CPE 325: Intro to Embedded Computer System

**Lab010**

**Software Reverse Engineering**

**Submitted by**: Nolan Anderson

**Date of Experiment**: 11/16/2020

**Report Deadline**: 11/16/2020

# Introduction

This lab covers how to use different utilities to reverse engineer different types of software. Here we are reversing a .out and .txt file to see what these programs do. The purpose of the .out file is to reverse the usernames and passwords based on command line executables such as readelf.exe which summarizes the content in the .out file. The second .txt file is used to reverse hex data and figure out how the code works based on using similar utilities such as Naken.

# Theory

**Theory Discussed in the lab demo video.**

# Results & Observation

**--------------------------------------------------------------- 1 ---------------------------------------------------------------**

- Do you think Figure 14 belongs to the same microcontroller as in Figure 13?

i. I would say that these devices are different simply due to the starting addresses of the different section headers. For example, figure 13 has the .data section at address 2400 and figure 14 the same section starts at address 1100.

- Where do you find this information about register addresses?

i. To find information about register addresses, you need to look up the msp430 .cmd file under:

**D:\CCS\CCS10.1.0.00010\_win64\ccs\ccs\_base\msp430\include\lnk\_msp430f5529.cmd** <- This will vary, find where you CCS is installed. Then you need to ctrl+f to find the address. Note that the device you are on the registers will change.

- Similarly, the next instruction ANDs content of 0x0223 with 127. Can you guess what this statement does?

i. This command **ands** the last bits of 0x0223 (most likely the port of the LED) with #127 to most likely turn the LED on port 0x0223 off. In the .cmd file 0x0223 is PBOUT\_H.

- The next instruction moves 0x4432 to R0 (PC). The means the PC points to 0x4432. Where command is executed after this?

- The command **xor.b #1, &0x0202 ;0xff80** will be executed. This toggles the other LED. So this code when one LED turns off, the other one is toggled. 0x0202 is PAOUT.

**--------------------------------------------------------------- 2 ---------------------------------------------------------------**

|  |
| --- |
|  |

Username: Abraham\_uname | Password: lincoln\_pass

Command to run: .\**msp430-elf-objdump.exe -s crack\_me.out**

* This command will output all of the different sections and their data in strings and in hex along with their addresses.

**--------------------------------------------------------------- 3 ---------------------------------------------------------------**

**a**. What is the magic number used? [1 pts] **b**. What is the class of this .out file? [1 pts] **c**. What machine was this file built for? [1 pts] **d**. What is the size of the header? [1 pts] **e**. How many section headers are there? [6 pts]

|  |  |
| --- | --- |
| * Run with **.\msp430-elf-readelf.exe -h crack\_me.out** in Windows Powershell.  1. 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 2. ELF32 3. Texas Instruments msp430 microcontroller 4. 52 (bytes) 5. 99 Section headers. 6. Command line source:  |  | | --- | |  | |

**--------------------------------------------------------------- 4 ---------------------------------------------------------------**

1. **Flasher output [10 pts]:**

|  |
| --- |
| **PS D:\CCS\CCS10.1.0.00010\_win64\ccs\tools\compiler\msp430-gcc-9.2.0.50\_win64\bin> D:\CCS\Flasher\MSP430Flasher.exe -n MSP430F5529 -w reverse\_me.txt -v -z [Vcc]**  **\* -----/|-------------------------------------------------------------------- \***  **\* / |\_\_ \***  **\* /\_ / MSP Flasher v1.3.20 \***  **\* | / \***  **\* -----|/-------------------------------------------------------------------- \***  **\***  **\* Evaluating triggers...done**  **\* Checking for available FET debuggers:**  **\* Found USB FET @ COM3 <- Selected**  **\* Initializing interface @ COM3...done**  **\* Checking firmware compatibility:**  **\* FET firmware is up to date.**  **\* Reading FW version...**  **\* Debugger does not support target voltages other than 3000 mV!**  **\* Setting VCC to 3000 mV...done**  **\* Accessing device...done**  **\* Reading device information...done**  **\* Loading file into device...done**  **\* Verifying memory (reverse\_me.txt)...done**  **\***  **\* ----------------------------------------------------------------------------**  **\* Arguments : -n MSP430F5529 -w reverse\_me.txt -v -z [Vcc]**  **\* ----------------------------------------------------------------------------**  **\* Driver : loaded**  **\* Dll Version : 31400000**  **\* FwVersion : 31200000**  **\* Interface : TIUSB**  **\* HwVersion : E 3.0**  **\* JTAG Mode : AUTO**  **\* Device : MSP430F5529**  **\* EEM : Level 7, ClockCntrl 2**  **\* Erase Mode : ERASE\_ALL**  **\* Prog.File : reverse\_me.txt**  **\* Verified : TRUE**  **\* BSL Unlock : FALSE**  **\* InfoA Access: FALSE**  **\* VCC ON : 3000 mV**  **\* ----------------------------------------------------------------------------**  **\* Starting target code execution...done**  **\* Disconnecting from device...done**  **\***  **\* ----------------------------------------------------------------------------**  **\* Driver : closed (No error)**  **\* ----------------------------------------------------------------------------**  **\*/**  **PS D:\CCS\CCS10.1.0.00010\_win64\ccs\tools\compiler\msp430-gcc-9.2.0.50\_win64\bin>** |

1. **Guess from observation on the board what the program does? [5 pts]**
2. This code blinks both LED’s infinitely when you press switch 1 (Port 2.1 that is).
3. **Using the naken utility and the steps shown in Section 5.2 of the tutorial, reverse engineer the hex file to assembly code. [5 pts]**

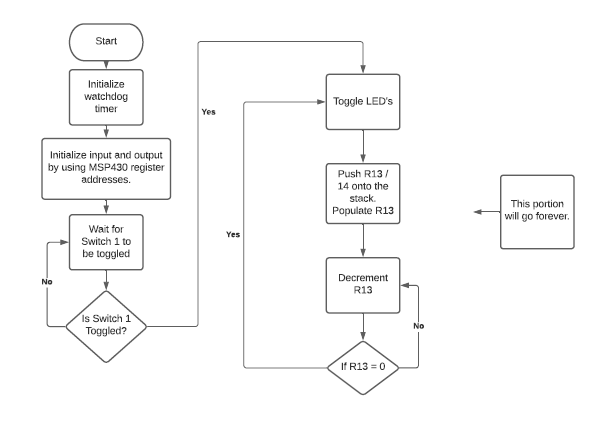
|  |
| --- |
| PS D:\CCS\CCS10.1.0.00010\_win64\ccs\tools\compiler\msp430-gcc-9.2.0.50\_win64\bin> D:\CCS\Flasher\MSP430Flasher.exe -r [reverse\_me.txt, MAIN]  \* -----/|-------------------------------------------------------------------- \*  \* / |\_\_ \*  \* /\_ / MSP Flasher v1.3.20 \*  \* | / \*  \* -----|/-------------------------------------------------------------------- \*  \*  \* Evaluating triggers...done  \* Checking for available FET debuggers:  \* Found USB FET @ COM3 <- Selected  \* Initializing interface @ COM3...done  \* Checking firmware compatibility:  \* FET firmware is up to date.  \* Reading FW version...done  \* Setting VCC to 3000 mV...done  \* Accessing device...done  \* Reading device information...done  \* Dumping memory from MAIN into reverse\_me.txt...done  \*  \* ----------------------------------------------------------------------------  \* Arguments : -r [reverse\_me.txt MAIN]  \* ----------------------------------------------------------------------------  \* Driver : loaded  \* Dll Version : 31400000  \* FwVersion : 31200000  \* Interface : TIUSB  \* HwVersion : E 3.0  \* JTAG Mode : AUTO  \* Device : MSP430F5529  \* EEM : Level 7, ClockCntrl 2  \* Read File : reverse\_me.txt (memory segment = MAIN)  \* VCC OFF  \* ----------------------------------------------------------------------------  \* Powering down...done  \* Disconnecting from device...done  \*  \* ----------------------------------------------------------------------------  \* Driver : closed (No error)  \* ----------------------------------------------------------------------------  \*/  PS D:\CCS\CCS10.1.0.00010\_win64\ccs\tools\compiler\msp430-gcc-9.2.0.50\_win64\bin> |

1. **Comment on each line of the assembly code generated from Q4c above to describe what each line is doing. [20 pts]**

|  |
| --- |
| **Addr Opcode Instruction Cycles**  **------- ------ ---------------------------------- ------**  **0x4400: 0x40b2 mov.w #0x5a80, &0x015c 5 // Setting the watchdog timer**  **0x4402: 0x5a80**  **0x4404: 0x015c**  **0x4406: 0xd3d2 bis.b #1, &0x0204 4 // Setting PADIR**  **0x4408: 0x0204**  **0x440a: 0xd0f2 bis.b #0x80, &0x0225 5 // Setting PBDIR\_H**  **0x440c: 0x0080**  **0x440e: 0x0225**  **0x4410: 0xc3e2 bic.b #2, &0x0204 4 // Clearing PADIR**  **0x4412: 0x0204**  **0x4414: 0xd3e2 bis.b #2, &0x0202 4 // Setting output direction P2.2 for PAOUT**  **0x4416: 0x0202**  **0x4418: 0xd3e2 bis.b #2, &0x0206 4 // Setting PAREN**  **0x441a: 0x0206**  **0x441c: 0xc3e2 bic.b #2, &0x0205 4 // Clearing PADIR\_H**  **0x441e: 0x0205**  **0x4420: 0xd3e2 bis.b #2, &0x0203 4 // Setting PAOUT\_H**  **0x4422: 0x0203**  **0x4424: 0xd3e2 bis.b #2, &0x0207 4 // Setting PAREN\_H**  **0x4426: 0x0207**  **// What's about to happen below is the auto generation waiting for an input, but it does it I think 3 times. As in it waits for input 3 times, maybe debouncing? Not sure.**  **0x4428: 0xb3e2 bit.b #2, &0x0200 4 // Anding with PAIN, a test. Most likely for the button press.**  **0x442a: 0x0200**  **0x442c: 0x23fd jne 0x4428 (offset: -6) 2 // And then if the switch has not been pressed, wait for the button to be pressed.**  **0x442e: 0x120d push.w r13 3 // If the button is pressed, push R13 onto the stack**  **0x4430: 0x403d mov.w #0x031d, r13 2 // Populate a large value**  **0x4432: 0x031d**  **0x4434: 0x831d sub.w #1, r13 1 // Sub R13**  **0x4436: 0x23fe jne 0x4434 (offset: -4) 2 // Loop until 0**  **0x4438: 0x413d pop.w r13 -- mov.w @SP+, r13 2 // Then pop R13**  **0x443a: 0x3c00 jmp 0x443c (offset: 0) 2 // And jump to 443c, which is next line? This is funny how it generates.**  **0x443c: 0xb3e2 bit.b #2, &0x0200 4 // Same thing as 0x4428. It waits for a button to be pressed or some tinput.**  **0x443e: 0x0200**  **0x4440: 0x23f3 jne 0x4428 (offset: -26) 2 // If it is not set jump to 4428 to restart (loop).**  **0x4442: 0xb3e2 bit.b #2, &0x0201 4 // Test the input high value**  **0x4444: 0x0201**  **0x4446: 0x23fd jne 0x4442 (offset: -6) 2 // If it is not set go back to 4442 ti wait for the high value to be set.**  **0x4448: 0x120d push.w r13 3 // Push r13**  **0x444a: 0x403d mov.w #0x031d, r13 2 // Populate a value**  **0x444c: 0x031d**  **0x444e: 0x831d sub.w #1, r13 1 // Sub 1**  **0x4450: 0x23fe jne 0x444e (offset: -4) 2 // Until 0, loop.**  **0x4452: 0x413d pop.w r13 -- mov.w @SP+, r13 2 // Pop R13 and push it onto the stack.**  **0x4454: 0x3c00 jmp 0x4456 (offset: 0) 2 // Jump to next line lol**  **0x4456: 0xb3e2 bit.b #2, &0x0201 4 // Test the high state of the input again**  **0x4458: 0x0201**  **0x445a: 0x23f3 jne 0x4442 (offset: -26) 2 // The following code below is a loop to toggle the LED's by decrementing**  **0x445c: 0xe3d2 xor.b #1, &0x0202 4 // the register values and then toggling when they're 0.**  **0x445e: 0x0202**  **0x4460: 0xe0f2 xor.b #0x80, &0x0223 5 // Toggle the LED**  **0x4462: 0x0080**  **0x4464: 0x0223**  **0x4466: 0x120d push.w r13 3 // Push R13 and R14 onto the stack.**  **0x4468: 0x120e push.w r14 3 //**  **0x446a: 0x403d mov.w #0x2844, r13 2 // Move 2844 into R13 to reset its value.**  **0x446c: 0x2844**  **0x446e: 0x431e mov.w #1, r14 1 // Move #1 into register R14**  **0x4470: 0x831d sub.w #1, r13 1 // Take 1 off of register R13**  **0x4472: 0x730e subc.w #0, r14 1 // Subtract with carray off of R14**  **0x4474: 0x23fd jne 0x4470 (offset: -6) 2 // If the Z flag is not set, keep decrementing.**  **0x4476: 0x930d cmp.w #0, r13 1 // Compare 0 to R13.**  **0x4478: 0x23fb jne 0x4470 (offset: -10) 2 // If the Z flag is not set, keep decrementing**  **0x447a: 0x413e pop.w r14 -- mov.w @SP+, r14 2 // If they are 0, pop R14 and R13 off of the stack**  **0x447c: 0x413d pop.w r13 -- mov.w @SP+, r13 2 // Increment the stack pointer and put it into R13 and R14 presumably to restart.**  **0x447e: 0x3c00 jmp 0x4480 (offset: 0) 2 // Jump to 4480? Weird...**  **0x4480: 0x4303 nop -- mov.w #0, CG 1 // No operation, move 0 into CG?**  **0x4482: 0x3fec jmp 0x445c (offset: -40) 2 // Jump to 445c.**  **0x4484: 0x4303 nop -- mov.w #0, CG 1 // Move 0 into CG?**  **0x4486: 0x4031 mov.w #0x4400, SP 2 // Go back to start of program.**  **0x4488: 0x4400**  **0x448a: 0x12b0 call #0x44a0 5 // Weird... Just goes to next line.**  **0x448c: 0x44a0**  **0x448e: 0x430c mov.w #0, r12 1 // Move 0 into R12**  **0x4490: 0x12b0 call #0x4400 5 // Restart program**  **0x4492: 0x4400**  **0x4494: 0x431c mov.w #1, r12 1 // Move 1 into R12**  **0x4496: 0x12b0 call #0x449a 5 // Goes to next line...**  **0x4498: 0x449a**  **// Maybe what's going on here is the program is just infinite looping as an interrupt. It's a little obscure.**  **0x449a: 0x4303 nop -- mov.w #0, CG 1 // No operation**  **0x449c: 0x3fff jmp 0x449c (offset: -2) 2 // jmp to next line**  **0x449e: 0x4303 nop -- mov.w #0, CG 1 // no operation**  **0x44a0: 0x431c mov.w #1, r12 1 // Move 1 into r12, Already done though???**  **0x44a2: 0x4130 ret -- mov.w @SP+, PC 3 // Return from interrupt**  **0x44a4: 0xd032 bis.w #0x0010, SR 2 // Set 10 as the status register**  **0x44a6: 0x0010**  **0x44a8: 0x3ffd jmp 0x44a4 (offset: -6) 2 // Jump to 44a4**  **0x44aa: 0x4303 nop -- mov.w #0, CG 1 // No operation wait for end of program** |

1. **Describe what the program is doing in a neat flowchart. You can also write a paragraph to describe in addition to the flowchart. [10 pts]**

## Flow Charts:



Essentially what this code does is it waits around for a user to press the switch on P2.1, and once it is pressed it infinitely blinks the LEDs at about 1 hz. The way it blinks them is by toggling every time register R13 goes from #2844-0, and then repopulates R13 with that same number to start the process all over again. From what I can see, it does this infinitely or until someone stops the program all together. It does not look like the switches do anything after they are pressed initially.

## Observations:

Reverse engineering is very useful, and somewhat scary. It was relatively simple to find those usernames and passwords. I am not sure if these files had any robustness coded in but if not it makes you wonder how secure your passwords are.

# Conclusion

This lab was by far my favorite, maybe I have a thing for reverse engineering. Anyways, I found this lab to be very informative and challenging enough to keep me interested. I did not face many issues other than just not reading the instructions fully, but at least I know how to do it now. Getting the LED’s to blink was troubling but once I figured out I was not supposed to use CCS it all went smoothly. I also accidentally created code for the crack\_me.out file and it was extremely long…

<https://drive.google.com/file/d/1CyaT0a_dhD_gGi7_y66G_6Zomp3HXL1r/view?usp=sharin>

Appendix

No code was specifically written for this assignment, but I will compile the different command line operations here.

**msp430-elf-readelf.exe –help**

**msp430-elf-objdump.exe –help**

**msp430-elf-strings.exe –help**

**msp430-elf-readelf.exe -h crack\_me.out**

* Determines type of machine code, data representation, entry points and more.

**msp430-elf-readelf.exe --section-headers crack\_me.out**

* Displays information about all sections.

**msp430-elf-objdump.exe -h crack\_me.out**

* Not sure exactly what this one does.

**msp430-elf-readelf.exe --symbols crack\_me.out**

* Displays all symbols in the ELF file.

**msp430-elf-readelf.exe --program-headers crack\_me.out**

* Displays all segments that are loadable into the memory.

**msp430-elf-objdump.exe -S crack\_me.out**

* Dumps source code together with disassembly.

**msp430-elf-objdump.exe -d crack\_me.out**

* Does not assume that source code is present.