PART2: Assembly and Reverse Engineering

Explain how you used reverse engineering to determine assembly language used:

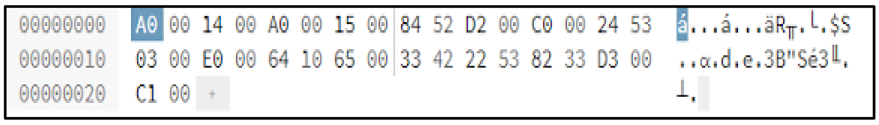
The methodology used to work out the which hard was utilized by the assembly was that of the use of converters and trial and error. I began with opening the .bin file in an online hex editor[1] and it threw out the results found in 'Figure 1'. I then proceeded to translate every 4 digits of the hex from the .bin file into 16 bits of binary with a hex-to-binary converter[2]. After that to get closer to working out instructions it required translating the resulting the binary, from the hex, into decimal format, as shown in 'Figure 2'. Therefore, it became possible to then work out instructions. From the scenario we know that "each instruction is 16 bits with the first 4 bits being the opcode." The rest of the bits, in fours, being arguments for each instruction.

I could then advance to finding out which specific hardware the instruction set was for. After seeing all the Opcodes, I then had a range from 0 to 14. Straight away this made CPU X inoperable with the instruction set as it only has 13 instructions. This left CPU Y and CPU Z. We later learned that the instruction set was offset by 1. Therefore, the first instruction, 1, became 0 with 2 becoming 1 and so on. To differentiate between CPU Y and CPU Z I ran trial by error testing to see if the assembly made sense with the instruction set off each piece of hardware.

It then became apparent that the hardware's instruction set the assembly made most logical sense, to me, with was for CPU Z. This is because accounting for the offset, the first instructions began with storing inputs from the keypad in the AC register.

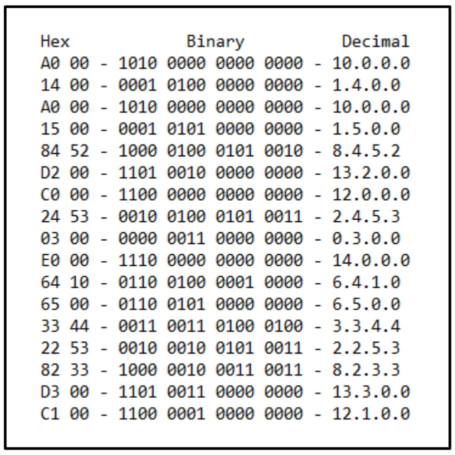
The CPU Z became more plausible as the perpetrator. We learn from (Rouse, 2012) "an accumulator is a register for short-term, intermediate storage of arithmetic and logic data in a computer's CPU (central processing unit)." It is reasonable to accept this definition as it is a commonly used in online web articles[3] and educational establishments[4] relating to computer architecture.

Figure 1.



```
00000000  A0 00 14 00 A0 00 15 00 84 52 D2 00 C0 00 24 53   á...á...äR╥.└.$S
00000010  03 00 E0 00 64 10 65 00 33 42 22 53 82 33 D3 00   ..α.d.e.3B"Sé3╙.
00000020  C1 00                                             ┴.
```

Figure 2.



```
Hex          Binary           Decimal
A0 00 - 1010 0000 0000 0000 - 10.0.0.0
14 00 - 0001 0100 0000 0000 - 1.4.0.0
A0 00 - 1010 0000 0000 0000 - 10.0.0.0
15 00 - 0001 0101 0000 0000 - 1.5.0.0
84 52 - 1000 0100 0101 0010 - 8.4.5.2
D2 00 - 1101 0010 0000 0000 - 13.2.0.0
C0 00 - 1100 0000 0000 0000 - 12.0.0.0
24 53 - 0010 0100 0101 0011 - 2.4.5.3
03 00 - 0000 0011 0000 0000 - 0.3.0.0
E0 00 - 1110 0000 0000 0000 - 14.0.0.0
64 10 - 0110 0100 0001 0000 - 6.4.1.0
65 00 - 0110 0101 0000 0000 - 6.5.0.0
33 44 - 0011 0011 0100 0100 - 3.3.4.4
22 53 - 0010 0010 0101 0011 - 2.2.5.3
82 33 - 1000 0010 0011 0011 - 8.2.3.3
D3 00 - 1101 0011 0000 0000 - 13.3.0.0
C1 00 - 1100 0001 0000 0000 - 12.1.0.0
```

Explain roughly what the assembly does:

[1] https://hexed.it
[2] http://www.binaryhexconverter.com/hex-to-binary-converter
[3] https://brainly.in/question/1550456
[4] https://www.coursehero.com/file/p6s92vb/ZERO-ADDRESS-INSTRUCTIONS-A-stack-organized-computer-does-not-use-an-address/

PART2: Assembly and Reverse Engineering

As the hardware's instruction set in which the assembly code applies was realized it was possible to put together the rational seventeen instructions, which can be seen in 'Figure 3'. Firstly, the assembly starts with loading an input from the keypad into the accumulator (AC) and storing it in the fourth register which is the first argument register. Followed by a similar process but stores the input from the keypad into register 5. After that we compare if the two inputs are equal, this then triggers a Boolean and stores the output of the Boolean in register 2 which is a return register. Followed by an LED which blinks for 2 seconds. However, the instruction in which LED should blink should have a number denoting the specific LED, but it is follows by a 0. From this, it could either be that an LED is either always on in the system or that from looking further ahead there is another LED, denoted as LED 1, means there could be two LEDs numbered 0 and 1 (possibly green and red as per some common systems).
After that, the following instruction adds the contents of registers 4 and 5 to register 3. Then loads contents of $3 to AC and displays it, leaving register 3 empty. Moving on to the phase after than we come across what I believe is the assembly code's attempt at making the process more complex than need be. Where it moves a value of 1 into register 4 and moves 0 to register 5, then subtracts register 3 from register 4 and stores value in register 4, but this would mean we would be taking 0 away from 1. We then add registers 2 and 5 together, 1 being in 2 and 0 being in 5 then store answer, 1, in register 3. Following that there is another compare between return registers 2 and 3 which at that moment both equal 1 and store the result in register 3. The result of this Boolean then triggers LED 1 to go off for 3 seconds. The depiction in 'Figure 4' is the use of algebra to simulate what the compiled assembly code, with inputs, may look like and overall helped me understand the transmission.

Figure 3.

```
Input number from AC
store $4
input number from AC
store $5
cmp $4 $5 $2
btfsc 2
btg 0
add $4 $5 $3
load $3
output $AC
mov $4 <1>
mov $5 <0>
subt $3 $4 $4
add $2 $5 $3
cmp $2 $3 $3
btfsc 3
btg 1
```

```
Input Number from keypad to AC
+a into $4
Input Number from keypad to AC
+a into $5
Boolean of 0 (false) or 1 (true) in $2 = True = Binary answer of 1 in $2
Blink LED for 2 seconds
Blink LED 0
+a from $4 and +a from $5 = 2a in $3
Load 2a from $3 into the AC
Output AC content to display = 0 in $3
+a into $4 = a into $4
+0 into $5 = 0 into $5
subtract $3 (0) from $4 (1) = a into $4
add $2 (1) to $5 (0) = store 1 into $3
Compare $2 and $3 (both equal 1) Boolean of True, store 1 into $3
Blink LED for 3 seconds
Blink LED 1
```

Figure 4.

Analysis of Counter-measures and/or flaws:

PART2: Assembly and Reverse Engineering

The security of this assembly code is could be compromised by the validation of inputs into the keypad. There could be an input limit in which if it goes over the limit it could adversely affect the ability of the LED in a way not intended. However, the return register 2 maintains the stored the inputs from registers 4 and 5 in the second and fourth instruction. Then when compared to the return register 3 the result of the Boolean is true and permits the LED to blink for 3 seconds.

However, this could lead to a buffer overflow. From the *Art of Software Security - Identifying and Preventing Software Vulnerabilities* we learn that "a buffer overflow is a software bug in which data copied to a location in memory exceeds the size of the reserved destination area" (Dowd, 2006). We can be assured this is a reputable source due to its nature, being a valid contribution to the field of software security and critically acclaimed and stored by educational institutions and reviewers from industry specialists, online[5].

Security Analysis of the Reverse Engineered assembly:

There are some implemented security measures preventing this assembly code being simply maliciously exploited. If we ran an assembly code with malicious intent, e.g. if the contents of registers 4 and 5 are not equal, the Boolean will not return true and the LED will not come on or if it comes on for longer than 2 seconds as a counter-measure it would be immediately telling and something would be incorrect with the code.

Furthermore, we have what I believe to be extra obfuscation, in addition to the instructions being offset by one, this could be an attempt at something similar to a ROT-13[6] data obfuscation method. Also, we have several instructions which makes no necessary difference to whether the Boolean returns true or not. After the contents of register 3 are outputted to AC, it then leaves register 3 is empty. All we needed to do was to make register 3 equal to register 2, which could be done by adding a value of 1, for when they are both compared. However, in instruction thirteen, for no functionally integral reason we subtract register 3 from register 4 and store it in register 4. But, this is just taking 0 away from 1 and keeping it the same. Fundamentally register 3 is still at 0. Then we add register 2 which maintains the original input, the Boolean result of 1, and register 5 with a value of 0 to 1 again but store it in register 3. We could have just moved a value of 1 into register 3 then compared register 2 and 3 and be done with it.

Code obfuscation are general methods for adding security, in the circumstance of someone achieving a static copy of your code. From *Reversing: Secrets of Reverse Engineering* we learn that "obfuscation is a generic name for a number of techniques that are aimed at reducing the program's vulnerability to any kind of static analysis" (Eilam, 2005). On the other hand, from *Art of Software Security - Identifying and Preventing Software Vulnerabilities* we learn that it has its limitations due to them believing that "…security by obscurity (or obfuscation) has earned a bad reputation in the past several years. On its own, it's an insufficient technique for protecting data from attackers; it simply doesn't provide a strong enough level of confidentiality" (Dowd, 2006).

[5] https://www.amazon.com/product-reviews/B004XVIWU2/ref=cm_cr_dp_d_cmps_btm?ie=UTF8&reviewerType=all_reviews
[6] https://en.wikipedia.org/wiki/ROT13

I believe on this matter they are correct. This is because, firstly their software security practices are meant to meet the standards of the PCI's[7] (Payment Card Industry) security standards. Moreover, data obfuscation is not the best for safekeeping for assembly. This is because with static code disassembly you have more time to try and find vulnerabilities and simply rotating or adding in unnecessary instructions into your code will not prevent malicious attackers with more resources and more time than hashing or encryption would.

Bibliography

Dowd, M. a. (2006). 7.1.2 Buffer Overflows. In M. a. Dowd, *The Art of Software Security - Identifying and Preventin Software Vulnerabilities* (p. 181). Addison Wesley Professional.

Dowd, M. a. (2006). 7.1.2 Buffer Overflows. In M. a. Dowd, *The Art of Software Security - Identifying and Preventin Software Vulnerabilities* (p. 51). Addison Wesley Professional.

Eilam, E. (2005). Basic Approaches to Antireversing. In E. Eilam, *REVERSING - Secrets of Reverse Engineering* (p. 361). Indianapolis, IN 46256: Wiley Publishing, Inc.

Rouse. (2012, April). *accumulator*. Retrieved from http://whatis.techtarget.com: http://whatis.techtarget.com/definition/accumulatorhttp://whatis.techtarget.com/definition/accumulator

[7] https://www.pcisecuritystandards.org/about_us/