Assignment 4
SoC Security Analysis
Due date: 11/27/2023 (Monday) at 11:59 pm, CST

Description

In this assignment, your objective is to play the role of an SoC hacker with the goal of exploiting the bugs in the SoC. You are provided with the details of various security vulnerabilities and clues about the possible exploit. You will be building the exploits and testing them through simulation. This exercise allows you to understand the severity and impact of various vulnerabilities and how they compromise the security of the design.

For this assignment, you are expected to

i. You will be given a buggy SoC design for which you have to design exploits and execute them through simulation on the buggy design.

ii. The design has numerous bugs, but this manual provides details for the three bugs B1, B2, and B3 which need to be exploited. You will build five exploits in total:
   a. Exploits E1, E2, and E3 for the three bugs B1, B2, and B3.
   b. Exploit E4 is more generic using B1, B2, and B3.
   c. Exploit E5 is for bonus points.

iii. Appendix B lists the details of the three bugs B1, B2, and B3 and the clues to build the exploits E1 to E5.

iv. For each of the exploits you build, you will be leaking/recovering a flag. The flag could be a password, protected data, etc.
   a. The exploit will be written in the form of a C program. All c program templates can be found in the openpiton/software folder
   b. The SoC setup provided to you comes with templates for the C program files to run each of the five exploits E1 to E5.
   c. You need to complete the exploit code in these template program files and submit the modified program files for each exploit. You can search for the keyword ‘EDIT’ in the C program files which indicates the locations where you need to modify the code.
   d. Your program files will be tested on the SoC design provided to you.
   e. You cannot directly hardcode the information about the flag in the program file. Also, you cannot change the kernel code since the attacker is an unprivileged user.
   f. You need to add comments to the program files. **Exploits without comments will not be graded. Check an example exploit in Appendix C which has a program file with comments inserted.**
   g. For each exploit, you need to include the following in your report along with the corresponding program file:

i. **Explicitly mention** if you were able to successfully run the exploit and report the flag. If the exploit did not run successfully, try to explain the issue (You will get partial points for the explanation in case your exploit did not work).
ii. A screenshot of the terminal at the beginning of the simulation of the program file showing the command you are using to start the simulation.
iii. A screenshot of the terminal at the end of the simulation.
iv. A screenshot of the `fake_uart.log` file.

h. Also, please include your feedback about this assignment in your report. Any feedback that can improve the lab for upcoming semesters is greatly appreciated.
   i. You are not required to give feedback for each exploit. You are required to give feedback on the Lab in its entirety.

Resources required for this assignment

1. Buggy design:

   The SoC design is an Openpiton SoC with Ariane (aka cva6) cores in it. It follows the RISC-V ISA. The design consists of a lot of design files and has multiple bugs inserted in it. You **need not** go through each and every file. Appendix B lists all the relevant files required to understand the bugs and create exploits. It is sufficient to only check those files.

2. Appendices from this manual:
   a. Appendix A: Setup instructions to simulate the buggy design.
   b. Appendix B: Bug and exploit details.
   c. Appendix C: Example program file with comments.
   d. Appendix D: Commonly made mistakes.

Due date and Deliverables

The due date to submit this assignment is 11/27/2023.

All the C program files you modified to create the exploits should be submitted along with a pdf report as a zipped file with <socLab_lastname_firstname>.zip as the file name in the Canvas.

TA office hours

- **Announced via Canvas.**

Rubrics

Maximum points: **100 (+20 bonus point)**

1. Report: 20 points.
2. Program files that can successfully run exploits E1, E2, and E3: 10 points each.

3. Program files that can successfully run exploit E4: 40 points.
4. Feedback on the assignment: 10 points.
5. Program files that can successfully run exploit E5: 20 bonus points.

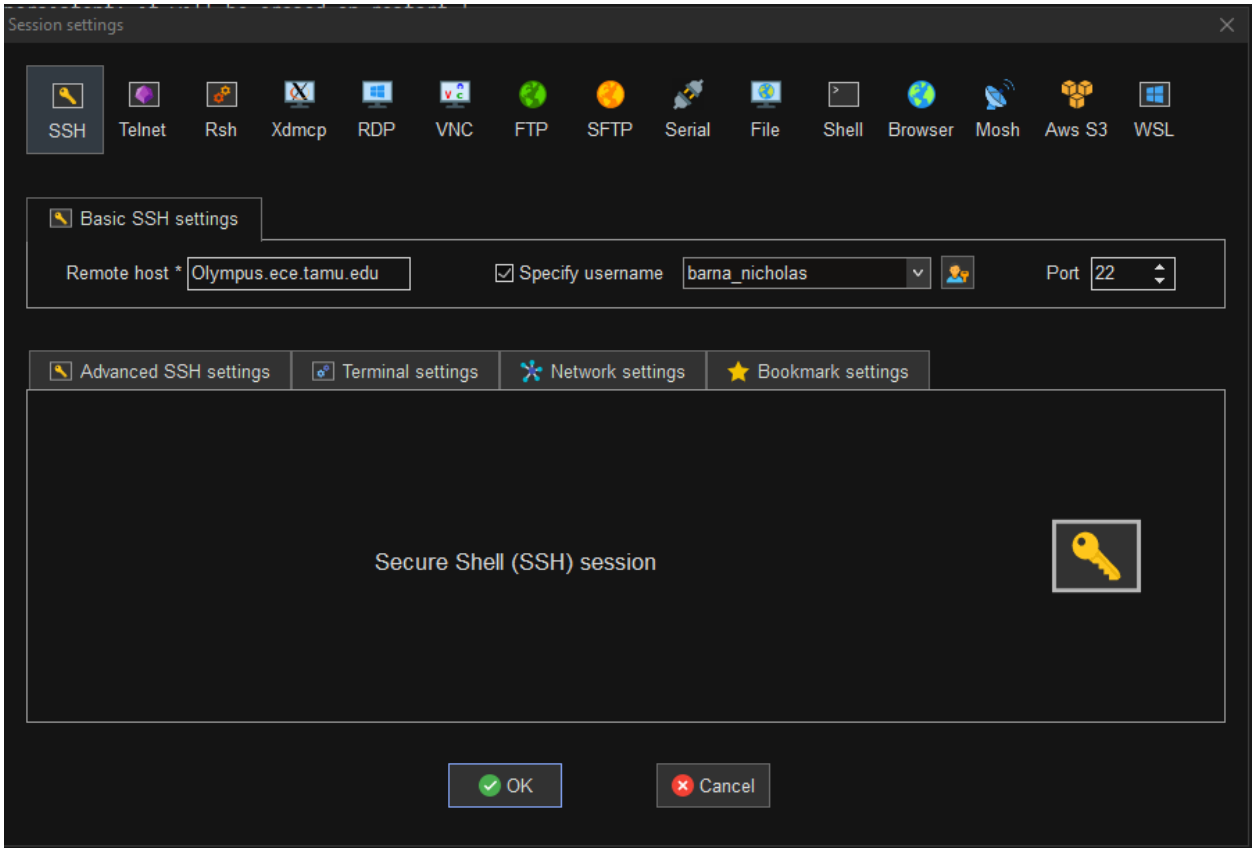
***If you do not leave comments, show correct screen shots, **explicitly** mention in the report that the exploit worked/did not work, or provide feedback we **will** take away points. It does **not** matter if your c-code was correct.

Appendix: A

Setup for the assignment:

You will be doing the assignment on the TAMU Olympus server. The setup documents are provided in the form of Gitlab repository.

1. Create an account in Gitlab if you don't already have one (https://gitlab.com/users/sign_in).
2. Login to the TAMU Olympus server:
    a. Download the Mobaxterm software.
    b. ssh into the Olympus server with this command: `ssh -Y <TAMU username>@Olympus.ece.tamu.edu.`
    Enter your TAMU password when prompted.
    c. Or set up a session with Olympus as shown below.



3. Load into the ecen 426 session with the following command
    a. load-ecen-426
4. Ensure you are in the home directory with the 'cd ~' command
5. Now create a new folder called 'assignment4'
    a. mkdir assignment4
6. Now cd into this folder
    a. cd assignment4

7. Copy the lab material into the assignment 4 folder and build it (This only needs to be done once on setup see “Simulating SoC” section for when you reenter your built environment):
    a. `cp -r /mnt/lab_files/ECEN426/assignments/soc_lab/soc-security-analysis-lab/openpiton/ .`
    b. `cd openpiton`
    c. `source piton/ariane_setup.sh`

```
[barna_nicholas]@n01-zeus ~/assignment4/openpiton> (19:28:37 11/13/23)
:: source piton/ariane_setup.sh

------------------------------------------------------------------------
openpiton/ariane path setup
------------------------------------------------------------------------

make sure that you source this script in a bash shell in the root folder of OpenPiton
not in bash (/usr/bin/bash), aborting

------------------------------------------------------------------------
setup complete. do not forget to run the following script
if you run the setup for the first time: ./piton/ariane_build_tools.sh
------------------------------------------------------------------------


[barna_nicholas]@n01-zeus ~/assignment4/openpiton> (19:29:11 11/13/23)
::
```

    d. `./piton/ariane_build_tools.sh (This will take around 5 minutes)`

```
/usr/bin/ld: skipping incompatible /lib/libm.so when searching for -lm
/usr/bin/ld: skipping incompatible /lib/libc.so when searching for -lc
make[2]: Leaving directory `/home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane/tmp/verilator-4.014/src/obj_opt'
make[1]: Leaving directory `/home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane/tmp/verilator-4.014/src'
Build complete!

Now type 'make test' to test.


------------------------------------------------------------------------
build complete
------------------------------------------------------------------------
```

    e. `sims -sys=manycore -x_tiles=2 -y_tiles=1 -vlt_build -ariane`
        i. You should see this once the setup is done:

```
/openpiton/piton/tools/verilator   -c -o verilated_dpi.o /home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane//tmp/verilator-4.014//include/verilated_dpi.cpp
/usr/bin/perl /home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane//tmp/verilator-4.014//bin/verilator_includer -DVL_INCLUDE_OPT=include Vcmp_top.cpp Vcmp_top___024u
nit.cpp Vcmp_top_tile__T2.cpp Vcmp_top_io_xbar_top_wrap.cpp Vcmp_top_dynamic_node_top_wrap.cpp Vcmp_top_noc_axilite_bridge__S8_SB1.cpp Vcmp_top_io_xbar_input_top_4.cpp Vcmp_top_io_xbar_output_t
op.cpp Vcmp_top_dynamic_output_top.cpp Vcmp_top_AXI_BUS__AC1_A40_AB40_AD1.cpp Vcmp_top_AXI_BUS__A40_AB40_AC1_AD1.cpp Vcmp_top_one_round.cpp Vcmp_top_pmp_entry__X40_P36.cpp Vcmp_top_sram__D100_N
80.cpp Vcmp_top_T.cpp Vcmp_top_aes_sbox.cpp Vcmp_top_S.cpp > Vcmp_top__ALLcls.cpp
/usr/bin/perl /home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane//tmp/verilator-4.014//bin/verilator_includer -DVL_INCLUDE_OPT=include Vcmp_top__Dpi.cpp Vcmp_top__
Syms.cpp > Vcmp_top__ALLsup.cpp
g++  -I.  -MMD -I/home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane//tmp/verilator-4.014//include -I/home/grads/b/barna_nicholas/assignment4/openpiton/piton/design
/chip/tile/ariane//tmp/verilator-4.014//include/vltstd -DVL_PRINTF=printf -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=0 -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-variable -Wno-unused-pa
rameter -Wno-unused-variable -Wno-shadow     -DVERILATOR -DPITON_DPI -lstdc++ -I/home/grads/b/barna_nicholas/assignment4/openpiton/piton/tools/pli/iop -I/home/grads/b/barna_nicholas/assignment4
/openpiton/piton/tools/verilator   -c -o Vcmp_top__ALLcls.o Vcmp_top__ALLcls.cpp
g++  -I.  -MMD -I/home/grads/b/barna_nicholas/assignment4/openpiton/piton/design/chip/tile/ariane//tmp/verilator-4.014//include -I/home/grads/b/barna_nicholas/assignment4/openpiton/piton/design
/chip/tile/ariane//tmp/verilator-4.014//include/vltstd -DVL_PRINTF=printf -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=0 -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-variable -Wno-unused-pa
rameter -Wno-unused-variable -Wno-shadow     -DVERILATOR -DPITON_DPI -lstdc++ -I/home/grads/b/barna_nicholas/assignment4/openpiton/piton/tools/pli/iop -I/home/grads/b/barna_nicholas/assignment4
/openpiton/piton/tools/verilator   -c -o Vcmp_top__ALLsup.o Vcmp_top__ALLsup.cpp
      Archiving Vcmp_top__ALL.a ...
ar r Vcmp_top__ALL.a Vcmp_top__ALLcls.o Vcmp_top__ALLsup.o
ar: creating Vcmp_top__ALL.a
ranlib Vcmp_top__ALL.a
g++    b_ary.o bw_lib.o cpx.o iob.o iob_main.o pcx.o my_top.o verilated.o verilated_dpi.o Vcmp_top__ALL.a    -o Vcmp_top -lm -lstdc++
make: Leaving directory `/home/grads/b/barna_nicholas/assignment4/openpiton/build/manycore/rel-0.1/obj_dir'
sims: stop_time Mon Nov 13 19:48:36 CST 2023
```

8. Simulating the SoC (**You need to do this every time you want to run an exploit**):
    a. If you haven't already you need to 'load-ecen-426'.
    b. Setup the environment (do this once every time you open a new terminal window or connect to the Olympus server):
        i. `cd openpiton`

```
[barna_nicholas]@n01-zeus ↪ (19:28:31 11/13/23)
:: cd assignment4/openpiton/

[barna_nicholas]@n01-zeus ~/assignment4/openpiton> (19:28:37 11/13/23)
::
```

        ii. `source piton/ariane_setup.sh`

```
[barna_nicholas]@n01-zeus ~/assignment4/openpiton> (19:28:37 11/13/23)
:: source piton/ariane_setup.sh

------------------------------------------------------------------------
openpiton/ariane path setup
------------------------------------------------------------------------

make sure that you source this script in a bash shell in the root folder of OpenPiton
not in bash (/usr/bin/bash), aborting

------------------------------------------------------------------------
setup complete. do not forget to run the following script
if you run the setup for the first time: ./piton/ariane_build_tools.sh
------------------------------------------------------------------------


[barna_nicholas]@n01-zeus ~/assignment4/openpiton> (19:29:11 11/13/23)
::
```

    c. Simulate:
        i. `cd build`
        ii. `./make_run_user_with_pk.sh <name of file to execute without the .c extension>`
        Ex: `./make_run_user_with_pk.sh helloworld`
        iii. To check the progress of the script above utilize the command 'tail -f fake_uart.log' in a separate tab on mobax tab

1. *** You do **NOT** need to let the simulation run until completion. As soon as the fake_uart.log shows the flag you can kill the sim with a 'ctrl+c'

iv. `The result will look something like this`

```
:: tail -f fake_uart.log
reading dtb for hart
    Setting the FUSE data
    Setting the register locks
entering supervisor mode
calling mret
copying to 00012000 from 00002000 with size 00000780
copying to 00010000 from 00000000 with size 0000182f
Starting!
Welcome to SoC Security Analysis Lab!
Done!
```

d. The setup comes with an example `helloworld` program and the template files in which you will create the exploits in `openpiton/software` folder.

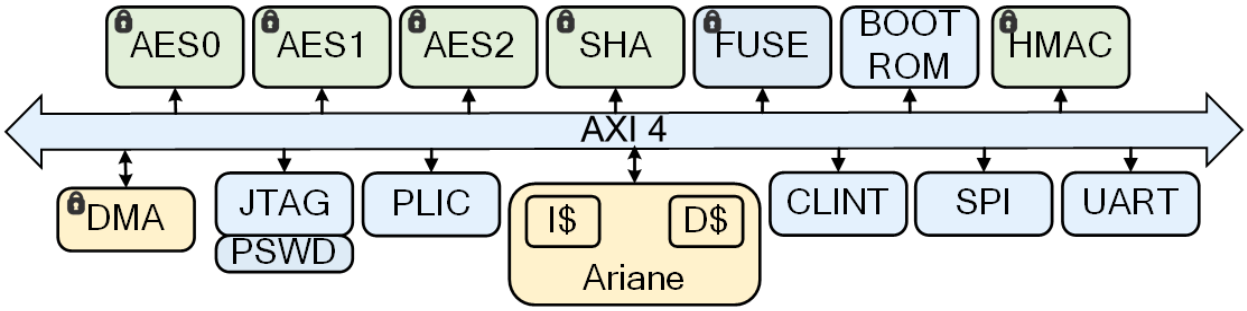
e. Note that the simulations take time to run. Depending on your program, the simulation can take upwards of 10 min. The `printf` statements in the program file are output to `fake_uart.log` file in the `openpiton/build` folder.

f. You might have to force-stop the simulation once you get all the print commands you are expecting by pressing `Ctrl + C` in the terminal where the simulation is happening.

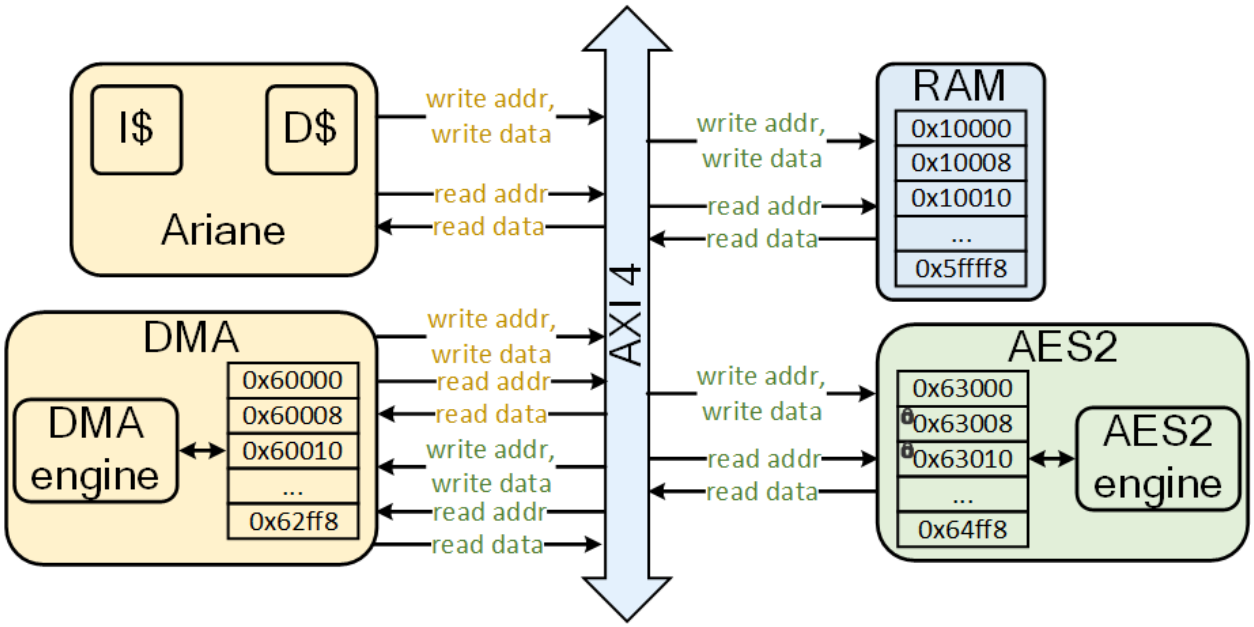
Appendix: B

This section describes the bugs B1 to B3 and gives a high-level overview of how the exploit should be built. In addition, a lot of low-level details about these exploits are provided in the form of comments to the template exploit programs provided along with the buggy design in the `openpiton/build` folder. We strongly recommend you check these comments along with the information provided in this section for greater understanding.

A simplified version of the SoC:



A simplified version of working of AXI bus:



Note: Addresses shown in this figure are dummy addresses. You don't have to figure out the actual addresses since they are all defined in the header files provided to you.

Openpiton SoC microarchitectural specification document can be found here. This document is not required to complete this lab and is provided only for your reference.

Using the DMA peripheral:

We will using Direct Memory Access (DMA) peripheral in all our exploits. The setup comes with a sample C program demonstrating the usage of the DMA peripheral. This program shows how to use the two kernel functions of DMA (`dma_transfer_from_perif` and `dma_transfer_to_perif`) to transfer data from and to peripherals.
Please go through the comments in this file that explain how to provide the correct arguments to these kernel functions. You can run this program with the simulation command:
`./make_run_user_with_pk.sh using_DMA`

This is the output you get:

```
$ tail -f fake_uart.log
uart working
reading dtb for hart
    Setting the FUSE data
    Setting the register locks
entering supervisor mode
calling mret
copying to 00012000 from 00002000 with size 00000780
copying to 00010000 from 00000000 with size 000018c0
This code shows the usage of DMA
Data read from SHA256 = 99991111 77773333 44441111 66663333
```

Bug B1 for Exploit E1: The plain text data used by the AES2 crypto peripheral is not erased after the encryption is done.

Code location: openpiton/piton/design/chip/tile/ariane/src/aes2/aes2_wrapper.sv

Code details:
It can be seen in the aes2_wrapper.sv that the data is written to the plain text registers (p_c register) in the first case statement (lines 100 to 108) but is never cleared and hence available to read in the second case statement (lines 159 to 166).

Exploit construction:
Because of the bug, an attacker can recover the plain text used for encryption by a previous user even when the data is completely removed from the software stack by directly accessing it from the AES2 registers. We will build an exploit for this bug where first, a user will use AES2 to encrypt his password. Then, we will run the attacker code that uses DMA to copy the data from the AES2 `p_c` registers to leak the user's password. More details are provided in the `openpiton/software/exploit1.c` program file in the form of comments. You have to edit this file to complete this exploit.

Bug B2 for Exploit E2: The AES2 crypto peripheral is leaking the Key0 value since the `default` case statement is not covering all the cases.

Code location: openpiton/piton/design/chip/tile/ariane/src/aes2/aes2_wrapper.sv, the case statement in `always @(*) block` (lines 154 to 181).

Code details:

It can be seen that, first, the read data (`rdata` signal) is set to `key0[address[8:3]]` (line 155). This shouldn't have been a problem since the case statement (lines 156 to 180) will be overwriting the `rdata` signal. But, in the specific case where the address does not match with any of the case values, we go to the `default` case (lines 177 to 180). In the default case, if the `start` signal is not '1', then, the case statement does not write any value to the read data signal leaving the key0 value from line 155 in it.

Exploit construction:
Because of the bug, an attacker can set the read address such that it does not match with any of the case statement values. The control will then go to the default case. The `start` signal is only set to '1' when an encryption or decryption operation is in progress. If not, the `start` will be set to '0'. In this case, the attacker can pick values for the read address to get all the key0 data. More details are provided in the `openpiton/software/exploit2.c` program file in the form of comments. You have to edit this file to complete this exploit.

Note: The AES2 key in hardware is stored in FUSE memory of hardware (`openpiton/piton/design/chip/tile/ariane/src/fuse_mem/fuse_mem.sv` at line 71 to 74). FUSE memory is a protected memory where sensitive information required for the hardware is stored. Attacker cannot directly access the data in FUSE memory. Through this exploit, we are able to leak the Key0 of AES2 without accessing FUSE memory by exploiting AES2 peripheral which keeps a of copy of this key to run the encryption.

Bug B3 for Exploit E3: The kernel function for AES2 crypto peripheral does not check if the pointer to plain text is coming from the user space or kernel space.

Code location: openpiton/pk/pk/syscall.c, the `sys_aes2_start_encry` kernel function at lines 775 to 796.

Code details:
It can be seen from line 775 of the `sys_aes2_start_encry` function that the pointer for the initial vector (`st`) is being checked if it is pointing to user range but not the plain text pointer (`pt`).

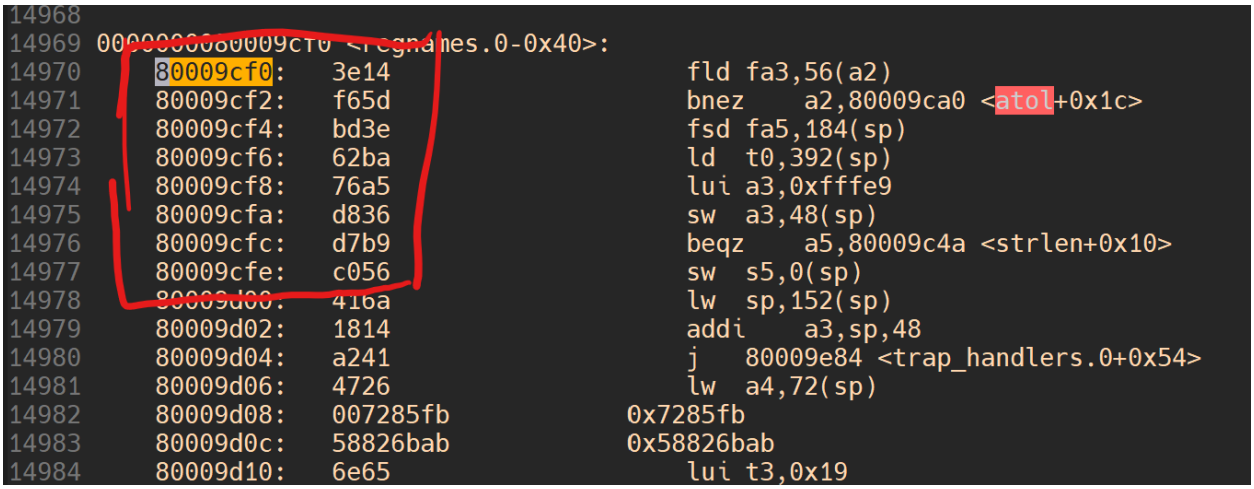
Exploit construction:
Because of the bug, an attacker can set the plain text pointer to a kernel memory space and run AES2 encryption on it. This will result in AES2 encrypting the data in the kernel memory space and output the cipher text.
Since we can run decryption also using AES2, we can decrypt this cipher text data to leak the kernel data.
The flag for this exploit is a data in a kernel variable (line 194 of `openpiton/pk/pk/pk.c`). Here, the kernel function `kernel_func1` is doing some operation on the data in line 194. Our goal is to leak this data.

```
191 // some kernel function that operates on data
192 void __attribute__((optimize("O0"))) kernel_func1() {
193
194     uint32_t  volatile data[4]  = {0xf65d3e14, 0x62babd3e, 0xd83676a5, 0xc056d7b9};
195
196     // some kernel functions here
197
198     // remove the data from the stack
199     memset((uint32_t *)data, 0, sizeof(data));
200
201     return;
202 }
```

We will be building this exploit by first figuring out the kernel memory address of the data we want to leak. This can be done by manually checking the `openpiton/build/diag.dump` file which disassembles the executables you simulate. In this dump file, search for the `kernel_func1` function. This function will load the data from a `atol` location which is the address of the kernel memory data. For example, if this is what you have in your `diag.dump` file:

```
206
207 00000000800002e8 <kernel_func1>:
208     800002e8:   1101                    addi    sp,sp,-32
209     800002ea:   ec06                    sd  ra,24(sp)
210     800002ec:   e822                    sd  s0,16(sp)
211     800002ee:   1000                    addi    s0,sp,32
212     800002f0:   0000a797            auipc   a5,0xa
213     800002f4:   a0078793            addi    a5,a5,-1536 # 80009cf0 <atol+0x6c>
214     800002f8:   6398                    ld  a4,0(a5)
215     800002fa:   fee43023            sd  a4,-32(s0)
216     800002fe:   679c                    ld  a5,8(a5)
217     80000300:   fef43423            sd  a5,-24(s0)
218     80000304:   fe040793            addi    a5,s0,-32
219     80000308:   4641                    li  a2,16
220     8000030a:   4581                    li  a1,0
221     8000030c:   853e                    mv  a0,a5
222     8000030e:   0e7090ef            jal ra,80009bf4 <memset>
223     80000312:   0001                    nop
224     80000314:   60e2                    ld  ra,24(sp)
225     80000316:   6442                    ld  s0,16(sp)
226     80000318:   6105                    addi    sp,sp,32
227     8000031a:   8082                    ret
228
```

then, the address to the kernel data is 0x80009cf0 (line 213). So, in the `exploit3.c` file, you will have to change line 51 to `uint32_t *pt = 0x80009cf0;`. Note that this address will keep changing based on the changes you make to the C code, so, make sure you check this address from your `diag.dump` file. You can crosscheck that this pointer is correct by searching for the address location 80009cf0 in the `diag.dump` file and you will see the kernel data (from line 194 of `openpiton/pk/pk/pk.c`) is same as the data in this location as shown below:



We will set the plain text pointer to this address in our exploit code, run the AES2 encryption and decrypt the encrypted data to obtain the kernel data. More details are provided in the `openpiton/software/exploit3.c` program file in the form of comments. You have to edit this file to complete this exploit.

Exploit E4:

Code location: openpiton/piton/design/chip/tile/ariane/src/aes0/aes0_wrapper.sv

Scenario:
The objective of this exploit is to leak the JTAG password used by the kernel function `kernel_func2` (lines 206 to 238 of `openpiton/pk/pk/pk.c`). This password is fetched from the hardware (lines 211 to 217) instead of being loaded from kernel memory space like kernel_func1 did in exploit E3. Hence we cannot use the method from exploit E3.

- Thus, the JTAG password has to be leaked indirectly.
- The `kernel_func2` function encrypts this JTAG password using AES0 crypto peripheral. This peripheral can only be accessed by the kernel code.
- Although the JTAG password is sent to AES0 as plain text for encryption, we cannot read this plain text directly as we did in exploit E1 because AES0 does not allow the plain text from hardware to be read directly once the encryption is done.
- But, AES0 has a bug that allows us to read the encrypted (ciphered) text.
- Thus, using an exploit similar to E1, we can get the encrypted text for the JTAG password. This will give us the encrypted version of flag encrypted with key0 of AES0.
- Still, we cannot recover the plain text since user programs cannot run decryption on AES0.
- But, we know that AES0 is a 192-bit CTR mode AES which takes 128-bit plain text and an initial vector to generate 128-bit cipher text.
- So, if we can somehow get the initial vector and AES key used for encrypting the JTAG password, we can use them along with the cipher text to run an offline AES engine and recover the JTAG password!

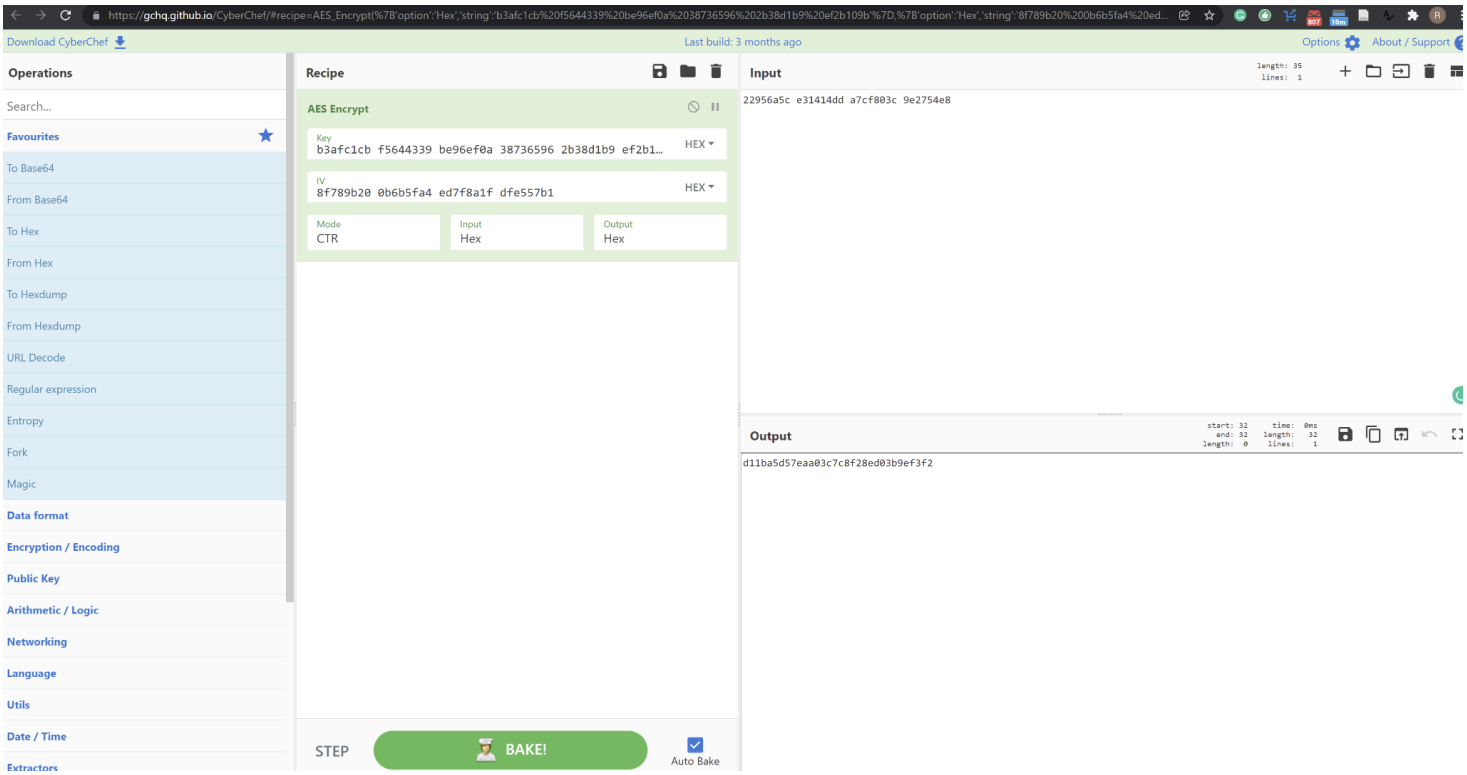
Exploit construction:

Based on the scenario explained above, we will be using the following exploit flow:

1. We will use an exploit similar to E3 to get the initial vector used by `kernel_func2`. This is possible since `kernel_func2` loads the initial vector from the kernel memory. You will need the address of the kernel data like the one you used for exploit 3. This time you will look for the `kernel_func2` function in `openpiton/build/diag.dump` file. You can use the method described in exploit3 to crosscheck if you got the correct address.
   You should be able to get the initial vector at line 220 of `openpiton/pk/pk/pk.c` with this step.
2. We will use an exploit similar to E2 to leak the Key0 of AES0 (AES0 also has the same bug as AES2 which leaks the key!).
   You should be able to get the copy of key0 stored in AES0 peripheral originating from lines 117 to 122 of `openpiton/piton/design/chip/tile/ariane/src/fuse_mem/fuse_mem.sv` with this step.
3. We will use an exploit similar to E1 to get the encrypted (cipher) text from AES0. This is the encrypted JTAG password.
   (Here is the cipher text you should be getting: 775bfc8a 11fd7bab df06e8a2 8c323ad2)
4. We now have the cipher text, the initial vector, and the key. We know the mode of AES crypto engine from the specification document of the SoC. Thus, we can use an offline AES engine (like this) to recover the JTAG password (Check the screenshot below. Make sure you sent the Mode as **CTR**, input and output formats as **HEX.** The values used in the image are dummy, replace them with the key, initial vector and cipher text you got. The cipher text goes into the Input field).

Partial credit will be given for exploit E4 if only a few of the above steps are completed.

The flag we are recovering with this exploit is the JTAG password present in line 21 of `openpiton/piton/design/chip/tile/ariane/src/fuse_mem/fuse_m`



`em.sv`.

Exploit E5 (bonus points):

You will be targeting the HMAC crypto engine for this exploit. HMAC has bugs similar to the B1 and B2 bugs present in AES2.

1. But the information about the bugs or exploits is not provided to you.
2. You have to check the HMAC file (`hmac_wrapper.sv`) to identify the bugs and run the exploit.
3. You get 10 points each for running exploits similar to E1 and E2.
   The flags you will be leaking are plain text data (stored in `data` register in `hmac_wrapper.sv`) and key0 of HMAC. In case of the `data` register, it is fine even

if you leak one 32-bit word (i.e., data[0]) and dont have to leak all the 16 32-bit words.

Code location: openpiton/piton/design/chip/tile/ariane/src/hmac/hmac_wrapper.sv

Appendix: C

Example program with comments:

The example program below shows a C program with lots of comments. You do not need to write this many comments in your submission, but you should have enough comments to explain your code.

```
////////////////////////////////////////////////////////////////////////////////////////////////
//
//  - This file is only to help the students with basic functions to interact with
//    the DMA in the SoC
//  - You can use copy and paste any part of the code from this file in your assignment. But
//    this file itself cannot be submitted as exploit code.
//
//  - Command to run this file: ./make_run_user_with_pk.sh using_DMA
//
////////////////////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

#include "ariane_api.h"  // includes all the API functions to access the peripherals

void main(void)
{
    printf("This code shows the usage of DMA\n");

    ////////////////////////////////////////////////////
    // Data transfer to peripheral using DMA
    ////////////////////////////////////////////////////
    // DMA transfers data in chunks of 32 bits
    // Max length supported is 64*32-bit words
    // In this case we are transfering 4 32-bit words to the first 4 data words
    // in the SHA256 peripheral

    // write down the data u want to send as a array of 64 bit integers
    uint64_t wdata[4]  = {0x99991111, 0x77773333,0x44441111, 0x66663333} ; // here we use 64-bit integer array instead
                                                                           // of 32-bit integer array bcz DMA fetches
                                                                           // one 64-bit word at a time

    // call DMA to transfer to peripheral
    // DMA transfer function declaration: dma_transfer_to_perif(uint32_t *sAddress, uint64_t dAddress, uint32_t length, int wait)
    // the function has 4 arguments
    // uint32_t*sAddress: a pointer to where the data to copy from is. The pointer should point to a array of 64 bit-integers
    // uint64_t dAddress: this tells the physical address of where the data should be copied to.
    //                    In this case, we want to copy the data to data registers of SHA.
    //                    The base address of SHA is SHA256_BASE. The 'data' registers are starting at index 1
    //                    in the case statement of SHA256 wrapper file (openpiton/piton/design/chip/tile/ariane/src/sha256/sha256_wrapper.sv).
    //                    so, the address will be SHA256_BASE + (1*8). We are multiplying the index 1 with 8 since
    //                    wrapper verilog files ignore the first 3 bits of address, i.e., to get offset of 1, we need to do 0x8
    // uint32_t length: the number of 32-bit words we want to transfer
    // int wait: this tells if the function should wait for transfer to complete or return while the transfer is ongoing. It
    //           is recommended to always set this to 1
    dma_transfer_to_perif(wdata, SHA256_BASE + (1*8), 0x4, 1);


    ////////////////////////////////////////////////////
    // Data transfer from peripheral using DMA
    ////////////////////////////////////////////////////
    // DMA transfers data in chunks of 32 bits
    // Max length supported is 64*32-bit words
./software/using_DMA.c [RO]
```

Appendix: D

Here is a list of common mistakes that students make in this lab. Make sure you do not repeat these mistakes:

- Not attempting the assignment: You will get partial credit even if the answer is incorrect. So, attempt all the problems to maximize your score.
- Not including the screenshots or including only partial screenshots or not specifying if you were able to successfully run the exploit in the report: You will lose credit if any information is missing in the report. Check "For this assignment, you are expected to" section for more details.
- Not changing the "student_name" variable: Make sure you set this variable to your name.
- Not submitting feedback for the assignment: Feedback carries 10 points.
- Printing incorrect variable (for example, printing "rdata" variable when you are supposed to print the variable "rdata2").
- Make sure you use "pt" and "ct" correctly in E3.
- Using incorrect kernel data location or offset when doing DMA read/write operations in exploit 4.
- Not including AES engine screenshot for exploit 4.
- Not writing comments in the C file.
    - You must write your own comments. This must be different from the existing ones.
- Not including the exploit C files you modified.
- Running the simulation for more time than needed: This does not affect your grade but will waste your time when doing the assignment. Once you get the output you want in the `fake_uart.log` file, you dont need to keep waiting for the simulation to end by itself. You can just press Ctrl + C in the terminal to stop the simulation. But make sure that you want till the required output is printed in the `fake_uart.log` file.