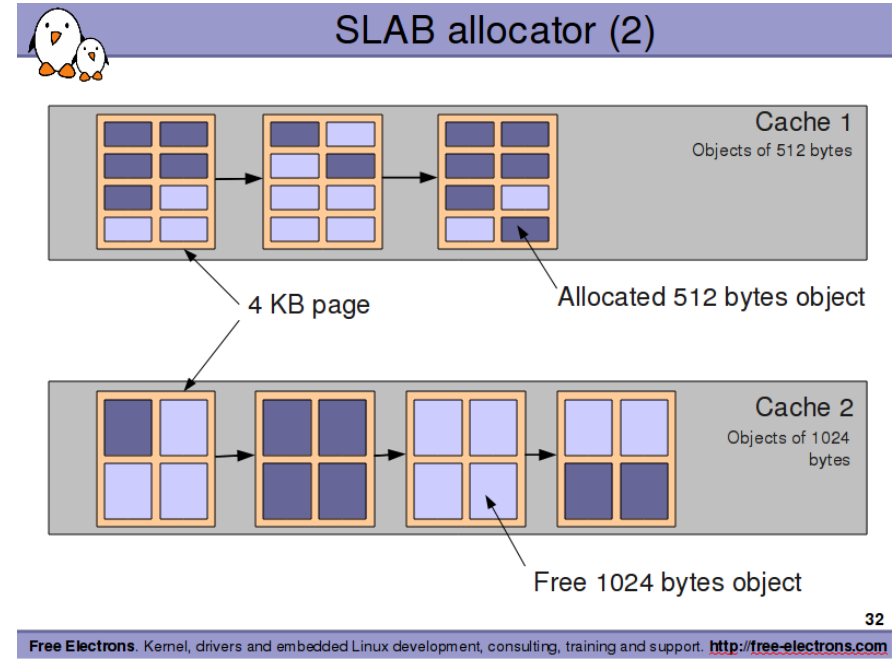
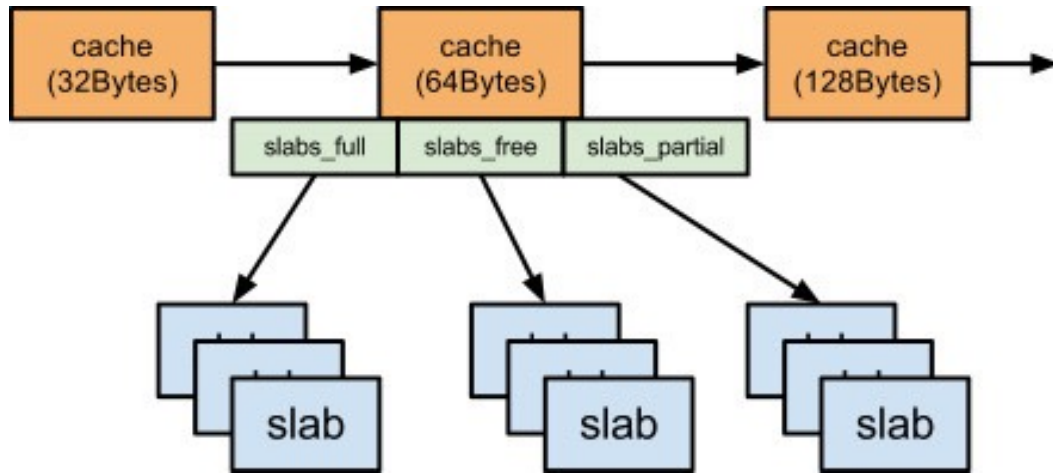


WHEN IT COMES TO UAF

ANNOYING PROBLEM RE-FILLING

- Our target: *struct sock*
- `kmem_cache_alloc("PING", priority & ~__GFP_ZERO);`
- Custom use cache ;(

ping



SLAB CACHE

A specific area for the allocation of kernel objects of particular type

Here we meet the type called “PING” xD



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

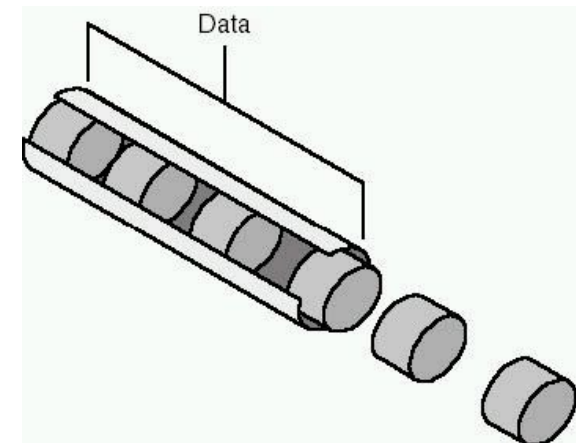




WHAT USED TO RE-FILL

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 packet
- Content control: data of packet





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 a slab with totally PING socket objects
- 2. Free all of them and spray kcalloc-x buffers
- Exactly possible, but ... **out of control**

MR. DAYDREAM

By Roger Hargreaves

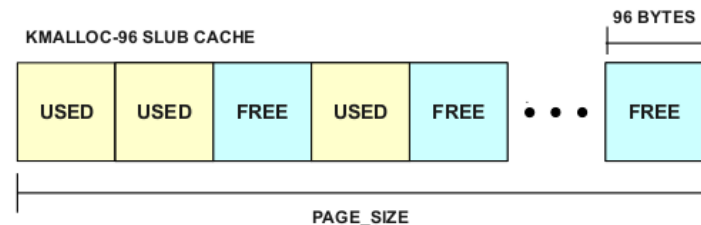


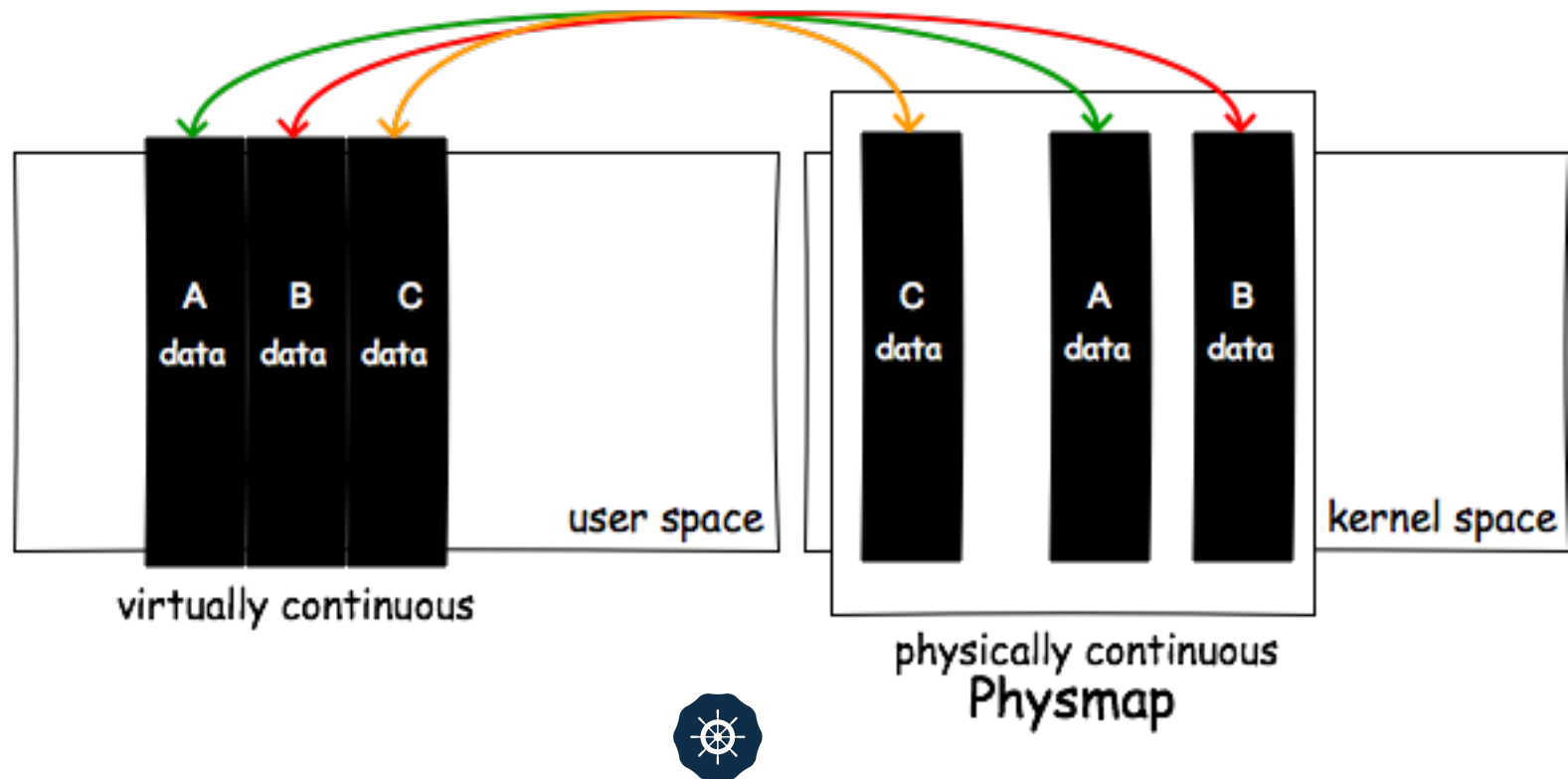


SLUB HELPS US?

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, ...





THE RETURN OF PHYSMAP

Physmap, also called direct-mapped memory, is memory in the kernel which would directly map the memory in the user space into the kernel space.



THE RETURN OF PHYSMAP

Again, based on the natural weakness of the system:
MEMORY REUSE

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



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 kernel theoretically**

Architecture		PHYS_OFFSET	Size	Prot.
x86	(3G/1G)	0xC0000000	891MB	RW
	(2G/2G)	0x80000000	1915MB	RW
	(1G/3G)	0x40000000	2939MB	RW
AArch32	(3G/1G)	0xC0000000	760MB	RW X
	(2G/2G)	0x80000000	1784MB	RW X
	(1G/3G)	0x40000000	2808MB	RW X
x86-64		0xFFFFF80000000000	64TB	RW (X)
AArch64		0xFFFFFC0000000000	256GB	RW X

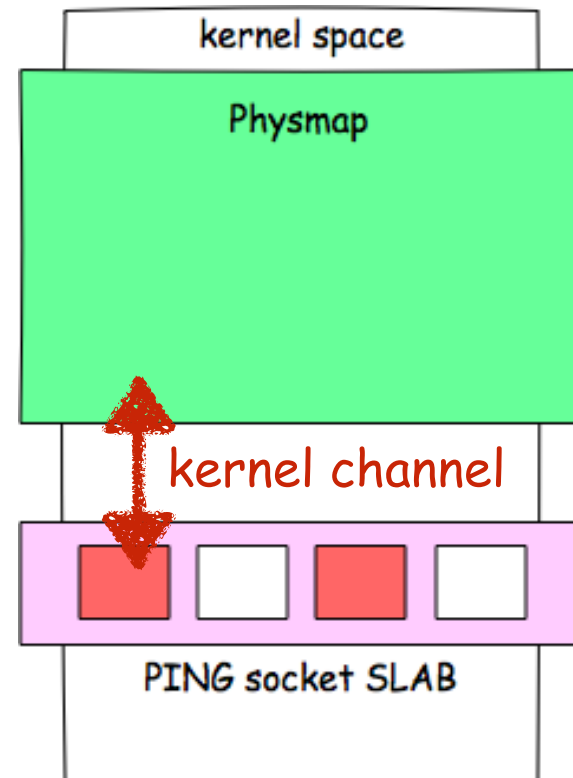
Table 1: physmap characteristics across different architectures (x86, x86-64, AArch32, AArch64).



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?

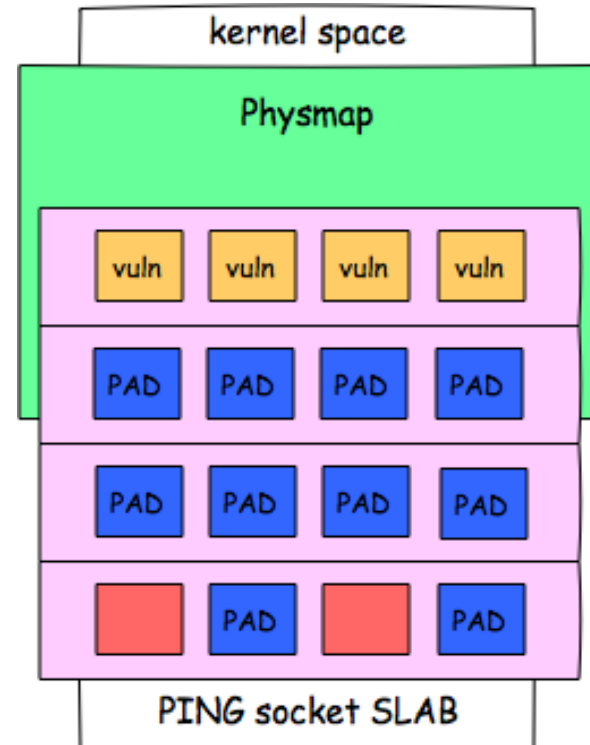




RELIABLE MEMORY COLLISION

Make **PING** socket objects and physmap **closer** ;)

- **PADDING** PING objects
- Lift up the allocation
- **TARGET** PING objects
- Used to pwn




```
60 int inet_ioctl(struct socket *sock,
61 unsigned int cmd, unsigned long arg)
62 {
63     struct sock *sk = sock->sk;
64     int err = 0;
65     struct net *net = sock_net(sk);
66
67     switch (cmd) {
68     case SIOCGSTAMP:
69         err = sock_get_timestamp(sk,
70 (struct timeval __user *)arg);
71         break;
72     case SIOCGSTAMPNS:
73         err = sock_get_timestamps(sk,
74 (struct timespec __user *)arg);
75         break;
76     [...]
```

```
79 int sock_get_timestamps(struct sock *sk,
80 struct timespec __user *userstamp)
81 {
82     struct timespec ts;
83     if (!sock_flag(sk, SOCK_TIMESTAMP))
84         sock_enable_timestamp(sk, SOCK_TIMESTAMP);
85     ts = ktime_to_timespec(sk->sk_stamp);
86     if (ts.tv_sec == -1)
87         return -ENOENT;
88     if (ts.tv_sec == 0) {
89         sk->sk_stamp = ktime_get_real();
90         ts = ktime_to_timespec(sk->sk_stamp);
91     }
92     return copy_to_user(userstamp, &ts,
93 sizeof(ts)) ? -EFAULT : 0;
94 }
95 EXPORT_SYMBOL(sock_get_timestamps);
```



HATE TO THROW A DICE

Find a 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 for **padding (lifting up)**.
- Allocate hundreds of PING socket objects **which is later to be re-filled**.
- 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.

Summary



CONQUER KERNEL UAF

~~separated allocation AND multi-core/thread
AND uncontrollable creation and content~~

- 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



PC CONTROL

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

```
97 int inet_release(struct socket *sock)
98 {
99     struct sock *sk = sock->sk;
100
101     if (sk) {
102         long timeout;
103
104         [...]
105
106         if (sock_flag(sk, SOCK_LINGER) &&
107             !(current->flags & PF_EXITING))
108             timeout = sk->sk_lingertime;
109         sock->sk = NULL;
110         sk->sk_prot->close(sk, timeout);
111     }
112     return 0;
113 }
114 EXPORT_SYMBOL(inet_release);
```



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**



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 **phsypmap**? **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 ;(

return
oriented
PROGRAMming



ROP TIPS

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 ≥ 4.3
 - **First 64bit root case** in the world as known
 - S6 & S6 Edge root
- DEMO ;)





PART IV

Future



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





ACKNOWLEDGEMENT

Thanks for contributions and inspirations

- wushi
- jfang
- Leo.C

- Liang Chen
- Slipper
- Peter

KEEN TEAM

Pictures in the slide come from Google



THANK YOU

Wen Xu

@antlr7

hotdog3645@gmail.com

