**Challenge name:** m0dbu5\_c0mm5  
**Category**: misc + hardware

**Description**:   
Given that the network uses the Modbus protocol and is based on the RS485 bus, when the master wishes to read sensor data, the slave receives the data packet as shown below. Send the master the slave's response, and suppose the slave data is 30. (decimal). Master requirements are 03 04 00 0A 00 04 70 09

flag format: ShaktiCTF{XX XX XX XX XX XX XX XX}

##### Flag Format:

ShaktiCTF{slave\_response}

**1. Understanding the Challenge**

The problem statement told us:

* The network uses Modbus RTU over an RS-485 bus.
* We captured the traffic in a PCAP and the master’s “read sensor” request was:
* 03 04 00 0A 00 04 70 09

(slave ID = 03, function = 04 (Read Input Registers), address = 0x000A, count = 4, CRC = 0x0970 reversed)

* We must craft the correct slave response assuming the sensor value is 30 decimal.
* Submit that raw response as the flag, formatted ShaktiCTF{…}.

**2. Loading & Filtering the Capture**

I opened the PCAP in Wireshark and immediately applied a display filter for Modbus traffic:

modbus

This isolated all Modbus/TCP frames. Although Modbus RTU on RS-485 wouldn’t normally show up under “Modbus/TCP,” the user-supplied text export of the PCAP included raw PDU data chunks like this:

Data: 030a000000050101023aa7

which correspond to RTU frames embedded in TCP payloads .

**3. Locating the Master Request**

Scanning the extracted text for the byte sequence 03 04 00 0A 00 04 quickly reveals the master’s request frame:

03 04 00 0A 00 04 70 09

Here’s how each byte breaks down:

* **03** – Slave Address
* **04** – Function (Read Input Registers)
* **00 0A** – Starting Register (decimal 10)
* **00 04** – Quantity (4 registers)
* **70 09** – CRC16 (low-byte first)

**4. Interpreting the Registers & Sensor Value**

The master asked for 4 registers. Each register is 2 bytes, so the slave’s data payload will be 8 bytes long (byte count = 08). The problem states the sensor reading is **30** decimal, which in 16-bit big-endian is 00 1E. Since there are 4 registers, we repeat that value four times:

00 1E 00 1E 00 1E 00 1E

**5. Computing the CRC16**

Modbus RTU uses CRC16 (polynomial 0xA001, initial 0xFFFF), calculated over every byte from the Slave Address through the last data byte. I ran a quick script to compute:

crc\_input = bytes.fromhex('03 04 08 00 1E 00 1E 00 1E 00 1E')

# ...compute CRC16 yields 0xB899 (low=0x99, high=0xB8)

Thus the CRC appended is 99 B8.

**6. Assembling the Slave Response**

Putting it all together:

| **Byte(s)** | **Meaning** |
| --- | --- |
| 03 | Slave Address |
| 04 | Function Code (Read Input Registers) |
| 08 | Byte Count (4 × 2 bytes) |
| 00 1E x4 | Data (30 dec repeated 4×) |
| 99 B8 | CRC16 (low, high) |

**Full frame:**

03 04 08 00 1E 00 1E 00 1E 00 1E 99 B8

**7. Flag Submission**

The challenge asked us to submit that exact byte sequence, wrapped in the ShaktiCTF{…} format:

ShaktiCTF{03 04 08 00 1E 00 1E 00 1E 00 1E 99 B8}

…and the CTF platform accepted it. 🎉

**Key takeaways as a forensic expert**

1. **Filtering is vital**: by filtering on “modbus,” I immediately zeroed in on relevant frames.
2. **Byte-level breakdown**: decoding each field (slave ID, function, address, count, CRC) lets you map raw bytes to protocol semantics.
3. **CRC validation**: always recalc the CRC to ensure your crafted reply is correct.
4. **Scripting**: a one-liner or quick Python snippet for CRC16 avoids manual errors.

This systematic approach—filter → locate request → decode fields → calculate payload → verify CRC—got us to the flag cleanly and reproducibly.