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 2rd 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

PART II

Bug Hunting

FUZZING

Open source kernel syscall fuzzer

- Trinity
  - https://github.com/kernelslacker/trinity
  - Scalability
  - Ported to ARM Linux

FUZZING

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 00200200
[ 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 : [<c08b18bc>]    lr : [<c09f7d9c>]    psr: 20010013
[ 3354.779541] sp : e99a5ee0  ip : c08a67ac  fp : 00000000
[ 3354.779724] r10: 00000000  r9 : e99a4000  r8 : c000e928
[ 3354.779846] r7 : 0000011b  r6 : 00000053  r5 : 00000000  r4 : eb3cd200
[ 3354.779907] r3 : 00000003  r2 : 00200200  r1 : 00000000  r0 : c144ed98
[ 3354.780029] Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
[ 3354.780120] Control: 10c5787d  Table: ab97406a  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_sport = 0;
        sock_prot_inuse_add(sock_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);
```

```c
static inline void __hlist_nulls_del(struct hlist_nulls_node *n)
{
    struct hlist_nulls_node *next = n->next;
    struct hlist_nulls_node **pprev = n->pprev; 2
    *pprev = next; 3
    if (!is_a_nulls(next))
        next->pprev = pprev;
}

static inline void hlist_nulls_del(struct hlist_nulls_node *n)
{
    __hlist_nulls_del(n);
    n->pprev = LIST_POISON2; 1
}
```

LIST_POISON2 == 0X200200

ping_unhash (__hlist_nulls_del) **TWO** times → 0x200200 not mapped → kernel crash

```
int inet_dgram_connect(struct socket *sock,
                       struct sockaddr * uaddr,
                       int addr_len, int flags)
{
    struct sock *sk = sock->sk;

    if (addr_len < sizeof(uaddr->sa_family))
        return -EINVAL;
    if (uaddr->sa_family == AF_UNSPEC)
        return sk->sk_prot->disconnect(sk, flags);

    if (!inet_sk(sk)->inet_num && inet_autobind(sk))
        return -EAGAIN;
    [...]
}
EXPORT_SYMBOL(inet_dgram_connect);
```

```
int udp_disconnect(struct sock *sk, int flags)
{
    struct inet_sock *inet = inet_sk(sk);

    sk->sk_state = TCP_CLOSE;
    [...]
    if (!(sk->sk_userlocks & SOCK_BINDPORT_LOCK)) {
        sk->sk_prot->unhash(sk);
        inet->inet_sport = 0;
    }
    sk_dst_reset(sk);
    return 0;
}
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) is called twice ;)
- What's **PUT**?

```
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_sport = 0;
        sock_prot_inuse_add(sock_net(sk), sk->sk_prot, -1);
        write_unlock_bh(&ping_table.lock);
    }
}
EXPORT_SYMBOL_GPL(ping_unhash);
```

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

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

PART III

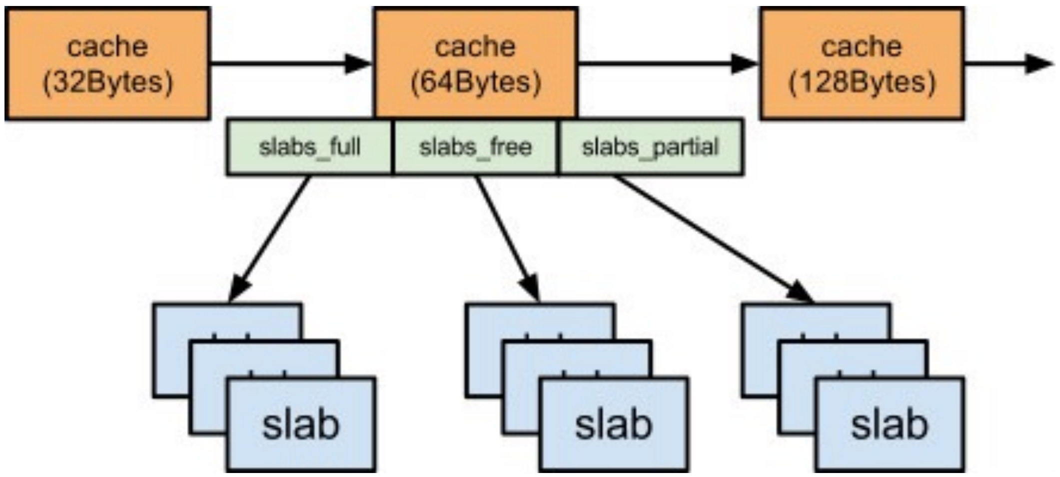
Exploitation

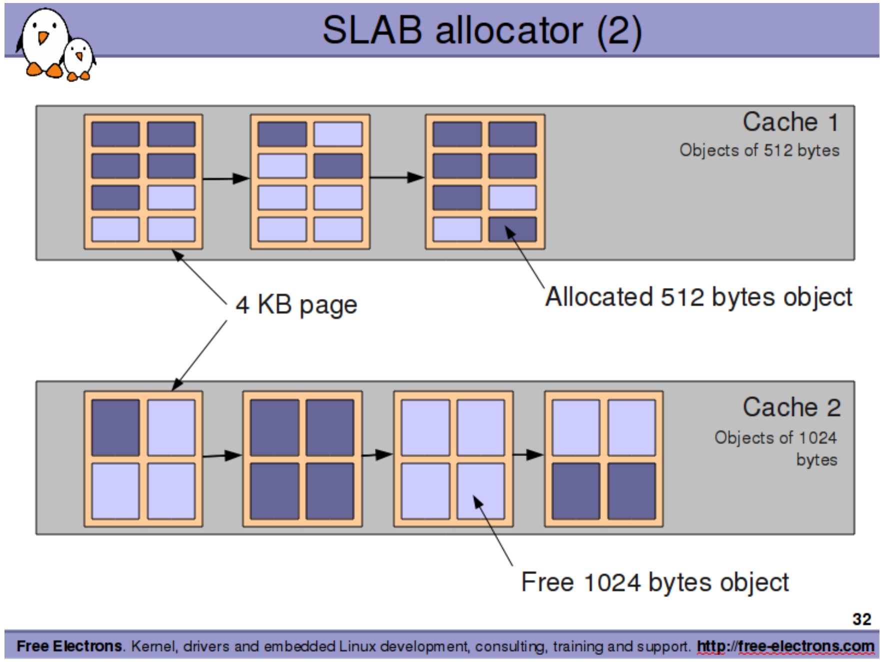
WHEN IT COMES TO UAF

ANNOYING PROBLEM RE-FILLING

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

PING PONG





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 control msg
- **Content control:** content of control msg

INTUITIVE IDEA

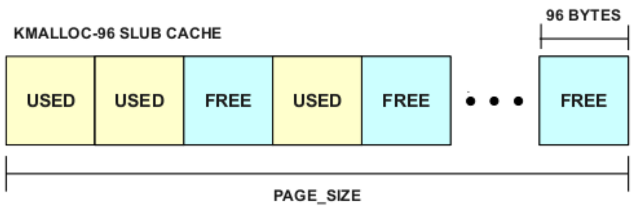
Basically, a completely free slab has large probability to be recycled for future allocation

- 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**

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, …**

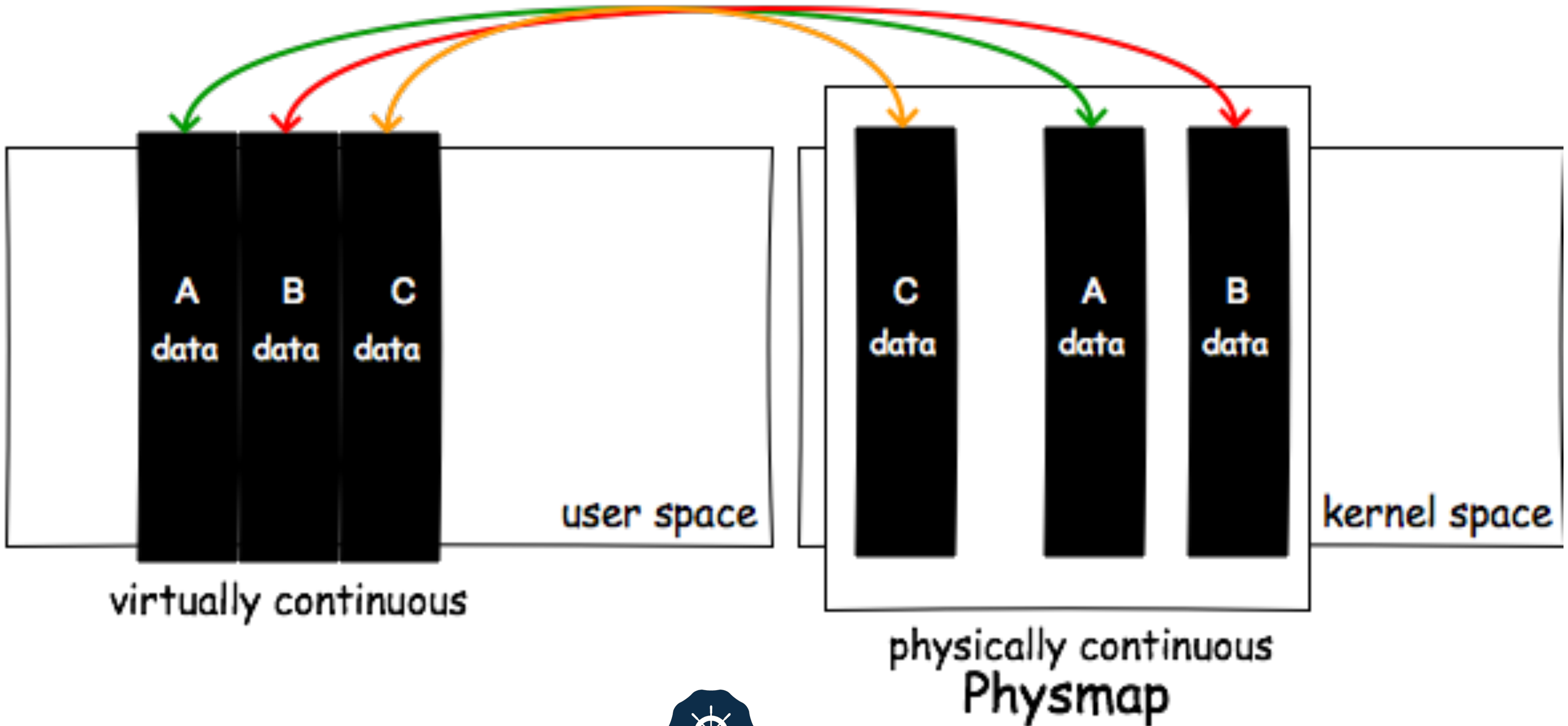


KERNEL MEMORY CONTROL

Universal Solution #1

RET2DIR

- ret2dir: Rethinking Kernel Isolation (USENIX 14’)
  - Vasileios P. Kemerlis Michalis Polychronakis 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?



THE RETURN OF PHYSMAP

Physmap, the 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

- **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

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		0xFFFF880000000000	64TB	RW(**X**)
AArch64		0xFFFFFFC000000000	256GB	RW**X**

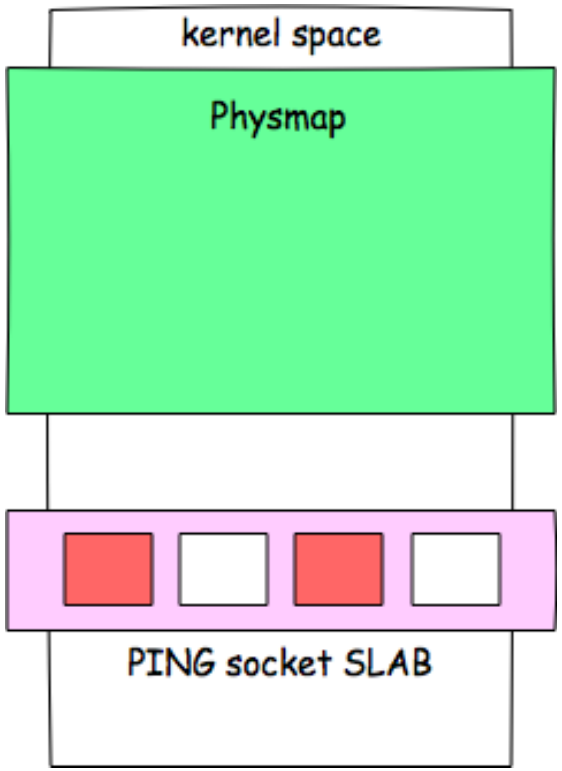
Table 1: physmap characteristics across different architectures (x86, x86-64, AArch32, AArch64).

Table [1] from ret2dir: Rethinking Kernel Isolation (USENIX 14')

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

```c
int inet_ioctl(struct socket *sock,
    unsigned int cmd, unsigned long arg)
{
    struct sock *sk = sock->sk;
    int err = 0;
    struct net *net = sock_net(sk);

    switch (cmd) {
    case SIOCGSTAMP:
        err = sock_get_timestamp(sk,
            (struct timeval __user *)arg);
        break;
    case SIOCGSTAMPNS:
        err = sock_get_timestampns(sk,
            (struct timespec __user *)arg);
        break;
    [...]
```

```c
int sock_get_timestampns(struct sock *sk,
    struct timespec __user *userstamp)
{
    struct timespec ts;
    if (!sock_flag(sk, SOCK_TIMESTAMP))
        sock_enable_timestamp(sk, SOCK_TIMESTAMP);
    ts = ktime_to_timespec(sk->sk_stamp);
    if (ts.tv_sec == -1)
        return -ENOENT;
    if (ts.tv_sec == 0) {
        sk->sk_stamp = ktime_get_real();
        ts = ktime_to_timespec(sk->sk_stamp);
    }
    return copy_to_user(userstamp, &ts,
        sizeof(ts)) ? -EFAULT : 0;
}
EXPORT_SYMBOL(sock_get_timestampns);
```

HATE TO THROW A DICE

Find an info leak to know whether our targeting PING socket object has already been covered by physmap or not

black hat USA 2015

KEEN TEAM

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.

Summary

UNLEASH KERNEL UAF

~~separated allocation AND multi-core/thread~~
~~AND incontrollable 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

```
int inet_release(struct socket *sock)
{
    struct sock *sk = sock->sk;

    if (sk) {
        long timeout;

        [...]

        if (sock_flag(sk, SOCK_LINGER) &&
            !(current->flags & PF_EXITING))
            timeout = sk->sk_lingertime;
        sock->sk = NULL;
        sk->sk_prot->close(sk, timeout);
    }
    return 0;
}
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

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 ;(

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 ;)


SAMSUNG
13:15
Thu, June 18
BACKUP PASSWORD

PART IV

Future

FUTURE

64bit devices could be more secure

- LIST_POISON2 in 64bit Android kernel
  - 0x~~200200~~ 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

PWNIE

- 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