Supervising the Supervisor: Reversing Proprietary SCADA Tech

Jean-Baptiste Bédrune
`jbbedrune at quarkslab.com`

Alexandre Gazet
`agazet at quarkslab.com`

Florent Monjalet
`fmonjalet at quarkslab.com`

May 29, 2015

Quarkslab

Abstract

SCADA[1] systems can be found in the core of many critical infrastructures, such as nuclear plants, water distribution circuits or alarm systems.

This article is about the security study we carried out on recent, proprietary and state-of-the-art SCADA technologies. We will mainly focus on the methodology followed to reach our goals, as well as on some techniques we used. This will cover fuzzing, black-box and white-box reverse-engineering. We will talk about the reverse-engineering of the industrial protocol, a part of the protocol stack of a SCADA supervisor and the PLC[2] firmware.

This study revealed multiple security breaches in the assessed technologies and allowed to reveal most of this protocol's cryptographic system. To illustrate one of these vulnerabilities, we will present an effective attack.

Disclaimer: The terms used to refer to the studied technologies are deliberately imprecise.

1 Introduction

1.1 Motivations

In response to some recent attacks on critical industrial systems (for example Stuxnet[8], that managed to put out of order uranium centrifuges in Iran, or more recently Havex, a RAT targeting SCADA system), some vendors have done, and are still doing, a lot of efforts to raise their products' security and resilience against malicious attacks.

The history of (in)security in SCADA systems makes the analysis of new, more secured systems very interesting.

[1]Supervisory Control And Data Acquisition, see section 1.2 for more information.
[2]Programmable Logic Controller

1.2 What is a SCADA System?

Before getting to the main topic of this article, this paragraph quickly presents what SCADA systems are and the purpose of some of the components we will be dealing with in this article.

1.2.1 Definitions

The SCADA acronym refers to a subset of what is called “industrial control systems” (ICS).

An ICS is an information system that is aimed at controlling physical systems: sluice gates, motors, temperature or pressure sensors, etc. They can notably be found in power plants, water distribution circuits, alarm or access control systems, video monitoring, etc. Actually, many systems can fit into this definition.

The SCADA part refers to the monitoring and control of the physical process.

1.2.2 Components

The SCADA part of an ICS is mainly composed of three elements:

Programmable Logic Controllers (PLC) These devices embed a program transforming electrical inputs into electrical outputs (for example adjusting a rotation speed according to a temperature). These devices usually contain ARM or MIPS type CPUs, and sometime have a real OS installed. The way outputs are computed from inputs is defined by a user program that can be modified at any time. It is the main difference with a more classical electronic circuit.

Programming stations used to create and download the user program on the PLC.

Supervisory stations (also called HMI, for Human Machine Interface) are meant to monitor the state of the physical process and to trigger actions (such as closing a sluice gate, change the rotation speed of a motor, etc.). It is a kind of abstraction of the underlying physical process. The monitoring and actions are done through the reading and writing of state variables stored in the PLCs, thanks to a dedicated industrial protocol.

These stations often consist in a graphical interface on which measures and figures will be shown, and on which some buttons will allow to trigger actions or adjust physical settings.

1.2.3 Particularities

Industrial devices, and particularly PLCs tend to be in function for many years (some can run for 30 years). They also are very expensive and complicated to replace: the devices themselves are quite expensive, but changing them also means reprogramming all the new ones and integrating them to an existing system, or rebuilding it from scratch. One of the consequences of this phenomenon is to find a lot of old equipment in industrial systems.

When they are running, some ICSs (particularly the most critical ones) are very delicate to update. When there is no complete duplication of the system, it has to be stopped for update. When the duplication degree allows to remove

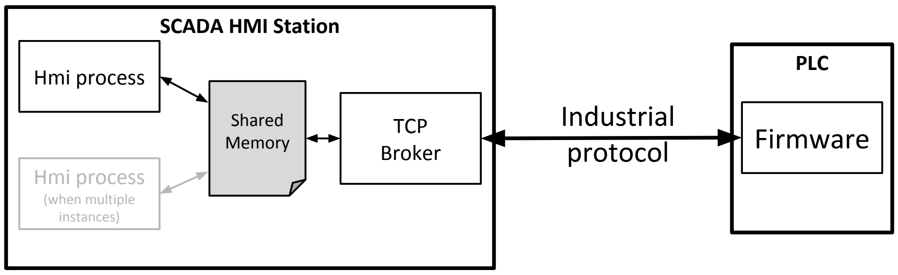

Figure 1: Simplified SCADA Architecture

some components of the system without stopping it, it is still necessary to ensure that the update does not disrupt the system. A simple update on these systems can become a very delicate problem, and it is therefore quite common to encounter industrial systems that are not kept up to date, even for major security updates.

Because of their critical nature, even a denial of service on one PLC can result in a serious loss (of money, products or gear). In general, the priority is to avoid any situation that could modify the usual behaviour of the device; authenticity and integrity are often more important than confidentiality of data or communications in this kind of network. These constraints in term of stability and security contrast with the field reality: often old and not up to date equipment. This is why the robustness and stability of industrial equipment is a priority.

1.3 The Target of Study

We chose to focus on a popular industrial equipment vendor, mainly selected for its market share in Europe and its efforts to raise its product security.

We studied one of their most recent PLC, their SCADA (supervision) software (referred to as HMI in what follows) as well as two versions of their proprietary industrial protocol. Figure 1 shows how these elements are organized.

A first PLC network scan reveals that only two TCP services are exposed: a web server (HTTP and HTTPs, both being configurable and deactivable) and a TCP server for the industrial protocol. This type of protocol allows the supervision (client side) and the PLC (server side) to exchange information as well as reprogramming the PLC. It can be carried over TCP or various serial buses.

Many vulnerabilities that were published for this kind of equipment are related to the embedded web server, however very few directly affect the industrial protocols (and even less when they are proprietary). The advantage of directly looking at the industrial protocol is that it is highly improbable (and often impossible) to disable the associated service: without it, the PLC becomes much less interesting. In contrast, the web server is totally optional (although quite useful) and can be disabled. For example, on the PLC we studied, it is disabled by default.

This is why we chose to focus on the industrial protocol, here carried on TCP. There are two versions of it:

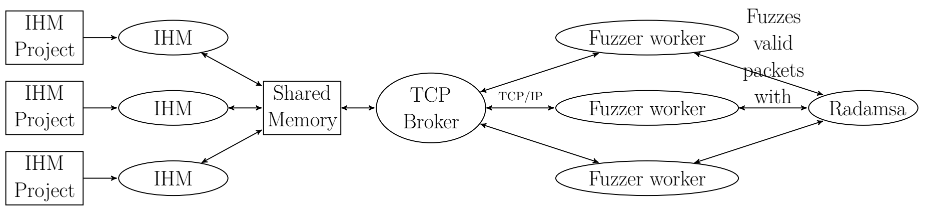

Figure 2: Fuzzing architecture

- the older one, that had no (or almost no) security mechanism: this one is rather well known by the community;
- the newer one, on which we focused the most, that implements some actual security mechanisms (that will be studied later), but almost no public resource is available.

2 Fuzzing a Poorly Known Protocol

In order to get familiar with the products of the vendor we chose, we decided to begin by implementing a part of the older protocol and then fuzz the HMI with a "mock" PLC (a Scapy program) that speaks this protocol. The latter has the advantage of being relatively well known by the community, which greatly facilitates its implementation.

2.1 Fuzzing Architecture

As said before, the goal is to act as a fake PLC to fuzz the HMI, that is to say sending it malformed packets, hoping to trigger an unexpected behaviour.

The fuzzing architecture in itself (shown in Figure 2) is relatively simple:

- Multiple HMI processes run on a VM.
- Multiple workers of a same fuzzer run on another VM.
- The HMI being the client of the communication, we have to ensure it connects to the fuzzer. This is done by configuring the HMI project files accordingly.
- The fuzzer is a Scapy legitimate client implementation, whose payloads are fuzzed with `radamsa`[1].

That said, a trick is involved here: the HMIs do not directly communicate with the fuzzer. If it does not prevent from analyzing a crash provoked, it is however way harder to retrieve the network capture that caused it. Indeed, the TCP Broker do not offer any documented (or easy) way to know which HMI instance uses which TCP connection.

To solve this problem, we had to find out how to associate a network communication to an HMI process PID. We did not find any way to change the destination TCP port of the communication (it cannot be changed in the HMI project) and the source port does not give any information on the PID that uses the TCP connection (because of the TCP Broker).

A trick based on IP aliasing has finally been used: each project is configured to connect to a different IP address[3] and the workers are configured to listen on all these IPs on the same network interface. With this solution, the script starting and catching the HMIs crashes on the client side generates HMI project files with different IPs, and is able to associate an HMI PID with the IP of the project that was passed as an argument to it. It then asks to the fuzzer the last capture associated to this IP address. This capture is necessarily the one that made the HMI crash, since all the HMIs started at a given moment connect to distinct IPs.

This allowed us to put up a fuzzing architecture where resources of each VM are used at their best and where it is possible to run n VMs containing i HMI processes each and m VMs running j workers, for all m, n, i and j such as $n * i = m * j$.

2.2 Results

After a very short time (around 5 minutes of fuzzing a single process) the first crash occurred.

It was caused by an improper check on a bound, allowing to access an arbitrary offset in an object table. A virtual method of the accessed object is then called by the program. The crash happens during the negotiation part of the protocol, in the first packets.

Whether this bug is exploitable or not has not been confirmed yet, but it can at least make an HMI crash with 3 packets, without any kind of authentication necessary. In this protocol, the HMI is the client and the PLC the server. To perform this attack, it is necessary to succeed in intercepting a TCP session establishment and answer in place of the PLC with the spurious packets. That said, another manipulation allows to easily *reset* the TCP connection, making things a lot easier.

We since reported it to the vendor and it has been patched by the vendor.

3 Reverse-engineering of a proprietary protocol

Once we got a grasp on the architecture, we chose to start the study of the new version of the protocol. This one offers password based authentication giving access to different levels of privilege on the PLC (read/write/reprogram, in a rough approximation). Very few public work exist on this version of the protocol; it seems interesting to audit it to ensure that it is as secure as it is meant to be.

3.1 First Approach: Black-Box Analysis

The studied protocol is not text based. Facing an unknown protocol and many megabytes of DLLs, we choose to start the study using black-box analysis. In practice, it means we studied the content of network captures, specially crafted using the PLC and the IHM to exhibit specific features of the protocol.

[3]The PLC IP address offset can easily be retrieved (hence changed) in an HMI project file.

The approach we used relied on differential analysis (inspired by [7] and [2]) Thanks to a tool that allows to realign the hex-dumps of packets (`hexlighter`[4]) and to highlight the differences between two consecutive packers, we learn a lot on the protocol's structure.

Let us take a test sample where the supervision is supposed to read all the PLC's input and output bits, while we make them vary by changing the input voltages. In this example, we will only look at the answers from the PLC to the supervision.

Hexlighter allows (amongst other features) to color the hex-dumps in a way that differences stand out. The result is hardly readable on these printed pages, thus a graphical representation replacing each byte by a colored square has been proposed. Figure 3 shows the result of the technique by filtering only the answers fro the PLC:

- Each line represent packet's payload (roughly 110 bytes here)
- A white square is used if the byte is identical to the same one in the previous line (packet).
- A black square represent a lake of value (for example when a line is shorter). They are not used in this example.
- A green square represent a difference with the previous line (packet).
- The brighter the green color is (a gray in a paper print), the greater the difference is (in absolute value).
- In the bottom figure, each line is compared to the first one rather than to the previous one.

With the representation, many information clearly stand out:

- A stable state (the packet's form does not vary anymore) is reached at the fourth packet sent (these packets are not shown on the figure for clarity purpose). We can deduce that the first three packets may be part of a sort of handshake, the other ones are the communication's body. Let us forget the handshake for now.
- A sequence number is clearly visible at offset 12 of all the packets (the size of this field will be verified by letting the value increase until it is reset to 0). On figure 3, it takes the form of a vertical line at offset 12 (slightly shaded on the bottom figure as the difference with the sequence id of the first packet grows with time).
- A second sequence number appears at offset 73. This one grows by steps of 2; thus the gradation more pronounced on the bottom figure.
- At the end of each packet, a 32-byte field with high entropy differs greatly for all packets. This field strongly suggests the presence of cryptography, possibly a HMAC or a hash.
- One byte (actually a single bit) varies in the middle of some packets. These variations are coherent with the I/O variations applied to the PLC during

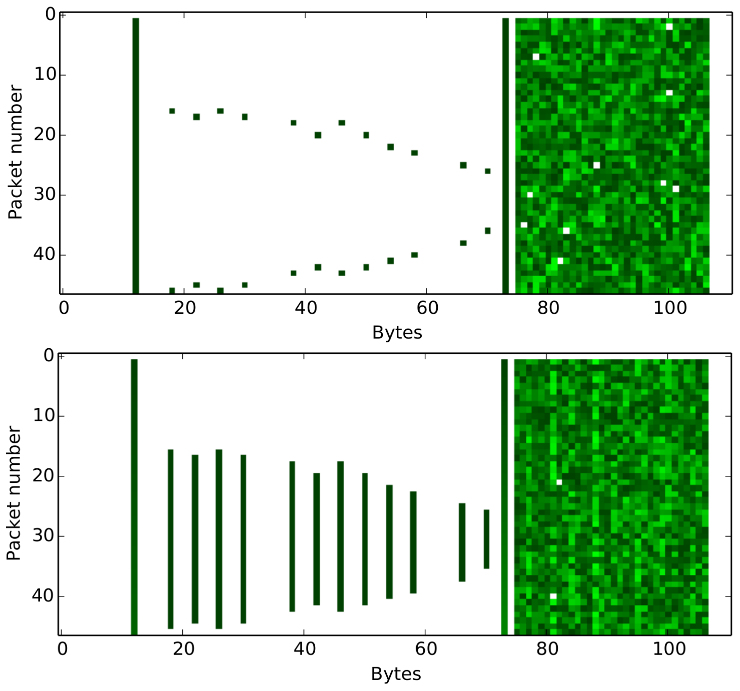

Figure 3: Representing differences between successive similar packets
Each line represents one packet. A the top, difference with the previous packet, at the bottom, difference with the first packet.

> the capture; we deduce that these fields represent the PLC variables read by the supervision. The green lines in the middle of packets on the bottom figure represent the time lapse where a variable value is different from its value in the first packet.

Looking closer and comparing the packets in different ways (for example the fifth packet from multiple different connection), one can deduce the semantics of many fields. Each parameter change (write instead of read, value modification, new session, etc.) tends to reveal its impact in the packets *diffs*. This method's key to success is to be able to easily associate a parameter change with its impact in the network traffic.

3.2 White-Box Analysis

Black-box analysis have limitations; it allowed us to quickly identify and recover the semantics of interesting fields from the protocol; however some questions cannot be answered without a step of reverse engineering on the binaries that implement the protocol. This is especially true for the high entropy 32 byte field; for now we can only suspect a HMAC or a hashing algorithm is implied. Our main objective is to discover its origin and how it is generated.

The easiest way is to start the analysis of the supervision software; it runs on a well-known, easy to instrument (contrary to the PLC), x86 Windows platform.

The first issue we encountered was the complexity of the whole program: multiple processes communicate through shared memory pages, they are heavily multi- threaded, rely on asynchronous events, and embed dozen of DLLs whom names are often not really evocative.

3.2.1 Tracing the Data

The first naive approach we used was to trace data going up from the `recv` system call up to the verification of the field we were interested in. The many copies of the read buffer, the way they are exchanged between the different processes and their asynchronous nature make both static and dynamic analysis difficult. We did not spent to much time on this method as it turned out to be quite inefficient. Nevertheless, it allowed us to discover many interesting code paths and to identify the specific process responsible for the verification of the suspect field (not yet the specific function though).

Tracing the whole program, or even the whole system and then using tainting could have been an interesting approach, however it required heavy instrumentation and we can find shortcuts.

3.2.2 Detecting Cryptographic Algorithms

We can make an educated guess that the 32 bytes we are interested in imply, one way or another, some sort of hash algorithm; may it be for message integrity (however a size of 32 bytes seems a bit disproportionate), or a HMAC-like feature used to ensure authenticity.

The usual and quick method to find this kind of algorithms is to look for the constants that characterize them in the binaries. For that purpose, we ran `signsrch`[6] on all the DLLs used by the IHM while it is connected to the PLC. One of them stood out (let us call it `hmi_core.dll`)), here is the output produced by `signsrch`:

```
  offset   num  description [bits.endian.size]
  --------------------------------------------
  xxxxxxxx 1036 SHA1 / SHA0 / RIPEMD-160 initialization [32.le.20&]
  xxxxxxxx 2053 RIPEMD-128 InitState [32.le.16&]
  xxxxxxxx 876  SHA256 Initial hash value H (0x6a09e667UL) [32.le.32&]
  xxxxxxxx 1016 MD4 digest [32.le.24&]
  xxxxxxxx 1299 classical random incrementer 0x343FD 0x269EC3 [32.le.8&]
  [...]
  xxxxxxxx 1290 __popcount_tab (compression?) [..256]
  xxxxxxxx 874  SHA256 Hash constant words K (0x428a2f98) [32.le.256]
  xxxxxxxx 894  AES Rijndael S / ARIA S1 [..256]
  xxxxxxxx 897  Rijndael Te0 (0xc66363a5U) [32.be.1024]
  xxxxxxxx 899  Rijndael Te1 (0xa5c66363U) [32.be.1024]
  xxxxxxxx 901  Rijndael Te2 (0x63a5c663U) [32.be.1024]
  xxxxxxxx 903  Rijndael Te3 (0x6363a5c6U) [32.be.1024]
  xxxxxxxx 915  Rijndael rcon [32.be.40]
  [...]

- 18 signatures found in the file in 7 seconds
```

The main suspect SHA-256 is found, as well as other hash and encryption algorithms (especially AES, we will come back on this later).

Setting up a breakpoint on the `sha2_process` function of the SHA-256 implementation in `hmi_core.dll`, one can verify that this function is actually called during the communication, and is even called each time a packet is sent or received.

Back to static analysis, one can find the SHA-256 functions are used to compute a HMAC. Debugging the program gave us further information, the packet's payload is passed as input to the HMAC algorithm alongside what can be called a session key of 24 bytes. Moreover, for each packet, the HMAC output matches wit the 32 bytes with high entropy located at the end of the packet.

Using a HMAC for each packet allow to ensure packet authenticity as the key is specific to the session and exchange in a secure (confidential) way. We have made an important progress here, now the question is where does this key come from? How is it exchanged?

3.3 Uncovering the Cryptosystem

3.3.1 Session Key Generation

Luckily for us, finding the generation of the session key was a matter of few breakpoints on write to follow the interesting buffers.

Identifying the origin of the session key, generated by a PRNG, unveiled a first major issue: the output of the PRNG doesn't seem random at all. Indeed, the PRNG is not seeded with a proper entropy source; contrary to what is expected, the seed is a constant buffer. These findings are later confirmed by the fact that the sequence of generated session keys is always the same for a given execution of the supervisory software. In this situation it becomes possible to brute-force this key in a very short time-lapse (we simply generate the session key sequence) and then to steal an already authenticated session thus completely bypassing the password-based authentication. An implementation of this attack is proposed in section 4.

3.3.2 Session Key Exchange

The attack based on the biased PRNG is an interesting first step however it is an implementation error, quite easy to fix, thus let us continue to analyze the cryptosystem. The next question is how the HMI provides the session key to the PLC.

Back to the network capture analysis, we quickly identify the packet that seems to contain the session key. Here is the logic:

- The first packet that is sent by the supervision is always the same (except for the sequence id) and is sent before the generation of the session key.
- The PLC answer is constant, except for 3 scattered bytes and a block of 20 bytes that are always different
- The second packet from the supervision always vary in a signicative way from one session to another. A block of 132 bytes is the most interesting; it's the only block that is large enough to contain the session key (either it is encrypted or encoded).
- The PLC answer is only made of an applicative ack.
- The third packet sent by the client (supervision) is authenticated with a HMAC.
- The same is true for the third packet sent by the server (PLC).

It seems obvious that the session key is actually exchanged in the second packet sent by the client.

As a reminder, an implementation of AES is present in `hmi_core.dll`. No asymmetric encryption algorithm being detected in the DLL (at least based on simple constants matching), AES thus becomes a candidate of choice for exchanging secrets. Debugging the HMI process one can confirm that AES related functions are actually called when the suspicious second packet is forged.

Back to static analysis, we have been able to reconstruct the mode of operation from the implementation: it is an AES 128 GCM (close to a CTR mode for the sake of simplicity). The idea behind CTR mode of operation is to *xor* the clear text with a *keystream* that is generated by deriving the IV to get an arbitrarily long bytes stream (the *keystream*), each derivated IV being then ciphered by block of 128 bits (for AES 128).

With the knowledge of the key (it can be dynamically recovered by debugging the HMI) we are now able to decipher the session key used for a given communication. Thus one can verify that the session key is indeed ciphered with AES 128 GCM (alongside other data) and the encrypted data are the last 72 bytes from the larger 132 bytes field previously identified.

3.3.3 First Shared Secret Exchange

For the record, here is what we know:

- The client of the protocol is authenticated with a password (the mechanism will not be presented in this article but is considered so far as reliable).
- A HMAC associated with a session key ensures that the client that authenticated is the one that is sending the packets (starting with the third one)
- A session key is generated on supervision side, then encrypted with AES and sent to the PLC.
- How does the PLC retrieve the AES key necessary to decipher the session key?

We reiterate over the same method as previously explained: mixing static and dynamic analysis we quickly isolate the part of the protocol that could be in charge of that exchange: the suspects are the first 60 bytes from the larger 132 bytes field previously identified.

Correlating these data with the ones manipulated by the process prior to the AES encryption, we locate the part of the code that seems to "encrypt" the key.

To our great surprise, the code that perform this encryption is heavily obfuscated; at the time this article is written, its analysis is still a work in progress.

3.3.4 Conclusion on the Cryptosystem

Figure 4 is a summary of the cryptosystem with respect to our analysis. The function *obf_enc* stands for the obfuscated encryption primitive that we still

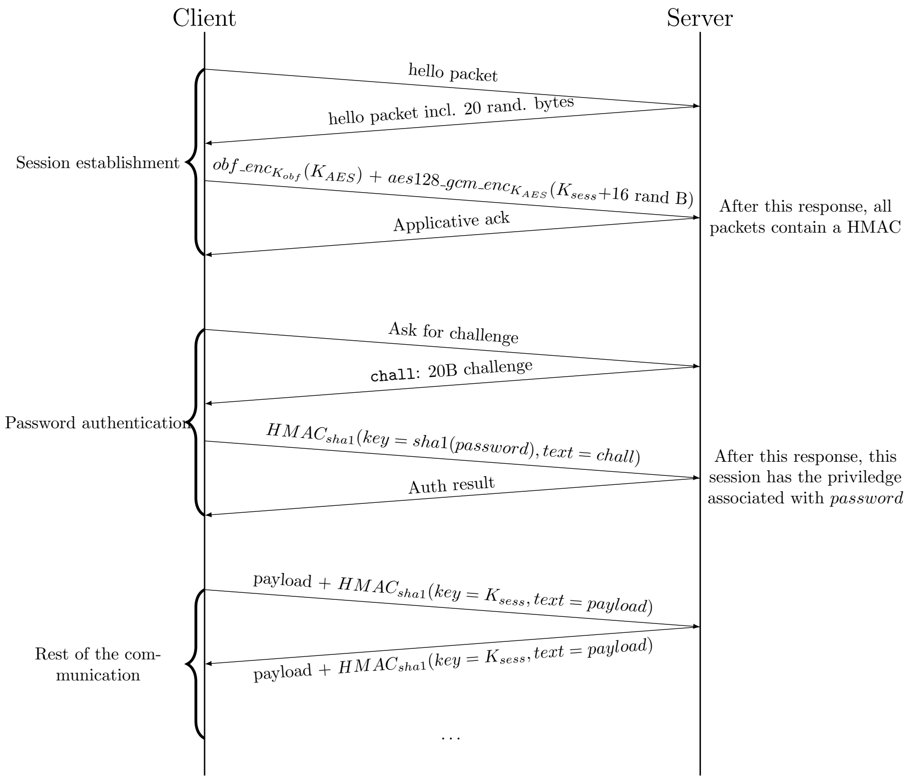
Client
Server
hello packet
hello packet incl. 20 rand. bytes
Session establishment
$obf\_enc_{K_{obf}}(K_{AES}) + aes128\_gcm\_enc_{K_{AES}}(K_{sess}$+16 rand B)
After this response, all packets contain a HMAC
Applicative ack
Ask for challenge
chall: 20B challenge
Password authentication
$HMAC_{sha1}(key = sha1(password), text = chall)$
After this response, this session has the priviledge associated with *password*
Auth result
payload + $HMAC_{sha1}(key = K_{sess}, text = payload)$
Rest of the communication
payload + $HMAC_{sha1}(key = K_{sess}, text = payload)$
...

Figure 4: Cryptosystem summary

have not totally recovered. The password based authentication mode was not yet fully examined at the time this article is written.

The cryptosystem mostly relies on standard and solid primitives that we have been able to identify. However more time is required to analyze the obfuscated part and thus give a motivated opinion on the complete system. There are still unknown parts that could lead us to think that these algorithms make use of stored secrets (maybe in the firmware or PLC's hardware) to exchange sensible data. These secrets could be common to all the PLC of a same model and with the same firmware version.

At this point, it seems sensible to look on the PLC side and more precisely look at the firmware, in particular to verify if the counter-part of the obfuscated functions was also obfuscated on the firmware. The hypothesis is that equipment with lower resources may use lighter or no obfuscation.

In case this hypothesis is confirmed, it could probably allow to easily identify which algorithm is used. That part of the analysis is covered in section 5. However before digging into that part, we will describe the attack that allows to steal an authenticated session.

4 Stealing an authenticated session with a man-in-the-middle

As explained in section 3.3.1, a lack of entropy makes the output of the PRNG predictable: the generated sequence is the same for every instance of the client. This part describes the steps which allowed us to write a code demonstrating the possibilities of this attack.

4.1 Summary of the Problem

All the security of a session relies on the fact that only the client and the server know the shared secret, that we call the “session key”. The values generated by the PRNG are totally predictable, because of a defect in the seeding process: the PRNG is always initialized with constant values. It is hence possible to enumerate all the session keys in the order they are generated.

4.2 The Attack

4.2.1 Description

From one single authenticated packet, it is possible to verify if a given session key has been used to generate the HMAC that authenticates the packet. Since it is possible to sequentially enumerate the session keys generated by the HMI, it is possible to generate and test the keys one by one in a very reasonable time. This attack allows to recover the session key that authenticates the captured packet; this session key will allow the attacker to authenticate any other packet, allowing him to perform actions with the privileges of the stolen session.

The first objective is to modify the responses to the read requests sent by the client, in order to modify the view of the state of the PLC from the client side. The second one is to write arbitrary values on the PLC with the privileges of the session, without being detected. Figure 5 shows the input and output LEDs, and a part of the supervision interface (on the top of the figure). Note that the red and green dots in the supervision screen are the LEDs of the PLC, and that their states are synchronized.

This attack needs a read-access to the network. It might be possible to get interesting results without being in a man-in-the-middle position, provided that at least one authenticated packet can be captured. This line of attack will not be studied here.

Step 1: Man-in-the-middle on the TCP Connection

In order to modify and intercept the packets exchanged between the client and the server in the TCP connection, the first step is to make all the packets transit by the attacker’s computer. This can be achieved for example with an ARP spoofing attack.

Our man-in-the-middle relies on NFQueues. Packets are read in user space, modified and sent back to the kernel. The command used to sent the routed packets into an nfqueue is:

```
sudo iptables -I FORWARD -j NFQUEUE --queue-num 1 [<various filters>]
```

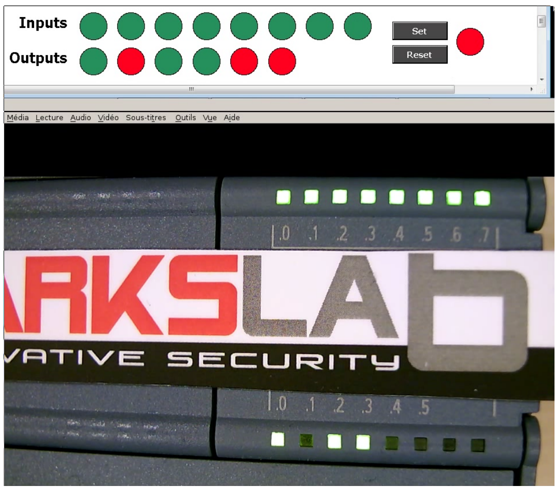

Figure 5: Preview of the inputs and outputs of the PLC, and of a part of the supervision interface (on top)

A Python binding for the nfqueue PLC has been used. This binding does not implement the `set_payload` function. To be able to implement the attack, modified packets are actually dropped by the nfqueue, then sent back using a raw socket after having modification.

As we will see later, some modifications on packets change their size, which breaks the synchronization of the `seq` and `ack` on both ends of the TCP connection. Our code, which is only a proof of concept, does not handle this problem of synchronization, hence the TCP connection is reset in some cases.

Step 2: Retrieving the Session Key

This part is quite simple. The HMAC of a packet is easy to locate: it is the 32-byte block with a high entropy seen before. The session keys are generated in the same order as the HMI does, and tested one by one. The number of sessions established by an HMI since its start-up begin reasonably low, the key is retrieved very quickly.

Step 3: Authenticating Arbitrary Packets

Knowing the session key, it is possible to authenticate arbitrary packets. The simplest way is to start from an existing packet, modify it, then authenticate it by computing the HMAC. A verification step can be done by launching the brute force on the modified packet and checking that the retrieved key is the same as the one used to authenticate the packet.

Step 4: Modifying the Responses to the Read Requests

To keep it simple, the attack will be built against a specific HMI project. It is possible to make something more generic, but it requires more time and brings no interesting improvement to the proof of concept.

For a given project, the read and write requests and their associated responses always have the same payload. To control the values received by the supervision

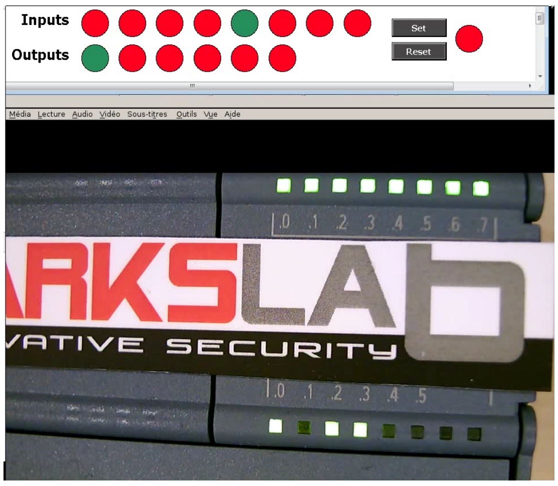

Figure 6: Desynchronization between the PLC and the supervision interface

(the client), one needs to locate in the responses the offset to the values of the PLC variables, and to set them to the wanted value. Authentication data is then added using the method described previously. Finally, packet is sent to the supervision.

This method allows us to fill our first objective: control what is received by the supervision, and hence what and operator would see on the HMI of an industrial site. An example of implementation of this attack can be seen Fig. 6: the state of the supervision interface, on top, is not consistent with the real state of the LEDs of the PLC.

Step 5: Generating Write Requests

This part is a little bit more complicated. To carry it out, the easiest way is to substitute a read request for a write request, which will be almost entirely forged. Substituting an existing request avoids forging the TCP part of the packet.

A write request that is accepted by the PLC must be written. Our existing data is a legitimate write request generated by the supervision protocol, and previously captured. To determine which fields needs to be changed to create a valid request for an arbitrary session, we can study the differences between two legitimate write requests, or between a forged, invalid packet and a valid packet. Most of these fields are easy to rebuild, because the depend either on the payload of the current packet, or on the previous packet sent by the supervision, whatever it is.

Nevertheless, one field is more complicated to forge: a sequence number unique to the write requests. If it is not consistent, the PLC will refuse the request. To retrieve this sequence number, three methods can be used:

- Be lucky enough to intercept a write request done by the HMI. Not every HMI do write requests, hence this scenario is not always reliable.
- Reset the connection: this forces the supervision to send a new password authentication request. This request happens to be a *write*-type request,

allowing us to retrieve the wanted sequence number. Note there is no password authentication, this attack is unnecessary.

- Brute force the sequence number until the PLC accepts the write request. This solution is less viable and, in some cases, less discreet.

Using a trial and error method, we managed to generate valid write requests for an arbitrary variable, for a given project. Only the modification of output variables has been implemented.

4.2.2 Consequences

Our example is restricted to supervision sessions, but can also be extended to administration sessions. It could allow an attacker to steal a reprogramming session of a PLC to upload an arbitrary program. Every communication established by a vulnerable client that uses this protocol can be compromised with this attack.

A security fix for this vulnerability has been released by the manufacturer.

This attack is quite realistic: in practice, it is often possible to gain access to an industrial system. Some of them are connected to enterprise networks or even directly to the Internet[4]. When it is not the case, they can use wireless networks (WiFi, GSM) to operate. Even when these attack vectors are not available, another event might make this attack possible: for example, a connection from a maintenance agent's laptop. These networks are most of the time flat, what makes the attack practicable.

5 Accessing the Firmware

The firmware can be downloaded from the manufacturer's website. A valid account is necessary to access the download page.

The most recent firmware available for this PLC model was the one already installed on the PLC.

The firmware format is specific to the manufacturer. It is mainly composed of a header, which contains a table of sections, followed by the content of each section. A CRC-32 of the content of each section is present in this header.

A section named `A00000` contains the whole code of the PLC. The last section, `FW_SIG`, contains a signature.

5.1 Unpacking the Firmware

The `A00000` is fully compressed. The compression algorithm is *a priori* unknown, the unpacking code being available only after decompression... During a firmware update, the PLC checks the firmware signature, unpacks the code of the section `A00000` and writes in a Flash memory.

As the Flash memory could not be dumped easily, the compression function had to be studied in black box.

Some blocks have been particularly interesting for the study, as they started by data whose unpacked version was known: it was mainly pages from the Web server of the PLC (HTML or CSS files).

[4]See [5] for an internet scan of Modbus (another industrial protocol) speaking devices)

Some data in the packed firmware, including pieces of Web pages, seem to be stored in plain text. However, a null byte is inserted every 9 bytes:

```
00 04 3C 68 74 6D 6C 3E 3C  ..<html><
00 62 6F 64 79 3E 0A 3C 74  .body>.<t
00 61 62 6C 65 20 63 65 6C  .able cel
00 6C 73 70 61 63 69 6E 67  .lspacing
00 3D 22 31 30 22 3E 0A 00  .="10">..
```

Thus, data seem to be stored in 9-byte blocks. Sometimes, the text is partially stored in plain text. It these cases, this is not a null byte which is inserted, but for example 41 or 10, as in the lines 2 and 4 of the following example:

```
00 3C 6D 65 74 61 20 68 74  .<meta ht
41 74 08 22 63 6F 6E 74 03  At."cont.
00 2D 74 79 70 65 22 20 63  .-type" c
10 6F 6E 74 03 3D 22 74 65  .ont.="te
```

The plain text is easy to guess: `<meta http-equiv="content-type" content="te`. A few bytes have been replaced:

- `p-equiv=` has been replaced by the byte 0x08 at line 2;
- `ent` has been replaced by the byte 0x03 at lines 2 and 4.

The original bytes have been replaced by their length!

The lines where bytes have been replaced do not start by a null byte, but by 0x41 or 0x10. This byte is actually a mask: the i^{th} bit of this byte indicates if the i^{th} byte of the block has to be copied, or if it is a length. Thus, a mask of $0x41 = (1 << 6) + (1 << 0)$ means the bytes $2 = 8 - 6$ and $8 = 8 - 0$ contain lengths.

The length of any compressed block can now be determined. Moreover, part of the data can be unpacked. For example, the following example:

```
40 73 09 68 61 6E 64 68 65  @s.handhe
00 6C 64 2C 20 6F 6E 6C 79  .ld, only
04 20 73 63 72 65 09 61 78  . scre.ax
```

will be decompressed in something like: `sXXXXXXXXXhandheld, only screXXXXXXXXXax`, where the bytes set to `X` are unknown.

This is strongly reminiscent of a LZ compression, where a dictionary is built as the decompression process runs.

How to retrieve the missing bytes knowing only their length? Conventional LZ algorithms encode the length and the distance to the block to be copied. Here, only the length is encoded.

A rather tedious manual analysis showed that whenever a length was found, the four bytes preceding it were already present in the data already unpacked. Hence, the previous occurrence of “scre” was already followed by a string whose 9 first bytes were the same as the expected plain text:

```
media="handheld, only screen and (max-width: 767px)
```

Unpacked data will then be:

```
sXXXXXXXXXhandheld, only screen and (max
```

This was a good progress. What we did was, every time a length was found, to search for the 4 previous bytes in the already unpacked data and copy the bytes following them. However, some unpacked data using this method was not consistent.

We made the hypothesis that, for performance reasons, the unpacking algorithm was not looking back the 4 bytes preceding a length each time a length was found, but that a hash table was built during the unpacking process. This assumptions was actually correct. The problem is that, to limit the memory usage, the hash table has a small size. Various inputs produce the same hash entry and collide, which leads to problems in the unpacking process if we don't know the hash algorithm used to compute the indexes of the hash table. There is then a cascade of errors and, at the end, the output data becomes completely wrong.

As the way to build the hash table is unknown, we were in a dead end. The solution was given by reading a list of various public LZ algorithms. The WikiBooks page on data compression [9] has been really useful: it mentions the LZP algorithm. It is exactly this algorithm, more precisely LZP3, which is used in the firmware.

This algorithm is detailed in [3]. An implementation has been developed. The firmware was then fully unpacked. The end of the unpacked section contained a CRC-32 of the whole section, which allowed us to ensure the output data was correct.

5.2 Firmware Signature

Access to the whole code of the firmware was essential to understand the signature mechanism. A static analysis showed that ECDSA-256 was used. In order to avoid parsing problems, particularly when reading the section table, a simple solution is used by the vendor: all the firmware is signed, except the 78 last bytes, which contain a fixed size signature. Data in the header of the `FW_SIG` section are actually not taken into account to verify the signature.

The curve and the generator used for the signature are standard ones (ANSI X9.62 P-256). The hash function is SHA-256.

The integrity mechanism of each section relies on a CRC-32, which is not secure from a cryptographic point of view. Nevertheless, no exploitation is possible if the global signature of the firmware is not circumvented. The implementation has been thoroughly studied. No vulnerability has been found on this mechanism.

5.3 Memory Layout

The format of the resulting firmware is unknown (no ELF, no bFLT, etc.); it seems to be a monolithic binary. In order to understand it correctly, having an idea about the memory mapping of the binary would be perfect.

During the examination of the binary, a table of sections with a rather simple format was identified. It can be found by looking, for example, for the string `.text`, which is the name of the code section. Here is a readable dump of the section table:

```
Section name         Address     Size        Permissions    Unknown
.exec_in_lomem       0x0         0x7f24     -- --x (0x1 )  0x0
.bitable             0x40000     0x40       -- --x (0x1 )  0x0
.sdramexec           0x40040     0x4d4      -- --x (0x1 )  0x0
.syscall             0x40540     0x18       -- --x (0x1 )  0x0
.th_initial          0x41040     0x2c70     i- --x (0x21)  0x0
.secinfo             0x43cc0     0x318      i- r-- (0x22)  0x0
.fixaddr             0x44000     0x0        -? --- (0x4 )  0x0
.fixtype             0x44000     0x0        -? --- (0x4 )  0x0
.text                0x44000     0xe490f0   i- --x (0x21)  0x0
.rodata              0xe8d100    0x3f4a6c   i- r-- (0x22)  0x0
.data                0x1281b80   0x27114    i- rw- (0x2a)  0x0
.bss                 0x1e01040   0x7ce53c   -? -w- (0xc )  0x0
.uninitialized       0x3641040   0x394247c  -? -w- (0xc )  0x0
CLSI_CACHED_MEM_POOL 0x6f834c0   0x0        -? --- (0x4 )  0x0
.dram_uncache        0x7ff0000   0x0        -? --- (0x4 )  0x0
MAP_MAC_MEM          0x7ff0000   0x494      -? -w- (0xc )  0x0
.iram0               0x10030000  0x7aa0     -? -w- (0xc )  0x0
.iram1               0x10040000  0xc35c     -? -w- (0xc )  0x0
.crctable            0x1004f400  0x400      -? -w- (0xc )  0x0
.softboot            0x1004f800  0x700      -? -w- (0xc )  0x0
.bootinfo            0x1004ff00  0x1c       -? -w- (0xc )  0x0
.dtcm                0x10010000  0x2a00     -? -w- (0xc )  0x0
```

The `Unknown` field is always set to zero, but is present in each entry of the table.

The `i` permission means "initialized when loaded in memory": this is data that matches parts of the binary directly loaded into memory. This information is merely empirical, and the first section does not fit into this case: it contains firmware code but has no permission `i`.

The role of the `?` permission is unknown. It seems that it could allow a certain kind of reading when `i` is not present. It could mean "allow reading of uninitialized data".

Here is how the firmware is loaded into memory:

- The first 0x40 bytes are a header, *a priori* not mapped.
- Bytes 0x40 to 0x7f64 (0x40 + size of `.exec_in_lomem`) are loaded at address 0.
- Bytes 0x7f64 to 0x8040 are ignored.
- The other sections are loaded in order, without ignoring any byte. Thus, the following sections are loaded from the firmware:
  - `.bitable` (a header)
  - `.sdramexec`
  - `.syscall`
  - `.th_initial` (code initializing the BSS, among other things).
  - `.secinfo` (section table)
  - `.text`

  - `.rodata`
  - `.data`

  All the other sections are *a priori* left uninitialized. The `.bss` section will be, for example, initialized at boot time by the code located in `.th_initial`.

Maybe some of these sections are related to physical devices. It must be noted that the firmware frequently dereferences unmapped addresses, according to the section table, looking like code accessing hardware components. These addresses are often very close to `0xffffffff`.

The beginning of the data is a header describing the memory layout of the PLC. It is immediately followed by the code. The code of the PLC is now fully available for a static analysis.

Unfortunately, the obfuscated algorithms embedded in the supervision software were also protected on the PLC.

6 Conclusion

The approach we followed to reverse engineer an industrial system has been explained in this paper. The fuzzing part allowed us to get familiar with the architecture and to identify interesting features in the involved technologies.

The black-box reverse engineering part gave us quickly enough information to target the key elements which needed a deeper analysis. In this last step, much of the authentication system has been understood, a vulnerability has been identified and exploited. Finally, the analysis of the firmware gave us new horizons to look for other vulnerabilities on the PLC.

Thereafter, this information will allow us to have a better understanding of these technologies, or to make more specific fuzzers by knowing all the data formats used by the protocol.

Unlike what has been previously seen, the company which manufactures the PLC made a significant effort to secure its devices, even if there is still some way before reaching a confidence level close to what can be found in classical information systems. There have also been very reactive at fixing the vulnerabilities we identified. Things are moving in the right direction.

References

[1] Aki Helin. Radamsa. `https://code.google.com/p/ouspg/wiki/Radamsa`.

[2] Aleksandr Timorin. SCADA deep inside: protocols and security mechanisms. `http://fr.slideshare.net/AlexanderTimorin/scada-deep-inside-protocols-and-security-mechanisms-40672525`, 2014.

[3] Charles Bloom. Lzp: A new data compression algorithm. In James A. Storer and Martin Cohn, editors, *Data Compression Conference*, page 425. IEEE Computer Society, 1996.

[4] Florent Monjalet. Hexlighter. `https://github.com/fmonjalet/hexlighter`.

[5] Pierre Lalet. Scanning internet-exposed modbus devices for fun & fun. http://pierre.droids-corp.org/blog/html/2015/02/24/scanning_internet_exposed_modbus_devices_for_fun___fun.html, 2015.

[6] Luigi Auriemma. signsrch. http://aluigi.altervista.org/mytoolz/signsrch.zip.

[7] Rob Savoye. Reverse Engineering of Proprietary Protocols, Tools and Techniques. In *FOSDEM*, 2009.

[8] Symantec. W32Stuxnet. http://www.symantec.com/security_response/writeup.jsp?docid=2010-071400-3123-99, 2010.

[9] WikiBooks. Data compression/dictionary compression. http://en.wikibooks.org/wiki/Data_Compression/Dictionary_compression, 2014.