Rogue7: Rogue Engineering-Station attacks on S7 Simatic PLCs

Eli Biham[1] Sara Bitan[1] Aviad Carmel[1] Alon Dankner[1] Uriel Malin[2]
Avishai Wool[2]

[1] Faculty of Computer Science, Technion, Haifa 3200003, Israel
{biham,sarab,dankner}@cs.technion.ac.il, aviad.car@gmail.com,
[2] School of Electrical Engineering, Tel-Aviv University, Ramat Aviv 6997801, Israel
uriel.malin@gmail.com, yash@acm.org

Abstract. The Siemens industrial control systems architecture consists of Simatic S7 PLCs which communicate with a TIA engineering station and SCADA HMI on one side, and control industrial systems on the other side. The newer versions of the architecture are claimed to be secure against sophisticated attackers, since they use advanced cryptographic primitives and protocols. In this paper we show that even the latest versions of the devices and protocols are still vulnerable. After reverse-engineering the cryptographic protocol, we are able to create a rogue engineering station which can masquerade as the TIA to the PLC and inject any messages favourable to the attacker. As a first example we extend attacks that can remotely start or stop the PLC to the latest S7-1500 PLCs. Our main attack can download control logic of the attacker's choice to a remote PLC. Our strongest attack – the stealth program injection attack – can separately modify the running code and the source code, which are both downloaded to the PLC. This allows us to modify the control logic of the PLC while retaining the source code the PLC presents to the engineering station. Thus, we can create a situation where the PLC's functionality is different from the control logic visible to the engineer.

1 Introduction

Programmable Logic Controllers (PLCs) are commonly used in Industrial Control systems (ICSs) to implement process critical logic. They are the core of ICSs using equipment such as thermostats, barometers, valves, engines and generators. ICS operates critical infrastructure such as power plants, chemical plants, water treatment plants, railways, and other transportation systems that are vital to our modern life.

Since 2010, ICSs, and in particular their configuration and monitoring interfaces, have become popular targets for cyber attacks, the most well known of which is Stuxnet [10]. In response, vendors hardened these interfaces by adding cryptographic protection.

PLCs are offered by several vendors such as Siemens, Allan-Bradley, Mitsubishi, and Modicon. Each vendor has its own proprietary firmware, programming, communication protocols and maintenance software. However, the basic software system architecture is similar: the PLC itself contains variables and logic to control its inputs and outputs. The PLC code is written on an engineering station in the vendor's control logic language. The control logic is then is compiled into an executable format, and downloaded to the PLC. The operating PLCs are monitored and managed via dedicated machines running Human Machine Interface (HMI) software. Modern networked PLCs, HMIs, and engineering stations all communicate over TCP/IP, but the higher-level protocols in use are typically proprietary.

Siemens S7 Programmable Logic Controllers (PLCs) in the Simatic family [30] are estimated to have over 30% of the worldwide PLC market [9]. In addition to the PLCs themselves the Simatic line of products includes the "Totally Integrated Automation Portal" (TIA), which functions as the engineering station, and can also function as an HMI. The TIA (or HMI) and the PLCs communicate over the S7 network protocol. The most recent versions of the S7 protocol include cryptographic mechanisms to protect the communication — and most importantly, a cryptographic message integrity code, whose goal is to protect the communication from adversarial manipulation. Our focus in this research is the cryptography on which the integrity protection relies, and the attacks that it admits.

1.1 Attacks Against ICSs

Digital attacks that cause physical destruction of equipment do occur [13]. Perhaps most well known is the attack on an Iranian nuclear facility in 2010 (Stuxnet) to sabotage centrifuges at a uranium enrichment plant. The Stuxnet malware [10,11,21] used a Windows PC to attack a Simatic S7 PLCs. It targeted Siemens's Simatic S7-315 and S7-417 PLCs and worked by changing centrifuge operating parameters in a pattern that damaged the equipment—while sending normal status messages to the HMI.

More recently, cyber-attacks on ICS controlling electrical distribution have caused widespread blackouts in Ukraine [22,24]. In 2014, the German Federal Office for Information Security announced a cyber attack at an unnamed German steel mill, where hackers manipulated and disrupted control systems to such a degree that a blast furnace could not be properly shut down, resulting in "massive"-though unspecified-damage [8].

At BlackHat USA 2015 Klick et al. [20] demonstrated injection of malware into a the control logic of a Simatic S7-300 PLC, without service disruption. In a follow on work, Spenneberg et al. [33] demonstrated the feasibility of a PLC worm. The worm spreads internally from one PLC to other target PLCs. During the infection phase the worm scans the network for new targets (PLCs).

More attacks against Simatic S7 include web session hijacking [15], a remote start/stop attack [5], a replay attack [4], and man in the middle attacks [32]—all against the S7-1200 PLCs. A preliminary attempt to understand some of the cryptographic protections in the S7 protocol can be found in [23]. As we shall see, the latest versions of the Siemens Simatic products and the S7-1500 PLCs introduced stronger cryptographic protections into the S7 protocol, that are designed to prevent such attacks. Despite these defenses, we are able to reprogram these PLCs over the network from a rogue engineering station.

A different type of attack takes advantage of vulnerabilities in PLC implementations. These attacks, while significantly more powerful, have scarcely been explored. One such attack [29], targeted the Allen Bradley ControlLogix L61 PLC. It took the approach of crafting a counterfeit firmware update, and uploading it onto the PLC. It bypassed the basic update validation methods by simply updating the package's CRC checksum.

There is a large body of work focused on anomaly detection in industrial control systems (ICS). Surveys of techniques related to learning and detection of anomalies can be found in [1,2,6]. In particular model-based anomaly detection [7,35,12] has been shown to be well-suited to anomaly detection in ICS. Kleinman and Wool [17,19] demonstrated that a similar methodology is successful also in systems running the Siemens S7 protocol. More recently Kleinmann et al. [18] showed that if the communication protocol is unauthenticated (as is the case for Modbus) then stealthy semantic-level attacks can be mounted to achieve adversarial control of the plant while presenting a benign view to the operators observing the HMI.

Table 1. S7 protocol versions usage

	S7-1200	S7-1500 V1.1	S7-1500 V1.8	S7-1500 V2.1
TIA V12	$P2^-$	P2	P2	P2
TIA V14	$P2^-$	P2	P3	P3
TIA V15	$P2^-$	P2	P3	P3

1.2 S7 protocol versions

As Siemens regularly updates their software and firmware, it is important to refer to the protocol versions the various TIA and PLC firmware versions. There are currently two major versions of the S7 protocol used by the PLC S7-1500, each has many subversions that were adapted over time. Protocol version 2 (named after the value passed in the headers) is used by the older versions of TIA and PLC firmware, up to TIA V12 and PLC S7-1500 firmware 1.5.[1] In this paper we refer to the subversion used by our equipment as P2. Notice that are differences between the S7-1500 P2 subversion which includes integrity protection, and the S7-1200 P2 subversion which does not, although for both the version number in the header is 2. Hence, we denote the S7-1200 P2 protocol by $P2^-$. The newer versions of TIA, e.g., V14 and V15, and newer PLC firmware, e.g., versions 1.8, 2.0 and 2.1, support protocol version 3, which we refer to as P3.[2] Actual use of protocol version 3 requires that both TIA and PLC support this new protocol. Table 1 shows the selected protocol for TIA and firmware versions that we tested.

1.3 Our contributions

In this paper we demonstrate that even cryptographically secured ICSs are not necessarily secure. We chose the Siemens S7 architecture and in particular the S7-1200 and S7-1500 as our test platform since they are relatively common and because the vendor states that they are well protected against various attacks.

Our main contribution is the ability to create a rogue engineering station that can impersonate the TIA to the latest S7-1500 PLCs running various firmware versions, using the P3 protocol. Using such impersonation we can apply many malicious operations. In particular, we are able to:

1. Extend the remote PLC start/stop attack to S7-1500 PLCs.
2. Remotely download a replayed control logic program to the PLC.
3. We found that the S7 program download message contains two copies of the control logic: the source (uncompiled) code and the binary (compiled) code. We are able to modify each of them independently, causing source-binary inconsistency.
4. The S7 environment allows the engineering station to upload the source code back from the PLC in order to display it to the engineer. Therefore, in our most advanced attack – the stealth program injection attack – we can maintain the source program as the engineer expects to see, while programming the PLC to run a malicious (different) binary that the engineer will never see.

[1] Version 1 in the protocol header is used for the initial handshake.

[2] According to [31] the P3 protocol is also available on the S7-1200 PLCs, which use firmware versions that are more recent that the ones we tested.

Table 2. Summary of our attacks on Simatic S7 PLCs

Attack	Attack type	PLC	Protocol
Start/Stop	MitM and impersonation	S7-1200	$P2^-$
Start/Stop	MitM and impersonation	S7-1500	P2
Start/Stop	Impersonation	S7-1500	P3
Download	MitM and impersonation	S7-1200	$P2^-$
Download	MitM and impersonation	S7-1500	P2
Download	Impersonation	S7-1500	P3

These attacks are made possible by a detailed reverse-engineering analysis by which we gained insights into the S7 P3 protocol and its content, as well as the key generation and cryptographic primitives used by it. This analysis covered:

1. Understanding the structure of S7 P3 protocol and its fields.
2. Understanding the cryptographic functions used in the protocol.
3. Understanding the cryptographic handshake and the generation of the common keys.

As part of our research we also analyzed versions of the S7 protocol running on previous generations of PLCs, of the S7-1200 family. Besides the rogue engineering station attacks we described above, we found additional attacks against the older PLCs running the P2 protocol:

1. A man in the middle attack on PLC S7-1200.
2. A man in the middle attack on TIA V12 with S7-1500 PLC.

All our attacks are network based, and can be successfully launched by any attacker with network access to the PLC. A summary of our attacks appears in Table 2. In parallel to this paper submission disclosed our findings to Siemens.

We note that our main findings in themselves are not "vulenrabilities" that can be quickly patched: rather, once the obscurity of the protocol is unveiled, we find that the attacks are consequences of the cryptographic design choices used in the S7 protocol.

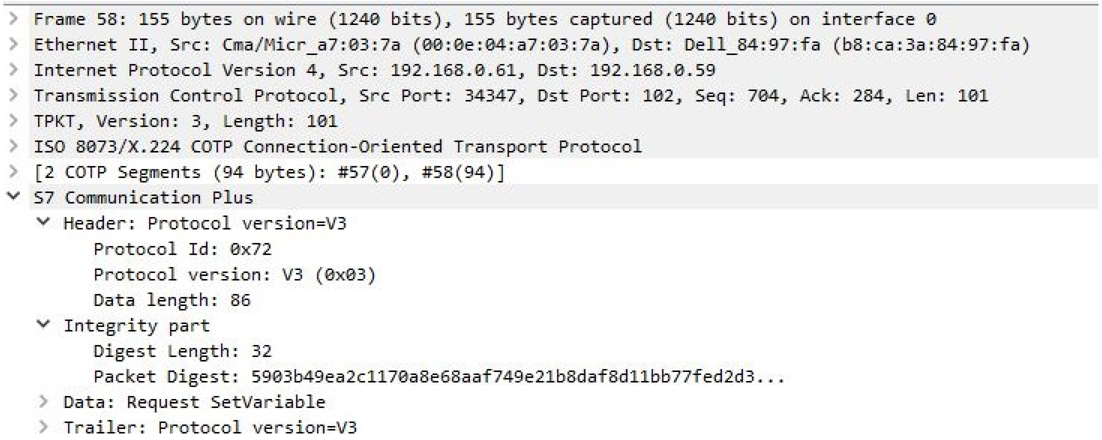
1.4 Structure of this paper

The rest of this paper is organized as follows: Section 2 describes the Siemens Simatic S7 environment and protocols. In Section 3 we describe our discoveries about the S7 use of cryptography, focusing on the integrity check, key derivation and key exchange. In Section 5 we describe our Start/Stop attack, and in Section 6 we describe the program download attack. Section 7 includes a discussion of possible countermeasures, and conclusions. Additional details are provided in the Appendices.

2 Preliminaries

2.1 S7 communication over TCP/IP

The S7 protocol is proprietary and little is published about it. Before delving into the details of our attacks on the protocol, we summarizes some of its key features. The information in this section is based on the reverse engineering work of [25][14][27][37] augmented by our own analysis of live S7 traffic.



```
> Frame 58: 155 bytes on wire (1240 bits), 155 bytes captured (1240 bits) on interface 0
> Ethernet II, Src: Cma/Micr_a7:03:7a (00:0e:04:a7:03:7a), Dst: Dell_84:97:fa (b8:ca:3a:84:97:fa)
> Internet Protocol Version 4, Src: 192.168.0.61, Dst: 192.168.0.59
> Transmission Control Protocol, Src Port: 34347, Dst Port: 102, Seq: 704, Ack: 284, Len: 101
> TPKT, Version: 3, Length: 101
> ISO 8073/X.224 COTP Connection-Oriented Transport Protocol
> [2 COTP Segments (94 bytes): #57(0), #58(94)]
v S7 Communication Plus
  v Header: Protocol version=V3
      Protocol Id: 0x72
      Protocol version: V3 (0x03)
      Data length: 86
  v Integrity part
      Digest Length: 32
      Packet Digest: 5903b49ea2c1170a8e68aaf749e21b8daf8d11bb77fed2d3...
  > Data: Request SetVariable
  > Trailer: Protocol version=V3
```

Fig. 1. The S7 packet structure as shown within WireShark. Note the unique protocol stack including COTP and TPKT, and Integrity part.

2.2 The S7 PLC platform

The Siemens Simatic S7 product line was introduced in 1995, and includes both older PLC models (S7-200, S7-300 and S7-400), and new generation PLCs, the S7-1200, and the S7-1500 (introduced in 2009 and 2012 respectively). In addition to the PLCs themselves the Simatic line of products includes the "Totally Integrated Automation Portal" (TIA), which functions as the engineering station, and can also function as an HMI, i.e., it can also be used to control the PLC, by sending commands to start or stop running, and watch or update the PLC's inner variables.

The TCP/IP implementation of the S7 protocol relies on ITOT [28] and communicates across the well known TCP port 102. S7 works on top of the ISO Connection Oriented Transport Protocol (COTP) [16,26] and TPKT [28]. Both TPKT and COTP add their own headers (inside the TCP segment). Thus the S7 message is encapsulated within the COTP packet. See Figure 1 for an example of a Wireshark view of an S7 packet.

The S7 protocol defines formats for exchanging S7 messages between devices. Its main communication mode follows a client-server pattern: the HMI or TIA (client) device initiates transactions (called queries) and the PLC (server) responds by supplying the requested data to the client, or by taking the action requested in the query.

Two different protocol flavours are implemented by Simatic S7 products: The older Simatic S7 PLCs implement an S7 flavor that is identified by the protocol number 0x32 (*S7comm*), while the new generation PLCs implement an S7 flavor that is identified by the protocol number 0x72 (*S7CommPlus*[3]). The newer S7CommPlus protocol, which is the focus of this paper, supports security features.

Throughout this paper, we use the term *message* for the messages that the S7 transmits. Long messages are typically divided into *fragments* (while short messages form a single fragment). Each fragment is then preceded by an S7 fragment header. An empty fragment (consisting of only a header) follows the last fragment of a message. The term *packet* refers to a fragment encapsulated by the required TPKT, COTP, TCP and IP headers.

[3] The name is taken from the wireshark dissector [37].

2.3 The S7comm-plus protocol overview

The S7 protocol supports various operations that are performed by the TIA. In this paper we discuss the first three operation and attack the first two:

- Start/Stop the control program currently loaded in the PLC memory.
- Download a control program to the PLC.
- Upload the current control program from the PLC to the TIA.
- Read the value of a control variable.
- Modify the value of a control variable.

All the above operations are translated by the TIA software to S7 *messages*, that are transmitted to the PLC. The PLC acts upon the messages it receives, performs the operations, and responds.

All messages are transmitted in a context of a *session*. Each session has a session ID (chosen by the PLC). A session begins with a four-message handshake used to select the cryptographic attributes of the session including the protocol version and keys. All messages after the handshake are integrity protected.

The S7comm-plus protocol also provides different flavours of integrity protection and key exchange:

- S7-1200 communication does not support integrity protection.
- S7-1500 PLCs communicating with P2 protocol use per-message integrity protection with a simple key exchange handshake.
- S7-1500 PLCs communicating with P3 protocol use per-fragment integrity protection with a cryptographic challenge-response key exchange for each session.

3 The S7 cryptographic protection

In order to protect the ICS from an adversary who gains network access to the PLC, Siemens integrated cryptographic protection into the newer version of its S7 proprietary protocol. The protection mechanisms include encryption of specific payloads (e.g., control program source and binary), authentication and integrity protection.

The message cryptographic protection mechanism consist of the following modules:

1. A key exchange protocol, that the two parties (PLC and TIA) use to establish a secret shared key, which we call the session key.
2. A message integrity protection algorithm, that calculates a MAC (Message Authentication Code) value, based on the session key and the message bytes.
3. A payload encryption algorithm.

Siemens implemented several variants of the S7 cryptographic protection. In the S7-1200 models, messages are not integrity protected. We observed that some of the fields in the S7 protocol used by the S7-1200 are encrypted, but we didn't investigate it.

On the other hand, the protocol used by the S7-1500 PLCs includes full integrity protection of all the messages exchanged between the TIA and the PLC (after the handshake).

The next two sub-sections describe the message integrity mechanisms and the key exchange protocols used by various TIA and S7 PLC firmware versions.

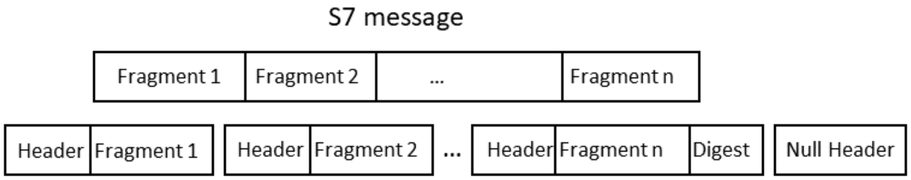


Fig. 2. S7 integrity protection in protocol P2

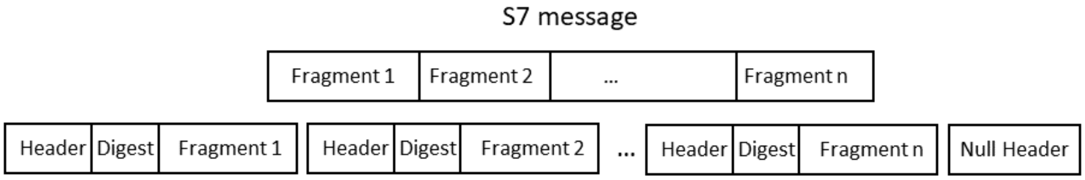


Fig. 3. S7 integrity protection in protocol P3

3.1 Message authentication and integrity protection

As mentioned above, the messages that the TIA and S7-1500 PLCs exchange are integrity protected by a message authentication code. It is calculated under a (symmetric) secret key, which we denote by $sessionKey$, shared between the PLC and the TIA.

The handshake is performed in the first four messages of a session, which perform the cryptographic session key establishment. We discuss the handshake later in Section 3.2. All the further messages in the session (starting from the fifth message) are then integrity protected. Depending on the TIA and the PLC model and version, the protocol used is either P2 or P3.

We discovered that the per-message integrity protection uses HMAC-SHA256 as the MAC, using the shared $sessionKey$ as the MAC key, using the full 256-bit output of HMAC-SHA256 as the MAC.

In the earlier (P2) version of the protocol, the MAC is placed in the integrity object at the end of the message, i.e., in the last fragment, just before the null fragment header—see Figure 2.

Notice that large messages (such as the program download messages we discuss in Section 6) are fragmented to many fragments, sent over many TCP/IP packets. Since the MAC is sent at the end of the message, it is only at the last packet of the message that the receiver can verify the integrity of the message (see Figure 2). All earlier packets cannot be authenticated (thus should not be parsed nor used) before the last packet is received.

In the newer version of the protocol (P3) integrity protection is applied at the fragment level, replacing the single MAC value at the end of the message. A cryptographic digest is placed at each fragment between the fragment header and the fragment data (see Figure 3).

Technically, the TIA (and PLC) software calls the HMAC-SHA256 API in an un-designed way. HMAC APIs (following hash functions APIs) have three API functions: *init*, *update*, and *finalize*. The intended use is that *init* initializes a *context* structure, then many *update* calls are made with all the fragments, incrementally updating the context with the message, and only at the end a *finalize* call is made (once) to extract the digest from the context. The design of such functions never intended to allow several *finalize* calls on the same message or to mix *finalize* calls between *update* calls. Therefore, most implementations let *finalize* modify the context, even though no new fragment is added during *finalize*.

We were thus surprised to find out that the integrity is computed as follows

- init
- update (with the first fragment)
- finalize (extract digest for the first fragment)
- update (with the second fragment)
- finalize (extract digest for the second fragment)
- update...
- finalize...
- update (with the last fragment)
- finalize (extract digest for the last fragment).

As the implementations of HMAC-SHA256 used by the TIA is one in which *finalize* modifies the context though it does not add any fragment, all digests but the first one are not valid HMAC-SHA256 digests. Moreover, the security proofs of HMAC do not hold for this incremental variant of HMAC. In fact, this incremental variant is less secure than HMAC-SHA256 [3].

CVE-2019-10929 was allocated to this vulnerability.

3.2 S7 Key Establishment

The P2 protocol uses a simplistic key synchronization scheme, which is equivalent to usage of a list fixed keys in a sequence. During each new handshake the next key is calculated by both parties. We observed that these keys are same and in the same order for all S7-1500 PLCs communicating the P2 protocol (at least with the PLC models that we have in our lab). Moreover, the same sequence of keys is used each time a TIA is restarted, regardless of whether it is the same TIA instance or another instance. The first ten keys are listed in Table 3. We also wrote a program that generates the sequence of the keys, by mimicking the TIA key generation algorithm. It is too lengthy to describe here.[4]

In the P3 protocol, Siemens replaced the simplistic P2 key generation process by a more sophisticated challenge-response protocol, that involves elliptic-curve public-key cryptography for the key exchange. The four-message handshake of the P3 protocol is outlined in Figure 4.

The first request message [M1] is an *Hello* message that the TIA sends to initialize a new session. The PLC responds with message [M2], which contains the PLC firmware version, and a 20-byte challenge *ServerSessionChallenge*. The PLC firmware version determines the

[4] According to [31] this vulnerability was disclosed to Siemens in the past as part of [32], analyzed as a failure in a random number generator within the TIA, and corrected in later versions of the TIA software (CVE-2015-1601).

Index	Key
1	`0xca1bd6399c718a9886a5b3ca************************`
2	`0x9bad94613b6b36ddba562a88************************`
3	`0x857a27307a932c0e0376598d************************`
4	`0x480d391238b1d59c26e8a65e************************`
5	`0x99634177751406a5ad1793a0************************`
6	`0x4a2a336fe6eac56f5fb16b14************************`
7	`0xc5129d05329cfee1b113e45f************************`
8	`0x0e7ad010269c1696aae1cdce************************`
9	`0x7135a24727104103e60f57ba************************`
10	`0x8197b43e537e66eb4a3a9818************************`

Table 3. P2 key list (first 10 keys, half of each key is kept hidden)

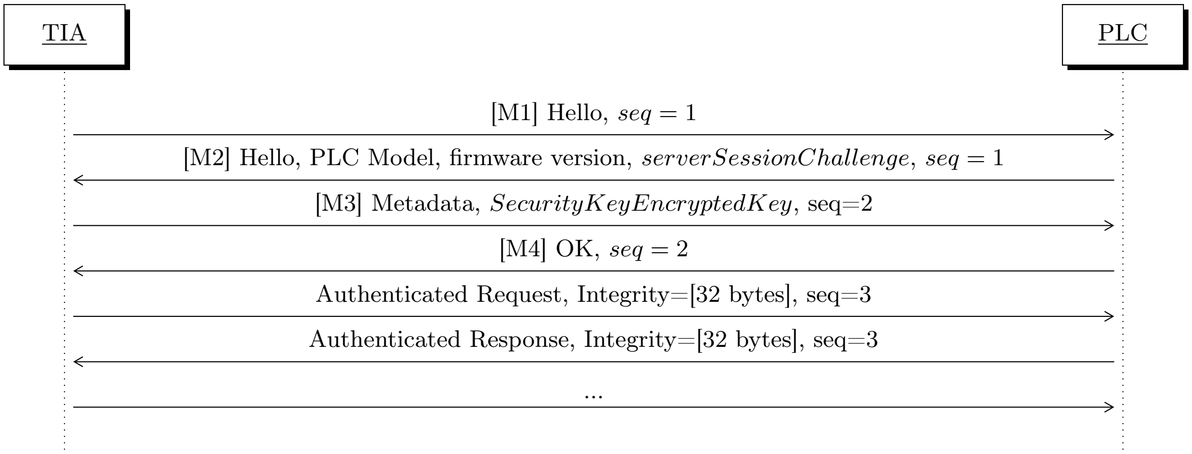

TIA
PLC
[M1] Hello, $seq = 1$
[M2] Hello, PLC Model, firmware version, $serverSessionChallenge$, $seq = 1$
[M3] Metadata, $SecurityKeyEncryptedKey$, seq=2
[M4] OK, $seq = 2$
Authenticated Request, Integrity=[32 bytes], seq=3
Authenticated Response, Integrity=[32 bytes], seq=3
...

Fig. 4. The session establishment handshake.

elliptic-curve public-key pair to be used in the key exchange. Appendix A.1.3 elaborates on this issue.

Upon receiving the [M2] message, the TIA activates a derivation algorithm to randomly select a "Key Derivation Key" (*KDK*) and to generate the $sessionKey$ from the PLC's *ServerSessionChallenge* and the *KDK*. The details of the derivation algorithm appear in Appendix A.

The third handshake message [M3] (recall Figure 4) transfers the *KDK* to the PLC, using the public-key scheme. [M3] contains two main parts:

1. A data structure which the S7CommPlus dissector [37] calls *SecurityKeyEncryptedKey*, which contains, among other things the *KDK* encrypted with the PLC's public key. See subsection A.1.2 for the exact description of the *SecurityKeyEncryptedKey* data structure and the algorithm that builds it.
2. Two 8-byte key fingerprints, of the PLC public key ID and the KDK, respectively.

The details of [M3] construction and verification appear in the Appendix. When the PLC verifies [M3] successfully it returns OK in the fourth message [M4]. All the following messages in the session are protected with the derived $SessionKey$.

3.3 S7 key exchange vulnerabilities and Implications

The S7 key exchange protocol contains several significant flaws, which we later exploit in the attacks we describe in Sections 5 and 6.

- The P3 key exchange uses one-way group authentication. A PLC of a given model and firmware version has the necessary private key and is able to successfully decrypt the *KDK*, and derive the *SessionKey*. I.e., when the TIA verifies the correct integrity protection of message #5, it authenticates the PLC. However, the PLC does not authenticate the TIA: it only confirms the session freshness, by successfully decrypting the encrypted PLC challenge.
- The design of the P3 key exchange implies that all PLCs running the same firmware version use the same public-key, and thus can impersonate each other. It seems that if some PLC's firmware is analyzed and its private key is extracted, then the security of the whole line of PLCs sharing the same firmware version will be compromised.
- There is no tethering (or pairing) of the TIA to the PLC: the PLC does not ensure that the currently-communicating TIA is the same TIA that successfully communicated with it earlier in another session.
- The P2 key exchange uses a static sequence of symmetric keys. The same keys are used by all the PLCs and all TIA installations. We extracted this key sequence, exposing the communication between the PLC and the TIA to Man-in-the-Middle attacks.

4 The attack architecture

To launch our attacks we implemented a toolbox, consisting of two types of a rogue TIA engineering station, and an S7 proxy. A rogue engineering station is a station which is controlled by the attacker, who can optionally modify its binaries to change it behaviour. Our toolbox includes:

1. A rogue station. It either builds the S7 packets by itself, or reads them from a pre-recorded file, and modifies them before sending to the PLC. We use a rogue station to implement our start/stop attacks.
2. An advanced rogue station. It consists of a real TIA connected through a man in the middle S7 proxy that modifies the packets that the TIA and the PLC transmit to each other. The proxy takes over the TIA, and receives the session key from it. Therefore the TIA is part of the rogue station. The proxy uses the key to correct the integrity protection. We use the advanced rogue station to implement our program download attack on S7 protocol version P3.
3. An S7 proxy. This proxy serves as a man in the middle between a victim TIA and a victim PLC. We use it to implement a man in the middle attack on program download of S7 versions P2 and P2$^-$.

4.1 Impersonating the TIA

Based on our understanding of the S7 key exchange we implemented a Python code impersonating the TIA. The crucial action of the impersonation code is to pick a random *KDK*, and to calculate the *sessionKey* by combining the *KDK* with the PLC's challenge. Once our code derives the *sessionKey* and successfully completes the 4-message handshake, it is able to generate a correct integrity code on any further S7 message that is required.

We implement some of the impersonation functionality by coding various functions in Python, and calling some Python libraries (such as HMAC-SHA256 and AES). In addition, we used the native DLL functions to compute the remaining fields of the key exchange message [M3]. These few functions are called through Python's *ctypes* library wraping the *OMSp_ core_ managed.dll* DLL.

Note that we could have chosen fixed values for the *KDK* and for the seed used to randomize the public-key encryption (for each firmware version's public-key). Based on these choices it is possible to calculate almost all the complex [M3] key exchange fields in advance. In this case, only two DLL functions actually need to be called after receiving the PLC's challenge: the fingerprinting function $f()$ used within the *sessionKey* calculation, and the internal Checksum function. See Appendix A for details.

5 A start/stop attack against Simatic S7 PLCs

Using our findings on the S7 protocol cryptographic protection and its vulnerabilities we succeeded to maliciously and remotely start and stop Simatic PLCs. The simplest attack is Beresford's attack [5] targeting the S7-1200 PLC, which we replicated. The more secure S7-1500 P2 protocol requires additional hacks to overcome the cryptographic authentication. Finally the latest improved cryptographic authentication and key exchange of the P3 protocol requires even further hacks. We describe these attacks in sequence.

5.1 S7-1200 with the P2$^-$ protocol

Our first step was to re-implement Beresford's attack [5] against the S7-1200 PLC using the P2$^-$ protocol, which has no device or message authentication. We describe our attack flavor here for completeness, since the more advanced attacks on the S7-1500 rely on it.

In P2$^-$ the first packet that the PLC sends to the TIA contains a two-byte random number, which the TIA uses as a session ID. The PLC verifies that this ID appears in all further messages sent by the TIA to the PLC.

In our attack, we recorded an S7 session containing a start (or stop) command. We wrote a program that reads the recorded messages and communicates with a victim PLC. The program sends the recorded packets to the PLC, whose new content is based on values transmitted by the PLC. The only required change in the replayed payload is to inject the correct session ID. The TCP and IP headers are freshly created. We use standard tools (e.g., TCPLiveReplay [34]) for this purpose.

5.2 S7-1500 with the P2 protocol

The S7-1500 P2 protocol uses cryptographic keys for authenticating messages. The content of each message is authenticated using HMAC-SHA256 under a session key which is shared between the PLC and the TIA. As mentioned earlier, we found that the S7-1500 P2 protocol uses a fixed sequence of keys (recall Table 3).

Our attack follows the lines of the attack on S7-1200, while also replacing the authentication tag by a digest that fits the modified message. Notice that in this attack the attacker can force the S7 session to always be the first, hence, it always uses the first key in the list. Therefore, the digest is HMAC-SHA256(first key in the list, modified message).

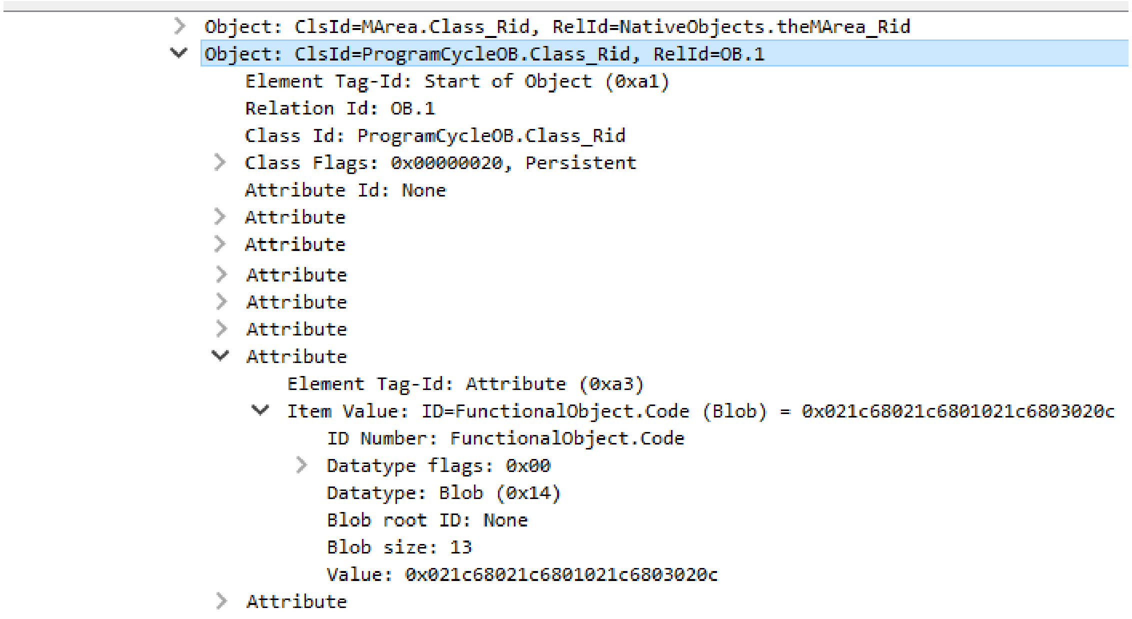
5.3 S7-1500 with the P3 protocol

In order to launch a successful start/stop attack on the S7-1500 P3 protocol one must first compromise the key exchange protocol. We exploited the fact the key exchange uses one way authentication and impersonated a TIA station. We have written a rogue TIA program, that implements the P3 key exchange we describes in Section 3.2. Once the key exchange is completed, the program transmits S7 start and stop messages that we recorded from a legal TIA session, with the appropriate modifications to the session ID and the integrity fields. The rogue TIA enabled us to start and stop the S7-1500 which communicates using the P3 protocol at our will.

6 The program download attack and the Stealth Program Injection Attack against Simatic S7 PLCs

Before describing our attacks we first discuss some aspects of control-logic program download from a TIA to a PLC. A program download is typically performed by a TIA that already knows the PLC model and its IP address. To do so the the control engineer performs a procedure that consists of the following steps:

- Create a new "project" or use an existing "project" in the TIA.
- Configure the PLC model and the PLC's IP address within the project.
- Write a control program using the TIA's dedicated GUI. The program may consist of several modules called OBs (Object Blocks). The program may also be written in one of several languages supported by Siemens PLCs (e.g., LAD and STL).
- The engineer may choose between several compilation and download options. He may decide to compile the program, or alternatively to compile and download the program to the PLC. In the latter case he can choose between full download or download of changes only. It is important to note that compilation is performed locally in the TIA, without any communication with the PLC. The compilation produces several outputs, whose total size is typically between hundreds of kilobytes and many megabytes. The compilation outputs are kept in the TIA's memory ready for download.
- Once the engineer runs a download command, the TIA sends the compiled outputs to the PLC in a single download message (potentially fragmented into smaller fragments).
- After a successful download, the program typically runs automatically. The program periodically reads the PLC's inputs, and writes into its outputs based on the control algorithm. In parallel, a SCADA HMI (typically running on a separate computer) may communicate with the PLC during the run, and fetch or modify information that is used for process control, monitoring and alerts.
- The engineer may also upload a program from the PLC into his project, and later edit it, compile it and download it to the same PLC, or to any other PLC.



```
> Object: ClsId=MArea.Class_Rid, RelId=NativeObjects.theMArea_Rid
v Object: ClsId=ProgramCycleOB.Class_Rid, RelId=OB.1
      Element Tag-Id: Start of Object (0xa1)
      Relation Id: OB.1
      Class Id: ProgramCycleOB.Class_Rid
    > Class Flags: 0x00000020, Persistent
      Attribute Id: None
    > Attribute
    > Attribute
    > Attribute
    > Attribute
    > Attribute
    v Attribute
          Element Tag-Id: Attribute (0xa3)
        v Item Value: ID=FunctionalObject.Code (Blob) = 0x021c68021c6801021c6803020c
              ID Number: FunctionalObject.Code
            > Datatype flags: 0x00
              Datatype: Blob (0x14)
              Blob root ID: None
              Blob size: 13
              Value: 0x021c68021c6801021c6803020c
    > Attribute
```

Fig. 5. The S7 program download message in the P2$^-$ protocol variant.

The S7 protocol variants include specific mechanisms that are dedicated to the protection of the program download message. Next we describe the structure of the message, and its protection in the different protocol variants. We then discuss the vulnerabilities of this protection, and explain how we exploit them to inject malicious control program that cannot be viewed by the engineer or the operator to the S7 Simatic PLC. Last, we describe three variants of our attack for the three S7 protocol variants.

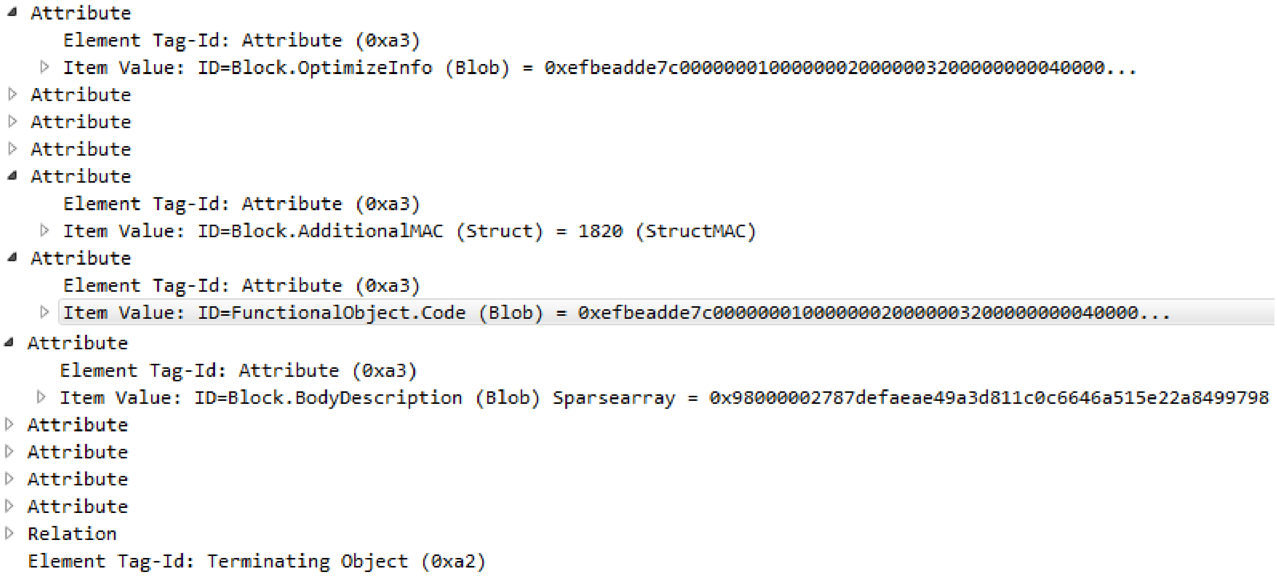
6.1 The S7-1500 program download message

We used network sniffing to identify the messages that the TIA sends to the Simatic S7 PLCs when the user performs the "program download" operation. Our findings on the structure of this message in the three protocol variants is summarized in this sub-section. As mentioned before, the S7 is a request response protocol. Each request message consists of a request header, and a request set. Respectively, a response message is composed of response header and response set. Both headers contains a function code, which identifies the requested operation. Specifically *CreateObject* request builds a new object in the PLC memory[5]. A single S7 message might contain multiple objects. Each object contains multiple attributes. Objects and attributes have unique class identifiers.

We found out that the program download message creates an object of class *ProgramCycleOB*. Figure 5 shows a Wireshark dump of the ProgramCycleOB object from the program download message in the P2$^-$ protocol variant. Using network sniffing, we identified the ProgramCycleOB attributes which contain the values of the run and source objects.

- The object *FunctionalObject.Code* contains the binary executable that the PLC runs (see Figure 5). This is the compiled program in the PLC's internal language.

[5] The names of the functions, the classes and the attributes are taken from the S7CommPlus dissector [37].



```
◢ Attribute
     Element Tag-Id: Attribute (0xa3)
   ▷ Item Value: ID=Block.OptimizeInfo (Blob) = 0xefbeadde7c0000000100000002000000032000000000400000...
▷ Attribute
▷ Attribute
▷ Attribute
◢ Attribute
     Element Tag-Id: Attribute (0xa3)
   ▷ Item Value: ID=Block.AdditionalMAC (Struct) = 1820 (StructMAC)
◢ Attribute
     Element Tag-Id: Attribute (0xa3)
   ▷ Item Value: ID=FunctionalObject.Code (Blob) = 0xefbeadde7c0000000100000002000000032000000000400000...
◢ Attribute
     Element Tag-Id: Attribute (0xa3)
   ▷ Item Value: ID=Block.BodyDescription (Blob) Sparsearray = 0x98000002787defaeae49a3d811c0c6646a515e22a8499798
▷ Attribute
▷ Attribute
▷ Attribute
▷ Attribute
▷ Relation
  Element Tag-Id: Terminating Object (0xa2)
```

Fig. 6. The S7 program download message in the P3 protocol variant.

- Another attribute related to the run object is *Block.AdditionalMac*. This attribute is used only by the P2 and the P3 versions of the protocol .
- We identified two attributes carrying the source object: *Block.OptimizedInfo* and *Block.BodyDescription*. These attributes consist of the original program in a computerized form and are equivalent to the program written by the engineer. They are stored in the PLC and can later be uploaded, upon request, to a TIA project.

Figure 6 shows the attributes of the run and source objects in a download message from protocol P3.

6.2 The attack on P2$^-$

When communicating with S7-1200, the TIA sends the run object in plaintext, without any attempt to encrypt it. For example in Figure 5 the run object (FunctionalObject.Code) is 0x021c6801021c6803020c.

We analyzed the run object structure, and identified the instruction and operands syntax. As a result, we can easily build control programs with some desired functionality either from scratch or by modifying existing run objects.

We implemented a Man-in-the-Middle (MitM) attack on the P2$^-$ program download message. We modified the run object on the fly, by changing PLC instructions to other desired instructions (e.g., changing "set" to "reset"), or by modifying the instruction operands, or both. We can easily do this, since in the P2$^-$ protocol the download message is not integrity-protected, and the run object is not encrypted.

Our changes forced different output values in the PLC outputs (e.g., toggling 1 to 0, or vice-versa), and produced visible changes in output on the PLC leds.

Alternatively, it is easy to write an S7 program injector that takes a source object and a run object and impersonates a TIA to download them to the victim PLC.

Note that since we did not change the source object, when the engineer or the operator uploads the control program from the PLC, the PLC will response with the original source object, which corresponds to the original binary object, and not the the program that actually runs in the PLC.

6.3 Manipulating P2 and P3 encrypted program objects

The main difference between the P2^{-} download message and the P2 download message is that the source and run objects are encrypted. We discovered that in P2, the run and the source attributes contain encrypted blobs that begin with the magic 0xdeafbeef (see Figure 6).

Encryption of these objects is done as part of the compilation, which can be done offline (i.e., without communicating the PLC). It is therefore clear that the encryption keys are not session dependent. We observed that it is possible to record an S7 download message from one session, fetch the source and run objects, and plug them into a download message from another session. This is true for both the P2 and P3 S7 protocol variants. Therefore, we did not attempt to understand the encryption process, nor did we extract the encryption keys: we can use an offline TIA to create and compile the control program of our choice, and then download the program to the victim PLC later.

Another vulnerability that we found in the cryptographic protection of the program objects is that the run object and the source object are not protected by a shared message authentication code. This enables us to mix and match between inconsistent source and run objects, so one program runs in the PLC, while another is displayed to the engineer when he uploads the source from the PLC. Specifically, integrity protection is attached to the run object through the addition of the Block.AdditionalMac attribute. Unfortunately the MAC is applied only to attribute FunctionalObject.Code, enabling us to substitute both attributes with another program's recorded attributes, without modifying the source objects: Block.OptimizedInfo and Block.BodyDescription. Note that even-though these attributes are encrypted, it does not prevent us from manipulating them.

6.4 The attack on protocol P2

Once we construct the desired malicious encrypted source and run objects on a legitimate offline TIA, we are ready to download them to the PLC. Since the P2 handshake protocol is susceptible both to man in the middle and impersonation attacks (see Sub-section 3.3, we can either plug the malicious object in a download message as part of a man in the middle attack that replaces the program, or use our rogue S7 program injector that impersonates the TIA and downloads the pre-built program to the victim PLC. Notice that in both cases the P2 protocol requires us to correct the cryptographic message authentication, which we can easily replay together with the source and binary objects, as mentioned in the previous section.

6.5 The attack on protocol P3

As described in Sub-section 3.2, protocol P3 contains an improved key-exchange during the protocol handshake. However, our rogue engineering station is able to impersonate the TIA to the PLC in the P3 protocol. Therefore, we can adapt our S7 stealth program injector of the P2 protocol with a modified steps required for the P3 protocol.

We implemented this S7 stealth program injector, and used it to download a pre-recorded encrypted programs to a victim S7-1500 PLC, and even downloaded one binary program to the PLC along with a different source program. When an engineer uploads the source back from the PLC, he gets only the source program, and the TIA notifies only on changes in the source, and ignores changes made by our attacks in the binary program.

7 Countermeasures and Conclusions

In this paper we showed that even the latest versions of the S7 devices and protocols are vulnerable to attacks. After reverse-engineering the cryptographic protocol, we are able to create a rogue engineering station which can masquerade as the TIA to the PLC and inject any messages favourable to the attacker. We presented attacks that can remotely start or stop the PLC, and can download control logic of the attacker's choice to a remote PLC. Our strongest attack – the Stealth Program Injection Attack – can separately modify the running code and the source code, which are both downloaded to the PLC. This allows us to modify the control logic of the PLC while retaining the source code the PLC presents to the engineering station. Thus, we can create a situation where the PLC's functionality is different from the control logic visible to the engineer.

The main gap in the S7 cryptographic handshake is that the TIA is not authenticated to the PLC: only the PLC is authenticated to the TIA. Fundamentally, this allows us to create a rogue engineering station (once the veil of obscurity was lifted from the protocol). This gap can be addressed cryptographically — e.g., by having each TIA instance use its own private key, whose public-key is shared and retained by the PLC. An alternative is to introduce a "pairing" mode, in which the PLC and TIA establish a long-lived shared secret during the first session. Either way, the PLC must refuse to communicate with any device claiming to be a TIA which is not the previously-authenticated TIA.

According to [31], the recommended counter-measure against rogue programming of the PLC is by activating the password-protected access control mechanism on each PLC.

A second gap is that all PLCs of the same model and firmware version share the same private-public key pair. This gap can be used in two ways. We used it in a generic way to conduct impersonation attacks on all the S7-1500 PLCs, which use the fact that all PLCs use the same key. We did not, however, extract the private key from the PLCs. If the private key is extracted from one PLC of a particular version, then stronger attacks, specifically full man in the middle attacks with on-they-fly session-hijacking, and also PLC impersonation attacks against a TIA station (without any valid PLC), become possible.

Acknowledgments

We would to acknowledge various contributions of Anat Levitanus, Yaron Gutmark, Rami Eilabouni, Loui Diab, Saeed Essa, Yosrie Mansour, Amir Hetzroni, Noi Madar, Ron Freudenthal, Oren Milman, Aharon Kupershtok, Avichai Tendler, Amit Kleinmann, Ralf Ramsauer. We would also like to thank Thomas Wiens for being able to use his early versions S7comm protocol dissector [37] and for various information exchanges regarding the protocol.

References

1. Cristina Alcaraz, Lorena Cazorla, and Gerardo Fernandez. Context-awareness using anomaly-based detectors for smart grid domains. In *International Conference on Risks and Security of Internet and Systems*, volume 8924, pages 17–34, Trento, 2014. Springer International Publishing.
2. A. Atassi, I. H. Elhajj, A. Chehab, and A. Kayssi. *The State of the Art in Intrusion Prevention and Detection*, chapter 9: Intrusion Detection for SCADA Systems, pages 211–230. Auerbach Publications, January 2014.
3. Gal Benmocha, Eli Biham, and Stav Perle. On the security of Siemens incremental implementation of HMAC. In preparation.
4. D. Beresford. Exploiting Siemens Simatic S7 PLCs. In *Black Hat USA*, July 2011.
5. D. Beresford. Siemens SIMATIC S7-1200 CPU START/STOP module., 2014. `https://www.exploit-db.com/exploits/19833/`.
6. Chia-Mei Chen, Han-Wei Hsiao, Peng-Yu Yang, and Ya-Hui Ou. Defending malicious attacks in cyber physical systems. In *IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013*, pages 13–18, Aug 2013.
7. S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes. Using model-based intrusion detection for SCADA networks. In *Proceedings of the SCADA Security Scientific Symposium*, pages 127–134, 2007.
8. Thomas De Maizière. Die Lage Der IT-Sicherheit in Deutschland 2014. *The German Federal Office for Information Security*, 2014. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2014.pdf?__blob=publicationFile`.
9. Electrical engineering Blog. The top most used PLC systems around the world. Electrical installation & energy efficiency, May 2013. Available at: `http://engineering.electrical-equipment.org/electrical-distribution/the-top-most-used-plc-systems-around-the-world.html`.
10. N. Falliere. Exploring stuxnet's PLC infection process, 2010. `http://www.symantec.com/connect/blogs/exploring-stuxnet-s-plc-infection-process`.
11. N. Falliere, L.O. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 2011.
12. Niv Goldenberg and Avishai Wool. Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems. *International Journal of Critical Infrastructure Protection*, 6(2):63–75, 2013.
13. Siobhan Gorman. Electricity grid in U.S. penetrated by spies. *The Wall Street Journal*, page A1, April 8th 2009.
14. T. Hergenhahn. LIBNODAVE, exchange data with Siemens PLCs, February 2011. Available at: `http://libnodave.sourceforge.net`.
15. Siemens SIMATIC S7-1200 vulnerabilities, 2014. `https://ics-cert.us-cert.gov/advisories/ICSA-14-079-02`.
16. ISO 8073: Information processing systems – open systems interconnection – connection oriented transport protocol specification, 1986.
17. A. Kleinmann and A. Wool. Accurate modeling of the Siemens S7 SCADA protocol for intrusion detection and digital forensics. *Journal of Digital Forensics, Security and Law*, 9(2):37–50, 2014.
18. Amit Kleinmann, Ori Amichay, Avishai Wool, David Tenenbaum, Ofer Bar, and Leonid Lev. Stealthy deception attacks against SCADA systems. In *3rd Workshop on the Security of Industrial Control Systems & of Cyber-Physical Systems (CyberICPS), LNCS 10683*, pages 93–109, Oslo, Norway, September 2017.
19. Amit Kleinmann and Avishai Wool. A statechart-based anomaly detection model for multi-threaded scada systems. In *International Conference on Critical Information Infrastructures Security*, pages 132–144. Springer, 2015.
20. Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, and Volker Roth. Internet-facing PLCs-a new back orifice. In *Blackhat USA 2015, Las Vegas, USA*, 2015.
21. Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.
22. Robert M. Lee, Michael J. Assante, and Tim Conway. Analysis of the cyber attack on the Ukrainian power grid. Technical report, SANS E-ISAC, March 18 2016. `https://ics.sans.org/media/E-SAC_SANS_Ukraine_DUC_5.pdf`.
23. Cheng Lei, Li Donghong, and Ma Liang. The spear to break the security wall of S7CommPlus. In *Blackhat Europe*, 2017. `https://www.blackhat.com/docs/eu-17/materials/eu-17-Lei-The-Spear-To-Break%20-The-Security-Wall-Of-S7CommPlus-wp.pdf`.
24. Gaoqi Liang, Steven R Weller, Junhua Zhao, Fengji Luo, and Zhao Yang Dong. The 2015 Ukraine blackout: Implications for false data injection attacks. *IEEE Transactions on Power Systems*, 2016.
25. S. Marsching. A new EPICS device support for S7 PLCs. In *Proceedings of the 14th International Conference on Accelerator & Large Experimental Physics Control Systems (ICALEPCS2013)*, San Francisco, CA, USA, October 2013.
26. A. M. McKenzie. RFC 905: ISO transport protocol specification ISO DP 8073, April 1984.

27. D. Nardella. Snap7 1.2.0 - reference manual - rev. 3, January 2014. Available at: `http://snap7.sourceforge.net`.

28. M. T. Rose and D. E. Cass. RFC 1006: ISO transport services on top of the TCP: Version 3, May 1987.

29. Carl Schuett, Jonathan Butts, and Stephen Dunlap. An evaluation of modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 7(1):61–68, 2014.

30. Siemens. Modular PLC controllers SIMATIC S7, 2014. Available at: `http://www.automation.siemens.com/mcms/programmable-logic-controller/en/simatic-s7-controller`.

31. Siemens ProductCERT, 2019. `https://new.siemens.com/global/en/products/services/cert.html`. Personal communication.

32. Siemens Security Advisory. Vulnerabilities in SIMATIC step 7 v12 and v13, 2015. `http://www.Siemens.com/cert/pool/cert/Siemens_security_advisory_ssa-315836.pdf`.

33. Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. PLC-blaster: A worm living solely in the PLC. In *Black Hat Asia, Marina Bay Sands, Singapore*, 2016.

34. tcpliveplay man page. `https://tcpreplay.appneta.com/wiki/tcpliveplay-man.html`.

35. A. Valdes and S. Cheung. Communication pattern anomaly detection in process control systems. In *IEEE Conference on Technologies for Homeland Security (HST)*, pages 22–29, 2009.

36. Roy Ward and Tim Molteno. Table of linear feedback shift registers. *Datasheet, Department of Physics, University of Otago*, 2007.

37. T. Wiens. S7comm Wireshark dissector plugin, January 2014. Available at: `http://sourceforge.net/projects/s7commwireshark`.

A Details of the S7 P3-variant key exchange protocol

A.1 Cryptographic environment and primitives

All the cryptographic code used in the key exchange is located in the *OMSp_ core_ managed* DLL: a 64bit binary which contains both managed (.NET: C#) code and native code. We have been able to characterize the central functions used withing this DLL. We refer to functions we have not identified yet using generic names such as f_1, or sometimes via a name that indicates their role (like KDF for Key Derivation Function) when that is clear. Note that by linking to the DLL using python ctypes and preparing the proper arguments we are able to call arbitrary functions inside the DLL as needed.

The DLL is accompanied by a directory of compressed key files (located at *Siemens/Automation/Portal V14/Data/Hwcn/Custom/Keys*) organized by firmware version. In addition there are also generic family ("S7-1500 Family") key files.

These key files contain general information (such as key type, firmware version, etc.), a standard certificate which we haven't encountered during our research, and a 40-byte public key, representing a 160-bit elliptic-curve point we denote by Q. The curve's base point G seems to be hardcoded inside the DLL.

We identified that the key exchange scheme uses an Elliptic-curve El-Gamal-like exchange. The curve itself is not one of the standard curves implemented by OpenSSL. Furthermore, while the OpenSSL library is embedded inside the *OMSp_ core_ managed* DLL, the TIA code does not use it for the public-key operations; instead we found a loop-unrolled implementation designed specifically for a 160-bit curve.

The integrity protection session key, which we denote by $sessionKey$ is derived from the 16 middle bytes of the PLC's *ServerSessionChallenge* combined with a random 24-byte key derivation key (*KDK*) that the TIA selects. The main purpose of the handshake is to provide the *KDK* to the PLC so it can derive the same $sessionKey$.

Algorithm 1 *Session key derivation*

```
Input: $ServerSessionChallenge$ # [20 bytes]
$challenge = ServerSessionChallenge[2:18]$
$KDK = prng(24)$                                    ▷ Based on the Microsoft CryptGenRandom API
$sessionKey = HMAC\text{-}SHA256_{KDK}(f(challenge, 8)||challenge)[:24]$    ▷ Using a “fingerprinting” function $f$
Return: $KDK$, $sessionKey$
```

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="4">0xFEE1DEAD</td><td colspan="4">Length</td><td colspan="4">1</td><td colspan="4">1</td><td colspan="16">KDK ID Header</td></tr>
<tr><td colspan="16">Public key ID Header</td><td colspan="16"><i>EG1</i></td></tr>
<tr><td colspan="4"><i>EG1</i></td><td colspan="20"><i>EG2</i></td><td colspan="8"><i>Nonce</i></td></tr>
<tr><td colspan="12"><i>Nonce</i></td><td colspan="16">IV</td><td colspan="4">Encrypted Challenge</td></tr>
<tr><td colspan="12">Encrypted Challenge</td><td colspan="20">Encrypted KDK</td></tr>
<tr><td colspan="4">Encrypted KDK</td><td colspan="16">Encrypted Checksum</td><td colspan="12">——</td></tr>
</table>

Fig. 7. The *SecurityKeyEncryptedKey* structure of [M3]

A.1.1 Starting the handshake The first request message [M1] is a *Hello* message that the TIA sends to initialize a new session.

The PLC responds with message [M2], which contains the PLC firmware version, and a 20-byte challenge *ServerSessionChallenge*. The PLC firmware version determines the elliptic-curve public-key pair to be used in the key exchange — see Section A.1.3.

Upon receiving the [M2] message, the TIA then activates Algorithm 1 to randomly select a *KDK* and to generate the *sessionKey* from the PLC's *ServerSessionChallenge* and the *KDK*.

A.1.2 The key exchange message [M3] The third handshake message *M3* (recall Figure 4) transfers the *KDK* to the PLCs, using the public-key scheme. [M3] contains a structure called *SecurityKeyEncryptedKey* in the S7CommPlus dissector [37] (See Figure 7). This structure starts with a magic value 0xFEE1DEAD, the structure length (180 bytes), and some flags which we always found to be ‘1’.

To initialize the cryptographic elements and populate some of the in [M3] the TIA activates Algorithm 2, which works as follows:

1. Generates a 24-bytes random quantity R and maps it to the elliptic curve's domain: we call the resulting random curve point the *PreKey*. It seems that the *PreKey* is represented in a projective representation, as a 60-byte quantity.
   This *PreKey* will be sent to the PLC as part of $M3$ using an elliptic-curve El-Gamal public-key exchange, see below.
2. From the random point *PreKey*, it uses a key-derivation-function KDF to derive 3 16-byte quantities, see Algorithm 2. We identify these quantities as follows:
   (a) A Key Encryption Key (*KEK*). This is an AES key that is used to encrypt the *KDK* and the *Challenge*): the encrypted KDK and encrypted challenge are placed inside the [M3] message.
   (b) A Checksum Seed *CS*. This seed is used to generate 4096 pseudo-random bytes organized as four 256-word look-up tables, which we collectively call LUT. LUT is used by a (non-cryptographic) tabulation hash to calculate a checksum over the KDK and Challenge (See Algorithm 4).

Algorithm 2 Initialization

```
$R = prng(24)$                              ▷ Based on the Microsoft CryptGenRandom API
$PreKey$ = $EC\text{-}MAP(R)$                  ▷ Select a random point on the curve
$KEK, CEK, CS = KDF(PreKey)$               ▷ Derive AES keys and seed for lookup tables
$LUT[4][256]$ = $TB\text{-}HASH\text{-}INIT(CS)$      ▷ Initialize 4 lookup-tables for the tabulation hash
Return $PreKey, LUT, KEK, CEK$
```

Algorithm 3 Elliptic-Curve El-Gamal-like key exchange

```
procedure EC-ENC($PreKey$, $Q$)     [G is the base point]
    $point = \infty$
    $Nonce = prng(20)$
    while $point == \infty$ do
        $y = prng(20)$
        $point$ = $EC\text{-}MULT$ $(G, y, Nonce)$              ▷ Using the base point $G$
        $EG2 = point_X$
    end while
    $s$ = $EC\text{-}MULT(Q, y, Nonce)$                     ▷ Using the public-key $Q$
    $EG1$ =$EC\text{-}ADD(s, PreKey)_X$                     ▷ Encrypt the $PreKey$ using $s$
    return $EG1, EG2$, $Nonce$
end procedure
```

(c) A Checksum Encryption Key *CEK*. This is an AES key used to encrypt the checksum. The encrypted checksum is placed inside [M3].

A.1.3 Public Key ID and KDK ID headers The [M3] message includes 2 header fields that are include key fingerprints: these are 8-byte truncated SHA256 hashes of the relevant key, with some additional flags. See Algorithm 5.

The *Public key ID header* identifies the public key Q used to encrypt the *PreKey*, while the *KDK ID header* identifies the KDK used to derive the *sessionKey*.

The public key ID appears in the key-chain data structure in the TIA memory, and may enable the TIA to indicate which public key it is using in case there is more than one applicable key for a specific PLC model and firmware version. We are unsure of the role of the KDK key ID field in [M3], perhaps it is used by the PLC to check the validity of the KDK it derives, in addition to the validation offered by the Encrypted Checksum field.

A.1.4 EC-Elgamal-like Encryption The TIA uses an EC-Elgamal-like public-key encryption to encrypt *PreKey* (See Algorithm 3). The TIA chooses 20-bytes *Nonce*, which is placed as-is in the [M3] structure and seems to be used to 'mask' the elliptic-curve calculations. It then chooses a 20-bytes random y, uses the base point G to calculate yG, and places its X coordinate into an [M3] field (*EG2*). Then it uses the public-key Q to calculate yQ, encrypts the *PreKey* point $(yQ + PreKey)$ and places the resulting point's X coordinate in an [M3] field (*EG1*).

A.1.5 Challenge and KDK Encryption The TIA encrypts the *Challenge* and the *KDK* by computing $AES\text{-}CTR_{KEK}(Challenge||KDK)$ and stores the results in the two relevant fields in [M3]. To do so the TIA picks a random 16-byte initialization vector (IV) for AES-CTR, which is also placed in [M3] (in plaintext). The counter-mode increments the counter

Algorithm 4 Checksum calculation

```
procedure CHECKSUM(ENC, CEK, LUT[4][256])        ▷ ENC is Encrypted Challenge||Encrypted KDK
    $currentChecksum = 0$
    for $block$ in $ENC$ do
        $currentChecksum = TB\text{-}HASH(currentChecksum \oplus block, LUT)$
    end for
    $currentChecksum[12] = currentChecksum[12] \oplus 40$        ▷ size of ENC is 40 bytes
    $finalChecksum = TB\text{-}HASH(currentChecksum, LUT)$
    return $AES\text{-}ECB_{CEK}(finalChecksum)$
end procedure
```

Algorithm 5 Key ID Derivation

```
procedure GetKeyId(key)
    return $sha256(key[:24] \,||\, \text{“DERIVE”})[:8]$        ▷ return an 8 byte fingerprint
end procedure
```

using a maximum-cycle 128-bit LFSR with taps at 128,127,126,121, called LFSR-4($n = 128$) in [36].

A.1.6 Checksum Encryption To provide authenticated encryption for the encrypted challenge and encrypted *KDK* a non-cryptographic checksum is computed, encrypted by AES-ECB using the CEK, and placed in the M3 structure (see Algorithm 4)

A.1.7 M3 decryption and handling by the PLC When the PLC receives the M3 message, it uses its private key to decrypts *PreKey*. It then derives the *KEK*, *CEK* and the *CS* hash configuration from the *PreKey*. Then it uses *KEK* to extract the *KDK* and *Challenge* and verifies TIA's freshness and the integrity of KDK. When the message is verified successfully, the PLC uses *KDK* to derive the *SessionKey* responses with an ACK message.

The next request message that the TIA sends, is already authenticated by an HMAC-SHA256 with the derived session key. All the rest of the packets in the session, both those which are sent by the PLC, and those which are sent by the TIA, are authenticated and integrity protected with HMAC-SHA256 with the session key.