Zer0Con2019

From Zero to Root: Building Universal Android Rooting with a Type Confusion Vulnerability

WANG, YONG (@ThomasKing2014)
Alibaba Security

Whoami

- WANG, YONG a.k.a. ThomasKing(@ThomasKing2014)
- Security Engineer of Alibaba Security
- Focus on Android/Browser vulnerability
- Android vulnerability hunting and exploitation since 2015
- Speaker at BlackHatAsia2018/HITB2018AMS
  - ReVent
  - Kernel Space Mirroring Attack

Agenda

- Overview
- Vulnerability analysis
- WrongZone rooting solution
- Conclusion

Agenda

- *Overview*
- Vulnerability analysis
- WrongZone rooting solution
- Conclusion

Challenges

- Fewer attack surfaces
  - SELinux - Fewer drivers
  - Secure Computing - Fewer syscalls

Challenges

- Fewer attack surfaces
  - SELinux - Fewer drivers
  - Secure Computing - Fewer syscalls
- More Mitigations
  - Privileged eXecute Never
  - Privileged Access Never
  - Kernel Address Space Layout Randomization
  - Control Flow Integrity

History

Year	CVE-ID	Name	Challenges
2013	CVE-2013-6282		
2014	CVE-2014-3153	Towelroot	SELinux
2015	CVE-2015-1805 CVE-2015-3636	PingPongRoot	PXN/SELinux
2016	CVE-2016-5195	DirtyC0w	PXN/SELinux
2017	CVE-2017-7533 CVE-2017-8890	ReVent	PXN/PAN/KASLR/SELinux/Seccomp

History

Year	CVE-ID	Name	Challenges
2013	CVE-2013-6282		
2014	CVE-2014-3153	Towelroot	SELinux
2015	CVE-2015-1805 CVE-2015-3636	PingPongRoot	PXN/SELinux
2016	CVE-2016-5195	DirtyC0w	PXN/SELinux
2017	CVE-2017-7533 CVE-2017-8890	ReVent	PXN/PAN/KASLR/SELinux/Seccomp
2018	CVE-2018-9568	WrongZone	PXN/PAN/KASLR/CFI/SELinux/Seccomp

Agenda

- Overview
- *Vulnerability analysis*
- WrongZone rooting solution
- Conclusion

Simple motivation

- Internal update
  - All servers must support IPv6
- The detail of IPv6 implementation in the Linux kernel
  - No idea
- RTFSC
  - Read The F***ing Source Code

SOCKET(2)

```
// net/ipv4/af_inet.c
static int inet_create(struct net *net, struct socket *sock, int protocol, int kern)
{…
        err = -ENOBUFS;
        sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot, kern);
        if (!sk)

// net/ipv6/af_inet6.c
static int inet6_create(struct net *net, struct socket *sock, int protocol, int kern)
{…
        err = -ENOBUFS;
        sk = sk_alloc(net, PF_INET6, GFP_KERNEL, answer_prot, kern);
        if (!sk)
```

SOCKET(2)

```
// net/core/sock.c
struct sock *sk_alloc(struct net *net, int family, gfp_t priority, struct proto *prot, int kern)
{…
        sk = sk_prot_alloc(prot, priority | __GFP_ZERO, family);
        if (sk) {
                sk->sk_family = family;
                sk->sk_prot = sk->sk_prot_creator = prot;
                sock_lock_init(sk);
                sk->sk_net_refcnt = kern ? 0 : 1;
                if (likely(sk->sk_net_refcnt))
                        get_net(net);
                sock_net_set(sk, net);
                atomic_set(&sk->sk_wmem_alloc, 1);
```

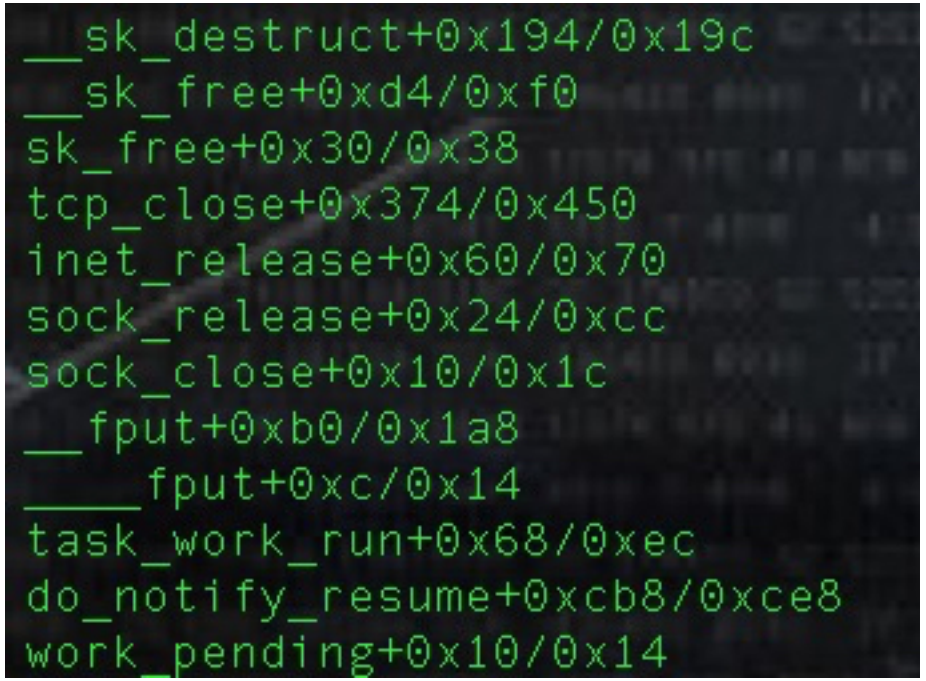
SOCKET(2)

```
// net/core/sock.c
static struct sock *sk_prot_alloc(struct proto *prot, gfp_t priority, int family)
{…
        slab = prot->slab;
        if (slab != NULL) {
                sk = kmem_cache_alloc(slab, priority & ~__GFP_ZERO);
                if (!sk)
                        return sk;
                if (priority & __GFP_ZERO)
                {…
                }
        } else
                sk = kmalloc(prot->obj_size, priority);
```

CLOSE(2) - socket

```
// net/core/sock.c
static void __sk_destruct(struct rcu_head *head)
{…
        sk_prot_free(sk->sk_prot_creator, sk);
}

static void sk_prot_free(struct proto *prot, struct sock *sk)
{…
        slab = prot->slab;
        if (slab != NULL)
                kmem_cache_free(slab, sk);
        else
                kfree(sk);
```



SETSOCKOPT(2) - IPV6_ADDRFORM

```
// net/ipv6/ipv6_sockglue.c
static int do_ipv6_setsockopt(struct sock *sk, int level, int optname, char __user *optval, unsigned
int optlen)
{…
        case IPV6_ADDRFORM:
        …
        if (sk->sk_protocol == IPPROTO_TCP) {
                struct inet_connection_sock *icsk = inet_csk(sk);
                …
                sk->sk_prot = &tcp_prot;
                icsk->icsk_af_ops = &ipv4_specific;
                sk->sk_socket->ops = &inet_stream_ops;
                sk->sk_family = PF_INET;
                tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
```

Think

- How is the usage of the two fields?
  - sk_prot_creator: allocate/free
  - sk_prot: other operations

Think

- How is the usage of the two fields?
  - sk_prot_creator: allocate/free
  - sk_prot: other operations

- Can the unusual state be transmitted to another object?

Think

- How is the usage of the two fields?
  - sk_prot_creator: allocate/free
  - sk_prot: other operations

- Can the unusual state be transmitted to another object?
  - ACCEPT(2)

ACCEPT(2)

```
// Classical TCP Server sample
int main(int argc, char *argv[])
{…
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        serv_addr.sin_port = htons(portno);
        bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)
        listen(sockfd,5);
        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
...
```

ACCEPT(2)

```
// net/core/sock.c
struct sock *sk_clone_lock(const struct sock *sk, const gfp_t priority)
{…
        newsk = sk_prot_alloc(sk->sk_prot, priority, sk->sk_family);
        if (newsk != NULL) {
                struct sk_filter *filter;
                sock_copy(newsk, sk);

static void sock_copy(struct sock *nsk, const struct sock *osk)
{…
        memcpy(nsk, osk, offsetof(struct sock, sk_dontcopy_begin));
        memcpy(&nsk->sk_dontcopy_end, &osk->sk_dontcopy_end,
                osk->sk_prot->obj_size - offsetof(struct sock, sk_dontcopy_end));
```

Type confusion

- After sock_copy invoked
  - newsk->sk_prot = sk->sk_prot;
    - newsk->sk_prot = &tcp_prot;
  - newsk->sk_prot_creator = sk->sk_prot_creator
    - newsk->sk_prot_creator = &tcpv6_prot;

- When closing the fd
  - sk_prot_free(sk->sk_prot_creator, sk);
  - kmem_cache_free(slab, sk);

Type confusion

- After sock_copy invoked
  - newsk->sk_prot = sk->sk_prot;
    - newsk->sk_prot = &tcp_prot;
  - newsk->sk_prot_creator = sk->sk_prot_creator
    - newsk->sk_prot_creator = &tcpv6_prot;

- When closing the fd
  - sk_prot_free(sk->sk_prot_creator, sk);
  - kmem_cache_free(slab, sk);
  - The TCPv4 object will be wrongly recycled in the TCPv6 slab cache!

Butterfly effect

```
// mm/slub.c
static inline void set_freepointer(struct kmem_cache *s, void *object, void *fp)
{
        *(void **)(object + s->offset) = fp;
}
```

- Variable ‘s’ is the slab cache of tcpv6_prot.
  - The offset of slab cache is A.
- The object is allocated in the slab cache of tcp_prot.
  - The offset of slab cache is B.

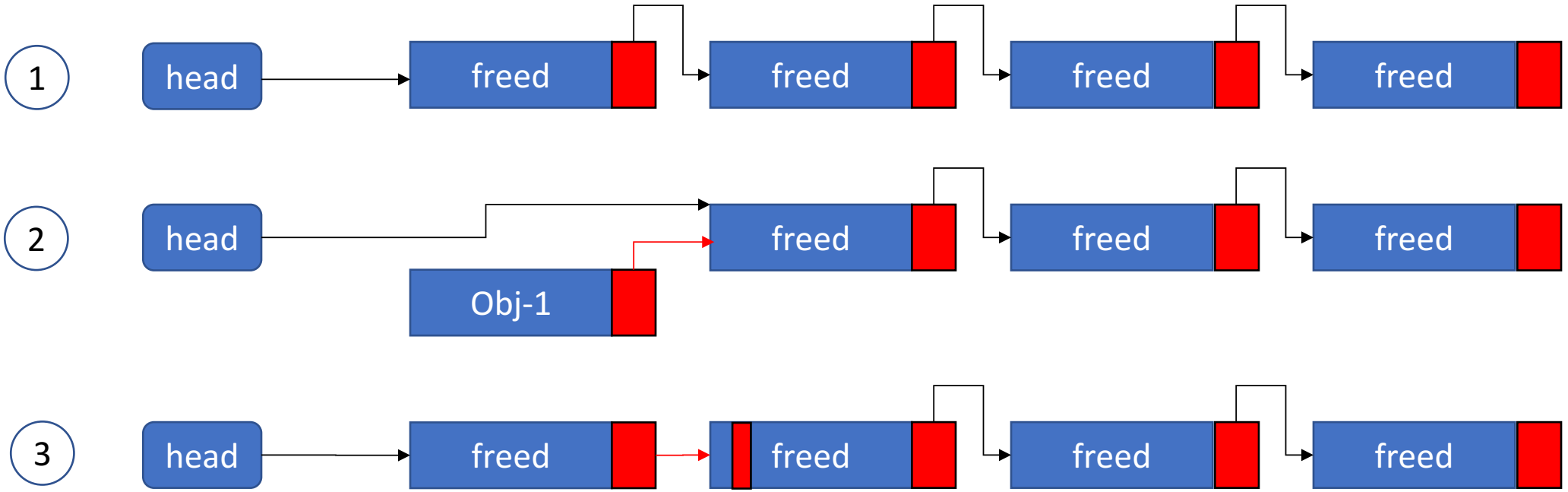
- If A is not equal to B, the free list will be corrupted.
- Otherwise, everything is fine.

Butterfly effect

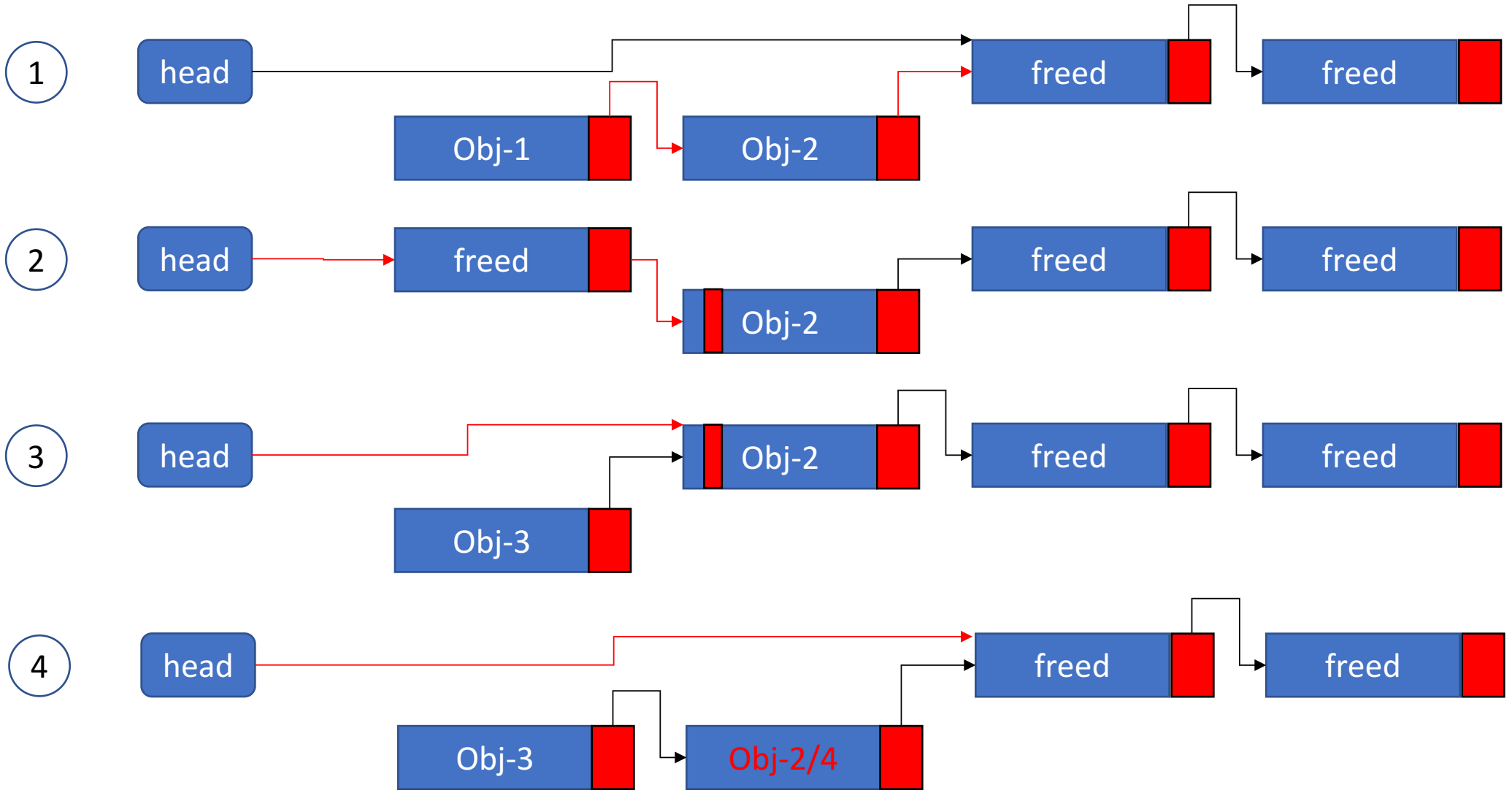
```
// mm/slub.c
static int calculate_sizes(struct kmem_cache *s, int forced_order)
{…
        if (((flags & (SLAB_DESTROY_BY_RCU | SLAB_POISON)) || s->ctor))
        {
                s->offset = size;
                size += sizeof(void *);
```

- The slab_flags contains the SLAB_DESTROY_BY_RCU bit.
  - The offset is set to the size of the sock object.
- TCPv6 sock object contains the structure of TCPv4 sock.
  - The free list is corrupted.

Butterfly effect



Butterfly effect - UAF



CVE-2018-9568

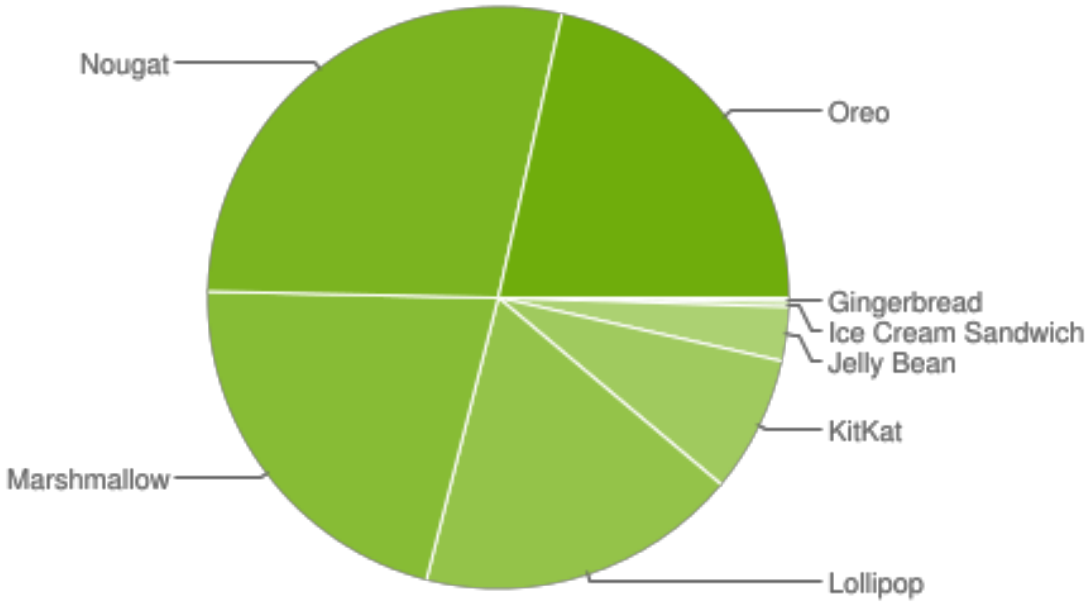
- Found in the Android msm-wahoo kernel.
  - Affect almost all the Android devices.
- Fixed in Linux upstream kernel. 😢
  - Collision
- Do not call out a security fix.
  - Bug fix.
  - Lots of branches are still affected.
- No public exploitable information

Agenda

- Overview
- Vulnerability analysis
- *WrongZone rooting solution*
- Conclusion

Rooting game

- Different version with different mitigations
  - Android 7 and older
  - Android 8
  - Android 9

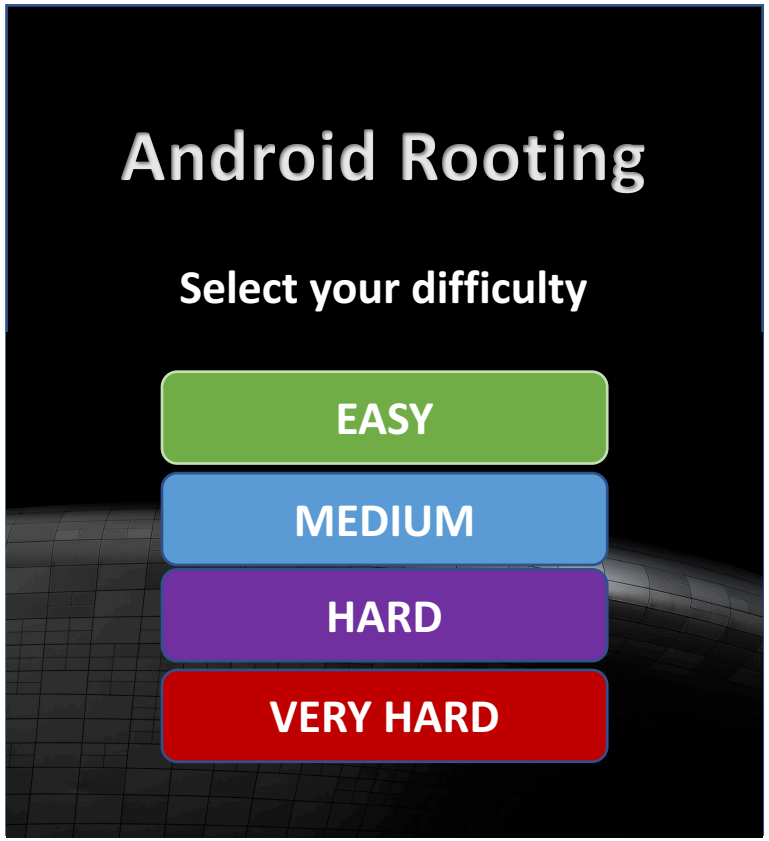


October 26, 2018

Rooting game

- Different version with different mitigations
  - Android 7 and older
  - Android 8
  - Android 9

- Difficulty of rooting game
  - Easy – PXN(Android 7 and lower)
  - Medium – PXN/PAN(Android 8 without KASLR)
  - Hard – PXN/PAN/KASLR(Android 8 with KASLR)
  - Very hard – PXN/PAN/KASLR/CFI(Android 9)



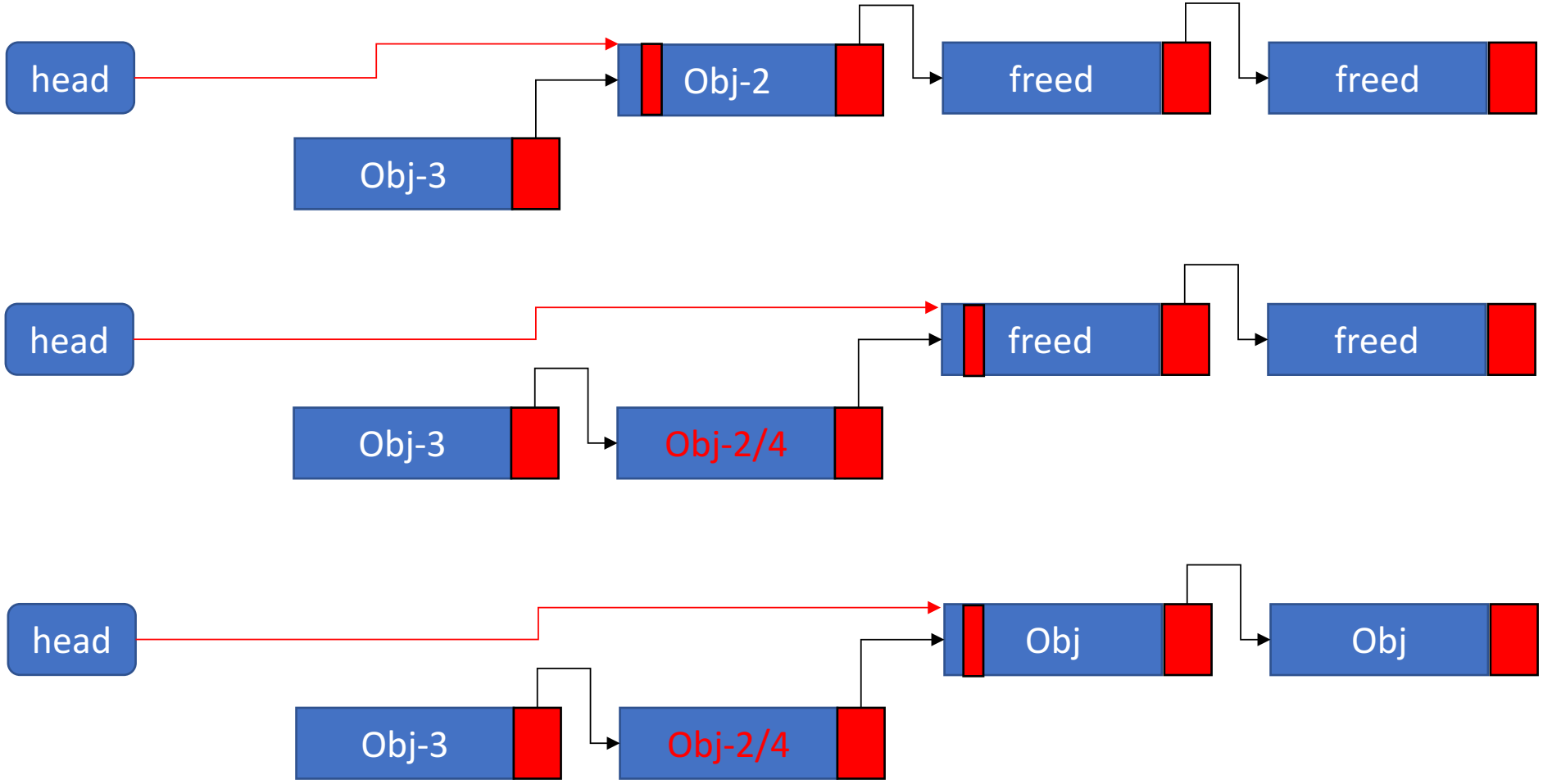
Android Rooting

Problem

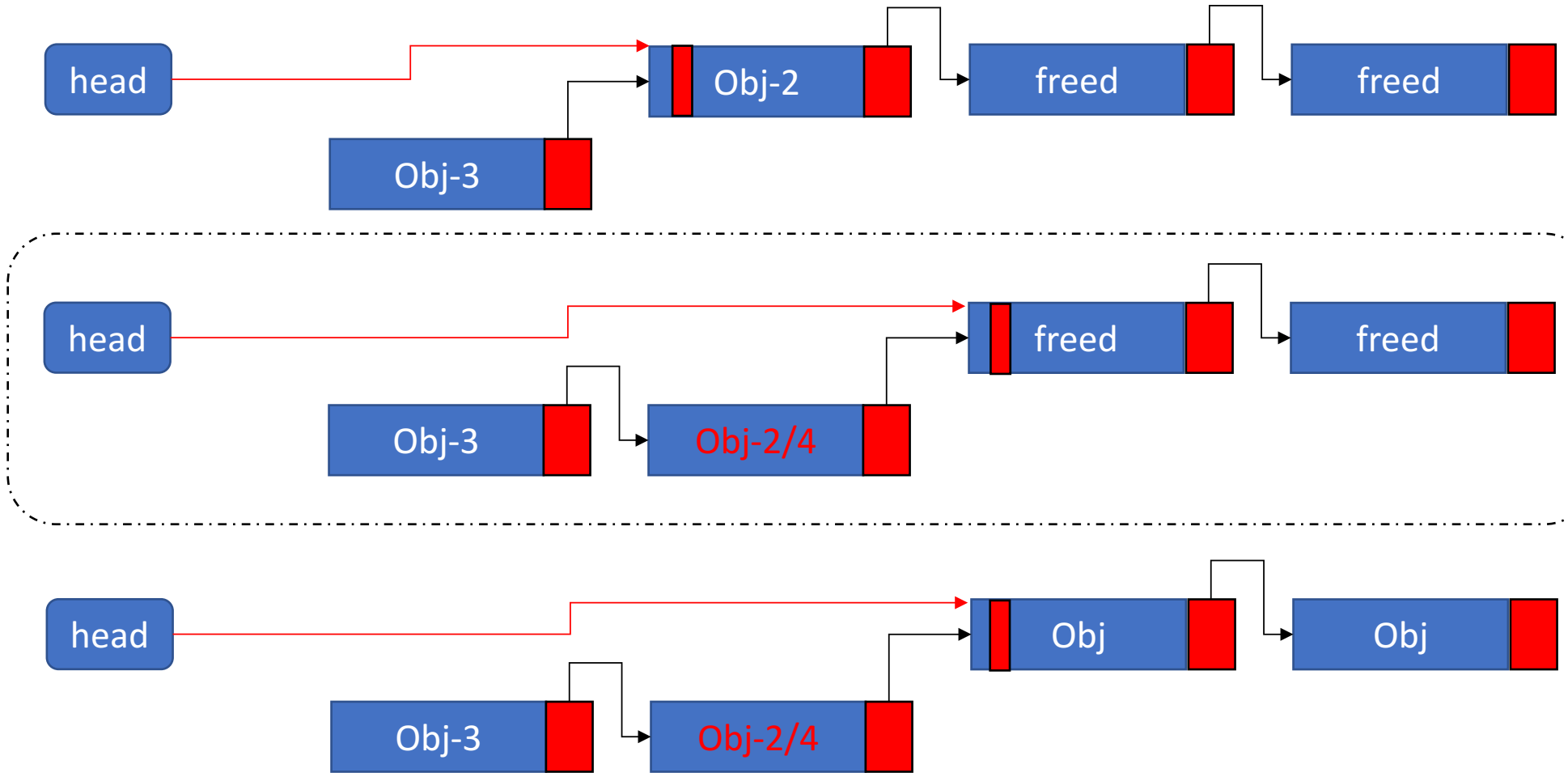
- Two sock objects share the same buffer.
  - Very similar to CVE-2015-3636(PINGPONGROOT)

- How can I create the bad sock objects reliably?

Different cases



Different cases



Slub allocator

- Buffers are not directly allocated in the kmem_cache_node

Slub allocator

- Buffers are not directly allocated in the kmem_cache_node
- One kmem_cache_cpu per CPU core
  - All the operations bind to a specific CPU core

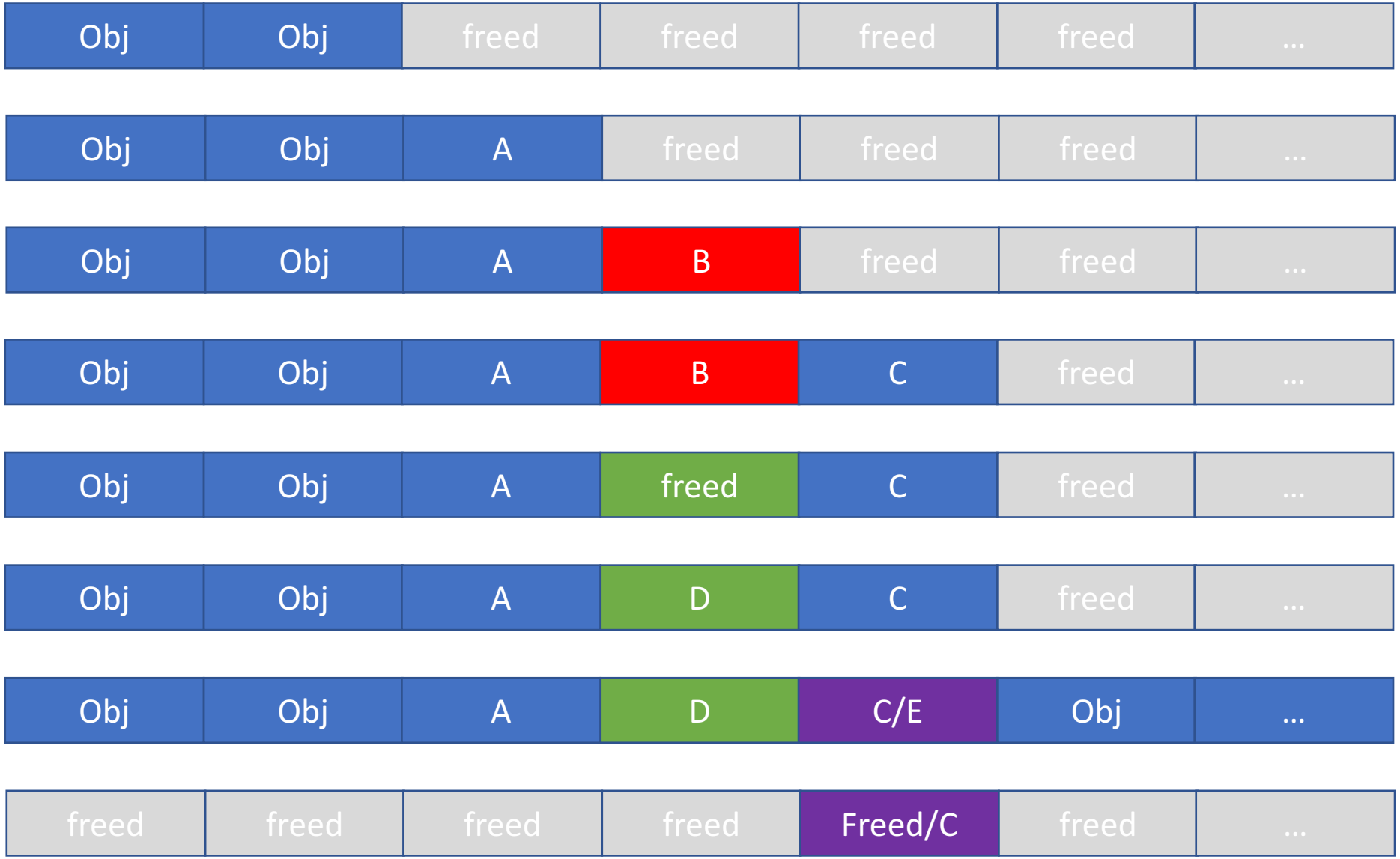
Slub allocator

- Buffers are not directly allocated in the kmem_cache_node
- One kmem_cache_cpu per CPU core
  - All the operations bind to a specific CPU core
- One UAF object per slab
  - Maximize the possibility of overlapping the uaf buffer

Heap Fengshui

- For each CPU core
  - Start one evil TCP server
  - Fill the holes with sock objects
  - Create one sock object A
  - Accept one evil sock object B
  - Create one sock object C
  - Free the object B
  - Create the sock object D
  - Create objects to run out of the current slab

Heap Fengshui



Control UAF socks

- Leak data from a UAF sock
  - ioctl(SIOCGSTAMPNS);

```
// net/core/sock.c
int inet_ioctl(struct socket *sock, unsigned int cmd, unsigned long arg)
{…
        case SIOCGSTAMPNS:
        err = sock_get_timestampns(sk, (struct timespec __user *)arg);

int sock_get_timestampns(struct sock *sk, struct timespec __user *userstamp)
{…
        ts = ktime_to_timespec(sk->sk_stamp);
        …
        return copy_to_user(userstamp, &ts, sizeof(ts)) ? -EFAULT : 0;
```

Control UAF socks

- Fill the UAF socks
  - 0. Close all the fds except the UAF fds.
  - 1. Call mmap syscall with 0x4000000 size.
  - 2. Fill the buffer with ‘0x0000000800000008’ magic number.
  - 3. Lock the buffer and request the time stamp.
  - 4. Check whether it is equal to 0x0000000800000008.
  - 5. If true, stop. Else, goto 1.

Control UAF socks

- Fill the UAF socks
  - 0. Close all the fds except the UAF fds.
  - 1. Call mmap syscall with 0x4000000 size.
  - 2. Fill the buffer with ‘0x0000000800000008’ magic number.
  - 3. Lock the buffer and request the time stamp.
  - 4. Check whether it is equal to 0x0000000800000008.
  - 5. If true, stop. Else, goto step 1.

- PC Control

```
// net/core/sock.c
int inet_ioctl(struct socket *sock, unsigned int cmd, unsigned long arg)
{…
        default:
        if (sk->sk_prot->ioctl)
                err = sk->sk_prot->ioctl(sk, cmd, arg);
```

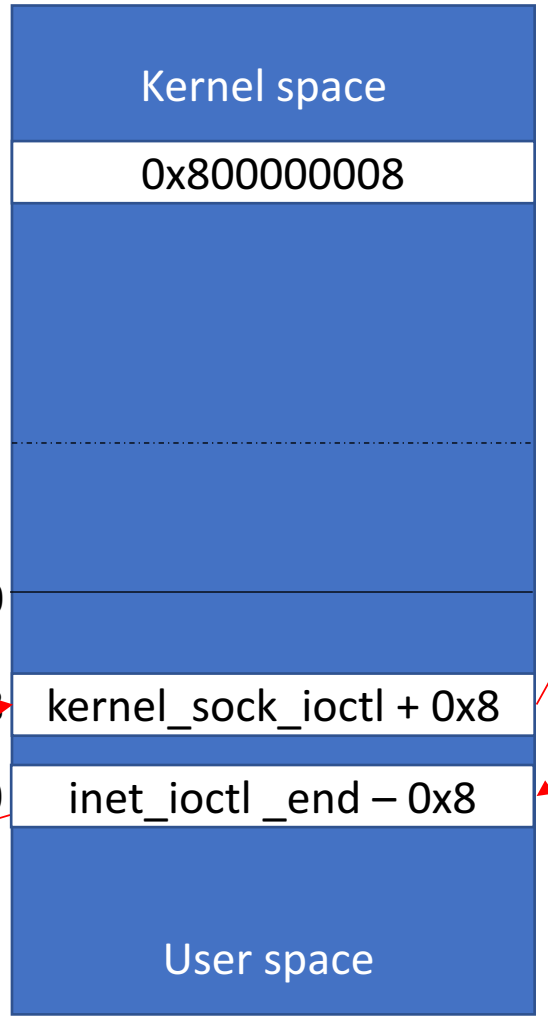
Bypass PXN

```
EXPORT inet_ioctl
                        ; DATA XREF: .text:FFFFF
                        ; .text:FFFFFFC001182D78

=  0
=  0x10
=  0x20
=  0x30
=  0x40

STP             X29, X30, [SP,#-0x10+var_s0]!
SUB             W3, W1, #8,LSL#12
MOV             X29, SP
SUB             W3, W3, #0x906
LDR             X4, [X0,#0x20] ; sk = sock->sk
CMP             W3, #0x4F
LDR             X0, [X4,#0x30]
B.LS            loc_FFFFFFC000E5EF94
LDR             X0, [X4,#0x28] ; sk->sk_prot
LDR             X3, [X0,#0x20] ; sk_prot->ioctl
CBZ             X3, loc_FFFFFFC000E5EFF8
MOV             X0, X4
BLR             X3

EF8C                    ; CODE XREF: inet_ioctl+
LDP             X29, X30, [SP+var_s0],#0x10
RET
```




```
EXPORT kernel_sock_ioctl
:1                      ; CODE XREF: cntl_s
                        ; socket_init_work_
                        ; DATA XREF: ...

= -0x20
= -0x10

STP             X29, X30, [SP,#var_20]!
MOV             X5, #0xFFFFFFFFFFFFFFFF
MOV             X3, SP
MOV             X29, SP
AND             X3, X3, #0xFFFFFFFFFFFFC000
STR             X19, [SP,#0x20+var_10]
LDR             X19, [X3,#8]
STR             X5, [X3,#8]
NOP
LDR             X3, [X0,#0x28]
LDR             X3, [X3,#0x48]
BLR             X3
MOV             X1, SP
```

current_thread_info()->addr_limit = -1

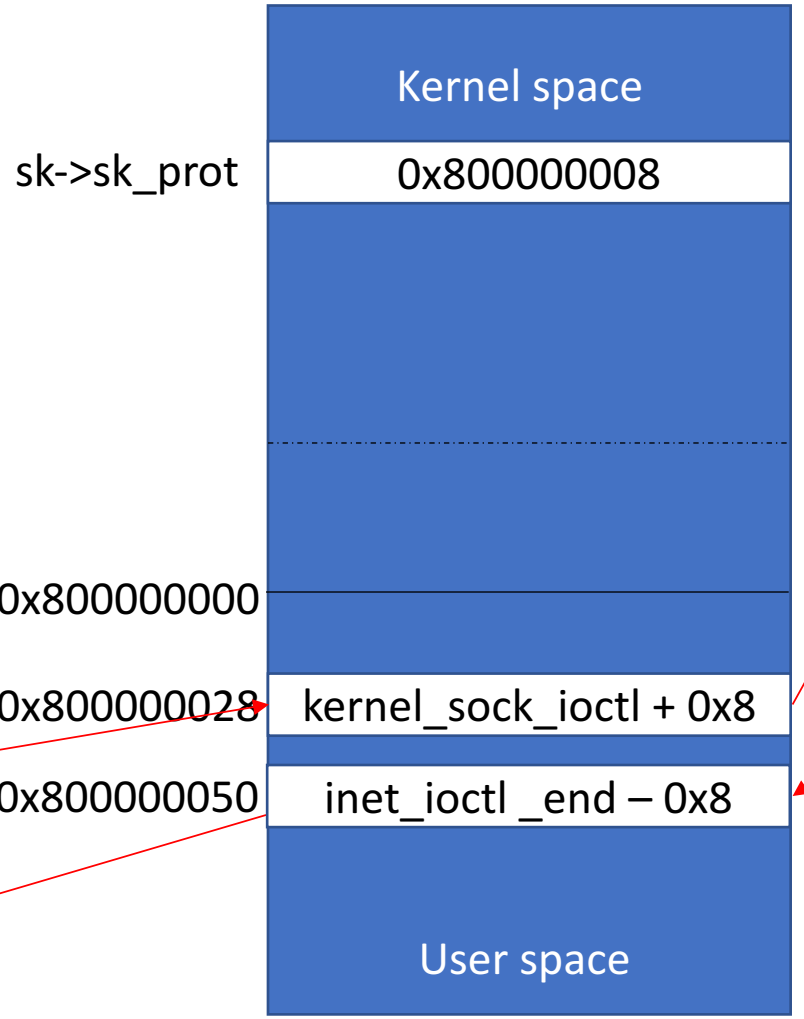
Bypass PXN

```
EXPORT inet_ioctl
                        ; DATA XREF: .text:FFFFF
                        ; .text:FFFFFFC001182D78

=  0
=  0x10
=  0x20
=  0x30
=  0x40

STP             X29, X30, [SP,#-0x10+var_s0]!
SUB             W3, W1, #8,LSL#12
MOV             X29, SP
SUB             W3, W3, #0x906
LDR             X4, [X0,#0x20] ; sk = sock->sk
CMP             W3, #0x4F
LDR             X0, [X4,#0x30]
B.LS            loc_FFFFFFC000E5EF94
LDR             X0, [X4,#0x28] ; sk->sk_prot
LDR             X3, [X0,#0x20] ; sk_prot->ioctl
CBZ             X3, loc_FFFFFFC000E5EFF8
MOV             X0, X4
BLR             X3

EF8C                    ; CODE XREF: inet_ioctl+
LDP             X29, X30, [SP+var_s0],#0x10
RET
```




```
EXPORT kernel_sock_ioctl
:1                      ; CODE XREF: cntl_s
                        ; socket_init_work_
                        ; DATA XREF: ...

= -0x20
= -0x10

STP             X29, X30, [SP,#var_20]!
MOV             X5, #0xFFFFFFFFFFFFFFFF
MOV             X3, SP
MOV             X29, SP
AND             X3, X3, #0xFFFFFFFFFFFFC000
STR             X19, [SP,#0x20+var_10]
LDR             X19, [X3,#8]
STR             X5, [X3,#8]
NOP
LDR             X3, [X0,#0x28]
LDR             X3, [X3,#0x48]
BLR             X3
MOV             X1, SP
```

current_thread_info()->addr_limit = -1

Arbitrary kernel R/W

Avoid crash

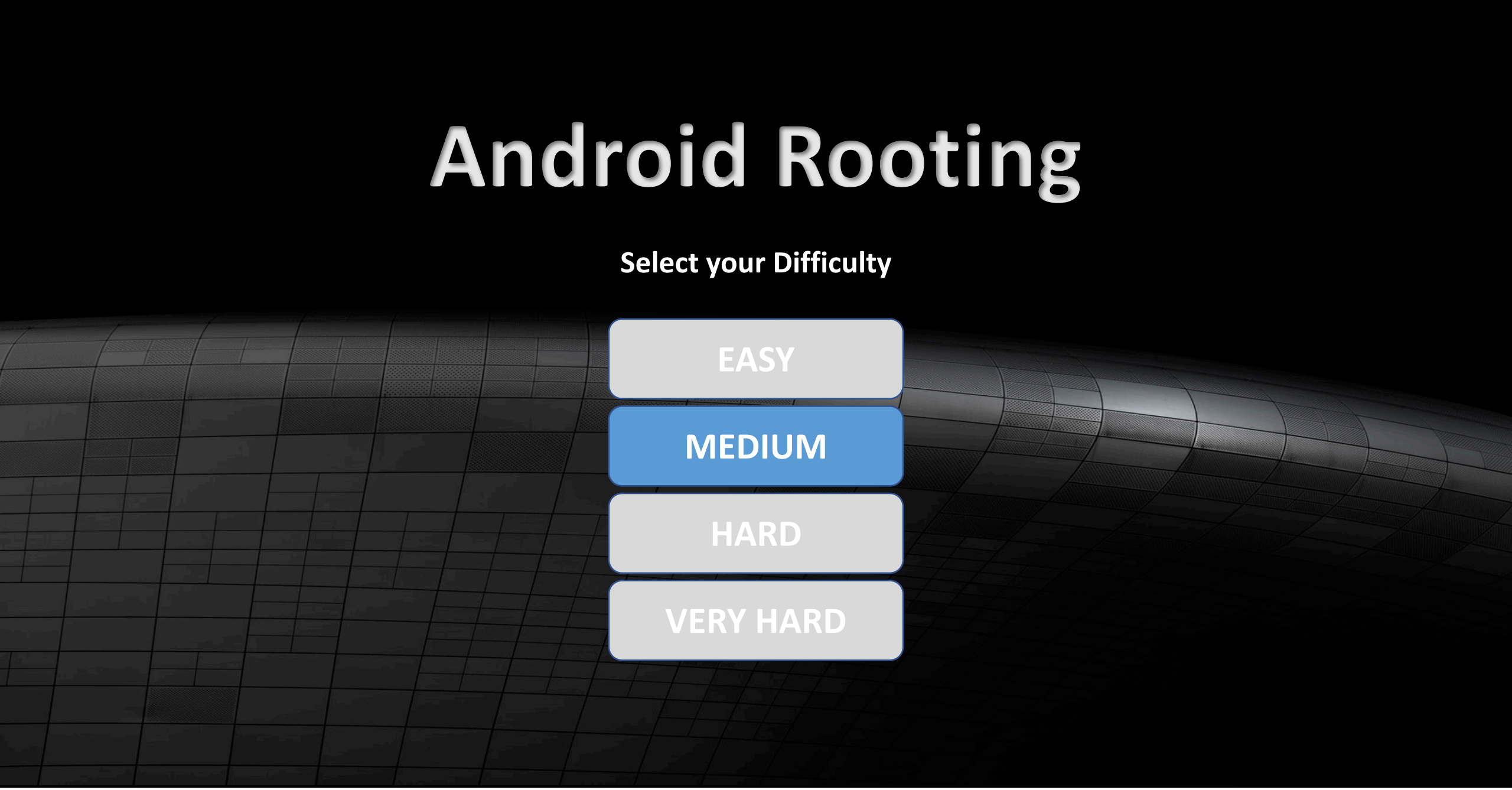
- Directly close the UAF fds leading to kernel crash
- Use an infinite sleep child process
  - Keep the reference count is 1
- Or remove the release routine of the UAF fds
  - Set the release pointer of f_op to NULL

```
// fs/file_table.c
static void __fput(struct file *file)
{…
        if (file->f_op->release)
                file->f_op->release(inode, file);
```

Easy mode

- Exploitation steps
  - Step 0: Start the evil servers and fill the holes
  - Step 1: Spawn client threads and create the UAF socks
  - Step 2: Repeatedly call mmap syscall and check if socks controlled
  - Step 3: Call ioctl syscall and gain arbitrary kernel R/W
  - Step 4: Overwrite uid, disable SELinux
  - Step 5: Remove the release routine and spawn a ROOT shell

Android Rooting



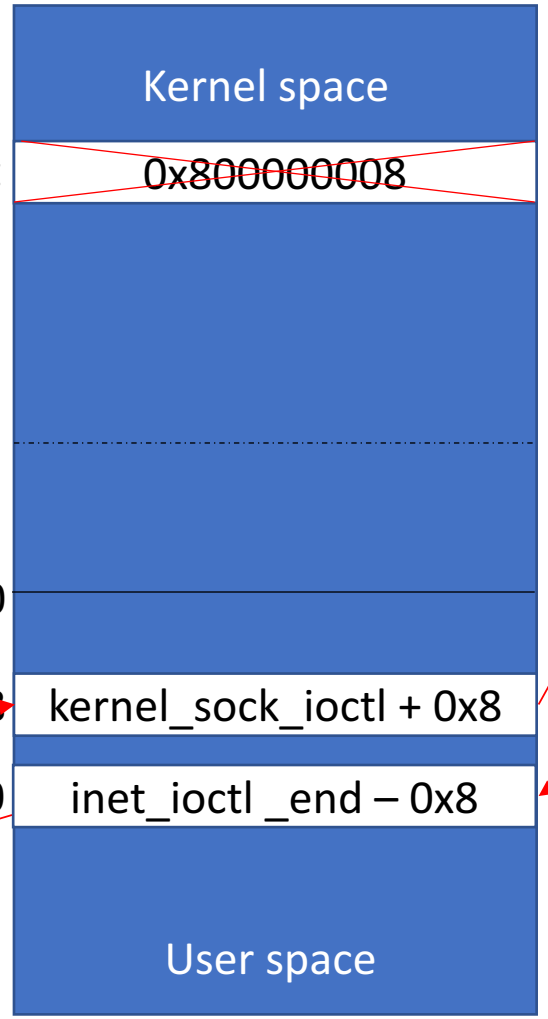
Medium mode

```
EXPORT inet_ioctl
                        ; DATA XREF: .text:FFFFF
                        ; .text:FFFFFFC001182D78

=  0
=  0x10
=  0x20
=  0x30
=  0x40

STP             X29, X30, [SP,#-0x10+var_s0]!
SUB             W3, W1, #8,LSL#12
MOV             X29, SP
SUB             W3, W3, #0x906
LDR             X4, [X0,#0x20] ; sk = sock->sk
CMP             W3, #0x4F
LDR             X0, [X4,#0x30]
B.LS            loc_FFFFFFC000E5EF94
LDR             X0, [X4,#0x28] ; sk->sk_prot
LDR             X3, [X0,#0x20] ; sk_prot->ioctl
CBZ             X3, loc_FFFFFFC000E5EFF8
MOV             X0, X4
BLR             X3

EF8C                    ; CODE XREF: inet_ioctl+
LDP             X29, X30, [SP+var_s0],#0x10
RET
```




```
EXPORT kernel_sock_ioctl
:1                      ; CODE XREF: cntl_s
                        ; socket_init_work_
                        ; DATA XREF: ...

= -0x20
= -0x10

STP             X29, X30, [SP,#var_20]!
MOV             X5, #0xFFFFFFFFFFFFFFFF
MOV             X3, SP
MOV             X29, SP
AND             X3, X3, #0xFFFFFFFFFFFFC000
STR             X19, [SP,#0x20+var_10]
LDR             X19, [X3,#8]
STR             X5, [X3,#8]
NOP
LDR             X3, [X0,#0x28]
LDR             X3, [X3,#0x48]
BLR             X3
MOV             X1, SP
```

Leak kernel heap address

- The lower 4 bytes is enough
  - Without KASLR, the higher 4 bytes is 0xFFFFFFC0.

- Keep X0 unchanged

```
// fs/file_table.c
void sk_free(struct sock *sk)
{
        if (atomic_dec_and_test(&sk->sk_wmem_alloc))
                __sk_free(sk);
}EXPORT_SYMBOL(sk_free);
```

Leak kernel heap address

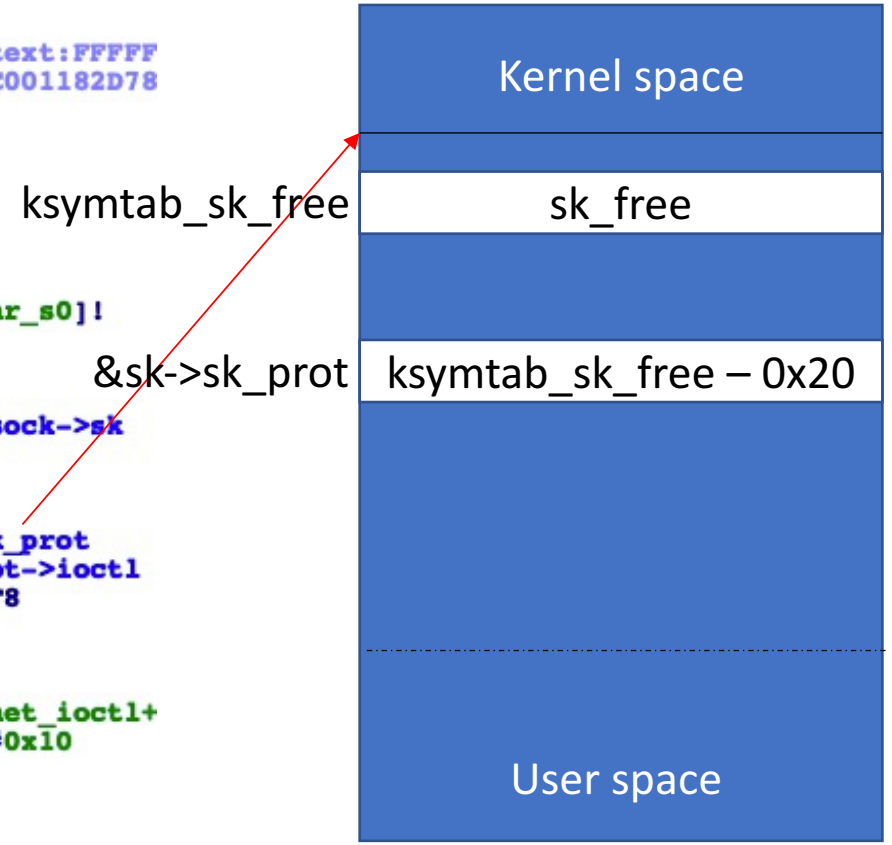
- The function is stored in kernel symbol table.

```
EXPORT inet_ioctl
                        ; DATA XREF: .text:FFFFF
                        ; .text:FFFFFFC001182D78

=  0
=  0x10
=  0x20
=  0x30
=  0x40

STP             X29, X30, [SP,#-0x10+var_s0]!
SUB             W3, W1, #8,LSL#12
MOV             X29, SP
SUB             W3, W3, #0x906
LDR             X4, [X0,#0x20] ; sk = sock->sk
CMP             W3, #0x4F
LDR             X0, [X4,#0x30]
B.LS            loc_FFFFFFC000E5EF94
LDR             X0, [X4,#0x28] ; sk->sk_prot
LDR             X3, [X0,#0x20] ; sk_prot->ioctl
CBZ             X3, loc_FFFFFFC000E5EFF8
MOV             X0, X4
BLR             X3

EF8C                    ; CODE XREF: inet_ioctl+
LDP             X29, X30, [SP+var_s0],#0x10
RET
```




```
EXPORT sk_free
                        ; CODE XREF: tun_relea
                        ; .text:FFFFFFC00069EF

=  0

STP             X29, X30, [SP,#-0x10+var_s0]!
ADD             X3, X0, #0x11C
MOV             X29, SP

9ABB0                   ; CODE XREF: sk_free+1
LDXR            W1, [X3]
SUB             W1, W1, #1
STLXR           W2, W1, [X3]
CBNZ            W2, loc_FFFFFFC000D9ABB0
DMB             ISH
CBNZ            W1, loc_FFFFFFC000D9ABCC
BL              __sk_free

9ABCC                   ; CODE XREF: sk_free+2
LDP             X29, X30, [SP+var_s0],#0x10
RET
```

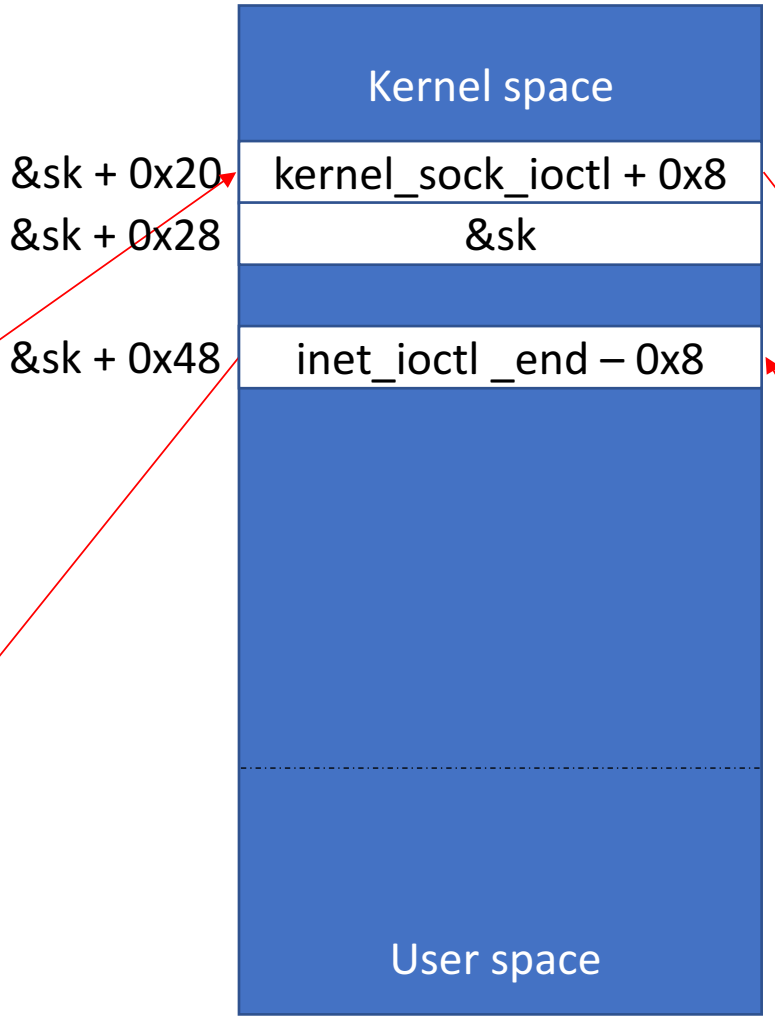
Bypass PXN and PAN

```
EXPORT inet_ioctl
                        ; DATA XREF: .text:FFFFFF
                        ; .text:FFFFFFC001182D78

=  0
=  0x10
=  0x20
=  0x30
=  0x40

STP             X29, X30, [SP,#-0x10+var_s0]!
SUB             W3, W1, #8,LSL#12
MOV             X29, SP
SUB             W3, W3, #0x906
LDR             X4, [X0,#0x20] ; sk = sock->sk
CMP             W3, #0x4F
LDR             X0, [X4,#0x30]
B.LS            loc_FFFFFFC000E5EF94
LDR             X0, [X4,#0x28] ; sk->sk_prot
LDR             X3, [X0,#0x20] ; sk_prot->ioctl
CBZ             X3, loc_FFFFFFC000E5EFF8
MOV             X0, X4
BLR             X3

EF8C                    ; CODE XREF: inet_ioctl+
LDP             X29, X30, [SP+var_s0],#0x10
RET
```




```
EXPORT kernel_sock_ioctl
:1                      ; CODE XREF: cntl_s
                        ; socket_init_work_
                        ; DATA XREF: ...

= -0x20
= -0x10

STP             X29, X30, [SP,#var_20]!
MOV             X5, #0xFFFFFFFFFFFFFFFF
MOV             X3, SP
MOV             X29, SP
AND             X3, X3, #0xFFFFFFFFFFFFC000
STR             X19, [SP,#0x20+var_10]
LDR             X19, [X3,#8]
STR             X5, [X3,#8]
NOP
LDR             X3, [X0,#0x28]
LDR             X3, [X3,#0x48]
BLR             X3
MOV             X1, SP
```

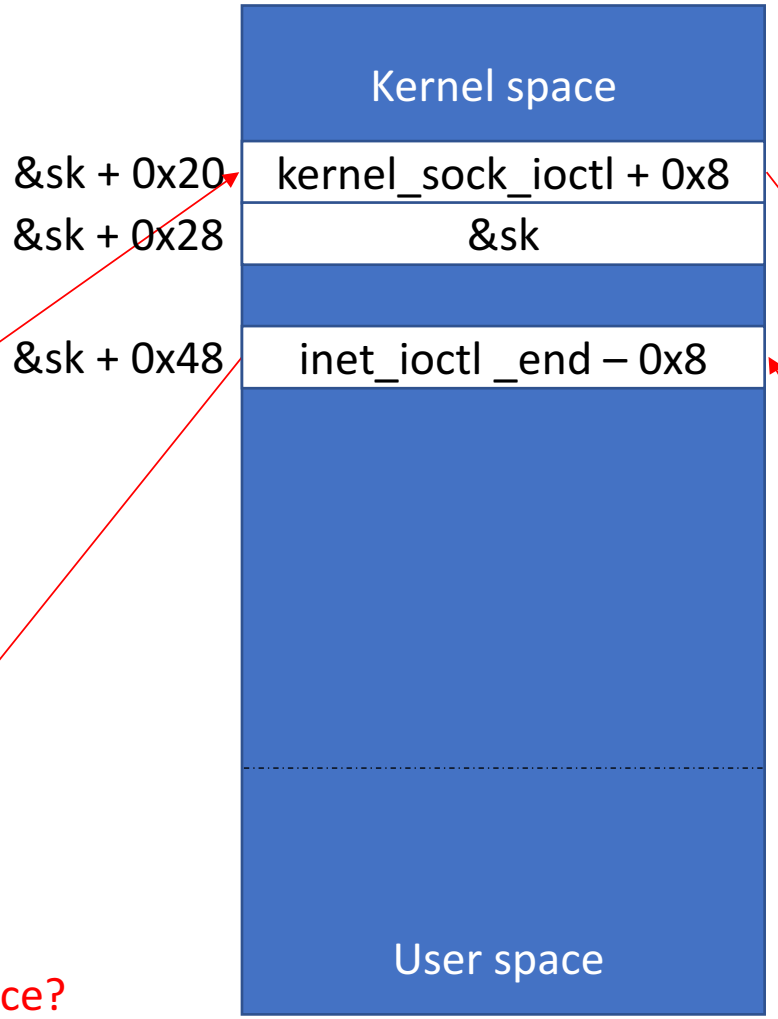
Bypass PXN and PAN

```
EXPORT inet_ioctl
                        ; DATA XREF: .text:FFFFF
                        ; .text:FFFFFFC001182D78

=  0
=  0x10
=  0x20
=  0x30
=  0x40

STP             X29, X30, [SP,#-0x10+var_s0]!
SUB             W3, W1, #8,LSL#12
MOV             X29, SP
SUB             W3, W3, #0x906
LDR             X4, [X0,#0x20] ; sk = sock->sk
CMP             W3, #0x4F
LDR             X0, [X4,#0x30]
B.LS            loc_FFFFFFC000E5EF94
LDR             X0, [X4,#0x28] ; sk->sk_prot
LDR             X3, [X0,#0x20] ; sk_prot->ioctl
CBZ             X3, loc_FFFFFFC000E5EFF8
MOV             X0, X4
BLR             X3

EF8C                    ; CODE XREF: inet_ioctl+
LDP             X29, X30, [SP+var_s0],#0x10
RET
```



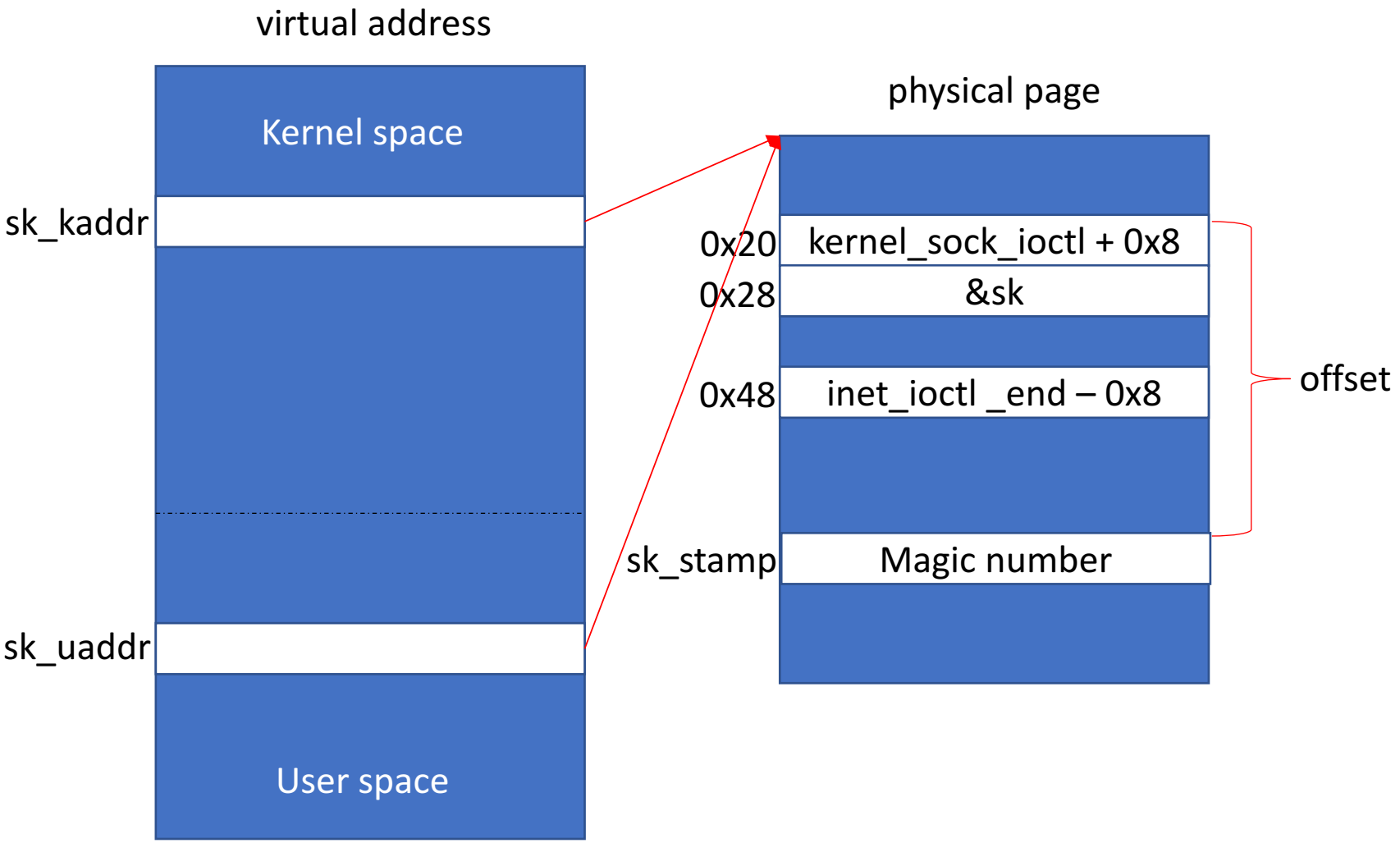

```
EXPORT kernel_sock_ioctl
:1                      ; CODE XREF: cntl_s
                        ; socket_init_work_
                        ; DATA XREF: ...

= -0x20
= -0x10

STP             X29, X30, [SP,#var_20]!
MOV             X5, #0xFFFFFFFFFFFFFFFF
MOV             X3, SP
MOV             X29, SP
AND             X3, X3, #0xFFFFFFFFFFFFC000
STR             X19, [SP,#0x20+var_10]
LDR             X19, [X3,#8]
STR             X5, [X3,#8]
NOP
LDR             X3, [X0,#0x28]
LDR             X3, [X3,#0x48]
BLR             X3
MOV             X1, SP
```

How can I put those values at the right place?

Ideal page overlap

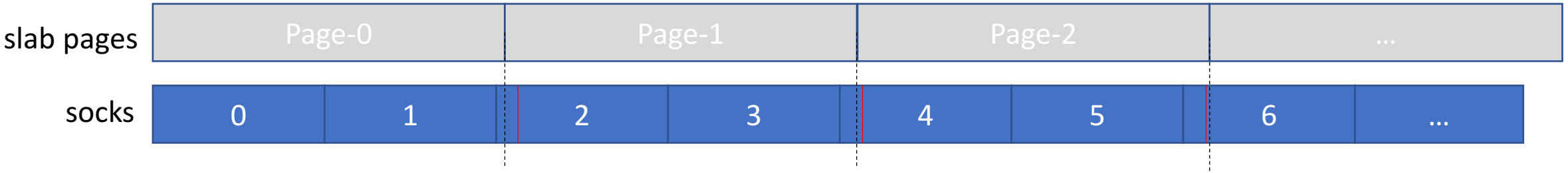


Potential crash

- The controlled sock object may not be stored in the same page.
- The physical pages for mmap syscall are probably not continuous.
  - User addresses are continuous.
- The gadget layout created through the controlled sock object’s user address can be wrong.
  - The problem also exists in easy mode.

Potential crash

- tcp_sk_sz=0x788;obj_sz=0x780;stamp_offset=0x240;prot_offset=0x28



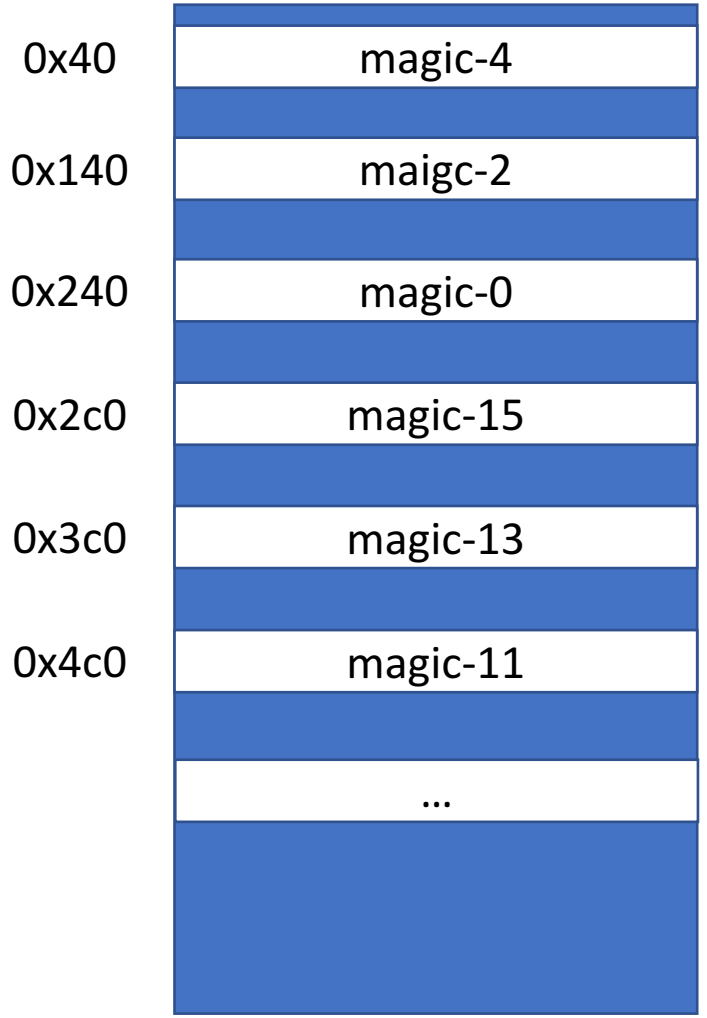
- For obj-2 and obj-4
  - sk_prot and sk_stamp are not in the same page.

Index Encoding

- Mark every user page
  - (index * obj_size + stamp_offset) % PAGE_SIZE
- Encode the index of sock objects
  - magic_value | index

Index Encoding

- Mark every user page
  - (index * obj_size + stamp_offset) % PAGE_SIZE
- Encode the index of sock objects
  - magic_value | index
- For obj_sz=0x780 and stamp_offset=0x240
  - 16 objects per slab(2^3 pages)



Medium mode

- Exploitation steps
  - Step 0: Start the evil servers and fill the holes
  - Step 1: Spawn client threads and create the UAF socks
  - Step 2: Call mmap syscall and mark the pages
  - Step 3: Check if socks controlled and filter the bad index
  - Step 4: If true, break. Else, go to step 3
  - Step 5: Call ioctl syscall and leak the kernel address of the controlled sock
  - Step 6: Rebuild gadget chain and flush the pages
  - Step 7: Call ioctl syscall and gain kernel arbitrary R/W
  - Step 8: Overwrite uid, disable SELinux
  - Step 9: Remove the release routine and spawn a ROOT shell

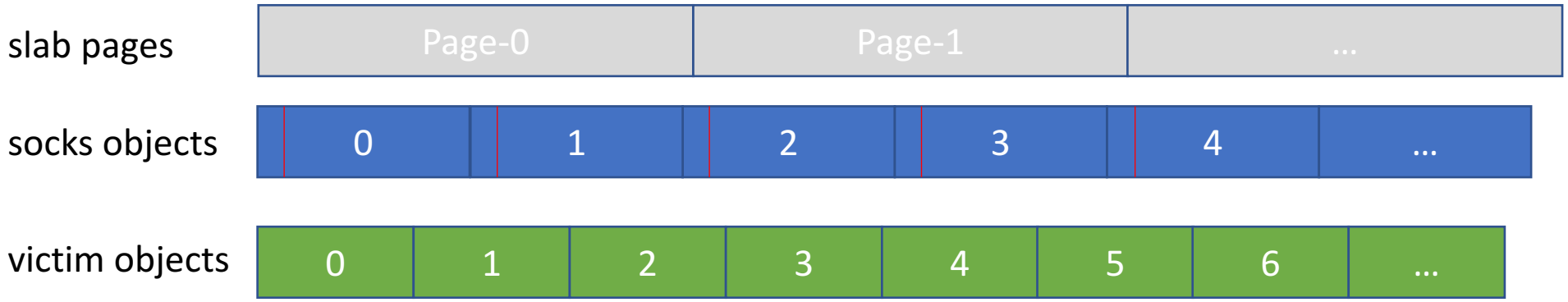
Android Rooting

Hard mode

- With KASLR mitigation
  - The addresses of kernel pointers are not known
- PC is uncontrollable without the kernel slide
  - PXN and PAN cannot be bypassed
- Leak kernel information from ‘sk_timestamp’ field
  - Fill the freed sock objects with another kind object

Ideal object overlap

- Object overlap



- Victim object
  - Kernel pointers set *P* is not empty.
  - At least one combination (index_sk, index_vim, p) satisfy:
    - index_sk*sk_sz + stamp_offset = index_vim*vim_sz + p, where
    - 0 <= index_sk < n_sk per slab, 0 <= index_vim < n_vim per slab
    - p$\in$*P*, both index_sk and index_vim are integer.

Victim candidate

- Tcp_sock objects are allocated in dedicated slab cache.
  - Unlike objects allocated in Kmalloc-N slab cache
  - Sendmmsg()/add_key()/iovecs spray are useless
  - Overlap the physical pages

Victim candidate

- Tcp_sock objects are allocated in dedicated slab cache.
  - Unlike objects allocated in Kmalloc-N slab cache
  - Sendmmsg()/add_key()/iovecs spray are useless
  - Overlap the physical pages
- Close a sock file descriptor
  - Free ’struct file’, ‘struct socket’, ‘struct sock’ and so on
  - Almost all the related slab caches are dedicated
- Other protocol objects are the best candidates
    - udp_sock/ping_sock/netlink_sock

Overlap trap

- The physical pages used by slab cache cannot be freed directly.
  - The freed victim objects are chained in the free list.

Overlap trap

- The physical pages used by slab cache cannot be freed directly.
  - The freed victim objects are chained in the free list.

- The physical pages associated with user virtual addresses can be freed!

Overlap trap

- The physical pages used by slab cache cannot be freed directly.
  - The freed victim objects are chained in the free list.
- The physical pages associated with user virtual addresses can be freed!
- The right sequence during exploiting
  - 1. lock a target user page and free others
  - 2. leak the kernel pointers

Bypass PAN

- Leaking the lower 4 bytes is not enough

Bypass PAN

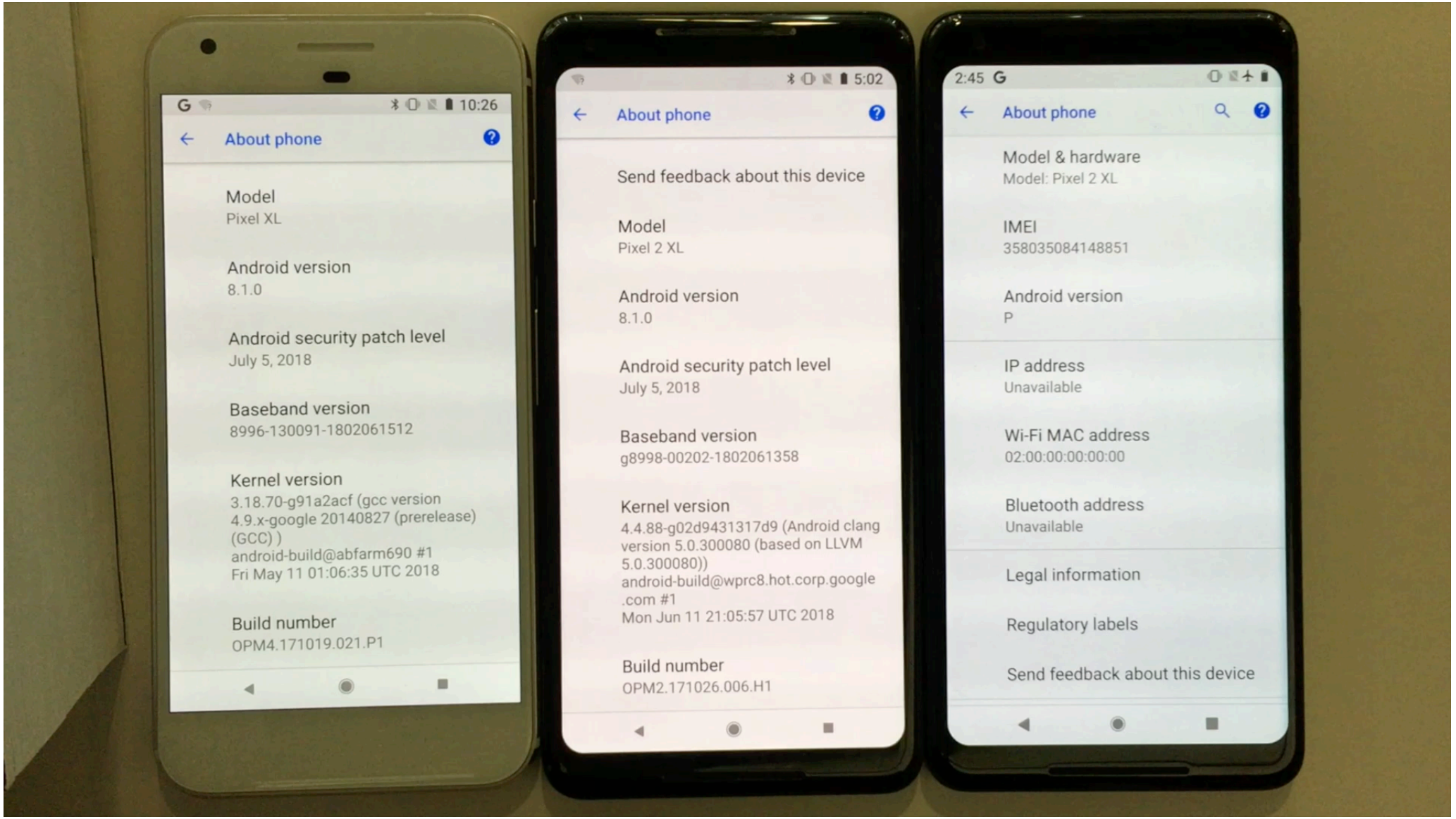
- Leaking the lower 4 bytes is not enough
- Call a function which can leak X0 into the controlled user page

```
                EXPORT klist_init
klist_init                              ; CODE XREF: device_private_init+44↑p
                                        ; bus_register+150↑p ...
                ADD             X3, X0, #8
                STR             X3, [X0,#8]
                STR             X3, [X3,#8]
                STRH            WZR, [X0]
                STRH            WZR, [X0,#2]
                STR             X1, [X0,#0x18]
                STR             X2, [X0,#0x20]
                RET
; End of function klist_init
```

Hard mode

- Exploitation steps
  - Step 0: Start the evil servers and fill the holes
  - Step 1: Spawn client threads and create the UAF socks
  - Step 2: Call mmap syscall and mark the pages
  - Step 3: Check if socks controlled and filter the bad index
  - Step 4: If true, break. Else, go to step 3
  - Step 5: Lock the target page and free other pages
  - Step 6: Repeatedly call socket syscall and check if kernel pointers leaked
  - Step 7: Revise the gadget pointers and flush the pages
  - Step 8: Call ioctl syscall and leak the kernel address of the controlled sock
  - Step 9: Rebuild gadget chain and flush the pages
  - Step 10: Call ioctl syscall and gain kernel arbitrary R/W
  - Step 11: Overwrite uid, disable SELinux
  - Step 12: Remove the release routine and spawn a ROOT shell

Demo


10:26
About phone
Model
Pixel XL
Android version
8.1.0
Android security patch level
July 5, 2018
Baseband version
8996-130091-1802061512
Kernel version
3.18.70-g91a2acf (gcc version 4.9.x-google 20140827 (prerelease) (GCC))
android-build@abfarm690 #1
Fri May 11 01:06:35 UTC 2018
Build number
OPM4.171019.021.P1
5:02
About phone
Send feedback about this device
Model
Pixel 2 XL
Android version
8.1.0
Android security patch level
July 5, 2018
Baseband version
g8998-00202-1802061358
Kernel version
4.4.88-g02d9431317d9 (Android clang version 5.0.300080 (based on LLVM 5.0.300080))
android-build@wprc8.hot.corp.google.com #1
Mon Jun 11 21:05:57 UTC 2018
Build number
OPM2.171026.006.H1
2:45
About phone
Model & hardware
Model: Pixel 2 XL
IMEI
358035084148851
Android version
P
IP address
Unavailable
Wi-Fi MAC address
02:00:00:00:00:00
Bluetooth address
Unavailable
Legal information
Regulatory labels
Send feedback about this device

Android Rooting

Select your difficulty

EASY

MEDIUM

HARD

VERY HARD

Control Flow Integrity

- Kernel Control Flow Integrity enabled in Android 9
  - Kernel version 4.9 and newer

- Some vendors has enabled KCFI in Android 8, 2017
  - The implementation is different

Samsung - KCFI

// net/core/sock.c

err = sk->sk_prot->ioctl(sk, cmd, arg);

```
A82D0C                          ; CODE XREF: inet_ioctl+48↑j
                                ; inet_ioctl+68↑j ...
  LDR             X0, [X4,#0x28]
  LDR             X6, [X0,#0x20]
  MOV             W0, #0xFFFFFDFD
  CBZ             X6, loc_FFFFFFC000A82D2C
  MOV             X0, X4
  MOV             W1, W3
  MOV             X2, X5
  BL              jopp_springboard_blr_x6

A82D2C                          ; CODE XREF: inet_ioctl+58↑j
                                ; inet_ioctl+70↑j ...
  LDP             X29, X30, [SP+var_s0],#0x10
  RET
.on inet_ioctl
```

Forward-edge check

Samsung - KCFI

// net/core/sock.c

err = sk->sk_prot->ioctl(sk, cmd, arg);

```
                EXPORT jopp_springboard_blr_x6
jopp_springboard_blr_x6                 ; CODE XREF: disable nonboot cpus+210↑p
                                        ; enable nonboot cpus+240↑p ...
                STP             X16, X18, [SP,#-0x10]!
                LDUR            W16, [X6,#-4]
                SUBS            W16, W16, #0xBE7,LSL#12
                CMP             W16, #0xBAD
                B.EQ            loc_FFFFFFC000BA6334
; ---------------------------------------------------------------------------
                DCB 0xDE, 0xC0, 0xAD, 0xDE
; ---------------------------------------------------------------------------

loc_FFFFFFC000BA6334                    ; CODE XREF: .text:FFFFFFC000BA632C↑j
                LDP             X16, X18, [SP],#0x10
                BR              X6
; ---------------------------------------------------------------------------
```

The called function must have the magic ‘0xBE7BAD’ before the first instruction.

Huawei - KCFI

```
                EXPORT inet_ioctl
inet_ioctl                              ; DATA XREF: .tex
                                        ; .text:FFFFFF800
                SUB             W3, W1, #8,LSL#12
                STP             X29, X30, [SP,#-0x10]!
                SUB             W3, W3, #0x906
                CMP             W3, #0x4F
                MOV             X29, SP
                LDR             X0, [X0,#0x20]
                B.LS            loc_FFFFFF8008EC2F8C
                LDR             X3, [X0,#0x28]
                LDR             X3, [X3,#0x20]
                CBZ             X3, loc_FFFFFF8008EC3000
                LDUR            W4, [X3,#-4]
                CMP             W4, #0x35C,LSL#12
                B.NE            loc_FFFFFF8008EC3008
                BLR             X3

loc_FFFFFF8008EC2F84                    ; CODE XREF: .tex
                LDP             X29, X30, [SP],#0x10
                RET


loc_FFFFFF8008EC3008                    ; CODE XREF: .tex
                BL              __cfi_report
```

Forward-edge check ‘0x35C000’

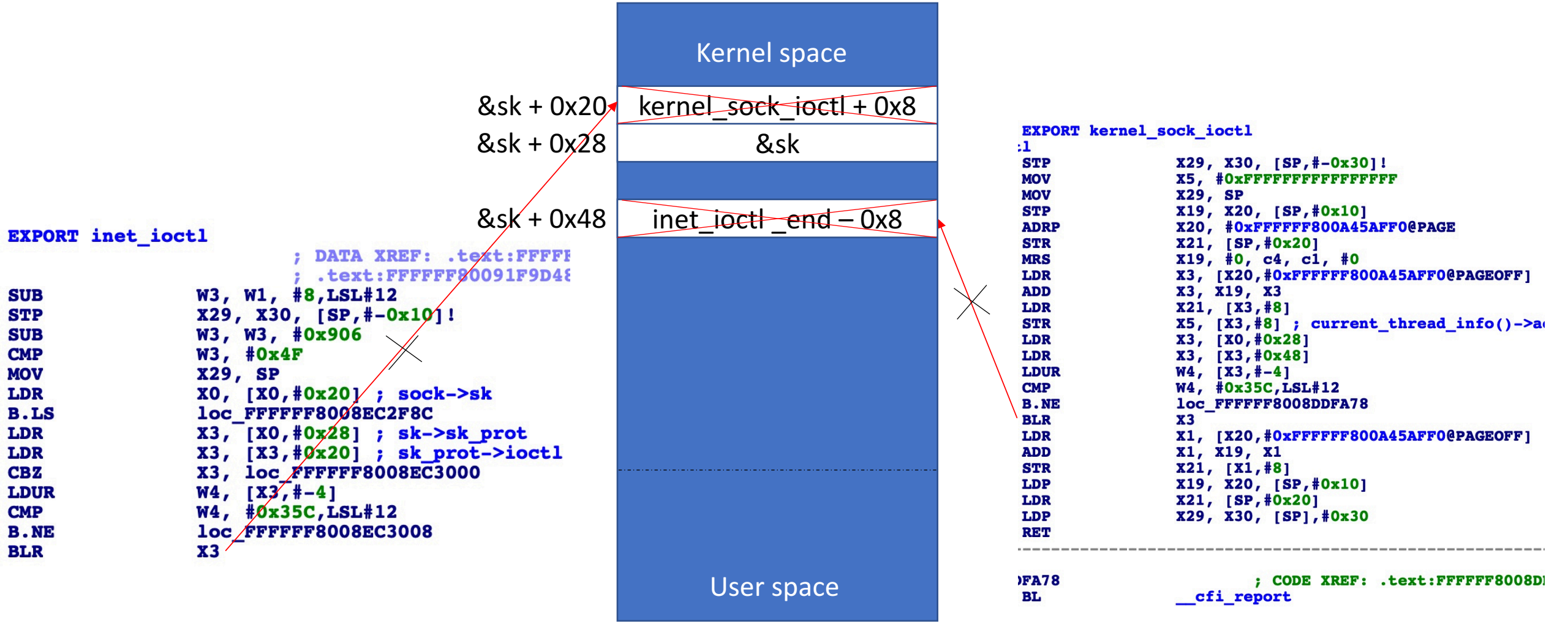
Pointer Authentication Code

- Forward-edge check
  - BLRA* instructions
- Backward-edge check
  - RETA* instructions
- Context-sensitive with a specific key
  - Xn/SP
  - iOS A12 - A/B key

Weakness

- Both KCFI implementation are just forward-edge check
  - No backward-edge check
- All the indirect branch perform the check
  - The magic number is context-free

Bypass KCFI


Kernel space
&sk + 0x20
kernel_sock_ioctl + 0x8
&sk + 0x28
&sk
&sk + 0x48
inet_ioctl _end – 0x8
User space
EXPORT inet_ioctl
; DATA XREF: .text:FFFFF
; .text:FFFFFF80091F9D48
SUB W3, W1, #8,LSL#12
STP X29, X30, [SP,#-0x10]!
SUB W3, W3, #0x906
CMP W3, #0x4F
MOV X29, SP
LDR X0, [X0,#0x20] ; sock->sk
B.LS loc_FFFFFF8008EC2F8C
LDR X3, [X0,#0x28] ; sk->sk_prot
LDR X3, [X3,#0x20] ; sk_prot->ioctl
CBZ X3, loc_FFFFFF8008EC3000
LDUR W4, [X3,#-4]
CMP W4, #0x35C,LSL#12
B.NE loc_FFFFFF8008EC3008
BLR X3
EXPORT kernel_sock_ioctl
:l
STP X29, X30, [SP,#-0x30]!
MOV X5, #0xFFFFFFFFFFFFFFFF
MOV X29, SP
STP X19, X20, [SP,#0x10]
ADRP X20, #0xFFFFFF800A45AFF0@PAGE
STR X21, [SP,#0x20]
MRS X19, #0, c4, c1, #0
LDR X3, [X20,#0xFFFFFF800A45AFF0@PAGEOFF]
ADD X3, X19, X3
LDR X21, [X3,#8]
STR X5, [X3,#8] ; current_thread_info()->a
LDR X3, [X0,#0x28]
LDR X3, [X3,#0x48]
LDUR W4, [X3,#-4]
CMP W4, #0x35C,LSL#12
B.NE loc_FFFFFF8008DDFA78
BLR X3
LDR X1, [X20,#0xFFFFFF800A45AFF0@PAGEOFF]
ADD X1, X19, X1
STR X21, [X1,#8]
LDP X19, X20, [SP,#0x10]
LDR X21, [SP,#0x20]
LDP X29, X30, [SP],#0x30
RET
)FA78 ; CODE XREF: .text:FFFFFF8008D
BL __cfi_report

Bypass KCFI

- Gadget chain
  - inet_ioctl()->kernel_sock_ioctl()->funX()
  - KERNEL_DS context
  - copy_to_user/put_user

Kernel arbitrary R/W

```
// fs/pipe.c
static long pipe_ioctl(struct file *filp, unsigned int cmd, unsigned long arg){
        struct pipe_inode_info *pipe = filp->private_data;
        …
        case FIONREAD:
                count = 0;
                buf = pipe->curbuf;
                nrbufs = pipe->nrbufs;
                while (--nrbufs >= 0) {
                        count += pipe->bufs[buf].len;
                        buf = (buf+1) & (pipe->buffers - 1);
                }
        return put_user(count, (int __user *)arg);  // KERNEL_DS
```

Very hard mode

- Exploitation steps
  - Step 0: Start the evil servers and fill the holes
  - Step 1: Spawn client threads and create the UAF socks
  - Step 2: Call mmap syscall and mark the pages
  - Step 3: Check if socks controlled and filter the bad index
  - Step 4: If true, break. Else, go to step 3
  - Step 5: Lock the target page and free other pages
  - Step 6: Repeatedly call socket syscall and check if kernel pointers leaked
  - Step 7: Revise the gadget pointers and flush the pages
  - Step 8: Call ioctl syscall and leak the kernel address of the controlled sock
  - Step 9: Rebuild gadget chain and flush the pages
  - Step 10: Call ioctl syscall and gain kernel arbitrary R/W
  - Step 11: Bypass vendors' mitigations
  - Step 12: Remove the release routine and spawn a ROOT shell

Demo


飞行模式
下午4:31
关于手机
设备名称
HUAWEI Mate 10
型号
ALP-AL00
版本号
ALP-AL00 8.0.0.153(C00)
GPU Turbo
EMUI 版本
8.0.0
Android 版本
8.0.0
IMEI
866158038399241
866158038464953
MEID
A000007FC996D5
处理器
HiSilicon Kirin 970
运行内存
4.0 GB
手机存储
可用空间：51.43 GB
总容量：64.00 GB
屏幕
2560 x 1440
Android 程序安全补丁级别
2018年7月1日

Agenda

- Overview
- Vulnerability analysis
- WrongZone rooting solution
- *Conclusion*

Takeaways

- A quite rare type confusion bug in Android kernel is discussed.
- The implementation of the vendors' KCFI mitigation and the ideas of bypassing all the mitigations on Android devices are detailed.
- Building a universal Android rooting solution is becoming more and more challenging, but it's still possible.

References

- https://patchwork.kernel.org/patch/10437615/
- https://towelroot.com
- https://www.blackhat.com/docs/us-15/materials/us-15-Xu-Ah-Universal-Android-Rooting-Is-Back.pdf
- https://dirtycow.ninja
- https://www.blackhat.com/docs/asia-18/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf
- http://www.linuxhowtos.org/data/6/server.c
- https://developer.android.com/about/dashboards
- https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-kemerlis.pdf
- https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html
- https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege.pdf
- https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf
- https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html
- https://unsplash.com/photos/_Apdet7E5yU

Zer0Con2019

Thank you

WANG, YONG (@ThomasKing2014)

ThomasKingNew@gmail.com