A Decade After Stuxnet: How Siemens S7 is Still an Attacker's Heaven

Colin Finck
c.finck@enlyze.com

Tom Dohrmann
t.dohrmann@enlyze.com

ENLYZE GmbH, Heliosstraße 6a, 50825 Köln, Germany

Industrial Control Systems have long evolved from specialized electronics communicating over proprietary bus systems to fully-fledged embedded computers based on commodity Ethernet connections. The Stuxnet computer worm of 2010 demonstrated to the general public that this development makes Industrial Control Systems susceptible to cyberattacks with physical consequences [10]. Siemens as the vendor of the affected Programmable Logic Controllers (PLCs) has released multiple new products since then, which double down on Ethernet connectivity in company networks and are expected to conform to higher security standards.

In this paper, we reverse-engineer the Siemens S7-1500 Software Controller PLC up to the communication protocol and show the violation of fundamental security principles. We show that substantial efforts have been put into obfuscating keys and code of the communication as well as modifying established cryptography primitives—all without increasing the effective security level. Along the way, we release a few tools to help with reverse-engineering that particular PLC firmware. Even though the analyzed variant of the protocol has been deprecated in 2022, it is still the most widespread one and updating to a newer variant is non-trivial. Unlike previous publications on this topic, we put an emphasis on providing sufficient details to enable other people to reproduce our research and build up on it. To that end, the S7-1500 Software Controller has been chosen, because it resembles the pervasive S7-1500 hardware PLC series while being a software-only PLC, making it very accessible to the broader research community.

1. Introduction

With their ubiquity in the manufacturing and processing industry, their importance for critical infrastructure such as power plants, grids or pipelines, and further applications in adjacent sectors like building automation, Industrial Control Systems (ICSs) can truly be considered the backbone of modern everyday life. A central component of today's Industrial Control Systems are Programmable Logic Controllers (PLCs). PLCs are compact modules that provide multiple input and output pins for working with analog and digital signals. Additional interfaces to bus networks allow for the connection of further peripherals such as sensors, actors or distributed I/O blocks. Due to this flexibility and their ability to be programmed graphically, PLCs have long replaced hard-wired logic.

From a Computer Science perspective, most current PLCs can be classified as embedded computers, with some of them being only little more than commodity x86 hardware in industrial cases. Former vendor-specific ports for connecting to proprietary bus interfaces have mostly been replaced by standard Ethernet ports. The same Ethernet ports are also used for programming the PLC. While this simplifies cabling and allows for easy integration into company networks, it also introduces a new path for adversaries to carry out cyberattacks with physical consequences.

Figure 1: A hardware PLC of the S7-1500 series [23]

The IEC 61131-3 norm standardized programming languages for PLCs, including variable declarations and their data types [24]. All relevant market participants follow this standard. Although IEC 61131-3 does a good job at unifying education on PLC programming and making PLC source code vendor-agnostic, it falls short on interoperability of PLC hardware and PLC communication protocols. This is one reason why most industrial automation companies created proprietary hardware and communication ecosystems. They usually have an interest in keeping up existing incompatibilities to other vendors in order to benefit from the resulting vendor lock-ins. Despite efforts such as OPC UA to exchange data between automation products of different companies, native vendor-specific communication protocols remain the only first-class citizens in the automation ecosystem. Consequently, a typical situation in the real world is a PLC by one vendor communicating with a Human-Machine Interface (HMI) by the same vendor using a vendor-proprietary protocol. As long as this situation persists, reverse-engineering PLCs and their communication protocols is not an exclusive domain of security researchers. It also becomes a necessity for companies like ENLYZE, who need to reimplement the communication protocols to record process variables from all kinds of PLCs. This task of reverse-engineering a target system for creating interoperability is also a prime example for legally permitted reverse-engineering under EU law [21].

The global PLC market is dominated by five companies that account for over 80% of the market. Siemens is leading this market with a share of around 31% [18]. A Siemens Industrial Control System featuring PLCs of the Siemens S7-300 and S7-400 series was the target of the infamous Stuxnet attack of 2010, which manipulated the control of specific frequency converters and reportedly destroyed around one-fifth of Iran's nuclear centrifuges [4, 10]. The current generation of Siemens PLCs called S7-1500 (Figure 1) has been introduced in 2012. Due to the revamped hardware and software of that generation as well as the time elapsed since the Stuxnet attack, customers can rightfully expect the S7-1500 series to feature a comprehensive security concept that guards against such attacks.

In this paper, we reverse-engineer the Siemens S7-1500 Software Controller PLC up to the communication protocol. Released in 2015, the S7-1500 Software Controller implements the features of an S7-1500 hardware PLC in a standalone x86 binary that is supposed to be run by a Siemens hypervisor next to a standard Windows operating system on a Siemens industrial computer. We show that Siemens put substantial efforts into obfuscation techniques and reinventing cryptographic primitives without improving the overall security. Unlike previous publications on this topic, we aim to provide sufficient details to enable other researchers to reproduce our findings and build up on them. This is common practice in the cybersecurity community and the only way to achieve enduring security, hence the industrial automation domain must not be exempted from this.

2. Legal and Ethical Considerations

Legal and ethical considerations played a major role in conducting our research and publishing this paper. While our focus has always been on researching PLCs to write compatible clients for reading process variables, we came across a lot of security-relevant peculiarities of the communication protocol. Publishing such details is always a double-edged sword: On the one hand, this is absolutely necessary to debunk false claims about the security of a device and advance the state of PLC security research. On the other hand, the same information could potentially aid malicious actors if disclosed unresponsibly. Further trade-offs have been discussed in [27] and were considered for this publication.

For this paper, we have decided to publish all our developed tools that help with reverse-engineering the PLC firmware. This allows other PLC security researchers to get on par with our research and with the state of previous publications, such as [1] and [3]. These tools do not disproportionally help malicious actors: Getting new insights still requires reverse-engineering, which continues to be a manual and time-consuming process.

We have also decided to publish all details regarding *public* cryptography keys. They are absolutely necessary to reproduce and verify our findings from a researcher's perspective. They are also required to achieve the initial goal of writing an independent client that can interoperate with modern Siemens PLCs. Note that this application of reverse-engineering is explicitly permitted under EU law [21].

At the same time, we are deliberately not releasing example code for an independent PLC client. Such a tool would only provide limited additional value to researchers, but disproportionally aid "script kiddies" and other adversaries in abusing industrial control systems.

Ironically, publishing these cryptographic details even has the potential to increase the effective security of PLC communication: We have encountered third-party implementations of PLC protocols, which do not implement the cryptographic parts correctly and use hardcoded values in parts where random numbers are required for a secure operation. Releasing our research could provide the required information to these vendors to fix their insecure implementations.

We have deliberately decided not to publish any PLC private keys in this paper. They turned out to be the most heavily guarded pieces of the firmware. Furthermore, they were not necessary to reach our goal of writing an independent PLC client.

The PLC communication protocol that we are analyzing in this paper has been deprecated by Siemens in 2022 [25], following a successful extraction of a PLC private key by Claroty [29]. As a consequence, Siemens has released a new version of the communication protocol based on the publicly standardized TLS 1.3 layer.

By publishing details about the legacy protocol, we are therefore not disclosing a new vulnerability. If we had found one during our research, we would have first initiated a coordinated disclosure process with Siemens ProductCERT. However, we would like to emphasize that the Siemens update policies, missing incentives for machine manufacturers, and the general infeasibility of quickly rolling out such major updates means that the new protocol has hardly reached existing machines [12]. Hence, the legacy protocol is likely going to stay with us for years to come, and the aforementioned precautions are still important.

Finally, choosing to publish on the S7-1500 Software Controller and not the S7-1500 hardware PLC has been part of our legal and ethical considerations:

1. Unlike the S7-1500 hardware PLC, the S7-1500 Software Controller runs next to a Windows operating system on a regular computer. The Windows OS as well as Siemens software written for Windows come with established automatic update processes. If any firmware update becomes necessary, this should be easier to manage for an S7-1500 Software Controller than for a physical S7-1500 PLC.

2. Although both types follow a similar design, the S7-1500 Software Controller communication protocol is sufficiently different from that of the S7-1500 hardware PLC. Any outcome resulting from this publication would only affect the Software Controller and not the more popular and harder to update hardware PLC.

3. Being a software-only PLC, the S7-1500 Software Controller is very accessible to the broader research community. This enables researchers to verify our findings easily, possibly without purchasing expensive hardware, but does not provide an edge to adversaries.

3. Relation to Previous Work

Multiple research teams have published on the security of the Siemens S7 PLC series before. Current versions of the S7-1200 and S7-1500 hardware PLCs as well as the S7-1500 Software Controller feature an encrypted communication protocol, which is what we are focusing on. This excludes several older publications on these PLC types, such as [26] from 2016, which exclusively deal with an abandoned unencrypted version of the communication protocol.

When starting to reverse-engineer the encrypted protocol of the S7-1500 in mid-2019 and researching prior work, only a single relevant publication from 2017 by Lei et al. turned up [19]. They provide a low-level analysis of the encryption bytes transmitted over the wire when connecting to an S7-1200 or S7-1500 with then-current firmwares using the Siemens TIA Portal programming software. This analysis is enriched with screenshots of the corresponding memory bytes in a debugger. While giving first hints as to what functions are involved in the encryption and their inputs and outputs, the work does not provide a description of the underlying algorithms. But unlike most publications on the topic, the paper by Lei et al. leaves all screenshots unredacted, making it easy to reproduce the findings when using the same version of the Siemens TIA Portal software.

Multiple parties were independently researching the latest S7 communication protocol in 2019. Shortly after our first successful implementation, Weißberg released his bachelor's thesis on the analysis of the protocol with a focus on the used cryptography [30]. Around the same time, Biham et al. presented a similar analysis at Black Hat USA 2019, including an application of the research in the form of a malicious engineering station [2]. Both publications build up on the work by Lei et al. and provide a detailed analysis of the message exchange, the theory behind it, and the used cryptographic primitives. They also give a first impression of the extensive code obfuscation measures taken by Siemens. Due to these measures and decisions taken by the authors, some important parts of the key exchange remain undocumented or were deliberately left out. As a result, the work can hardly be reproduced independently and these parts need to be researched from scratch.

Whereas the previously named publications analyzed the PLC from the perspective of the TIA Portal client software, Abbasi et al. presented an analysis of an S7-1200 from the hardware and operating system side at Black Hat Europe 2019 [1]. Although their research does not cover the communication protocol, it provides valuable insights into the proprietary Siemens ADONIS operating system running on Siemens PLCs, and techniques into reverse-engineering it. Some of these insights aided our research on the S7-1500 Software Controller.

Team82 at Claroty has recently built up on the published research and developed an exploit to extract hardcoded private keys of S7-1200 and S7-1500 PLCs [29]. Disguising an application as Siemens TIA Portal and communicating with a PLC was already possible before without knowing the private keys. However, Team82 has also revealed that the same keys can be used to extract passwords from the PLC and thereby bypass access level protections. This proves once more that custom cryptography with code obfuscation cannot make up for the mistake of relying on hardcoded private keys. Their publication led to the introduction of a new TLS-based cryptography now available on all S7-1200 and S7-1500 PLCs (both physical and software-only models). The previous communication protocol has been deprecated, but is likely going to stay with us for decades due to long machine lifetimes, the cost of updates, and the inability of applying updates in many setups [12].

It should be noted that all aforementioned publications exclusively analyze the S7-1200 and S7-1500 hardware PLCs. A quick look at the communication between TIA Portal and the S7-1500 Software Controller reveals that the cryptographic details are significantly different, and these have not been published before. First research on the S7-1500 Software Controller has been presented by Bitan & Dankner at Black Hat USA 2022 [3]. Their work shows a novel way of breaking one layer of obfuscation that Siemens added exclusively for the S7-1500 Software Controller. Although the authors spend a lot of time explaining their methodology, the publication falls short on details to make the result reproducible. In fact, screenshots have been deliberately redacted. This paper adds the missing details and builds up on the research to deliver a usable framework for reverse-engineering the S7-1500 Software Controller, as well as the first full description of its communication protocol.

Finally, it must be noted that all publications on the modern S7 communication protocol make heavy use of the S7comm Wireshark dissector plugin by Thomas Wiens [31]. His long-time and continuous work on the dissector deserves most credit here. To this end, our paper also follows the naming introduced by him and calls the protocol *S7CommPlus* hereafter.

4. Reverse Engineering Obstacles and Strategies

Siemens provides the S7-1500 Software Controller in several flavors: One option is to obtain it preinstalled on an embedded computer with DIN rail mounting. That computer is called "ET 200SP Open Controller" or "CPU 1515SP" and is available in multiple configurations, all having order numbers starting with "6ES7672-2". Multiple corresponding controller firmwares exist and are called "CPU 1505S" or "CPU 1505SP" (order numbers starting with "6ES7672-5").

Alternatively, you can obtain the S7-1500 Software Controller as a software-only package to be installed on specific Siemens industrial computers. The corresponding controller firmwares are called "CPU 1507S" and "CPU 1508S" (order numbers starting with "6ES7672-7" resp. "6ES7672-8").

In the following sections, we obtain a specific version of the firmware and describe our journey of reverse-engineering it.

4.1. Obtaining the Firmware Image

No matter what S7-1500 Software Controller model you want to analyze, the firmware is always a single self-contained binary called `CPU.ELF`. After installation, this file can be found in `C:\Boot\Siemens\SWCPU` on the simultaneously running Windows system. It is accompanied by a Siemens hypervisor that is split up between the two files `VMM_1ST_STAGE.ELF` and `VMM_2ND_STAGE.ELF`. You can find them in `C:\Boot\Siemens\SIMATIC_RT_VMM`.

In order to make our research reproducible, we didn't use a random firmware version from some machine, but extracted a specific one from a downloadable update package. Hence, the following chapters are specific to the firmware obtained via the `SIMATIC_CPU_1505SP_V21_9.exe` update package with the SHA-256 checksum `68a192e9d6203b74f546275b6393f1fd9e410983b3642c1ac3400b875a2af2c2`. This package can be run on an arbitrary Windows computer to extract the contained files into a directory. Afterwards, we opened the `Data1.cab` file from the extracted `InstData\S7_Vmm\Media` subdirectory and further extracted the `VMM_2ND_STAGE.ELF_4B3AB761_8023_596B_91E6_4235FAF5D2F3` file. Likewise, we extracted `CPU.ELF_C924BFD8_60A0_520A_A2EF_CBE66BEA1F2B` from the `Data1.cab` file of `InstData\CPU1505SP\Media`.

4.2. Decrypting the Firmware Image

While `VMM_1ST_STAGE.ELF` and `VMM_2ND_STAGE.ELF` are standard x86 ELF binaries, the actual firmware in `CPU.ELF` is encrypted. Bitan & Dankner have shown in [3] that the decryption code is fully contained in `VMM_2ND_STAGE.ELF`. Instead of performing a time-consuming static analysis and reimplementation, they have developed a harness to jump directly to the decryption code in `VMM_2ND_STAGE.ELF` and run it on a standard Linux system using the Intel Pin framework [16]. Unfortunately, their harness has not been released to the public. As a result, we had to redo their work and implement our own harness to decrypt the `CPU.ELF` file. This has been released at [11].

Note that this is not a new method, but utilizes the same Intel Pin framework. All credits for originally inventing this approach go to Bitan & Dankner.

Additionally, the hardcoded memory addresses in our harness are specific to the `VMM_2ND_STAGE.ELF` of the CPU 1505SP firmware version 21.9. Your mileage may vary with different firmware versions.

4.3. Dynamic Analysis

The ADONIS kernel found in the S7-1500 Software Controller firmware appears to be mainly written in C++ and makes heavy use of virtual inheritance. Reversing such code statically can be difficult, so our first attempt at gaining an understanding about the inner workings of the kernel was to boot it and analyze it dynamically.

The decoded firmware image contains a multiboot header, but it's at byte address `0x37C0` outside the first 8 KiB as expected by QEMU. Instead of modifying the QEMU multiboot loader accordingly, we implemented a small UEFI-based bootloader in Rust to load the kernel. This bootloader is now available at [7]. It has later been extended to patch the loaded kernel to our needs, for example to hook various debug print functions.

The kernel uses the hypercall instructions `VMCALL` and `VMMCALL` to communicate with the hypervisor. The hypercalls (Table 5) don't seem to follow any well-known standard and so are incompatible with KVM, Hyper-V, and Xen hypervisors. Instead of relying on hardware virtualization and the kernel for handling those hypercalls, we chose to modify QEMU's TCG emulator to implement the hypercalls.

ADONIS expects certain custom PCI devices to be present. These devices are emulated by the hypervisor. Their purposes are not yet known. QEMU was modified to add these devices and emulate them in such a way that the kernel progresses further at boot. Our modifications have been made available at [8].

Dynamic analysis was eventually abandoned, because it turned out to be more work than expected to get the kernel to boot and static analysis proved sufficient for our needs. We are releasing our work in progress anyway to let other researchers pick up from where we left off.

4.4. Static Analysis

The S7-1500 Software Controller code has been statically analyzed using the Ghidra software reverse engineering suite [22]. This initially proved challenging, because the ADONIS kernel runs in 64-bit long mode, but only uses 32-bit pointer addresses. This x32-like ABI was not natively supported by the tested reverse engineering tools (Ghidra, IDA, Binary Ninja), which all expect pointers to be 64-bit values when executing 64-bit code. To improve decompiler results, we modified Ghidra's SLEIGH files for x86_64 to truncate addresses to 32 bits. Our modifications have been made available at [6].

The firmware contains C++ Run-Time Type Information (RTTI) records that provide valuable insights into class names and the inheritance structure. We attempted to use the Ghidra-Cpp-Class-Analyzer plugin [28] to recover type information from RTTI and reconstruct vtables, but the original version failed on duplicate entries in the firmware binary. This issue could eventually be fixed by us and the fix has been submitted upstream [9]. The fixed analyzer now provides a detailed overview of the C++ classes in the firmware binary. In particular, many of the virtual classes we were interested in turned out to only have a single implementation.

Furthermore, names of several previously unidentified functions could be retrieved via their calls to the logging facility. A Ghidra script has been written to analyze the entire binary for such calls and rename functions accordingly. This script is now available as part of [6].

Eventually, we identified two functions using the magic value `0xfee1dead` found in the handshake. Starting from there we were able to identify other code used in the handshake.

5. Handshake of the Communication Protocol

The Siemens S7-1200 and S7-1500 PLCs, physical and software-only, communicate with the TIA Portal client software and HMIs over the S7CommPlus protocol. This is a binary protocol, which can be understood to a high degree these days thanks to the S7comm Wireshark dissector plugin by Thomas Wiens [31]. Each S7CommPlus packet is embedded in a Connection-Oriented Transfer Protocol (COTP) [15] packet, which is itself embedded in a Transport Packet (TPKT) [17] sent over a TCP/IP connection. We can only assume historical reasons for this peculiar stack of deprecated

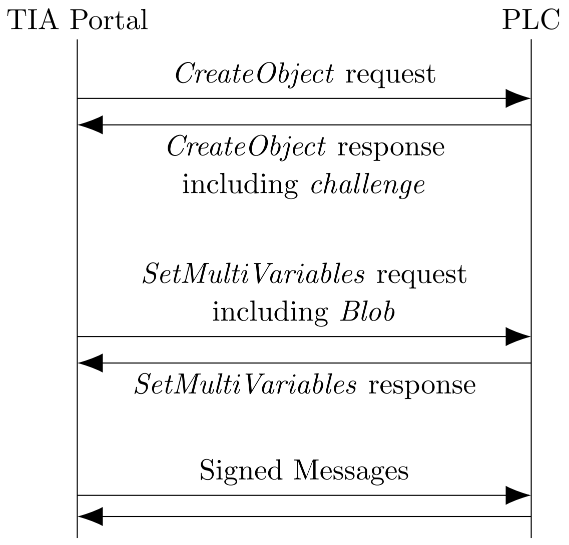

TIA Portal
PLC
CreateObject request
CreateObject response including *challenge*
SetMultiVariables request including *Blob*
SetMultiVariables response
Signed Messages

Figure 2: Legacy handshake of the S7CommPlus protocol

protocols, considering that the legacy S7 protocol used by the S7-200/300/400 series of PLCs already featured a similar architecture.

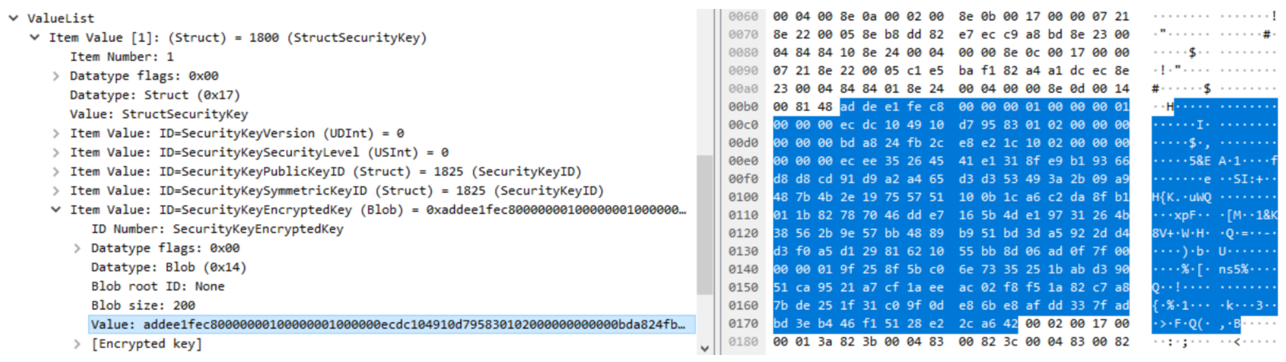
After initiating a connection from TIA Portal to the PLC via TCP/IP and the COTP *Connect Request* message, the PLC expects a cryptographic handshake. For Siemens PLCs since 2013 and the S7-1500 Software Controller, this handshake is based on heavily obfuscated code and modifications to established cryptographic primitives. Although Siemens deprecated this part when introducing TLS 1.3 transport encryption in 2022, you can only expect to see TLS in new automation projects. Existing projects would require a firmware upgrade on all affected devices and a (paid) upgrade to the TIA Portal V17 programming software. Currently, most machine manufacturers have no incentives to roll out such upgrades to their customers [12]. As a result, the legacy handshake is still far more widespread and expected to stay relevant for a long time. Our following analysis therefore focuses on the legacy handshake specific to the S7-1500 Software Controller.

The legacy communication protocol of the S7-1500 Software Controller uses an asymmetric challenge-response handshake (Figure 2). First, TIA Portal sends a *CreateObject* request and receives a challenge in the corresponding *CreateObject* response. This challenge is encrypted and placed in a *Blob* structure. In addition to the challenge, TIA Portal also encrypts a randomly generated symmetric key and places it in the same *Blob* structure. This key is later used to sign messages. The *Blob* structure is sent to the PLC in a *SetMultiVariables* request (Figure 3). The PLC decrypts and verifies the values in the *Blob* structure. The handshake and algorithms specific to S7-1200/1500 hardware controllers have been published in detail in [2, 19, 30] except for the core asymmetric cryptographic operations. These operations are partly obfuscated in both the PLC firmware and TIA Portal.

It should be noted that no part of this handshake uses any user-controlled secret. The PLC has a fixed public-private key pair and the remaining secrets are randomly generated at each connection attempt. This is why TIA Portal can connect to most PLCs without any authorization prompt, and so can third-party clients implementing the same protocol. It is possible to set a password and access levels, but our experience shows that this is hardly done in practice [12]. Additionally, Claroty has even managed to break into password-protected PLCs after extracting the PLC private key [29].

The algorithms used to encrypt the challenge and symmetric key differ between physical PLCs and the S7-1500 Software Controller. This becomes obvious once you notice the different sizes of the public key and the *Blob* structure between the variants.

The following subsections explain the cryptography unique to the S7-1500 Software Controller. They are not an exact description of what TIA Portal and the firmware do as multiple layers of obfuscation irrelevant to the actual cryptography have been omitted.



```
v ValueList
  v Item Value [1]: (Struct) = 1800 (StructSecurityKey)
      Item Number: 1
    > Datatype flags: 0x00
      Datatype: Struct (0x17)
      Value: StructSecurityKey
    > Item Value: ID=SecurityKeyVersion (UDInt) = 0
    > Item Value: ID=SecurityKeySecurityLevel (USInt) = 0
    > Item Value: ID=SecurityKeyPublicKeyID (Struct) = 1825 (SecurityKeyID)
    > Item Value: ID=SecurityKeySymmetricKeyID (Struct) = 1825 (SecurityKeyID)
    v Item Value: ID=SecurityKeyEncryptedKey (Blob) = 0xaddee1fec800000001000000010000000…
        ID Number: SecurityKeyEncryptedKey
      > Datatype flags: 0x00
        Datatype: Blob (0x14)
        Blob root ID: None
        Blob size: 200
        Value: addee1fec800000001000000010000000ecdc104910d795830102000000000000bda824fb…
      > [Encrypted key]
```

```
0060  00 04 00 8e 0a 00 02 00  8e 0b 00 17 00 00 07 21   ········ ·······!
0070  8e 22 00 05 8e b8 dd 82  e7 ec c9 a8 bd 8e 23 00   ·"······ ······#·
0080  04 84 84 10 8e 24 00 04  00 00 8e 0c 00 17 00 00   ·····$·· ········
0090  07 21 8e 22 00 05 c1 e5  ba f1 82 a4 a1 dc ec 8e   ·!·"···· ········
00a0  23 00 04 84 84 01 8e 24  00 04 00 00 8e 0d 00 14   #······$ ········
00b0  00 81 48 ad de e1 fe c8  00 00 00 01 00 00 00 01   ··H····· ········
00c0  00 00 00 ec dc 10 49 10  d7 95 83 01 02 00 00 00   ······I· ········
00d0  00 00 00 bd a8 24 fb 2c  e8 e2 1c 10 02 00 00 00   ·····$·, ········
00e0  00 00 00 ec ee 35 26 45  41 e1 31 8f e9 b1 93 66   ·····5&E A·1····f
00f0  d8 d8 cd 91 d9 a2 a4 65  d3 d3 53 49 3a 2b 09 a9   ·······e ··SI:+··
0100  48 7b 4b 2e 19 75 57 51  10 0b 1c a6 c2 da 8f b1   H{K.·uWQ ········
0110  01 1b 82 78 70 46 dd e7  16 5b 4d e1 97 31 26 4b   ···xpF·· ·[M··1&K
0120  38 56 2b 9e 57 bb 48 89  b9 51 bd 3d a5 92 2d d4   8V+·W·H· ·Q·=··-·
0130  d3 f0 a5 d1 29 81 62 10  55 bb 8d 06 ad 0f 7f 00   ····)·b· U·······
0140  00 00 01 9f 25 8f 5b c0  6e 73 35 25 1b ab d3 90   ····%·[· ns5%····
0150  51 ca 95 21 a7 cf 1a ee  ac 02 f8 f5 1a 82 c7 a8   Q··!···· ········
0160  7b de 25 1f 31 c0 9f 0d  e8 6b e8 af dd 33 7f ad   {·%·1··· ·k···3··
0170  bd 3e b4 46 f1 51 28 e2  2c a6 42 00 02 00 17 00   ·>·F·Q(· ,·B·····
0180  00 01 3a 82 3b 00 04 83  00 82 3c 00 04 83 00 82   ··:·;··· ··<·····
```

Figure 3: Blob structure in Wireshark

5.1. Asymmetric Key Exchange

The key exchange (Algorithm 1) is done using the Elliptic Curve Diffie-Hellman (ECDH) key exchange over a 192-bit prime field elliptic curve with these parameters:

$$
\begin{aligned}
p &= \texttt{0xfffffffffffffffffffffffffffffffffffffffffffff13} \\
a &= \texttt{-1} \\
b &= \texttt{0x6241e52b7bd8790514ebe1e51c8368cd9d56e1ae21de9cbc} \\
G_x &= \texttt{0x6f74ce776d67b1d7a49f8cf0e26b77bc677cf771962e4427} \\
G_y &= \texttt{0x7eaa7f6516d614857b4cda3e3f2fb5c642fc8285fb86575f}
\end{aligned}
$$

These parameters don't match any standardized curves and seem to be Siemens specific.

Algorithm 1 Elliptic Curve Diffie-Hellman Key Exchange performed by the Client

```
1: procedure ECDH($PlcPublicKey$)
2:     $Nonce \leftarrow \textsc{rng}(24)$
3:     $ClientPublicKey \leftarrow Nonce * G$
4:     $DerivedSharedSecret \leftarrow Nonce * PlcPublicKey$
5:     return $ClientPublicKey, DerivedSharedSecret_x$
6: end procedure
```

The PLC's public key is

$$
\begin{aligned}
PlcPublicKey_x &= \texttt{0x8e6d4846b080f387e3d48858c54a40b7fb28dc02b706e25f} \\
PlcPublicKey_y &= \texttt{0x12fe2110375f5e3627148ac04f1c5473042275e4b1091567}
\end{aligned}
$$

This key is shipped with TIA Portal in `C:\Program Files\Siemens\Automation\Portal V17\Data\Hwcn\Custom\Keys\Modules.PcStation.mpk`, which can be opened as a zip file. In that zip file, the public key for the S7-1500 Software Controller CPU 1505S is found in `6ES7672-5AC00-0YA0_V1.0.key`. The .key files are zlib-compressed and parts of the resulting content are base64-encoded in addition. Both layers can be trivially decoded using e.g. CyberChef [14]. To our knowledge, this public key also works for all other S7-1500 Software Controller CPUs in existence. It definitely does not work for S7-1200 and S7-1500 hardware PLCs.

5.2. Deriving Shared Keys

The 192-bit derived shared secret generated by the asymmetric key exchange is used to derive two shared keys (Algorithm 2). The algorithm has three steps:

1. The following matrix of values in the curve's prime field ($ScrambleMatrix$) is raised to the derived shared secret:

   $$\begin{pmatrix} \texttt{0xa5e873221ea059a595ba61bf27f9cdd5954ef57a747978e2} & \texttt{0x71ded36d796ac873a589cfe8e2831af1297e7e279053186c} \\ \texttt{0x55136a2069fe9c09984dcb47174c5b77d9c8b4a3db52cd7e} & \texttt{0x5a178cdde15fa65a6a459e40d806322a6ab10a858b868633} \end{pmatrix}$$

2. The resulting matrix is hashed using SHA-256, the hash is truncated, and the result is hashed again using SHA-256.

3. The resulting digest is split into two parts. Each part is separately encrypted using a modified AES algorithm and returned as a shared key.

To our knowledge, this algorithm has not been published before and Siemens invented it specifically for the S7-1500 Software Controller. Some steps have apparently been added solely for obfuscation purposes. An example is truncating the output and hashing it a second time in step 2. This step provides no additional security over hashing the output once using the proven SHA-256 hash function.

Algorithm 2 High-level description of the key derivation algorithm

```
procedure DeriveSharedKeys($DerivedSharedSecret$)
    $M \leftarrow ScrambleMatrix^{DerivedSharedSecret}$
    $Digest_0 \leftarrow \text{SHA256}(M)$
    $Digest_1 \leftarrow \text{SHA256}(Digest_0[0:23])$
    $Key_0 \leftarrow \text{ModifiedAES}(Digest_1[0:15])$
    $Key_1 \leftarrow \text{ModifiedAES}(Digest_1[16:31])$
    return $Key_0, Key_1$
end procedure
```

The modified AES algorithm (Algorithm 3) roughly follows the same steps as regular AES: An additional step has been added at the beginning and the SubBytes and MixColumns steps have been modified. The algorithm has been shortened to 5 rounds and the round keys are hardcoded.

Algorithm 3 Top level description of the modified AES algorithm

```
procedure ModifiedAES($S$)
    $S \leftarrow \text{Reorder}(S)$
    $S \leftarrow \text{ModifiedRound}(S,$ 6d3d0d8dc4bc3c2c5d060ed4beef620a, 0x34, 0x6d, $0)$
    $S \leftarrow \text{ModifiedRound}(S,$ b1a3b9553037fb819fba35d5a1a91f51, 0xf1, 0x82, $1)$
    $S \leftarrow \text{ModifiedRound}(S,$ f01b26b5138997cab2b1275b65934c7a, 0x06, 0xdf, $2)$
    $S \leftarrow \text{ModifiedRound}(S,$ da3b3bf7bb85e77422c5268502742193, 0xa2, 0x4a, $3)$
    $S \leftarrow \text{ModifiedRound}(S,$ dd13a6e62620cdf12621f699ea9ac26e, 0xb7, 0x30, $4)$
    return $S$
end procedure

procedure ModifiedRound($S, K, a, b, Round$)
    $S \leftarrow \text{AddRoundKey}(S, K)$
    $S \leftarrow \text{ModifiedSubBytes}(S, a, b)$
    $S \leftarrow \text{ShiftRows}(S)$
    $S \leftarrow \text{ModifiedMixColumns}(S, Round)$
    return $S$
end procedure
```

The newly introduced Reorder step (Algorithm 4) applies a fixed permutation to the initial state.

Algorithm 4 The newly introduced Reorder step

```
procedure REORDER($S$)
    $X \leftarrow 0$
    $Y \leftarrow 1$
    $S' \leftarrow []$
    for $i \leftarrow 0$ to 15 do
        $Index \leftarrow Y * 4 + X$
        $S'_{Index} \leftarrow S_i$
        $X \leftarrow (X + 1) \mod 4$
        if X = 0 then
            $Y \leftarrow (Y + 1) \mod 4$
        end if
        if $i$ is even then
            $Y \leftarrow (Y + 1) \mod 4$
        end if
    end for
    return $S'$
end procedure
```

The Reorder step can also be described by this matrix:

$$S' = \begin{bmatrix} S_{12} & S_1 & S_2 & S_7 \\ S_0 & S_5 & S_6 & S_{11} \\ S_4 & S_9 & S_{10} & S_{15} \\ S_8 & S_{13} & S_{14} & S_3 \end{bmatrix}$$

The SubBytes step (Algorithm 5) has been modified to xor a constant into both the index and substituted byte each.

Algorithm 5 The modified SubBytes step

```
procedure MODIFIEDSUBBYTES($S, a, b$)
    for $i \leftarrow 0$ to 15 do
        $Index \leftarrow S_i \oplus a$
        $SubByte \leftarrow AESSBox_{Index}$
        $S_i \leftarrow SubByte \oplus b$
    end for
    return $S$
end procedure
```

The MixColumns step (Algorithm 6) has been modified in two ways: The polynomial is different depending on the round and column, and the columns are transposed into rows.

Algorithm 6 The modified MixColumns step

```
1: procedure ModifiedMixColumns($S, Round$)
2:     $M_{0:3} \leftarrow$ ModifiedMixColumn($(S_0, S_4, S_8, S_{12}), Round, 0$)
3:     $M_{4:7} \leftarrow$ ModifiedMixColumn($(S_1, S_5, S_9, S_{13}), Round, 1$)
4:     $M_{8:11} \leftarrow$ ModifiedMixColumn($(S_2, S_6, S_{10}, S_{14}), Round, 2$)
5:     $M_{12:15} \leftarrow$ ModifiedMixColumn($(S_3, S_7, S_{11}, S_{15}), Round, 3$)
6:     return $M$
7: end procedure
8:
9: procedure ModifiedMixColumn($C, Round, ColumnIndex$)
10:     Rotate $C$ by $ColumnIndex$ positions to the right.
11:     $SubroundValue \leftarrow (0,0,0,0)$
12:     if $Round = 3$ & $ColumnIndex = 0$ then
13:         $SubroundValue_0 \leftarrow C_3$
14:     end if
15:     if $Round = 2$ & $ColumnIndex = 1$ then
16:         $SubroundValue_1 \leftarrow C_2$
17:     end if
18:     if $Round = 1$ & $ColumnIndex = 2$ then
19:         $SubroundValue_2 \leftarrow C_1$
20:     end if
21:     if $Round = 4$ & $ColumnIndex = 3$ then
22:         $SubroundValue_3 \leftarrow C_0$
23:     end if
24:     $M_0 \leftarrow C_1 \oplus C_2 \oplus C_3 \oplus$ GfMult2($C_0 \oplus C_1 \oplus SubroundValue_0$)
25:     $M_1 \leftarrow C_0 \oplus C_2 \oplus C_3 \oplus$ GfMult2($C_1 \oplus C_2 \oplus SubroundValue_1$)
26:     $M_2 \leftarrow C_0 \oplus C_1 \oplus C_3 \oplus$ GfMult2($C_2 \oplus C_3 \oplus SubroundValue_2$)
27:     $M_3 \leftarrow C_0 \oplus C_1 \oplus C_2 \oplus$ GfMult2($C_3 \oplus C_0 \oplus SubroundValue_3$)
28:     return $M$
29: end procedure
30:
31: procedure GfMult2($I$)
32:     if $I < 128$ then
33:         return $I * 2$
34:     else
35:         return $(I * 2 + 27) \mod 256$
36:     end if
37: end procedure
```

The ShiftRows and AddRoundKey steps have not been modified (Algorithm 7).

Algorithm 7 The unmodified steps of the AES algorithm

```
1: procedure AddRoundKey($S, K$)
2:     return $S \oplus K$
3: end procedure
4:
5: procedure ShiftRows($S$)
6:     for $i \leftarrow 0$ to 3 do
7:         Rotate $S_{i*4:i*4+3}$ by $i$ positions to the left
8:     end for
9:     return $S$
10: end procedure
```

While Siemens put enormous efforts into modifying the AES algorithm and obfuscating the hard-coded round keys, the actual purpose of the algorithm remains dubious. Due to the lack of any individual encryption key, it cannot serve as a secure encryption function. Similarly, it does not have a purpose as a hashing function, considering that it is a two-way function and the handshake already uses SHA-256 for hashing. We therefore come to the conclusion that the modified AES algorithm only adds to the obfuscation, but does not increase the overall security of the handshake in any way.

5.3. Challenge and Symmetric Key

The main goal of the whole key exchange algorithm is to securely transmit the challenge and a symmetric key. The challenge is provided by the PLC and the symmetric key is randomly generated by the client.

An ephemeral key and a randomly generated Nonce are used to encrypt the challenge and symmetric key using AES-GCM. Note that the Nonce is not directly transmitted, instead the first IV derived from the Nonce is transmitted. The ephemeral key is securely transmitted using the shared keys derived in the previous section. The first shared key is used to encrypt the ephemeral key using AES-128. To authenticate the resulting ciphertext, it is hashed using SHA-256 and encrypted with the second shared key using AES-128.

Algorithm 8 describes the full key exchange and the values of all involved variables. These variables are bundled together in a *Blob* structure (Table 1) and sent to the PLC.

Algorithm 8 The full key exchange

```
1: $ClientPublicKey, DerivedSharedSecret \leftarrow \text{ECDH}(PlcPublicKey)$
2: $Key_0, Key_1 \leftarrow \text{DeriveSharedKeys}(DerivedSharedSecret)$
3: $SK \leftarrow \text{rng}(16)$
4: $EncryptedSK \leftarrow \text{AES128}(Key_0, SK)$
5: $Digest \leftarrow \text{SHA256}(EncryptedSK)$
6: $EncryptedDigest \leftarrow \text{AES128}(Key_1, Digest)$
7: $N \leftarrow \text{rng}(12)$
8: $FirstIV \leftarrow \text{AESGcmCounter0}(N)$
9: $SymmetricKey \leftarrow \text{rng}(24)$
10: $Plaintext \leftarrow Challenge_{2:17} \| SymmetricKey$
11: $EncryptedChallenge, AuthenticationTag \leftarrow \text{AESGcm}(SK, N, Plaintext)$
```

Table 1: Blob structure

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0xfee1dead				Length (200)				1				1				Symmetric Key Checksum								Symmetric Key Flags							
PLC Public Key Checksum								PLC Public Key Flags								$ClientPublicKey_x$															
$ClientPublicKey_x$								$ClientPublicKey_y$																							
$EncryptedSK$																$EncryptedDigest$															
$FirstIV$																$EncryptedChallenge$															
$EncryptedChallenge$																								$AuthenticationTag$							
$AuthenticationTag$																															

5.4. Additional Obfuscations in the Implementation

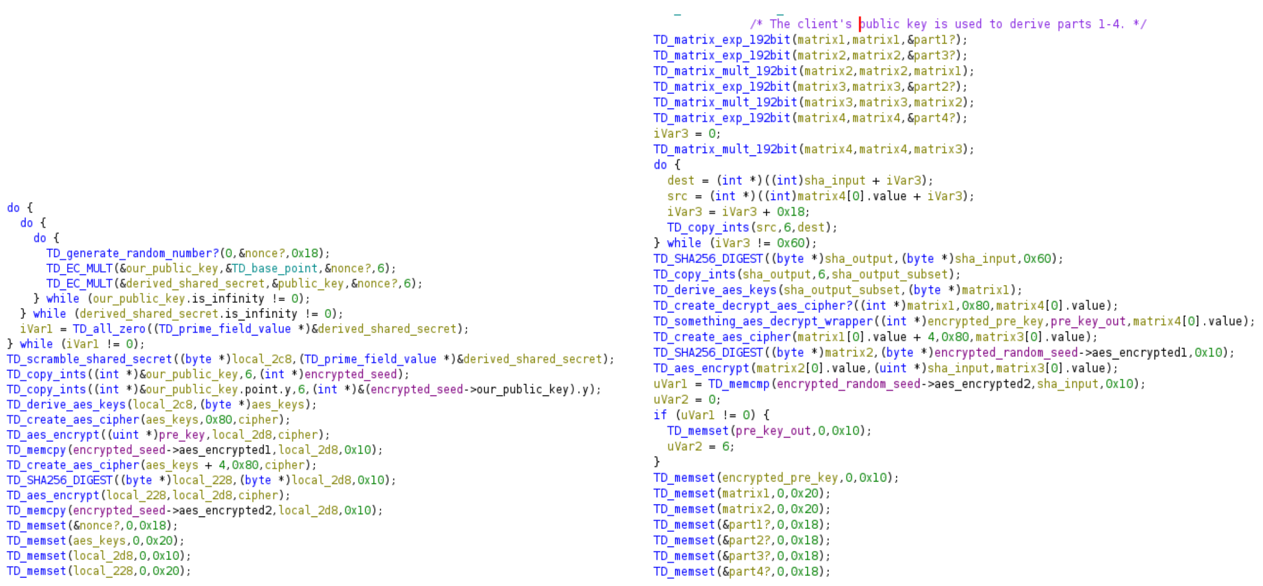
The previous chapters provide a high-level description of the algorithms involved in the handshake from the perspective of the client (TIA Portal or an HMI). However, when it comes to the actual implementation on the S7-1500 Software Controller side, Siemens added further layers of obfuscation.

A notable obfuscation lies in the implementation of the Elliptic Curve Diffie-Hellman key exchange. For the client, this follows the well-known textbook ECDH algorithm, which is also given in Algorithm 1 and with a screenshot of an annotated Ghidra decompilation in Figure 4a. Instead of a fixed key, the client uses a random Nonce for every connection.

On the other hand, the PLC has a fixed public key and therefore one would expect the ECDH algorithm to use the PLC's corresponding secret private key to derive the shared secret. However, doing it exactly like that would instantly reveal the secret private key in the disassembly. Siemens mitigated that risk by implementing another obfuscated function, which uses the private key indirectly by inlining the elliptic curve operations on it. The output of that function is not the shared secret, but four intermediate values relative to the PLC's private key and the client's public key. Four hardcoded matrices are raised to these values and then multiplied together to get the shared secret. Parts of this can be seen in the screenshot of the annotated Ghidra decompilation in Figure 4b.

We can only assume that Siemens obfuscated this part to such an extent in order to keep the hardcoded and unchangeable PLC private key secret under all circumstances. However, Claroty has published in [29] that they managed to retrieve the PLC private key of hardware S7-1500 PLCs anyway. They also demonstrated that the private key has further uses, such as maintaining the secrecy of PLC passwords – another reason why Siemens put these significant efforts into obfuscating the private key.

Ultimately, this obfuscation and the resulting secrecy of the PLC private key do not prevent attackers from disguising as a PLC in the handshake though. Instead of reimplementing the PLC side of the handshake with an ECDH textbook algorithm (and failing due to the undisclosed private key), they can always just run the obfuscated function and take the results of it. This technique of running obfuscated code instead of analyzing it has already been demonstrated in subsection 4.2 when decrypting the firmware image. Thanks to the x86_64 architecture of the S7-1500 Software Controller firmware, this method does not even require processor emulation.



```
do {
  do {
    do {
      TD_generate_random_number?(0,&nonce?,0x18);
      TD_EC_MULT(&our_public_key,&TD_base_point,&nonce?,6);
      TD_EC_MULT(&derived_shared_secret,&public_key,&nonce?,6);
    } while (our_public_key.is_infinity != 0);
  } while (derived_shared_secret.is_infinity != 0);
  iVar1 = TD_all_zero((TD_prime_field_value *)&derived_shared_secret);
} while (iVar1 != 0);
TD_scramble_shared_secret((byte *)local_2c8,(TD_prime_field_value *)&derived_shared_secret);
TD_copy_ints((int *)&our_public_key,6,(int *)encrypted_seed);
TD_copy_ints((int *)&our_public_key.point.y,6,(int *)&(encrypted_seed->our_public_key).y);
TD_derive_aes_keys(local_2c8,(byte *)aes_keys);
TD_create_aes_cipher(aes_keys,0x80,cipher);
TD_aes_encrypt((uint *)pre_key,local_2d8,cipher);
TD_memcpy(encrypted_seed->aes_encrypted1,local_2d8,0x10);
TD_create_aes_cipher(aes_keys + 4,0x80,cipher);
TD_SHA256_DIGEST((byte *)local_228,(byte *)local_2d8,0x10);
TD_aes_encrypt(local_228,local_2d8,cipher);
TD_memcpy(encrypted_seed->aes_encrypted2,local_2d8,0x10);
TD_memset(&nonce?,0,0x18);
TD_memset(aes_keys,0,0x20);
TD_memset(local_2d8,0,0x10);
TD_memset(local_228,0,0x20);
```

(a) Client side

```
            /* The client's public key is used to derive parts 1-4. */
TD_matrix_exp_192bit(matrix1,matrix1,&part1?);
TD_matrix_exp_192bit(matrix2,matrix2,&part3?);
TD_matrix_mult_192bit(matrix2,matrix2,matrix1);
TD_matrix_exp_192bit(matrix3,matrix3,&part2?);
TD_matrix_mult_192bit(matrix3,matrix3,matrix2);
TD_matrix_exp_192bit(matrix4,matrix4,&part4?);
iVar3 = 0;
TD_matrix_mult_192bit(matrix4,matrix4,matrix3);
do {
  dest = (int *)((int)sha_input + iVar3);
  src = (int *)((int)matrix4[0].value + iVar3);
  iVar3 = iVar3 + 0x18;
  TD_copy_ints(src,6,dest);
} while (iVar3 != 0x60);
TD_SHA256_DIGEST((byte *)sha_output,(byte *)sha_input,0x60);
TD_copy_ints(sha_output,6,sha_output_subset);
TD_derive_aes_keys(sha_output_subset,(byte *)matrix1);
TD_create_decrypt_aes_cipher?((int *)matrix1,0x80,matrix4[0].value);
TD_something_aes_decrypt_wrapper((int *)encrypted_pre_key,pre_key_out,matrix4[0].value);
TD_create_aes_cipher(matrix1[0].value + 4,0x80,matrix3[0].value);
TD_SHA256_DIGEST((byte *)matrix2,(byte *)encrypted_random_seed->aes_encrypted1,0x10);
TD_aes_encrypt(matrix2[0].value,(uint *)sha_input,matrix3[0].value);
uVar1 = TD_memcmp(encrypted_random_seed->aes_encrypted2,sha_input,0x10);
uVar2 = 0;
if (uVar1 != 0) {
  TD_memset(pre_key_out,0,0x10);
  uVar2 = 6;
}
TD_memset(encrypted_pre_key,0,0x10);
TD_memset(matrix1,0,0x20);
TD_memset(matrix2,0,0x20);
TD_memset(&part1?,0,0x18);
TD_memset(&part2?,0,0x18);
TD_memset(&part3?,0,0x18);
TD_memset(&part4?,0,0x18);
```

(b) PLC side

Figure 4: Excerpts of annotated Ghidra decompilations of the key exchange code

5.5. Intermediate Values

To make the previously introduced algorithms better comprehensible, Table 2 shows intermediate values calculated during a handshake for exemplary randomly generated values.

All values are given in hexadecimal. Values prefixed with `0x` are stored as a single big integer while all other values are stored as byte arrays.

Table 2: Intermediate values during a handshake

Input	**Name**	**Step**
`0x8e6d4846b080f387e3d48858c54a40b7fb28dc02b706e25f`	$PlcPublicKey_x$	ECDH
`0x12fe2110375f5e3627148ac04f1c5473042275e4b1091567`	$PlcPublicKey_y$	ECDH
`9a61448927a7ca9cca042704093702004959 45b4`	$Challenge$	Key Exchange

Randomly Generated Value	**Name**	**Step**
`0x140f25900db2ee9ff1fcea1f99e47e65b1edec84b48c7119`	$Nonce$	ECDH
`d49d08c3390b2fce1698f2ff6197948e`	SK	Key Exchange
`ca1bd6399c718a9886a5b3ca48f3607e7e792997f9421587`	$SymmetricKey$	Key Exchange
`bbdb4bad57c166df1f027a2a`	N	Key Exchange

Calculated Value	**Name**	**Step**
`0x44a5c5271a5a04d185aa4e622d8642f510da98f7e7e3b433`	$ClientPublicKey_x$	ECDH
`0x742e73a48557454f55c7cc8348d3a7f6f516f0eb218474f7`	$ClientPublicKey_y$	ECDH
`0x20ec8c108f20c78a56ec392bb7781226b34db0fd54cd2c3c`	$DerivedSharedSecret_x$	ECDH
`0x17a089d0769561cf533f3f271096312d89b373d0c4aa2ef8`	$M_{0,0}$	DeriveSharedKeys
`0xaa221afd220b8e44baae6927a2642edb520ef55d5aed198d`	$M_{0,1}$	DeriveSharedKeys
`0xd9eef4a83e821c72e9d2e920b2679defd223d17e79ba306c`	$M_{1,0}$	DeriveSharedKeys
`0xe85f762f896a9e30acc0c0d8ef69ced2764c8c2f3b55d01d`	$M_{1,1}$	DeriveSharedKeys
`af22e2debf3124ca46816d215567de4c73619b4a61c127c9e513dbd78a356649`	$Digest_0$	DeriveSharedKeys
`ad73ebf1a80184bed13a4c772a98f75f3841f9302af5ef90b9d254507098372d`	$Digest_1$	DeriveSharedKeys
`99ccb39bd62347dd9a4cf0ff56986b91`	Key_0	DeriveSharedKeys
`dbe035e522ba4609e0dd26de62460827`	Key_1	DeriveSharedKeys
`fc4ebc6f2b42a65fbab9b55e4d1c3118`	$EncryptedSK$	Key Exchange
`358c5c9eb4aa6d24770c82850acd96a662dc290c42f7f58b3a343cf2d5b3f0ce`	$Digest$	Key Exchange
`1db3f584de487b47e4833366dbaa4574`	$EncryptedDigest$	Key Exchange
`bbdb4bad57c166df1f027a2a00000001`	$FirstIV$	Key Exchange
`eb212e29a9c9a03ce8c72df820ae454ef84a9f2ef6bbc6862866e7e83008cbc2` `e5c83bf2614b7d7f`	$EncryptedChallenge$	Key Exchange
`6639598505ffe87a371c655a5098de81`	$AuthenticationTag$	Key Exchange

Additionally, Table 3 shows intermediate states in the modified AES algorithm when calculating the first derived key.

Table 3: Intermediate values of the modified AES algorithm

Value	**Step**	**Line**	**Description**
`ad73ebf1a80184bed13a4c772a98f75f`	ModifiedAES	1	Input
`2a73ebbead018477a83a4c5fd198f7f1`	ModifiedAES	2	Reorder
	ModifiedRound		Round 0
`474ee63369bdb85bf53c428b6f77795fb`	ModifiedRound	12	
`e2b7d8a821ca09c5155d556554775fe7`	ModifiedRound	13	
`e2b7d8a8ca09c5215565155de754775f`	ModifiedRound	14	
`2875b07752d2212aae4d9d01065f5e88`	ModifiedAES/ModifiedRound	3/15	
	ModifiedRound		Round 1
`99d6092262e5daab31f7a8d4a7f641d9`	ModifiedRound	12	
`c74ec3e45e78733c38ed49bd334765b6`	ModifiedRound	13	
`c74ec3e478733c5e49bd38edb6334765`	ModifiedRound	14	
`e25aec1411d44eb946d428b47a874709`	ModifiedAES/ModifiedRound	4/15	
	ModifiedRound		Round 2
`1241caa1025dd973f4650fef1f140b73`	ModifiedRound	12	
`257f94832de641425624dec10b160842`	ModifiedRound	13	
`257f9483e641422ddec15624420b1608`	ModifiedRound	14	
`e7c9a2d3bdfe8746409fb5fc17f7ae38`	ModifiedAES/ModifiedRound	5/15	
	ModifiedRound		Round 3
`3df29924067b6032625a937915838fab`	ModifiedRound	12	
`9119a80e037f6f2af00b8df3e3b7924b`	ModifiedRound	13	
`9119a80e7f6f2a038df3f00b4be3b792`	ModifiedRound	14	
`e8a832cc87b62580bb4c72406a932ab5`	ModifiedAES/ModifiedRound	6/15	
	ModifiedRound		Round 4
`35bb942aa196e8719d6d84d98009e8db`	ModifiedRound	12	
`23ce166e77cdff84d567f3afaa9eff60`	ModifiedRound	13	
`23ce166ecdff8477f3afd56760aa9eff`	ModifiedRound	14	
`99ccb39bd62347dd9a4cf0ff56986b91`	ModifiedAES/ModifiedRound	7/15	Output

5.6. Plaintext Mode

A surprising discovery was made at the end of our research. If the fourth value in the *Blob* is set to zero, the PLC ignores most values and expects the challenge and symmetric key to be transmitted as plaintext values.

It is not entirely clear to us why plaintext mode exists. We have never observed it being used in the wild. This leaves room for speculation: Either the fourth value could be a flag that purposefully disables encryption to help with debugging. Alternatively, the fourth value could be a version indicator, indicating that the handshake has been improved after initially introducing it in an insecure way.

If a client used plaintext mode, the connection parameters could be easily captured by an eavesdropper and used for a subsequent man-in-the-middle attack.

For completeness, the structure of the *Blob* in plaintext mode is depicted in Table 4.

Table 4: Blob structure in Plaintext Mode

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0xfee1dead				Length (200)				1				0				Symmetric Key Checksum								Symmetric Key Flags							
PLC Public Key Checksum								PLC Public Key Flags								*Challenge*															
ignored																															
ignored																															
SymmetricKey																								*ignored*							
ignored																															
ignored																															

6. Conclusion and Outlook

In this paper, we have shown our process of reverse-engineering the S7-1500 Software Controller, beginning with the decryption of a public firmware image to get the entire software-only PLC implementation and ending up with an algorithmic description of the communication protocol. We have found various instances of code obfuscation and unique modifications to well-established cryptographic algorithms in the legacy handshake. Despite these remarkable efforts by Siemens, our analysis shows that the handshake itself only provides limited message integrity protection. Taking latest research and common configurations into account, the PLC security system cannot be expected to withstand any serious cyberattacks. Specifically, when a PLC is operated in the default passwordless configuration, an attacker with knowledge of the algorithms can freely connect to it and issue arbitrary commands. In addition to that, researchers of Team82 at Claroty have shown that it's possible to extract the static PLC private key. This allows to forge messages on the wire via a man-in-the-middle attack, and surprisingly also allows to break the password protection as the last line of defense in some situations [29]. Finally, we have shown that changing a single bit in the handshake allows to bypass the cryptographic parts altogether.

We can only assume that Siemens implemented such a complex handshake for reasons of vendor lock-in. As per Kerckhoffs' principle, the security of a trustworthy cryptographic system must depend on the secrecy of the key and not on the secrecy of the algorithm. Due to the lack of a unique key per PLC, the security of the legacy handshake depends solely on the secrecy of the algorithm and thereby falls short on protecting the PLC communication from adversaries. By publishing the details of the cryptographic system, we want to raise awareness of its issues and enable other researchers to perform a thorough cryptoanalysis. This may reveal further deficiencies.

At the same time, we have published the steps and tooling of our reverse-engineering journey to let other researchers continue from there. This should enable a plethora of follow-up explorations: For once, it allows for a validation of the security of the current TLS-based handshake that is meant to overcome the shortcomings of the legacy handshake. Likewise, further research may also reveal additional surprises, just like the single bit to bypass the cryptographic handshake. Finally, additional work on running the S7-1500 Software Controller inside QEMU would make it accessible to even more researchers.

We have deliberately published the entirety of our research on the S7-1500 Software Controller, but left out any private keys or sample code to connect to a PLC. This has been done to support the legitimate work of fellow security researchers and creators of interoperable products, but without disproportionally benefitting adversaries or "script kiddies" interested in abusing PLCs. Choosing to publish on the S7-1500 Software Controller has been part of that decision. While similar to the physical Siemens PLCs in theory, the actual cryptography of the Software Controller is different. Should any follow-up research reveal further deficiencies, the impact would likely be limited to the less popular and easier updatable Software Controller PLC.

6.1. A call to the industry

Just 10 years ago, most machines in the field could be attacked via plaintext protocols lacking any kind of authentication [13]. The automation world is fortunately past that point, but this development opens a Pandora's box of new challenges: Improperly implemented cryptography, undisclosed vulnerabilities, and missing processes for responsible disclosure continue to threaten the security of automation systems. The good thing is that we have been there before in computer operating systems and established working practices in the meantime. Sadly, it looks like we need to do it all over again for the automation industry. For instance, a 90-day deadline for responsible disclosure of vulnerabilities does not work in an industry where any fix needs to go through at least 3 layers of red tape and requires a paid update. At the same time, this reality must not result in vulnerabilities being hoarded or research being redacted to a point where it cannot be independently verified anymore.

We are therefore calling all fellow researchers to follow our example and share *reproducible* but responsible research on the security of industrial automation products. The state of PLC security cannot be improved if people are forced to do the same work over and over again. History is apparently repeating itself, as only actively used exploits and a push from the security research community led to monthly OS security updates and mutually agreed responsible disclosure practices [5, 20]. Likewise, we also need to get the automation industry to the point where a fix can reach all affected machines in 90 days. Ultimately, the next generation of PLC communication protocols should be developed in the public to enable independent security reviews right from the beginning. We are already seeing third-parties implementing the unpublished S7 communication protocol incorrectly and thereby compromising its security even further.

As of now, adversaries have the upper hand and customers are losing: Considering that our initial attempt to break the obfuscation of a single S7-1500 took only 6 person weeks in 2019, organized adversaries certainly have the capacities to do the same and more. The existing culture of unpublished and underspecified research is making life harder for legitimate security researchers, who want to advance the state of industrial automation security. At the same time, the missing visibility into PLC security issues hampers awareness at customers. They don't even have an established process to quickly roll out security updates to all machines [12].

Time has come to realize that automation products are just embedded computers and need to be subjected to the same cybersecurity standards as the rest of the IT industry.

7. Acknowledgments

We would like to thank Alexander Gladis, Manuel 'HonkHase' Atug, the German Federal Office for Information Security (BSI), and Siemens for reviewing this paper and providing valuable feedback, especially on the legal and ethical considerations of our work.

References

[1] Ali Abbasi, Tobias Scharnowski, and Thorsten Holz. *Doors of Durin: The Veiled Gate to Siemens S7 Silicon.* Dec. 2019. URL: https://i.blackhat.com/eu-19/Wednesday/eu-19-Abbasi-Doors-Of-Durin-The-Veiled-Gate-To-Siemens-S7-Silicon.pdf.

[2] Eli Biham et al. *Rogue7: Rogue Engineering-Station attacks on S7 Simatic PLCs.* Aug. 2019. URL: https://i.blackhat.com/USA-19/Thursday/us-19-Bitan-Rogue7-Rogue-Engineering-Station-Attacks-On-S7-Simatic-PLCs-wp.pdf.

[3] Sara Bitan and Alon Dankner. *sOfT7: Revealing the Secrets of the Siemens S7 PLCs.* Aug. 2022. URL: https://i.blackhat.com/USA-22/Wednesday/US-22-Bitan-Revealing-S7-PLCs.pdf.

[4] William Broad, John Markoff, and David Sanger. *Stuxnet Worm Used Against Iran Was Tested in Israel.* Jan. 2011. URL: https://www.nytimes.com/2011/01/16/world/middleeast/16stuxnet.html.

[5] Steve Dent. *Google posts Windows 8.1 vulnerability before Microsoft can patch it.* Jan. 2015. URL: https://www.engadget.com/2015-01-02-google-posts-unpatched-microsoft-bug.html.

[6] Tom Dohrmann. *A Ghidra plugin for the ADONIS Kernel.* URL: https://github.com/enlyze/ghidra-adonis-processor.

[7] Tom Dohrmann. *A small bootloader for the kernel used in Siemens S7-1500 Software Controllers.* URL: https://github.com/enlyze/s7-1500-software-controller-loader.

[8] Tom Dohrmann. *ADONIS-related modifications to QEMU.* URL: https://github.com/enlyze/qemu/tree/soft-sps.

[9] Tom Dohrmann. *Ghidra-Cpp-Class-Analyzer PR: allow more than one type string.* July 2023. URL: https://github.com/astrelsky/Ghidra-Cpp-Class-Analyzer/pull/72.

[10] Nicolas Falliere, Liam Murchu, and Eric Chien. *W32.Stuxnet Dossier, Version 1.4*. Feb. 2011. URL: https://archive.org/details/w32_stuxnet_dossier.

[11] Colin Finck. *EnlyzeS7SoftwareControllerDecoder - ENLYZE version of pintool as presented in "sOfT7: Revealing the Secrets of Siemens S7 PLCs"*. URL: https://github.com/enlyze/EnlyzeS7SoftwareControllerDecoder.

[12] Colin Finck. *On the hype around the critical Siemens S7-1200/S7-1500 vulnerability CVE-2022-38465*. Dec. 2022. URL: https://building.enlyze.com/posts/on-the-hype-around-the-critical-siemens-s7-1200-1500-vulnerability/.

[13] Eric Forner and Brian Meixell. *Out of Control: SCADA Device Exploitation*. July 2013. URL: https://media.blackhat.com/us-13/US-13-Forner-Out-of-Control-Demonstrating-SCADA-WP.pdf.

[14] Government Communication Headquarters. *The Cyber Swiss Army Knife - a web app for encryption, encoding, compression and data analysis*. URL: https://gchq.github.io/CyberChef/.

[15] *Information processing systems — Open Systems Interconnection — Connection oriented transport protocol specification*. ISO 8073:1986. July 1986. URL: https://www.iso.org/standard/15096.html.

[16] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[17] *ISO Transport Service on top of the TCP Version: 3*. RFC 1006. May 1987. DOI: 10.17487/RFC1006. URL: https://www.rfc-editor.org/info/rfc1006.

[18] Andre Kukhnin et al. *Global Industrial Automation Industry Primer*. Apr. 2017. URL: https://plus.credit-suisse.com/r/V6BmZM2AF-WElY95.

[19] Cheng Lei, Li Donghong, and Ma Lian. *The spear to break the security wall of S7CommPlus*. Dec. 2017. URL: https://www.blackhat.com/docs/eu-17/materials/eu-17-Lei-The-Spear-To-Break%20-The-Security-Wall-Of-S7CommPlus-wp.pdf.

[20] Robert Lemos. *Microsoft details new security plan*. Oct. 2003. URL: https://www.cnet.com/news/privacy/microsoft-details-new-security-plan/.

[21] Federico Morando. "Software Interoperability: Issues at the Intersection between Intellectual Property and Competition Policy". Dissertation. Politecnico di Torino, 2009. DOI: 10.6092/polito/porto/2615059. URL: https://iris.polito.it/retrieve/handle/11583/2615059/27644/Federico_Morando-PhD_Thesis.pdf.

[22] National Security Agency. *Ghidra software reverse engineering (SRE) suite of tools*. URL: https://ghidra-sre.org.

[23] Palatinatian. *Siemens modular programmabler controller SIMATIC S7-1500*. CC BY 3.0. URL: https://commons.wikimedia.org/wiki/File:S71500.JPG.

[24] *Programmable controllers - Part 3: Programming languages*. IEC 61131-3:2013. Feb. 2013. URL: https://webstore.iec.ch/publication/4552.

[25] Siemens AG. *Remarks Regarding SSA-568427 (Weak Key Protection Vulnerability in SIMATIC S7-1200 and S7-1500 CPU Families)*. SSB-898115. Oct. 2022. URL: https://cert-portal.siemens.com/productcert/html/ssb-898115.html.

[26] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. *PLC-Blaster: A Worm Living Solely in the PLC*. Mar. 2016. URL: https://www.blackhat.com/docs/asia-16/materials/asia-16-Spenneberg-PLC-Blaster-A-Worm-Living-Solely-In-The-PLC-wp.pdf.

[27] Jennifer Stisa Granick. *Legal Risks of Vulnerability Disclosure*. Jan. 2004. URL: https://www.blackhat.com/presentations/win-usa-04/bh-win-04-granick.pdf.

[28] Andrew Strelsky. *Ghidra-Cpp-Class-Analyzer - Ghidra C++ Class and Run Time Type Information Analyzer.* URL: https://github.com/astrelsky/Ghidra-Cpp-Class-Analyzer.

[29] Team82 at Claroty. *The Race to Native Code Execution in PLCs: Using RCE to Uncover Siemens SIMATIC S7-1200/1500 Hardcoded Cryptographic Keys.* Oct. 2022. URL: https://claroty.com/team82/research/the-race-to-native-code-execution-in-plcs-using-rce-to-uncover-siemens-simatic-s7-1200-1500-hardcoded-cryptographic-keys.

[30] Felix Weißberg. "Analyse des Protokolls S7CommPlus im Hinblick auf verwendete Kryptographie". Bachelor's Thesis. Fachhochschule Münster, 2019. URL: https://os-s.net/publications/thesis/Bachelor_Thesis_Weissberg.pdf.

[31] Thomas Wiens. *S7comm Wireshark dissector plugin.* URL: https://sourceforge.net/projects/s7commwireshark/.

A. Preliminary Table of ADONIS Hypercalls

Table 5: Hypercalls

$\mathbf{eax_{in}}$	$\mathbf{ebx_{in}}$	$\mathbf{ecx_{in}}$	$\mathbf{edx_{in}}$	$\mathbf{eax_{out}}$	$\mathbf{ebx_{out}}$	$\mathbf{ecx_{out}}$	$\mathbf{edx_{out}}$	**Description**
`0x001`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x002`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x003`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x004`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x005`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x007`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x008`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x010`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x020`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x030`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x101`	*ignored*	*ignored*	*ignored*	*unknown*	*ignored*	*ignored*	*ignored*	Unknown
`0x102`	*ignored*	*ignored*	*ignored*	*unknown*	*ignored*	*ignored*	*ignored*	Unknown
`0x103`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x104`	Index	*ignored*	*ignored*	Success	Value1 (hi)	Value1 (lo)	Value2	Unknown
`0x105`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x106`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x202`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x204`	Index	*unknown*	*ignored*	Success	0	0	*ignored*	Unknown
`0x205`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x207`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x208`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x20a`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x20b`	Index	*ignored*	*ignored*	*unknown*	*ignored*	*ignored*	*ignored*	Unknown
`0x301`	Index	*ignored*	*ignored*	*unknown*	*ignored*	*ignored*	*ignored*	Unknown
`0x302`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x401`	IOAPIC ID	Register	*ignored*	Success	Value	0	*ignored*	Read I/O APIC Register
`0x402`	IOAPIC ID	Register	Value	Success	0	0	0	Write I/O APIC Register
`0x404`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x405`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x503`	Region ID	0	Property	Success	Value (hi)	Value (lo)		Get Memory Region Property
`0x504`	Name Ptr	0	*ignored*	Success	Region ID	*ignored*	*ignored*	Find Memory Region
`0x601`	0	*unknown*	*ignored*	*unknown*	*ignored*	*ignored*	*ignored*	Unknown
`0x602`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x604`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x606`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x607`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x608`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x60a`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x60c`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x60d`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x60f`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x610`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x611`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x612`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x614`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown
`0x701`	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	*unknown*	Unknown