ArmV8 64 bits: Syscall Security

Gustavo Kundlatsch, Paola de Oliveira, Pedro Souza

28 de setembro de 2022

1 Introdução

System calls, ou *syscalls*, são a maneira pela qual um programa qualquer pode requisitar uma funcionalidade do kernel de um sistema operacional, como controle de hardware (HD, câmera), criação de um novo processo ou realizar comunicação com alguma funcionalidade do sistema operacional (escalonamento, memory management) [11].

Syscalls são necessárias pois, na maioria das arquiteturas, não é todo programa que tem acesso às funções citadas anteriormente por questões de segurança, sendo necessário requisitar ao kernel que as execute. O sistema operacional possui o nível mais alto de privilégio, e fornece uma interface para que programas com níveis mais baixos acessem suas funcionalidades através de syscalls.

Neste trabalho, mostraremos o funcionamento das syscalls na arquitetura ARMv8 64 bits e na máquina Cortex-A53. A fonte principal de informações para a confecção do texto foi o manual de referência da arquitetura ARMv8 [1].

2 Exceções

Uma exceção é um sinal gerado para alterar o fluxo de processamento de um software. Alguns tipos de exceção são: eventos de debug; uma instrução indefinida detectada; interrupções. Exceções são divididas entre as exceções síncronas e exceções assíncronas.

2.1 Exceções Síncronas

As principais causas de exceções síncronas são:

- Tentativa de executar uma instrução `undefined`;
- Uso stack pointer desalinhado;
- Tentativa de executar uma instrução com o program counter desalinhado;
- **Instruções SVC, HVC ou SMC**.

A princípio, uma única instrução qualquer pode gerar uma série de exceções síncronas diferentes, entre os da instrução, sua decodificação e eventual execução. Por conta disso, toda exceção síncrona possui uma prioridade diferente, onde 1 é a prioridade mais alta.

2.2 Exceções Assíncronas

Na arquitetura ARMv8, exceções assíncronas são chamadas de interrupções. As interrupções por sua vez são separadas em duas categorias: físicas e virtuais. Interrupções físicas são sinais enviados ao Processing Element (PE, termo utilizado para se referir a um Core) de fora do PE, como erros de sistema. Interrupções virtuais são interrupções que o software em execução no EL2 pode ativar e tornar pendentes.

2.3 Níveis de Exceção

A arquitetura ARMv8 define 4 níveis diferentes de exceção, de 0 a 3, com níveis ascendentes de privilégios. O nível EL0 é chamado de nível sem privilégios; é o modo padrão de aplicativos em modo de usuário. O nível EL1 é o modo supervisor, o nível de privilégio do *kernel* e de funções associadas. Os sistemas com virtualização em hardware introduzem o nível EL2, o modo de *hypervisor*, dedicado a hipervisores e *Virtual Machine Monitors* (VMMs). Por fim, o EL3 é o nível *secure monitor*, que permite a troca de estado de segurança entre os estados Seguro e Não-seguro.

Os únicos níveis que precisam necessariamente ser implementados são os níveis EL0 e EL1, e os níveis implementados não precisam ser contíguos (i. e., é possível implementar apenas os níveis EL0, EL1 e EL3). O nível EL3 é o único que consegue trocar o estado de segurança, logo, não implementá-lo significa não ter acesso a qualquer estado de segurança. Similarmente, não implementar o nível EL2 resulta em não ter acesso a muitas das funcionalidades para virtualização.

2.4 Estados de Segurança

O nível EL3 permite a troca de estado entre os estados Seguro e Não-seguro. Esses estados definem o grau de permissão de acesso a endereços físicos de memória que o PE tem acesso:

- **Estado Seguro:** Nesse estado o PE consegue acessar tanto os endereços de memória Seguros quanto os Não-seguros.
- **Estado Não-seguro:** Nesse estado o PE consegue acessar apenas os endereços de memória Não-seguros.

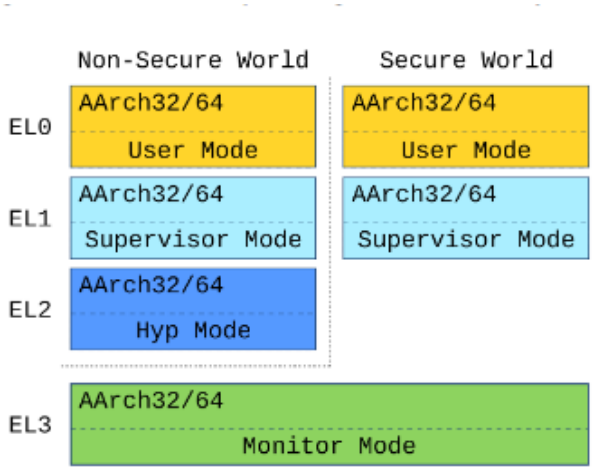

Figura 1: Hierarquia de níveis de exceção [9].

3 System calls

Algumas instruções ou funções do sistema só podem ser efetuadas em um nível específico de exceção. Se o código executando em um nível de exceção mais baixo precisa efetuar uma operação privilegiada, por exemplo, quando uma aplicação pede uma funcionalidade do kernel, é necessário que a aplicação gere uma *system call*, uma exceção síncrona realizada com o intuito de ocorrer a troca de contexto para o nível de exceção desejado e conseguinte execução de função privilegiada pelo nível superior de exceção.

Tradicionalmente, as syscalls são implementadas por meio do processo mostrado na Figura 2: o processo do usuário coloca os dados em um local predeterminado – geralmente em registradores ou na pilha – e então dispara um mecanismo específico para fazer com que o kernel assuma o controle da execução do processador (por meio de uma interrupção ou uma instrução específica).

Na arquitetura ARMv8, as syscalls são executadas através de instruções específicas de acordo com o nível de privilégio que se pretende usar. Os parâmetros da syscall podem ser passados por registradores ou codificados na própria syscall. [2] Uma system call pode ser gerada pela execução de uma instrução SVC, HVC ou SMC. Por padrão, uma instrução `SVC` gera uma exceção síncrona no nível EL1, permitindo que um aplicativo rodando no nível EL0 tenha acesso ao Kernel. Caso a implementação inclua o nível EL2, a instrução `HVC` pode ser usada para fazer uma chamada ao Hypervisor.

Abaixo é demonstrado como realizar uma chamada da syscall write() para escrever o tradicional “Hello world” na tela [8].

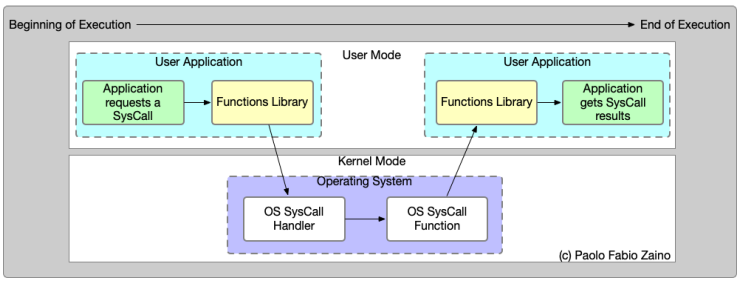

Figura 2: Funcionamento tradicional de uma system call [6].

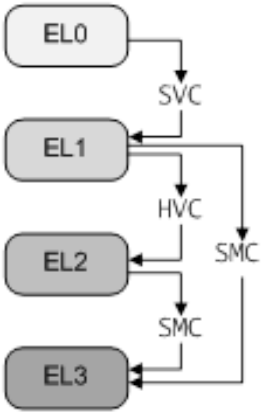

Figura 3: Níveis de exceção nas syscalls [4].

```
/* Our application's entry point. */
.data

/* Data segment: define our message string and calculate its length. */
msg:
    .ascii        "Hello, ARM64!\n"
len = . - msg

.text

/* Our application's entry point. */
.globl _start
_start:
    /* syscall write(int fd, const void *buf, size_t count) */
    mov     x0, #1      /* fd := STDOUT_FILENO */
    ldr     x1, =msg    /* buf := msg */
    ldr     x2, =len    /* count := len */
```

```
mov     w8, #64     /* write is syscall #64 */
svc     #0          /* invoke syscall */

/* syscall exit(int status) */
mov     x0, #0      /* status := 0 */
mov     w8, #93     /* exit is syscall #1 */
svc     #0          /* invoke syscall */
```

Para a execução do código acima, é feita a compilação utilizando um *cross-compiler*:

```
aarch64-linux-gnu-as -o hello.o hello.s
aarch64-linux-gnu-ld -s -o hello hello.o
qemu-aarch64 ./hello
```

Para realizar uma syscall com ARMv8, os argumentos são guardados nos registradores `$x0 - $x7`, e o número da função no registrador `$x8`. A função write() é representada pelo código 4 – cada função possui um número fixo e pré-definido pela arquitetura. Por fim, a instrução `svc $0` invoca a syscall.

De maneira similar podemos observar uma chamada da syscall exit(), que armazena o status no registrador `$R0` e possui código 1.

4 Problemas de segurança do kernel e syscalls

O kernel é responsável por organizar recursos e escalonar os outros processos do sistema. Ele tem acesso direto ao hardware, o que é permitido pois ele roda no modo privilegiado EL1. O kernel também monitora as aplicações em modo de usuário (EL0) em execução e suas alocações de memória e comunicações com o hardware.

Devido ao seu acesso irrestrito à memória e modo privilegiado, o kernel é crítico ao funcionamento correto do sistema e também pode ser alvo de ataques, logo, é fundamental que o kernel seja o mais seguro e livre de erros possível. Contudo, para garantir a segurança do kernel, também é necessário garantir a segurança da interface de system calls, que conecta o nível menos privilegiado EL0 ao nível supervisor EL1. Isso acontece pois, mesmo que a compilação individual do kernel e do espaço de usuário garantam segurança, o fato deles não serem compilados em conjunto significa que essas garantias podem ser quebradas na interface de comunicação entre eles [10].

4.1 Corrupção de memória

A corrupção de memória acontece quando um programa ganha acesso à memória que ele não deveria poder ter acesso, dando a ele a habilidade de causar comportamento inesperado no sistema. É possível violar a memória de forma espacial ou temporal.

Em C++, a linguagem usada para escrita do EPOS, pode ser difícil alcançar segurança de memória completa. Ponteiros para objetos individuais podem ser confundidos facilmente com arrays, e eles não precisam ter limites de tamanho ou deslocamento definidos na instanciação, então eles podem apontar para qualquer lugar sem ser claro se tal referência é uma violação. Além disso, a segurança temporal se torna difícil em lugares onde a gerência de memória é manual [10].

4.1.1 Violação espacial

Uma violação espacial de memória ocorre quando um ponteiro é usado para acessar uma região de memória que ele não foi definido para apontar [7]. Por exemplo, se existe um ponteiro para um buffer em memória, aconteceria uma violação se esse ponteiro pudesse acessar ou sobrescrever dados de um buffer adjacente, como acontece em um buffer overflow, em que são escritos mais dados dentro de um buffer do que ele consegue comportar, fazendo com que os dados sobrando "transbordem" para endereços de memória adjacentes. Isso pode ocorrer, por exemplo, em uma system call de escrita que recebe um ponteiro e o seu tamanho, e o tamanho do ponteiro passado como argumento não corresponde ao tamanho real do ponteiro, e também não há checagem sobre esse tipo de discrepância na implementação da syscall.

Se o buffer onde ocorre o overflow for uma variável temporária de uma função, ele possivelmente será armazenado na stack, que também armazena retornos de funções. Dessa forma, com um overflow na stack causado por, por exemplo, uma syscall write() sem checagens de parâmetro corretas, é possível mudar o endereço de retorno de uma função ao sobrescrever o valor original devido à violação de memória causada pelo overflow, o que faz com que seja possível redirecionar o fluxo de um programa de forma maliciosa ou no mínimo causar uma falha de segmentação.

4.1.2 Violação temporal

Uma violação temporal ocorre quando um ponteiro é usado fora do tempo de vida que foi originalmente definido, como no caso de um ponteiro ser desreferenciado depois de ter sido liberado (use-after-free). Um ponteiro desses é chamado de ponteiro pendente/*dangling pointer* e seu uso causa comportamento não definido, já que os dados para os quais ele apontava podem não estar mais naquele endereço de memória, ou pior, o endereço pode ter sido usado para alocar outra estrutura de dados, permitindo que o ponteiro antigo seja usado para corromper dados que agora têm outro propósito [7].

4.2 Desreferenciação de ponteiros corrompidos/não validados

Essa categoria cobre qualquer situação em que um ponteiro é usado enquanto o seu conteúdo foi corrompido ou não foi validado o suficiente. Um ponteiro corrompido normalmente é consequência de algum outro erro, como um buffer

overflow, que pode corromper um ou mais bytes do conteúdo do ponteiro, como descrito anteriormente. Esse tipo de situação dá a um possível atacante mais controle sobre o conteúdo da variável, o que leva diretamente a um ataque mais confiável [7].

Problemas causados por um ponteiro não-validado fazem mais sentido em um espaço de endereçamento combinado para o kernel e espaço de usuário, como é o caso da arquitetura ARMv8-A. Nela, o espaço de endereçamento do nível EL1 fica em endereços acima do espaço de endereçamento do nível EL0, e os registradores TCR_EL0 e TCR_EL1 são usados para definir o tamanho e portanto endereço limite desses espaços de endereçamento [3]. Dessa forma, funções internas do kernel podem usar os endereços limite dos espaços de endereçamento para decidir se um ponteiro específico aponta para o kernel ou para o espaço de usuário. No primeiro caso, normalmente são feitas menos checagens devido ao nível de privilégio maior, enquanto no segundo caso é preciso de mais cuidado ao acessar o endereço. Se essa checagem não estiver presente ou for aplicada incorretamente, um endereço de espaço de usuário pode ser desreferenciado sem o controle necessário [7].

4.3 Condições de corrida

Uma condição de corrida acontece quando dois ou mais atores (por exemplo, processos ou threads) querem realizar uma ação sob o mesmo objeto "ao mesmo tempo" e o resultado será diferente dependendo da ordem que cada ação ocorrer. Para que a condição de corrida ocorra, os atores precisam executar suas ações ou paralelamente, o que ocorre em um processadores de múltiplos cores, ou, pelo menos, concorrente,de forma intercalada uma com a outra, o que ocorre dentro de um único core devido à alternância de tarefas [7]. Ela pode ocorrer, por exemplo, devido a duas threads escrevendo em uma posição de memória compartilhada ao mesmo tempo.

Considerando o sistema operacional, esse tipo de situação não é desejado pois condições de corrida podem causar comportamento inesperado em caminhos fundamentais para o funcionamento correto do sistema. Para preveni-las, é preciso garantir algum tipo de sincronização entre os vários atores, mas falhas nessa sincronização ainda permitem condições de corrida e podem causar problemas acidentais ou serem exploradas por programas maliciosos, como, por exemplo, sobrescrever o código de um programa enquanto ele está sendo executado por uma syscall `execve()`. Ataques de condição de corrida típicos envolvem abrir um arquivo, validar um arquivo, executar uma subrotina, checar uma senha ou verificar um nome de usuário [5].

5 Experimento

A seguinte metodologia foi utilizada para testar os conceitos estudados na parte teórica:

1. Instalar as dependências do repositório Bare Metal Programming on Raspberry Pi 3 [12], que proveem um ambiente de implementação Bare Metal de ARMv8 64bits para Raspberry Pi 3;

2. Editar o tutorial `11_exception` para que o tratador de exceções implementado no `exc.c` seja capaz de tratar de syscalls;

3. Implementar nossa própria syscall e

4. Conseguir replicar algum exploit definido a partir dessa syscall.

O objetivo com essa metodologia era demonstrar como é possível realizar um ataque ao kernel utilizando uma syscall, como um ataque de escalada de privilégios ou de buffer overflow.

O ataque de escalada de privilégios consiste em se aproveitar do comportamento de uma syscall para obter um nível de privilégios elevado, como uma aplicação executando em EL0 recebendo acesso de kernel em EL1. A partir disso, o aplicativo malicioso pode utilizar recursos que só deveriam estar disponíveis para o sistema operacional.

Já os ataques de buffer overflow consistem em escrever mais dados no buffer (que pode ser a stack ou a heap) do que o esperado. Em um exploit clássico de buffer overflow, o invasor envia dados a um programa que escreve na pilha. O resultado é que as informações na pilha de chamadas são substituídas, incluindo o ponteiro de retorno da função. Os dados definem o valor do ponteiro de retorno para que, quando a função retornar, ela transfira o controle para o código malicioso contido nos dados do invasor.

Para tratar as exceções, o seguinte código foi utilizado (`start.S`):

```
.section ".text.boot"

.global _start

_start:
    // read cpu id, stop slave cores
    mrs     x1, mpidr_el1
    and     x1, x1, #3
    cbz     x1, 2f
    // cpu id > 0, stop
1:  wfe
    b       1b
2:  // cpu id == 0

    // set top of stack just before our code
    ldr     x1, =_start

    // set up EL1
    mrs     x0, CurrentEL
```

```
    and     x0, x0, #12 // clear reserved bits

    // running at EL3?
    cmp     x0, #12
    bne     5f
    // should never be executed, just for completeness
    mov     x2, #0x5b1
    msr     scr_el3, x2
    mov     x2, #0x3c9
    msr     spsr_el3, x2
    adr     x2, 5f
    msr     elr_el3, x2
    eret

    // running at EL2?
5:  cmp     x0, #4
    beq     5f
    msr     sp_el1, x1
    // enable CNTP for EL1
    mrs     x0, cnthctl_el2
    orr     x0, x0, #3
    msr     cnthctl_el2, x0
    msr     cntvoff_el2, xzr
    // enable AArch64 in EL1
    mov     x0, #(1 << 31)      // AArch64
    orr     x0, x0, #(1 << 1)   // SWIO hardwired on Pi3
    msr     hcr_el2, x0
    mrs     x0, hcr_el2
    // Setup SCTLR access
    mov     x2, #0x0800
    movk    x2, #0x30d0, lsl #16
    msr     sctlr_el1, x2
    // set up exception handlers
    ldr     x2, =_vectors
    msr     vbar_el1, x2
    // change execution level to EL1
    mov     x2, #0x3c4
    msr     spsr_el2, x2
    adr     x2, 5f
    msr     elr_el2, x2
    eret

5:  mov     sp, x1

    // clear bss
    ldr     x1, =__bss_start
```

```
    ldr     w2, =__bss_size
3:  cbz     w2, 4f
    str     xzr, [x1], #8
    sub     w2, w2, #1
    cbnz    w2, 3b

    // jump to C code, should not return
4:  bl      main
    // for failsafe, halt this core too
    b       1b

    // important, code has to be properly aligned
    .align 11
_vectors:
    // synchronous
    .align  7
    mov     x6, #0
    bl      exc_handler
    eret

    // IRQ
    .align  7
    mov     x6, #1
    bl      exc_handler
    eret

    // FIQ
    .align  7
    mov     x6, #2
    bl      exc_handler
    eret

    // SError
    .align  7
    mov     x6, #3
    bl      exc_handler
```

Esse código consiste na implementação inicial do tratador de exceções proposto pelo autor, mas modificado para que o próprio arquivo `exc.c` busque os argumentos type, esr, elr, spsr, far, current_el direto dos registradores utilizando assembly.

Esse código é executado, e então chama a função main do arquivo `main.c`:

```
#include "uart.h"
#include "mmu.h"
```

```
void main()
{
    unsigned long ret;

    // set up serial console
    uart_init();

    // set up paging
    mmu_init();

    __asm__ volatile (
        "str    x8, [sp, #-8]!\n\t"
        "str    x0, [sp, #-8]!\n\t"

        "mov    x0, #4\n\t"
        "mov    x8, #1\n\t"

        "svc    0\n\t"

        "mov    %0, x9\n\t"

        "ldr    x0, [sp], #8\n\t"
        "ldr    x8, [sp], #8\n\t"
        : "=r" (ret)
    );

    uart_puts("\nRetorno Syscall: ");
    uart_hex(ret);
    uart_puts("\n");

    // echo everything back
    while(1) {
        uart_send(uart_getc());
    }
}
```

O código assembly contido aqui apenas reserva os valores dos registradores `x0` (argumento de função) e `x8` (código da syscall), e depois os altera para 4 e 1, respectivamente. A syscall é invocada pela instrução `svc`. Quando a syscall é chamada, ela será tratada pelo código contido no `exc.c`:

```
void exc_handler()
{
    unsigned long x0, x1, x2, x3, x4, x5, x8, ret, \
                  type, esr, elr, spsr, far, current_el;
```

```
ret = 0;
__asm__ volatile (
    "mov    %0, x0\n\t"
    "mov    %1, x1\n\t"
    "mov    %2, x2\n\t"
    "mov    %3, x3\n\t"
    "mov    %4, x4\n\t"
    "mov    %5, x5\n\t"
    "mov    %6, x8\n\t"
    : "=r" (x0), "=r" (x1), "=r" (x2),
    "=r" (x3), "=r" (x4), "=r" (x5), "=r" (x8)
);
// Using x5, since our implementation
// uses registers x0-x4 for other purposes

__asm__ volatile (
    "mov    %0,  x6\n\t"
    "mrs    %1, esr_el1\n\t"
    "mrs    %2, elr_el1\n\t"
    "mrs    %3, spsr_el1\n\t"
    "mrs    %4, far_el1\n\t"
    "mrs    x7, CurrentEL\n\t"
    "and    x7, x7, #12\n\t"
    "mrs    %5, CurrentEL\n\t"
    : "=r" (type), "=r" (esr), "=r" (elr),
    "=r" (spsr), "=r" (far), "=r" (current_el)
);


// print out interruption type
switch(type) {
    case 0: uart_puts("Synchronous"); break;
    case 1: uart_puts("IRQ"); break;
    case 2: uart_puts("FIQ"); break;
    case 3: uart_puts("SError"); break;
}
uart_puts(": ");
// decode exception type
// (some, not all. See ARM DDI0487B_b chapter D10.2.28)
switch(esr>>26) {
    case 0b000000: uart_puts("Unknown"); break;
    case 0b000001: uart_puts("Trapped WFI/WFE"); break;
    case 0b001110: uart_puts("Illegal execution"); break;
    case 0b010101: ret = syscall_example(x8, x0); break;
    case 0b100000: uart_puts("Instruction abort, lower EL"); break;
    case 0b100001: uart_puts("Instruction abort, same EL"); break;
```

```
        case 0b100010: uart_puts("Instruction alignment fault"); break;
        case 0b100100: uart_puts("Data abort, lower EL"); break;
        case 0b100101: uart_puts("Data abort, same EL"); break;
        case 0b100110: uart_puts("Stack alignment fault"); break;
        case 0b101100: uart_puts("Floating point"); break;
        default: uart_puts("Unknown"); break;
    }
    // decode data abort cause
    if(esr>>26==0b100100 || esr>>26==0b100101) {
        uart_puts(", ");
        switch((esr>>2)&0x3) {
            case 0: uart_puts("Address size fault"); break;
            case 1: uart_puts("Translation fault"); break;
            case 2: uart_puts("Access flag fault"); break;
            case 3: uart_puts("Permission fault"); break;
        }
        switch(esr&0x3) {
            case 0: uart_puts(" at level 0"); break;
            case 1: uart_puts(" at level 1"); break;
            case 2: uart_puts(" at level 2"); break;
            case 3: uart_puts(" at level 3"); break;
        }
    }
    __asm__ volatile (
        "mov    x9, %0\n\t"
        :: "r"(ret)
    );

    return;
}
```

Nesse código, as exceções são roteadas para escreverem na tela seu tipo de interrupção e exceção. No caso de uma exceção do tipo syscall, trocamos a função `uart_puts` para `syscall_example(x8, x0)` (cujo retorno é guardado na variável `ret` para mais tarde ser retornado como resultado da syscall). A função `syscall_example` é uma implementação simples de syscall que multiplica o argumento recebido no registrador `x0` por 2:

```
unsigned long syscall_example(unsigned long code, unsigned long arg) {
    if(code == 1){
        uart_puts("Syscall Arg: ");
        uart_hex(arg);
        return arg*2;
    } else {
        return -1;
    }
}
```

A partir da syscall implementada, o próximo passo seria o desenvolvimento de um exploit conforme definido nos objetivos. Todavia, isso não foi alcançado pela equipe.

Referências

[1] ARM Holdings. Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/documentation/ddi0487/ea, 2019. [Online; accessed 29-November-2021].

[2] ARM Holdings. Arm cortex-a series programmer's guide for armv8-a - system calls. https://developer.arm.com/documentation/den0024/a/AArch64-Exception-Handling/Synchronous-and-asynchronous-exceptions/System-calls, 2019. [Online; accessed 29-November-2021].

[3] ARM Holdings. Learn the architecture: Aarch64 memory management - address spaces in aarch64. https://developer.arm.com/documentation/101811/0101/Address-spaces-in-AArch64, 2019. [Online; accessed 29-November-2021].

[4] ARM Holdings. Aarch64 exception and interrupt handling. https://developer.arm.com/documentation/100933/0100/Synchronous-and-asynchronous-exceptions, 2021. [Online; accessed 05-December-2021].

[5] Tanjila Farah et. al. Study of race condition: A privilege escalation vulnerability. In *Proceedings of the 21st World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2017*, 2017.

[6] Paolo Zaino. Operating systems: System calls (part i). https://paolozaino.wordpress.com/2013/05/22/system-calls-part-i/, 2013. [Online; accessed 29-November-2021].

[7] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Elsevier, 2011.

[8] Peter Nelson . 'hello world!' in arm64 assembly. https://peterdn.com/post/2019/02/03/hello-world-in-arm-assembly/, 2019. [Online; accessed 04-December-2021].

[9] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the shadows: Empowering arm for stealthy virtual machine introspection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 407–417, 2018.

[10] Jakob H. Weisblat. Improving security at the system-call boundary in a type-safe operating system. https://shorturl.at/oDENV, 2018. [Online; accessed 29-November-2021].

[11] Wikipedia contributors. System call — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=System_call&oldid=1056437221, 2021. [Online; accessed 29-November-2021].

[12] Zoltan Baldaszti. Bare metal programming on raspberry pi 3. `https://github.com/bztsrc/raspi3-tutorial`, 2021. [Online; accessed 08-December-2021].