An Oil and Gas Weak Spot: Flow Computers

Executive Summary

- Flow computers calculate oil and gas volume and flow rates; these measurements are critical not only to process safety, but are also used as inputs in other areas, including billing.
- Team82 is disclosing details on a path-traversal vulnerability in ABB TotalFlow flow computers and controllers.
- An attacker could exploit a vulnerable system to inject and execute arbitrary code.
- CVE-2022-0902 (CVSS v3: 8.1) was addressed in a firmware update.
- Affected products include:
    - ABB's RMC-100 (Standard)
    - RMC-100-LITE
    - XIO
    - XFCG5
    - XRCG5
    - uFLOG5
    - UDC products.
- ABB's security advisory can be found here.

Introduction

Flow computers are specialized computers that calculate volume and flow rates for oil and gas that are critical to electric power manufacturing and distribution. These machines take liquid or gas measurements that are not only vital to process safety, but are also used as inputs by other processes—alarms, logs, configurations—and therefore require accuracy to ensure reliability. These capabilities are described in the American Gas Association's AGA Report No. 9.

One other important aspect to the role of flow computers within a utility is billing. The most noteworthy and related security incident was the ransomware attack against Colonial Pipeline, which impacted enterprise systems, and forced the company to shut down production because it could not bill customers. Disrupting the operation of flow computers is a subtle attack vector that could similarly impact not only IT, but also OT systems; this led us to research the security of these machines.

Team82 focused on ABB flow computers because of their use within many large oil and gas utilities worldwide. We looked for vulnerabilities that could give an attacker the ability to influence measurements by remotely running code of their choice on the device.

As a result, Team82 found a high-severity path-traversal vulnerability (CVE-2022-0902) in ABB's TotalFlow Flow Computers and Remote Controllers. Attackers can exploit this flaw to gain root access on an ABB flow computer, read and write files, and remotely execute code.

ABB has made a firmware update available that resolves the vulnerability in a number of product versions; it also recommends network segmentation as a mitigation. More information, including affected product versions, is found in ABB's advisory.

How Flow Computers Work

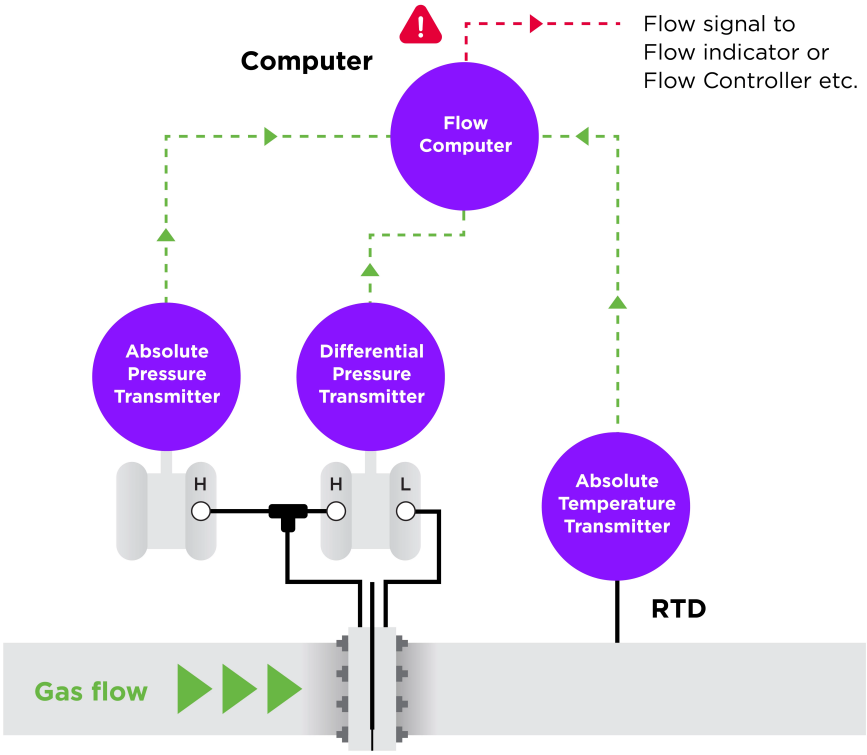
Flow measurement computations, especially gas flow, demand a substantial amount of processing power and thus are often calculated by a low-power CPU rather than a microcontroller.

Examples of different flow computers.

Flow meters read raw data from attached sensors that measure the volume of a substance in a number of ways, depending on what's being measured (gas or a liquid). Different examples of flow meters include: electromagnetic, vortex, differential pressure, thermal, coriolis, and others.

Examples of different types of flow meters (Source: ABB)

The following diagram describes how a flow computer usually measures gas flow:



Three types of sensors are used to calculate gas flow using a differential pressure technique: absolute pressure transmitters, differential pressure transmitters, and absolute temperature transmitters. Raw data is sent to the flow computer, which measures gas flow.

Researching ABB's µFLO G5 Flow Computer

The target of our research was ABB's µFLO G5 flow computers. This device can receive raw sensor data from other flow meters, perform flow calculations (following the AGA and ISO standards) and show/propagate the output.

µFLOG5 in its enclosure. (Source: ABB)

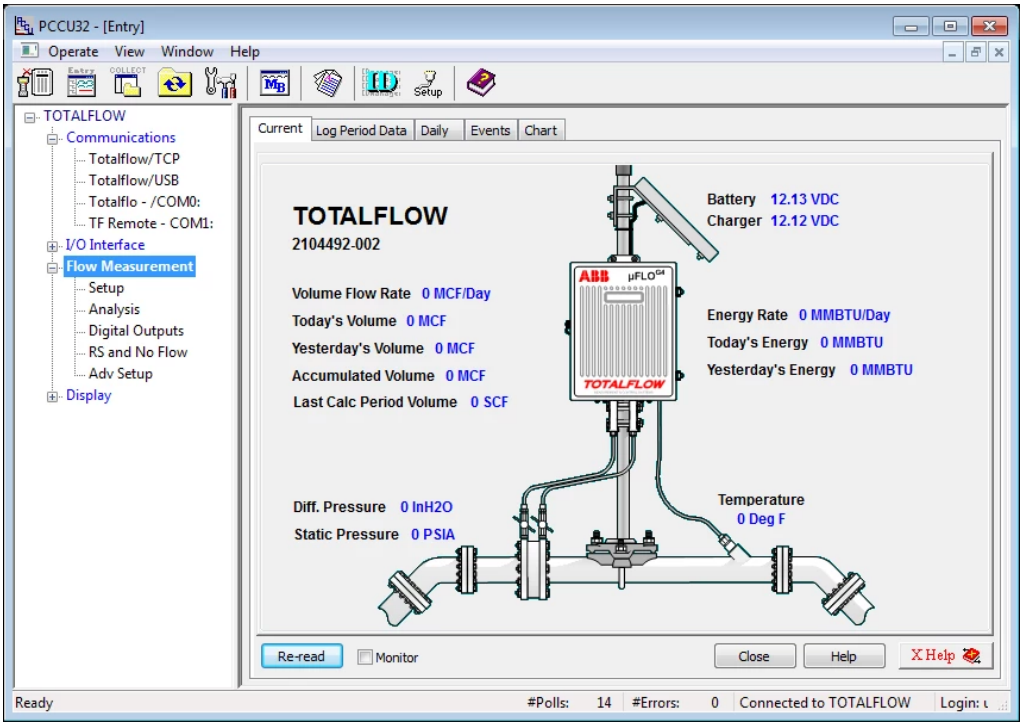
The µFLO G5 is a single board computer with IO ports (Ethernet, USB ,etc.), CPU, and other peripherals. The CPU is an ARMv8 processor, which is a 32-bit architecture. The device runs Linux as an operating system, which was good news for us, because this increased our chances to emulate the device in the lab.

From a security perspective, the µFLO G5 features three main mechanisms:

1. **Security switch:** A physical switch attached to the board that will enable/disable the use of the security passcode.
2. **Security passcode:** Two four-digit passcodes; one for reading data, and another authorizing writing of data.
3. **RBAC**: Role-based access control which assigns roles and permission to read and write specific attributes; this option is implemented only on the client side.

Client Application

The flow computer can be remotely configured with a designated configuration program, below.



Interface of ABB's PCCU Client showing measurement data.

The interesting thing to note is that configuration is done via a proprietary protocol designed by the ABB called TotalFlow. This protocol can be used on top of a serial or Ethernet (TCP) connection. Most of the communication between the client and the device—retrieval of the gas flow calculations, set and get device settings, import and export of the configuration files—is done over the TotalFlow protocol (TCP/9999).

Our goal in this research is to achieve remote code execution on the device. The proprietary protocol seemed to be a good attack vector to start with because undocumented protocols are usually less reviewed by security researchers.

Understanding ABB's TotalFlow Protocol

As we began our examination of the proprietary TotalFlow protocol, we knew two things: TotalFlow is 1) used to configure the device and send the flow measurements to the client, and 2), it listens on TCP port 9999.

Our goal is to be able to send and receive messages of our choice to test the implementation of the protocol. For this, we need to understand the protocol structure and build a simple client that constructs the payload. Luckily for us, the firmware is available online and is not encrypted, therefore we could easily extract it in order to analyze the application.

First, we wanted to find the binary that implements the protocol. Often the implementation of the protocol will reside at the main executable file or one that is directly linked to it.

Because our target is a Linux-based embedded device, the main binary will be executed at system's init. The good place to search for this is the init.d/inittab:

```
# Respawn deviceloader if it stops
null::respawn:/devLoader.exe
```

inittab content.

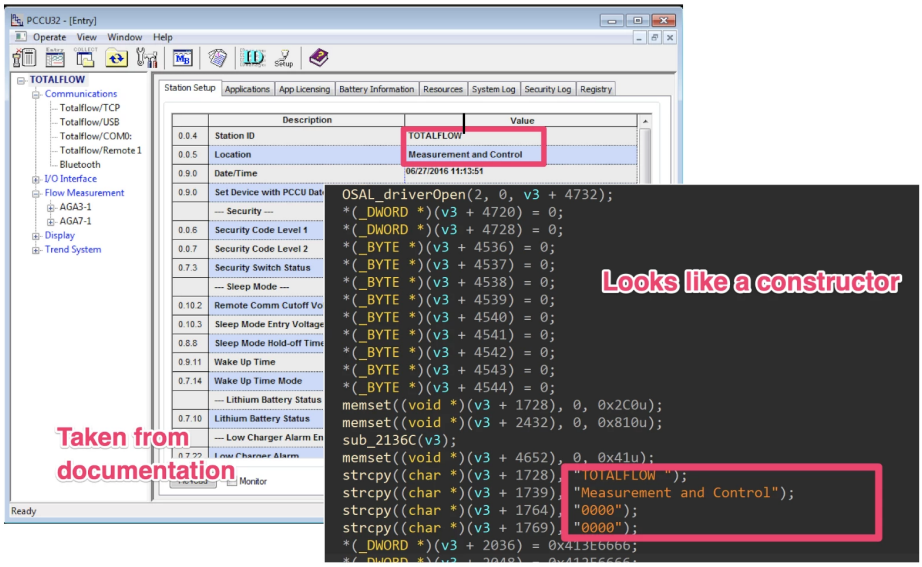
The init revealed the name of the binary: devLoader.exe (exe is a probably legacy name from when the device ran on Windows CE), which allowed us to reverse engineer it. The binaries within the firmware were stripped, but we had a lot of error-related logging strings, below, which was great for our research because it makes our life easier in finding interesting functions.

```
→  binaries strings -n 20 flash | grep -ie '^error' | sort | uniq | head -n 20
ERROR: Failed to load back up process. Warm configuration wont be backup up.
Error creating recursive lock instance for slot=%d
Error in saving RMC persistence files rmcAppDataObjIds.dat and rmcAppDataDefUUIDs.dat
Error parsing CDATA section
Error parsing PCDATA section
Error parsing comment
Error parsing document declaration/processing instruction
Error parsing document type declaration
Error parsing element attribute
Error parsing end element tag
Error parsing start element tag
Error processing xfRecord %hhu.%hhu.%hu=>%hhu.%hhu.%hu for %d regs
Error reading RMC App Def UUIDs file at %s
Error reading RMC App Obj ID file at %s
Error reading detailed config file at %s
Error reading device config file at %s
Error reading from file/stream
Error unlocking mutext for '%s' on count problem processing
Error unlocking mutext for '%s' on count problem processing, %s: %d
Error writing RMC App Def UUID file into RMC filesystem
```

A sample of error strings found within TotalFlow binaries.

There are few techniques we can start with to search for the relevant code for incoming packet handling.

1. Look for matching strings from the client application and the firmware, below:



Since there was a lot of memory initialization, it was likely we had found a constructor, which gave us a place to start.

1. Another good place is CRC checks. Embedded devices, especially ones that receive data from serial ports, use a CRC checksum to validate the accuracy of the received payload. Finding the place where CRC checks are validated is interesting because this will point to the payload that was received from the client. CRC checksum often uses hardcoded lookup tables which are easy to find within the binary.

2. Last but not least are the error strings; if you are fortunate, you will be able to find the relevant code just by looking at those. Sadly, in this case, we weren't so fortunate.

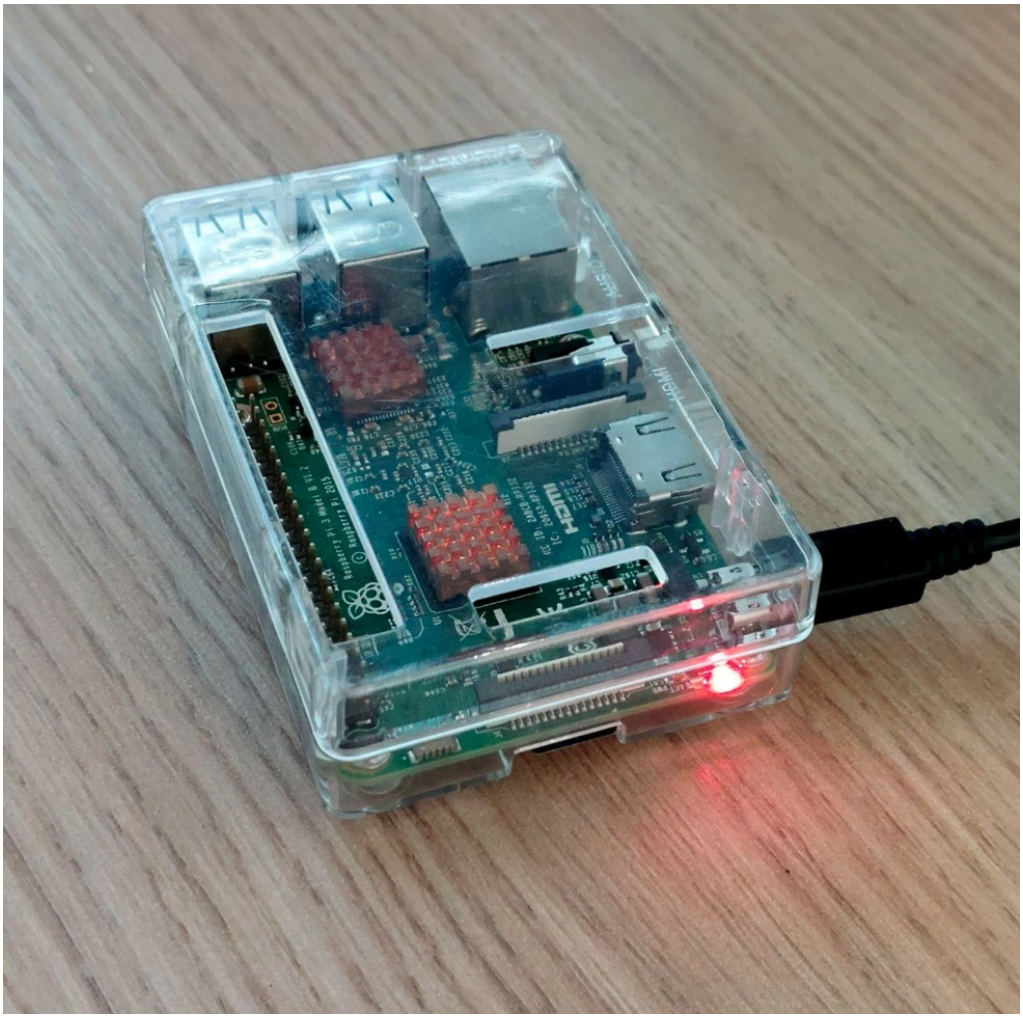
Now that we have a basic understanding of what we need to look for and where to find it, our next step is to create a setup of the device so we can dynamically debug it.

Emulating a Flow Computer on RaspberryPI

Although quite often we purchase devices we research, this time it was not an option. When the application of interest is within the user space and the device runs a familiar operating system on familiar architecture, it is often possible to

emulate the relevant part.

Therefore, we took one of our Raspberry Pis, copied the firmware's file system to it, and chrooted the directory.



Team82 used a Raspberry Pi to load ABB firmware and emulate a flow computer

The main disadvantage of emulating the device is that at some point the application will want to communicate with peripherals. Unfortunately, our Raspberry Pi doesn't have orifice plates attached to it, so any communication with them needs to be patched (changed within the binary).

The procedure to patch the application is straightforward:

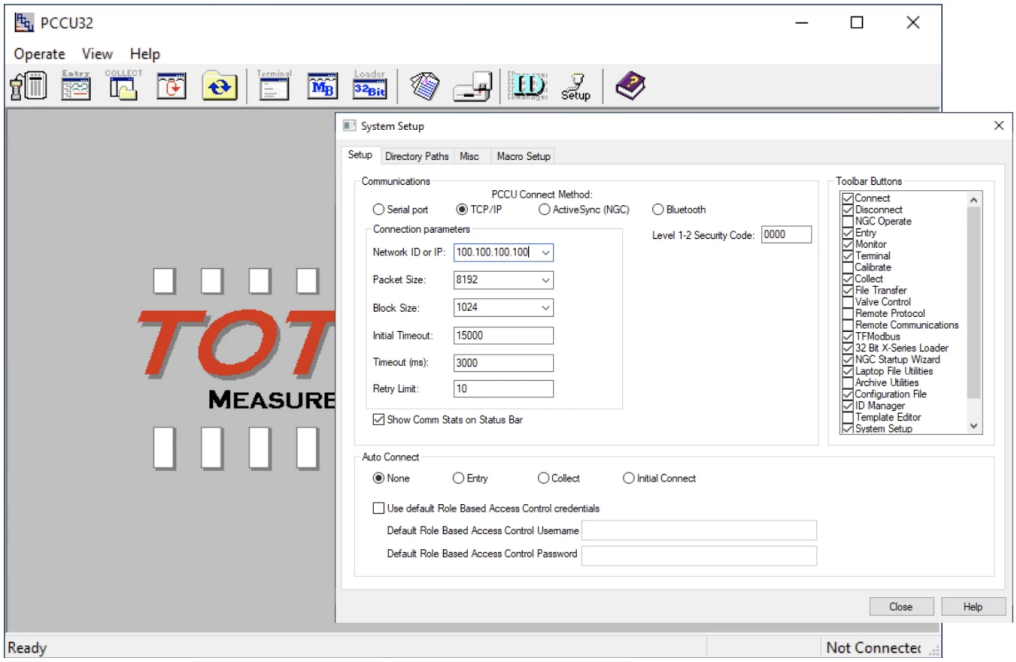
1. Run the binary
2. Wait for the binary to crash (due to emulation/setup issues)
3. Patch the function that causes the problem (e.g. skip a check)
4. Back to Step No. 1

In this research, seven functions were patched across two binaries; these functions communicate with the sensors and other hardware peripherals, which obviously do not exist in our simulated Raspberry Pi environment. Four hours later, we were good to go to the next step.

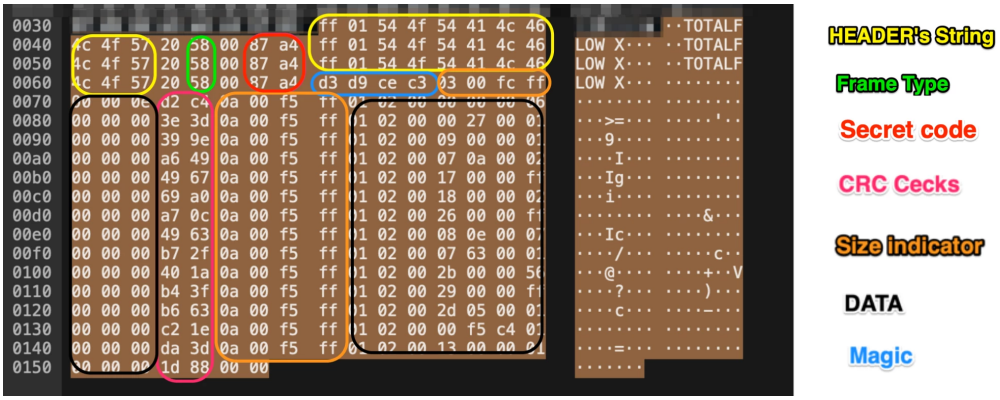
Debugging the TotalFlow Protocol

Now that we have a working setup, we were ready to analyze communication between the device and the client application. With a debugger, we can stop at the interesting functions that we have found by reverse engineering the binary and completing our understanding of the protocol.

We downloaded the client application (PCCU) from the ABB website and installed it on a Windows machine. We connected the application to the flow computer by providing the IP address of the Raspberry Pi:



The user interface for ABB's client application, PCCU.



A packet sent from client to device when first connected.

TotalFlow is a relatively simple protocol: Every setting within the device has its own TAG that is defined by a tuple called Registers (APP, ARRAY, INDEX).

For example, in the image below we see that the "SSH Service" setting is accessible by the (0.7.27) Register.

	Description	Value
0.0.15	Network ID	g5uflo
	--- Services ---	
0.7.27	SSH/SFTP Service (Port: 9696)	Disabled
0.7.29	Totalflow Service	Enabled

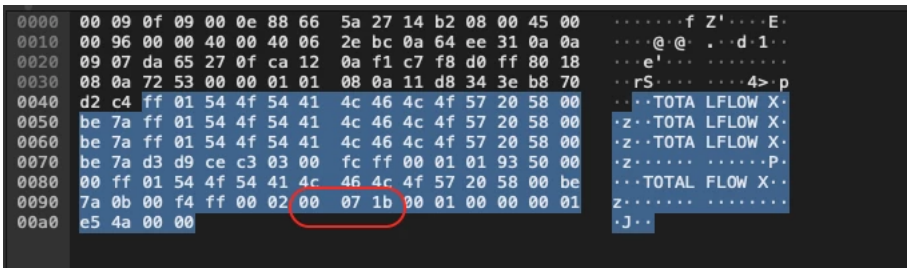
A sample of network settings from the PCCU client application.

The RegisterGet and RegisterSet functions, below, are responsible for changing/returning the Register's value:

```
int __fastcall CtfProtocol::RegisterGet(CtfProtocol *app, int array, unsigned int index, void *ptr_fot_result)
{
  int result; // r0
  int v5; // r1
  char *v6; // r1
```

```
int __fastcall CtfProtocol::RegisterSet(CtfProtocol *app, unsigned int array, unsigned int index, void *value)
{
  CtfProtocol *v4; // r4
  const char *v5; // r7
  int v6; // r5
  int result; // r0
```

The following is the payload that will enable SSH - as we set the triple tuple of the SSH settings to be enabled - app: 0 array: 07 index: 0x1b (27).



Now that we understood the protocol structure, we could write a simple python client with interesting functionalities such as read-write Registers, enable SSH, and more:

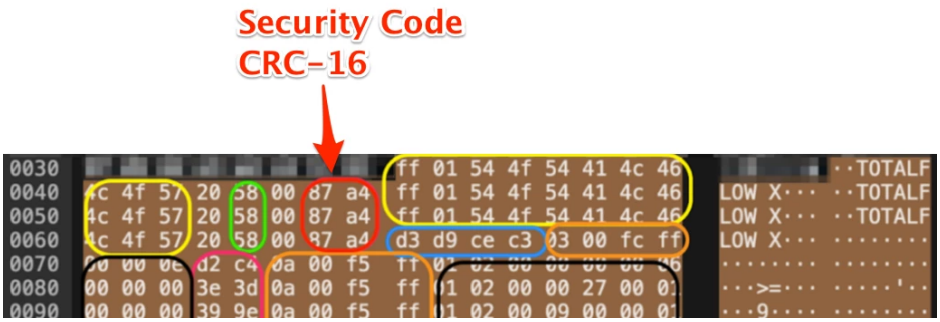
```
uFLOG5 mini client

optional arguments:
  -h, --help            show this help message and exit
  --read
  --write
  --src_file_name SRC_FILE_NAME
  --dest_file_name DEST_FILE_NAME
  --write_to_file
  --restart_device
  --enable_ssh
  --get_sec_code_hash
```

In order to do this, we need to be authenticated by providing the correct security code.

Authentication Bypass

Note the security code (the red rectangle) is a CRC-16 of the four-digit security passcode. Since the device sends an error message on incorrect code and there is no rate limit mitigation available on the device, we can easily bypass the authentication mechanism by enumerating all possibilities.

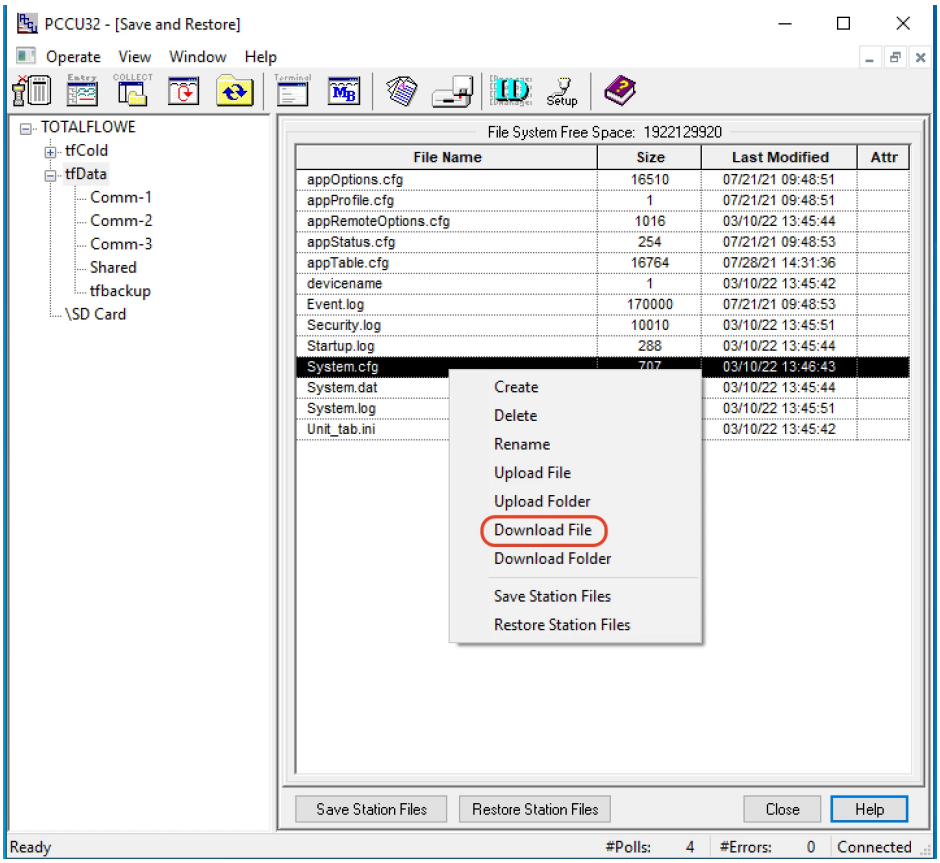


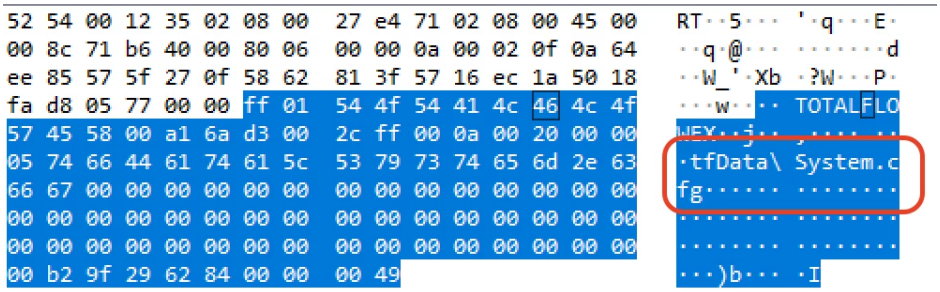
CRC-16 is a two-byte value, which has a maximum of 2^{16} possibilities. We can brute force all of the possible values in a range 0-65,535, which can take about four minutes. We can also optimize it by calculating, prior to the attack, the values from CRC-16 (0000) to CRC-16 (9999) and thus reducing the number of possibilities to 10,000.

Finding a Vulnerability

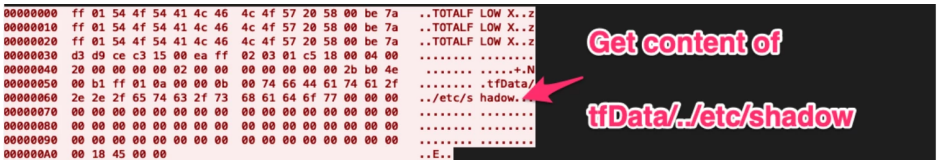
Now that we have an authentication bypass, it is time to look at functionalities available to authenticated users. Remember we said the configuration can be uploaded and downloaded? This is a good place to look for bugs because file operations are not always done securely.

This is what the file download procedure looks like in Wireshark:





We can see that the request contains a file name in the tfData directory. Lets check for a path traversal vulnerability by requesting the /etc/shadow file.




```
→  uflo_client git:(master) ✗ python3 ./uflog5_client.py --read --dest_f
ile_name tfData/../etc/shadow
[X] Reading 'tfData/../etc/shadow'...
[X] Got 479 bytes of tfData/../etc/shadow (decompressed)
b'root:6kv55iwIRTlJM:16563:0:99999:7:::\nbin:*:10933:0:99999:7:::\ndaemo
n:*:10933:0:99999:7:::\nadm:*:10933:0:99999:7:::\nlp:*:10933:0:99999:7::
:\nsync:*:10933:0:99999:7:::\nshutdown:*:10933:0:99999:7:::\nhalt:*:1093
3:0:99999:7:::\nuucp:*:10933:0:99999:7:::\noperator:*:10933:0:99999:7:::
\ntotalflow:$1$R2Blg7b2$3wrDwaC9eghg8/ujGV4Ge1:16351:0:99999:7:::\nnobod
y:*:10933:0:99999:7:::\ndefault::10933:0:99999:7:::\nmqtt:kJm1fWoHFuTFg:
17990:0:99999:7:::\navahi:*:::::::\ndbus:*:::::::\nsshd:*:::::::\n'
```

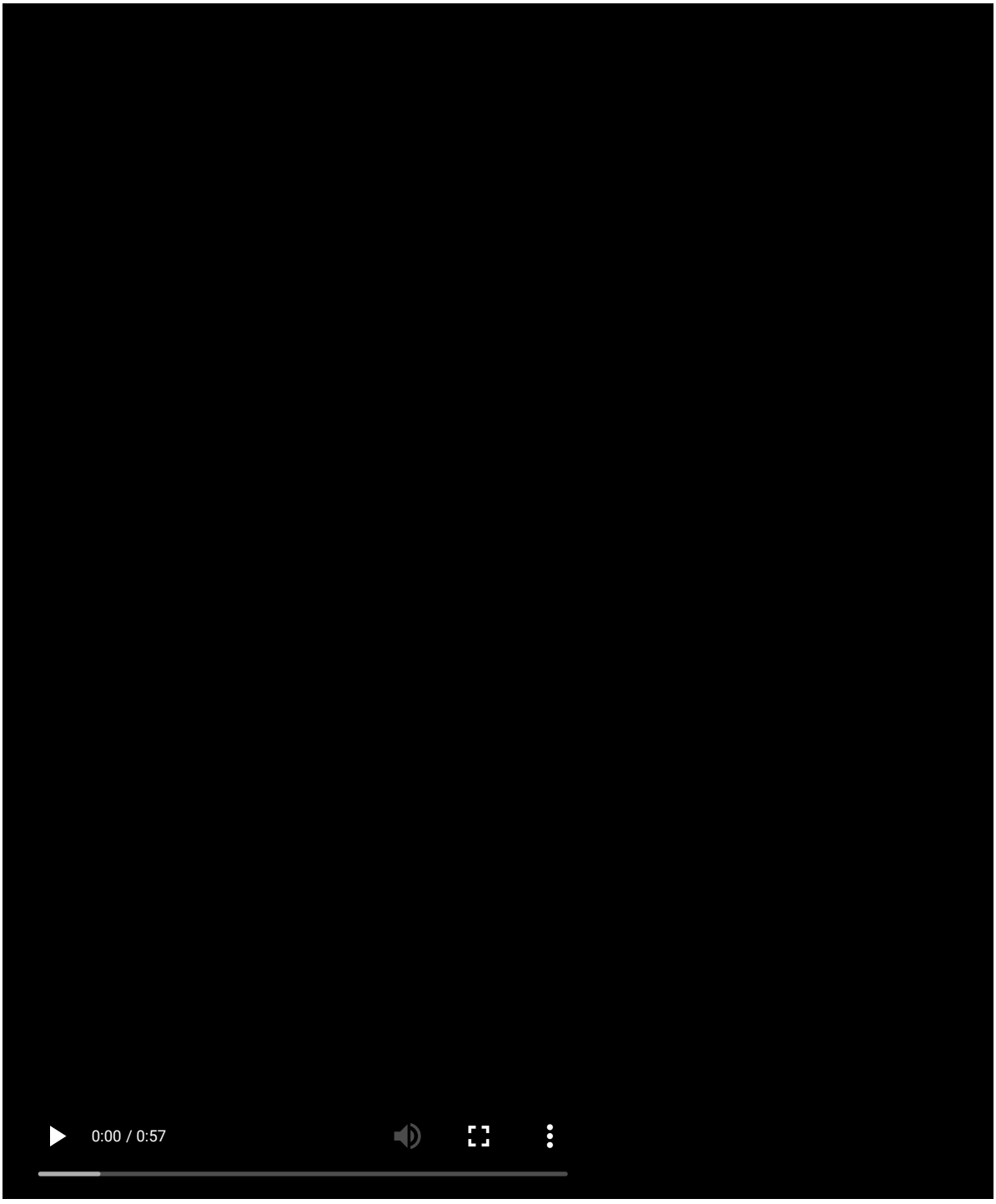
/etc/shadow content

Nice. It works.

Remote Code Execution

Now that we have an arbitrary read and write, it is simple to get code execution.

We chose the simplest approach, reading /etc/shadow and using hashcat cracking the root account password (which turned out to be root:root). Then we changed the SSH configuration file to enable root to connect using password. Then all that was left to do was to turn on the SSH daemon (using the TotalFlow protocol) and to connect to it.



Team82's proof of concept in action.

The Vulnerability

CVE-2022-0902

CWE-22 Path Traversal Vulnerability

Affected Products: ABB Flow Computers and Remote Controllers' Totalflow TCP protocol

CVSS v3 score: 8.1

This path traversal vulnerability can enable an attacker to take over flow computers and remotely disrupt the flow computers' ability to accurately measure oil and gas flow. These specialized computers calculate these measurements that are used as inputs in a number of functions, including configurations and customer billing.

A successful exploit of this issue could impede a company’s ability to bill customers, forcing a disruption of services, similar to the consequences suffered by Colonial Pipeline following its 2021 ransomware attack.

Team82 disclosed this vulnerability to ABB, which issued an update that addresses the issue. ABB also advises network segmentation as a mitigation strategy; further information is available in ABB’s advisory.