Assignment 2 - Firmware Security and Rehosting

TASK 1: UART Identification

Goal: Decode the UART message using the corresponding protocol decoder with the right parameters.

Explanation: To find the UART configurations used by the embedded Raspberry Pi Pico and identify which physical pins on the board were carrying UART data, we followed the systematic probing approach using the logical analyzer and the microcontroller datasheet. Since the pinout was not given, we connected the logical analyzer to the pins one by one until we found that one pin remained constantly high -like the UART TX. Also, we connected with GND. All other pins stayed flat, that means, no data. After that we attached the Async Serial analyzer and selected the input channel as channel 0 which was connected to UART TX. We initially captured raw digital transitions, and the analyzer showed decoded characters, but with periodic framing errors, most likely it was because the baud rate was not correct. And to determine the correct baud rate, we used a measurement tool to check the duration of a single data bit. We measured the width of several bits and found that each bit was approximately ≈ 104 µs per bit and using the baud rate formula Baud rate=1/bit width =1/104 µs =9600. This matched the standard UART baud rate of 9600 and all other settings were kept at default.

Settings: The correct UART settings identified were baud rate =9600, data bit = 8, parity bit = none, stop bit = 1, bit order = LSB first, Polarity = Idle High

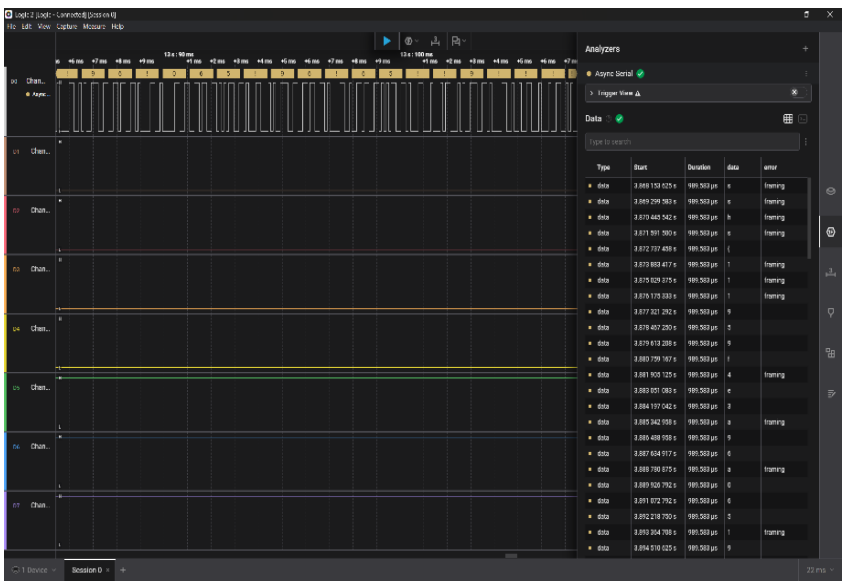

Illustration 1: *UART output by logic analyzer*

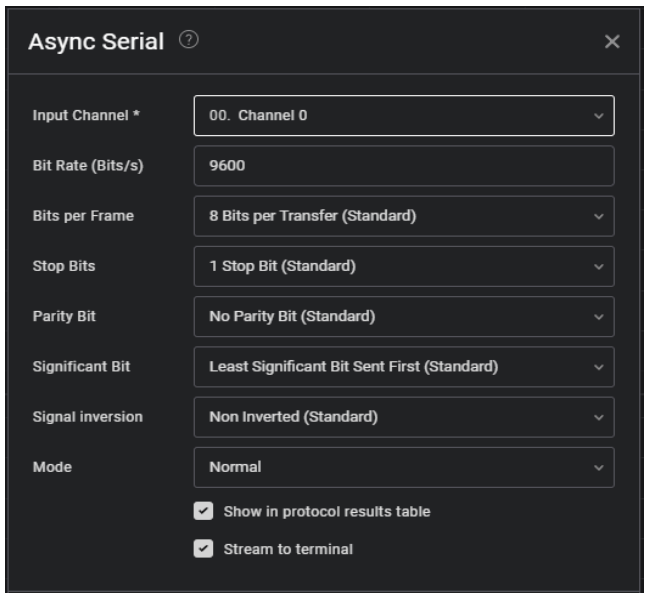

Illustration 2: *UART setting for the flag.*

These were verified by observing consistent decoded output and correct ASCII characters.

Result: **Flag: sshs{111959f4e3a96a065196a65149a2daa2}**

TASK 2 : SPI Identification

Goal: Decode the UART message using the corresponding protocol decoder with the right parameters.

Explanation: To identify the SPI wiring and configuration, we used both hardware probing and protocol analysis based on the microcontroller datasheet and pinout. The SPI requires a periodic square-wave clock (SCK), a chip-select line (CS) that pulls low before every transfer and Data line (MOSI and MISO) that only toggle when the clock is active. After testing multiple GPIO pins, only a specific group showed meaningful activity while the rest remained stable. By examining these signals on the logic analyzer, we identified the SPI bus. One pin showed a continuous, evenly spaced square wave, indicating the SPI clock (SCK). Another pin stayed high most of the time but briefly dropped low before each transmission, matching the behavior of an active-low chip-select (CS). The remaining two pins toggled only when CS was low and their transitions aligned with SCK edges, confirming them as MOSI (Master Out) and MISO (Master In). The combination of these four characteristic signals confirmed the correct SPI pin GP19=SCK (clock) connected to channel 3, GP20 =SPI1 TX(MISO) connected to channel 2, GP16=SPI1 RX(MOSI) connected to channel 1, GP17=SPI1 CSn connected to channel 0 on the Raspberry Pi Pico are default SPI1 pins. Initially, the decoding produced errors and fragmented bytes because the SPI mode and timing were not yet correct. We tested different combinations of CPOL (Whether the idle state of SCK is high or low)/CPHA (Whether data is sampled on the rising or falling edge), bit order (MSB-first or LSB-first) and observed the decoded output.

Settings: So, we were able to find the correct settings: clock state=CPOL=0, clock phase CPHA=0), Bit Order=MSB first, CS Polarity= active low, data bit =8

We systematically tested the SPI and monitored the decoded output. At first, the analyzer split the data into many tiny messages because the timeout was too short. This made the output unreadable. So, we increased the “command holdoff/timeout” to around 1–2 ms allowing the analyzer to group the entire SPI transaction into a single message. After adjusting the SPI mode and increasing the holdoff, the analyzer produced clean and consistent byte sequences, confirming the correct SPI configuration.

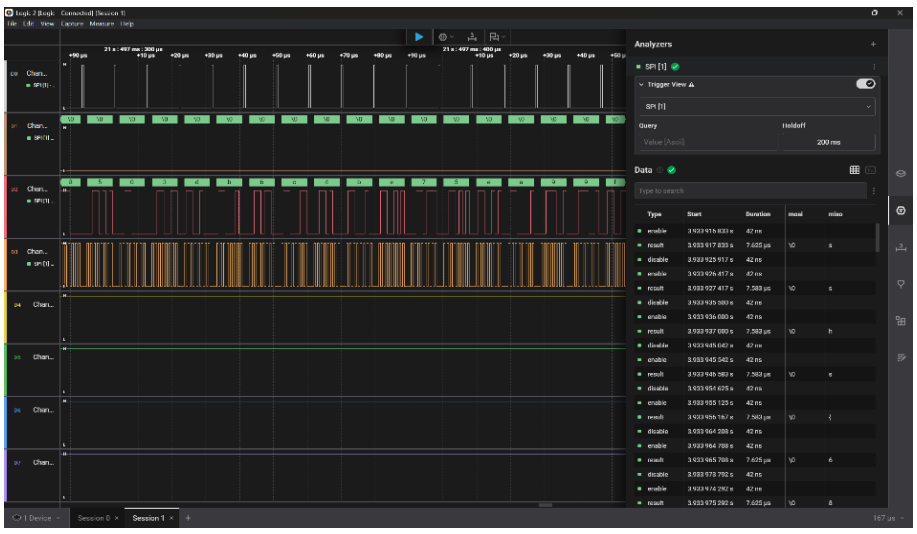

Illustration 4: *SPI signal output on logical analyzer*

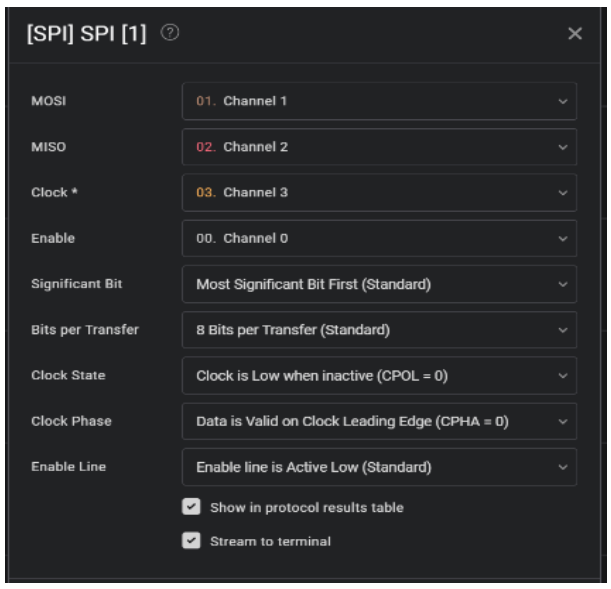

Illustration 5: *SPI setting*

These were verified by observing consistent decoded output and correct ASCII characters.

Result: **Flag: sshs{68503db6c4be75ea99ff3c3af4d82897}**

TASK 3: Rehosting

Goal: To rehost Assignment_2B_rehost() using unicorn and perform brute-force in order to fetch the flag at the correct PIN.

Explanation:

Stages Involved:

- mapping registers using mem_map().
- static analysis findings using Ghidra.
- selecting functions to hook such as printf(), scanf(), sleep(), puts().
- skipping delays such as sleep().
- Writing results to appropriate registers using reg_write().
- Automating stepping instructions using UC_ARM_REG_PC.
- detecting the required flag via puts().

Firstly, we performed correct firmware and registers mapping as per the 'regs.txt'. While doing static analysis using Ghidra, we discovered a few functions like sleep(), two appearances, that were causing delays to our brute force attempts and skipped printf() which was of the least use to us. We selected printf(), scanf(), sleep_ms(), and final puts() as our point of interest and started developing hooks around these functions. We began feeding PINs, starting from 0000, to scanf() and moved the program counter(UC_ARM_REG_PC) to the next line. As soon as the counter reached sleep() address, we skipped the address, and the subsequent lines of instructions such as printf() and sleep(), thereby stepping over both the delay functions. Then, the program counter moved line by line, allowing execution of the rest of the instructions and inserting data to the appropriate registers. Finally, we execute puts(), which displays whether the PIN gives a valid or invalid flag. The program stops when the correct PIN and flag are found.

```python
def bruteforce_pin():
    mu = unicorn.Uc(UC_ARCH_ARM, UC_MODE_THUMB)
    mapping(mu)


    try:
        for i in range(10000):
            pin= f'{i:04d}'
            test_data["pin"] = int(f'{i:04d}')


            # test_data["pin"] = i
            # pin=i
            # mu.hook_add(UC_HOOK_CODE, hook_all, test_data, begin=0x1000042c)
            mu.hook_add(UC_HOOK_CODE, hook_printf, test_data, begin=0x1000042e, end=0x1000043a)
            mu.hook_add(UC_HOOK_CODE, hook_scanf, test_data, begin=0x1000043c, end=0x10000440)
            mu.hook_add(UC_HOOK_CODE, hook_sleep_ms, test_data, begin=0x10000442, end=0x10000456)
            mu.hook_add(UC_HOOK_CODE, hook_puts, test_data, begin=0x10000492, end=0x10000498)
            # print("hooks addes")
            # print("PC at", hex(mu.reg_read(UC_ARM_REG_PC)))
            mu.emu_start(0x1000042c | 1, 0x10000498) #pop
            print("Testing Pin: {}".format(pin))
            if test_data["found"] == True:
                print("flag found = {}".format(test_data["flag"]))
                break
    except UcError as e:
        print("Unicorn Error:", e)
        print("Error code:", e.errno)
```

Result:

Testing Pin: 3832

Flag: sshs{9f74c94962e00ac4dad4b58e32f2d92a}

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

pxb@PXB:~/Music/rehost$ python3 ./rehost.py
Firmware Mapped...
Registers Mapped...
Flag: bytearray(b'\xe9\xbfu\x05\x9c"6\xc3&\xf1ER\x0e\xde\xae\xbdX\x8f\xa5\x99\xf7\xaf^t\xff\xd0\xd9\xe2\x96\xa3\xd6\x9d\xc8\x9fs\xa5\x9b6')
Testing Pin: 3830
Flag: bytearray(b'\xd9a\r4\n\xc6\xe0\xae\x0f\xa3@\xc0R}3?\xe4 \x84\xcd\xeevb\xd6\xb6\x8a\n\xddW@\xb1\x8a\x88\xc9\xc9\xa0\x16a')
Testing Pin: 3831
Flag: bytearray(b'sshs{9f74c94962e00ac4dad4b58e32f2d92a}')
Testing Pin: 3832
flag found = bytearray(b'sshs{9f74c94962e00ac4dad4b58e32f2d92a}')
pxb@PXB:~/Music/rehost$

Accounts
```