Securing the Industrial Internet

CyberX Threat Intelligence

Rockwell Automation MicroLogix Remote Code Execution

Contents

Introduction

Serving customers worldwide, CyberX enables real-time detection of cyber and operational incidents by providing complete visibility into operational networks. As part of the research and development efforts backing its flagship technology, CyberX has created its Industrial Threat Intelligence capabilities.

The CyberX Threat Intelligence team has discovered numerous zero-day vulnerabilities in Programmable Logical Controllers (PLCs) and industrial equipment, ranging from Denial-of-Service (DOS) to remote code execution.

CyberX is a member of the Industrial Internet Consortium (IIC) and ICS-ISAC and was recognized by Gartner as a Cool Vendor in Security for Technology and Service Providers, 2015.

Abstract

This document details the research that led to the finding of a remote code execution vulnerability on the Allen-Bradley MicroLogix family of controllers from Rockwell Automation.

The vulnerability has been acknowledged by the Department of Homeland Security, and received CVSS v3 base score of 9.8. Part of the innovative work described in this document, which includes the creation of a custom firmware, was also presented in the 2015 ICS Cyber Security Conference in Atlanta. Major part of the interest exhibited was due to the distinct nature of the research, arising when its results are compared to past vulnerabilities found in Rockwell Automation's equipment

The document focuses on Allen-Bradley MicroLogix 1100. However, the vulnerability also relates to MicroLogix 1400. These controllers are used for every type of control application worldwide, rendering the impact of this research extensive.

Preparation

Our target protocol was EtherNet/IP, as it has been the focal point for numerous customers. This led us to research the **Allen-Bradley MicroLogix 1100**.

When choosing a research target it is important to check the cost-effectiveness of its firmware availability.
Some vendors do not offer firmware updates, which means that we would need to extract the firmware out of the flash. This process is time consuming and not effective.

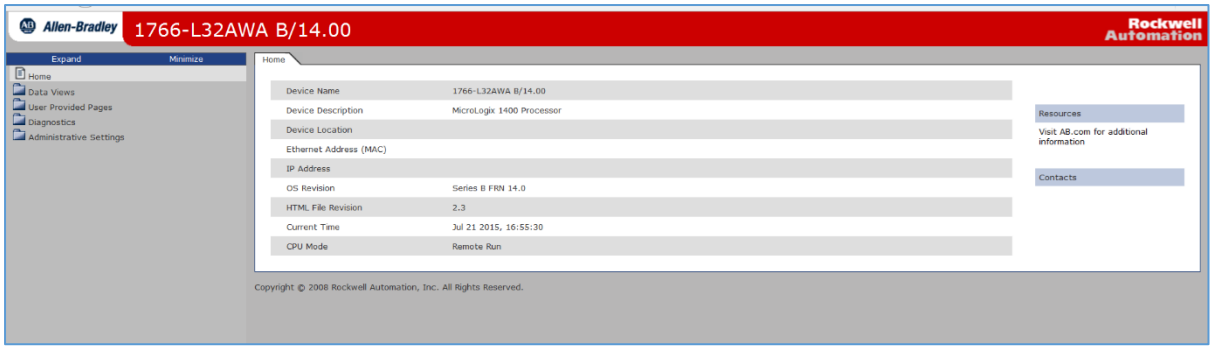
However, the firmware for ML1100 was easily available on the official vendor site.

Although this PLC is almost 10 years old, the latest firmware update was in 2014. Hence, we had to verify that the latest firmware version is installed on our device.

Before Diving In

The next step is to study everything we can about the device, in order to make the next steps easier. Connecting it with a simple RJ45 and port scanning it gives us some interesting information.

Port 80 - HTTP Server



The HTTP server is thin and contains statistics about the device, a user management page and some user defined pages.

The port may be fingerprinted by the HTTP Header **Server**, which its unique value is **A-B WWW/0.1**. Usually custom server headers might be an indication for a custom HTTP implementation.

Port 44818 – EtherNet/IP

This port is interesting because of the information it discloses.

```
Product name: 1763-L16AWA B/14.00
Vendor ID: Rockwell Automation/Allen-Bradley
Serial number: [DWORD]
Device type: Communications Adapter
Device IP: 192.168.90.90
```

The product name consists of the catalog number and the firmware version. For example, our catalog number is 1763-L16AWA which is ML1100, and the firmware version is B/14.00 which is version 14, the latest firmware.

Getting to know the firmware

We have extracted the file ML1100_R03R14_Os.bin from the flashing utility and decided to scan it with binwalk. The scan did not yield any significant results due to the fact that the image is a one large binary blob.

The next step was to determine the CPU type which this firmware runs on. Although Allen-Bradley states in their official documentation that the CPU architecture is unique and proprietary, a quick search on the internet revealed it is a ColdFire v2 CPU.

Loading it into IDA Pro with the ColdFire CPU configuration and instructing IDA to disassemble from the first instruction, showed us that the firmware starts with a JMP opcode.

Because we did not provide the real loading offset, IDA could not analyze the image. The flashing utility also includes a file by the name ML1100_R03R14_Os.nvs. This is a configuration file containing the value StartingLocation = 0x00008000, which is the real ROM offset. To determine the RAM offset and size, we looked into the offsets referenced by the disassembled opcodes, and chose values that cover their ranges.

Custom Firmware

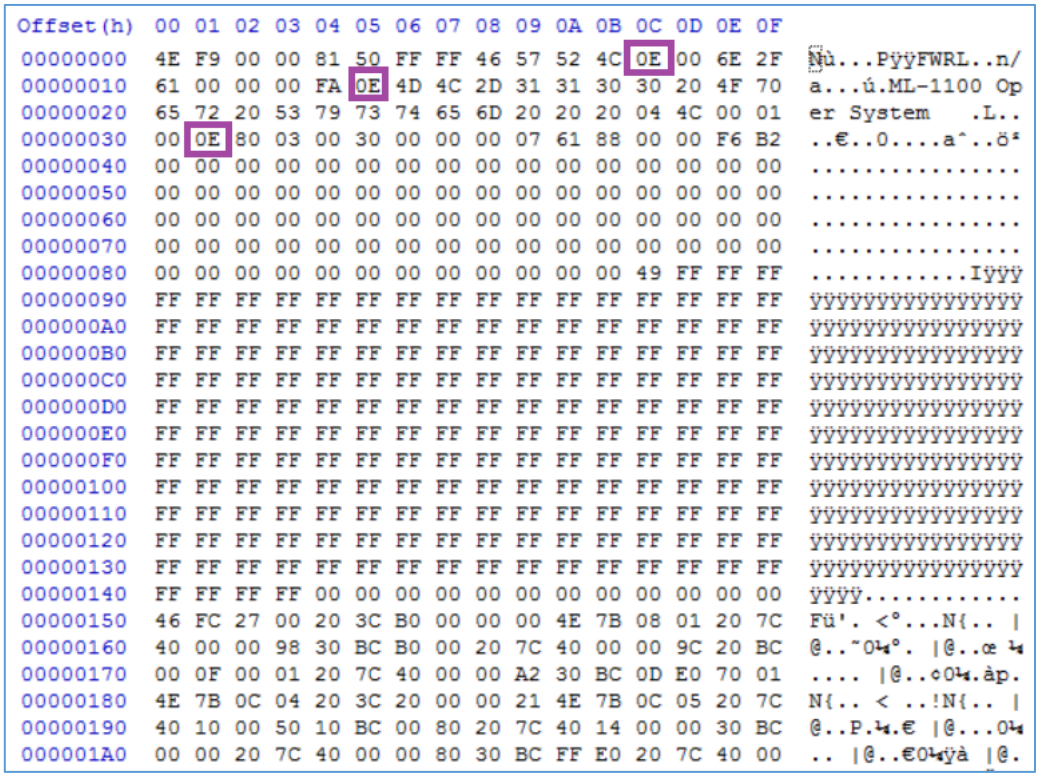
To better understand the firmware workings and to develop a working remote code execution exploit, we had to get a better overview of the memory. Since we could not find any memory leak vulnerabilities we have decided to create a custom firmware. This firmware shall allow us to dump memory.

The first attempt to patch and upload the firmware was not successful. This is due to the boot firmware returning an error, stating that we tried to upload a corrupted firmware. We have assumed that some checksum algorithm exists.

There are many kinds of checksums, so first we had to figure out whether the checksum is position dependent like CRC/MD5, or just a regular checksum that sums all the bytes or their LSB or anything similar to that. In order to test this we used the latest firmware and swapped 2 bytes of code. Uploading it to the device was successful. This led us to the conclusion that the error checking algorithm is position independent.

Now that we know this is just a plain old checksum algorithm, we started to study the header. By comparing several firmware images we noticed the changes between them and how they affect the bytes in the header. We concluded there are 2 checksums, one for the header and one for everything else.

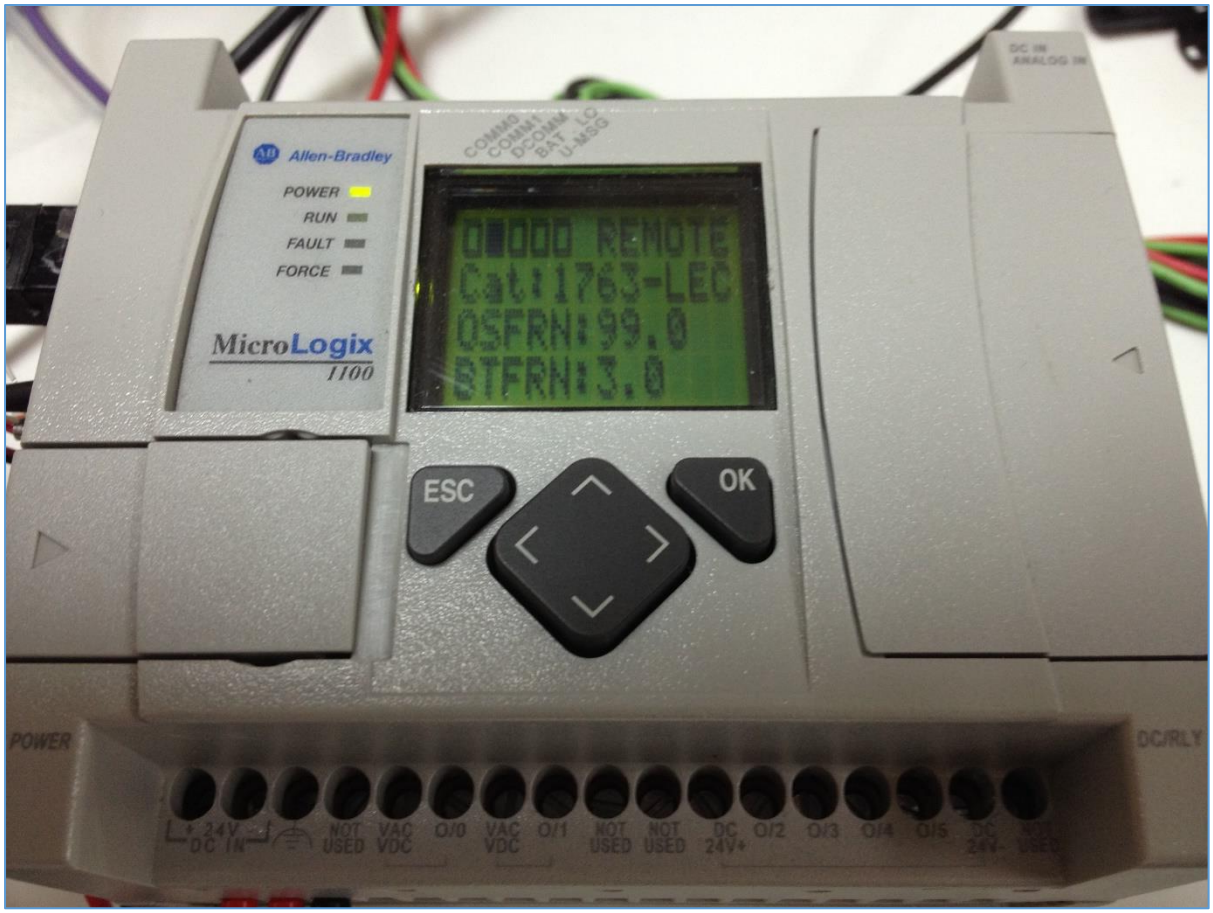
The first noticeable change is the version. The byte 0xE means its firmware version 14.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	4E	F9	00	00	81	50	FF	FF	46	57	52	4C	0E	00	6E	2F	Nù...PÿÿFWRL..n/
00000010	61	00	00	00	FA	0E	4D	4C	2D	31	31	30	30	20	4F	70	a...ú.ML-1100 Op
00000020	65	72	20	53	79	73	74	65	6D	20	20	20	04	4C	00	01	er System .L..
00000030	00	0E	80	03	00	30	00	00	00	07	61	88	00	00	F6	B2	..€..0....aˆ..ö²
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	49	FF	FF	FF	.....Iÿÿÿ
00000090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000100	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000110	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000120	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000130	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000140	FF	FF	FF	FF	00	00	00	00	00	00	00	00	00	00	00	00	ÿÿÿÿ.....
00000150	46	FC	27	00	20	3C	B0	00	00	00	4E	7B	08	01	20	7C	Fü'. <°...N{.. |
00000160	40	00	00	98	30	BC	B0	00	20	7C	40	00	00	9C	20	BC	@..˜0¼°. |@..œ ¼
00000170	00	0F	00	01	20	7C	40	00	00	A2	30	BC	0D	E0	70	01	.... |@..¢0¼.àp.
00000180	4E	7B	0C	04	20	3C	20	00	00	21	4E	7B	0C	05	20	7C	N{.. < ..!N{.. |
00000190	40	10	00	50	10	BC	00	80	20	7C	40	14	00	00	30	BC	@..P.¼.€ |@...0¼
000001A0	00	00	20	7C	40	00	00	80	30	BC	FF	E0	20	7C	40	00	.. |@..€0¼ÿà |@.

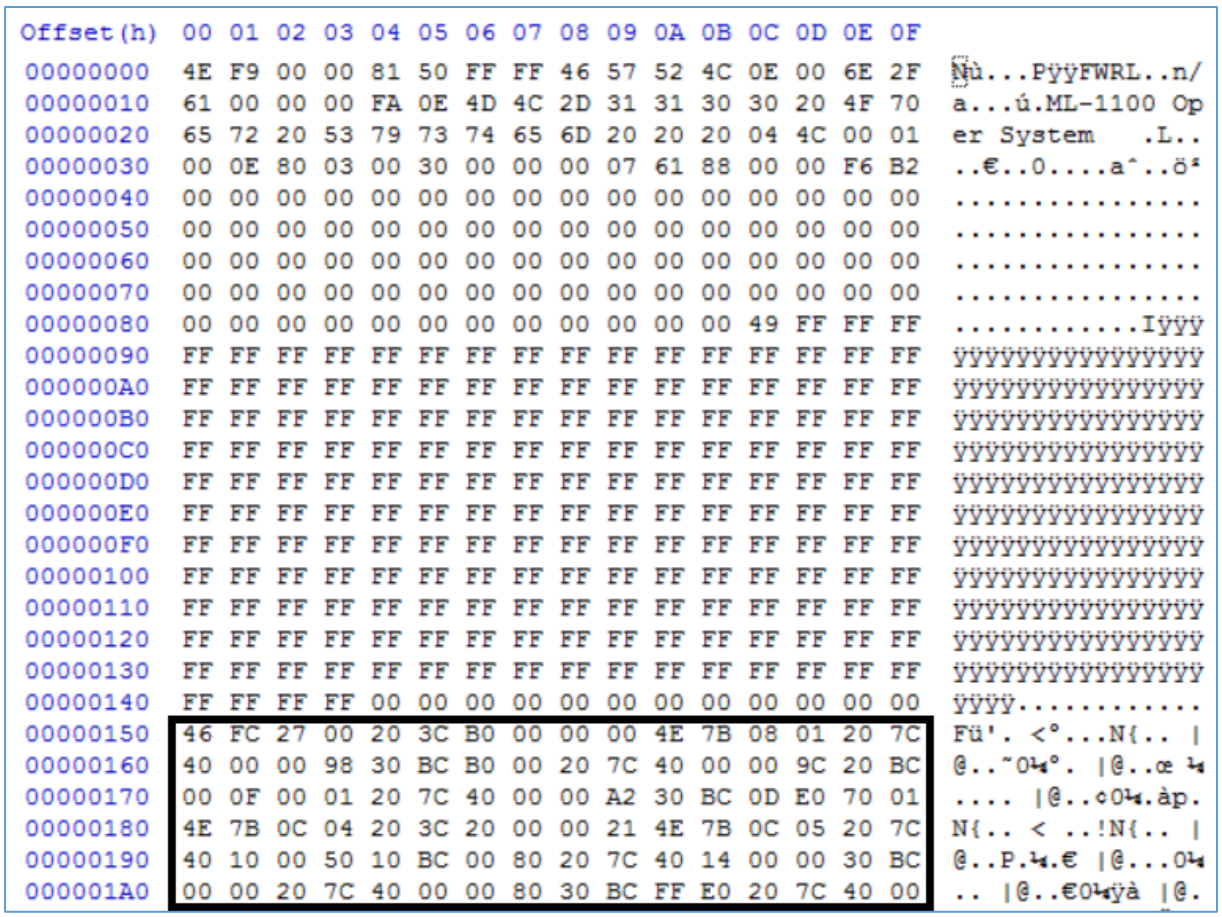
By changing only these bytes the firmware will not be accepted, which mean they are included in a checksum.

This gave us the ability to tweak the header. Below is a firmware with a nonexistent version 99:



Figuring out the checksum algorithm for the whole file is very problematic, since it requires guessing the fields that should be summed. Therefore we tried another approach.

First we defined where the code starts:



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	4E	F9	00	00	81	50	FF	FF	46	57	52	4C	0E	00	6E	2F	Nù...PÿÿFWRL..n/
00000010	61	00	00	00	FA	0E	4D	4C	2D	31	31	30	30	20	4F	70	a...ú.ML-1100 Op
00000020	65	72	20	53	79	73	74	65	6D	20	20	20	04	4C	00	01	er System .L..
00000030	00	0E	80	03	00	30	00	00	00	07	61	88	00	00	F6	B2	..€..0....aˆ..ö²
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	49	FF	FF	FF	.....Iÿÿÿ
00000090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000100	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000110	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000120	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000130	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000140	FF	FF	FF	FF	00	00	00	00	00	00	00	00	00	00	00	00	ÿÿÿÿ............
00000150	46	FC	27	00	20	3C	B0	00	00	00	4E	7B	08	01	20	7C	Fü'. <°...N{.. |
00000160	40	00	00	98	30	BC	B0	00	20	7C	40	00	00	9C	20	BC	@..˜0¼°. |@..œ ¼
00000170	00	0F	00	01	20	7C	40	00	00	A2	30	BC	0D	E0	70	01	.... |@..¢0¼.àp.
00000180	4E	7B	0C	04	20	3C	20	00	00	21	4E	7B	0C	05	20	7C	N{.. < ..!N{.. |
00000190	40	10	00	50	10	BC	00	80	20	7C	40	14	00	00	30	BC	@..P.¼.€ |@...0¼
000001A0	00	00	20	7C	40	00	00	80	30	BC	FF	E0	20	7C	40	00	.. |@..€0¼ÿà |@.

Then we made the following assumption:

[Global checksum] = [Header fields checksum] + [Code area checksum]

This means that patching the code area affects the global checksum. So instead of trying to find out which algorithm is needed to calculate the global checksum, we calculated once the whole code area checksum, and for every patch we applied we simply modified a few bytes in order to make sure the new checksum has the same value as the original.

This process allows us to maintain the same checksum as before, and to successfully upload the patched firmware image.

Dumping the memory

In order to achieve our goal of dumping the memory, we have chosen to modify an HTTP page handler.

Choosing a page handler will allow us to make sure the device boots and works properly, and to trigger the execution of our code only upon the sending of an HTTP request. This method has proven itself beneficial for debugging purposes, since it allows us to make sure that in case our custom patch is not stable it will crash only at that point.

Another important issue to note about the patching is the necessity to use only a limited set of instructions. While this appears to be a standard ColdFire CPU, the instruction set lacks a large amount of instructions, probably due to optimization. This caused our custom patch to fail. In order to overcome this issue, we have made sure we use only instructions that were observed in the original image.

Patching the page handler was comprised of stripping out of all of its functionality, and replacing it with our code. This code takes the first key and value from the HTTP arguments, converts it to 2x32 bit integers, and dumps all the memory between these two resulting integers.

Reversing the HTTP server

The HTTP server is a proprietary implementation by Allen-Bradley. A noticeable issue is a strong indicator for secure code development due to the limitation of size of almost all input buffers.

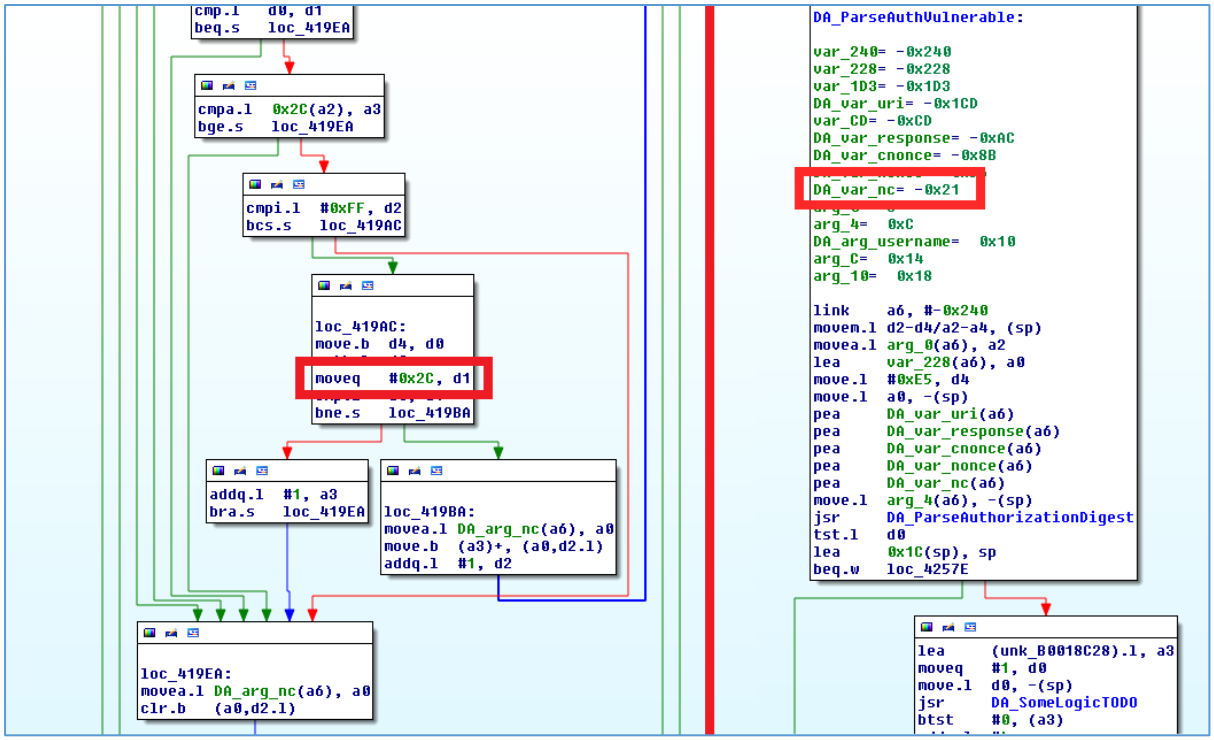
Mapping potential vulnerable functions

In order to facilitate the process of mapping potential vulnerable functions, we have decided to write IDA python scripts to map the functions under the HTTP parsing tree code that may cause a buffer overflow vulnerability.

The script that found the vulnerability did it by traversing the HTTP parsing code tree and mapping all the copy patterns in that tree.

Validating the function's vulnerability

In order for a function to be vulnerable to buffer overflow, it has to copy certain amount of data into a smaller buffer. The function we decided to focus on is responsible for parsing authorization digest header inside the authentication function.



Now that we know which function is vulnerable to buffer overflow, we can send some non-existent addresses and see how the device crashes. However, crashing the device is not enough to prove that it is possible to cause code execution.

Although there might be other ways, the easiest way to send our shellcode along with the exploit was to place it in the URI as an argument, since all of the arguments are parsed and placed into a constant address in the memory. The shellcode we have written prints the word " CyberX " as part of the PLC's menu. The images below display the PLC's menu before and after the execution of our exploit.

Before running the POC

After running the POC