From Pass-the-Hash to Code Execution on Schneider Electric M340 PLCs

Amir Zaltzman and Avishai Wool
School of Electrical Engineering, Tel Aviv University, ISRAEL
E-mails: amirzaltzman@mail.tau.ac.il, yash@eng.tau.ac.il

November 26, 2024

Abstract

The Schneider Electric industrial control systems architecture consists of Modicon PLCs which communicate with an engineering station and SCADA HMI on one side, and control industrial systems on the other side. After reverse-engineering the cryptographic protocol, we identify vulnerabilities through which we are able to masquerade as the engineering station to the PLC, cryptographically sign messages, and inject any messages favourable to the attacker. Moreover, we identify additional vulnerabilities in the PLC's memory management. We demonstrate that these primitives lead to remote code execution, installation of persistent root-kits, and potential re-programming the boot firmware over the network.

1 Introduction

Programmable Logic Controllers (PLCs) are widely used in Industrial Control Systems (ICSs), where they perform essential process control functions. These controllers manage key equipment such as thermostats, barometers, valves, engines, and generators. ICSs are used to oversee critical infrastructure, including power generation facilities, chemical processing plants, water treatment facilities, railways, and other vital transportation systems essential to contemporary 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 [20, 6]. In response, vendors hardened these interfaces by adding cryptographic protection.

PLCs are available from multiple vendors, including Siemens, Allen-Bradley, Mitsubishi, Schneider Electric, and others. Each vendor provides unique firmware, programming languages, communication protocols, and maintenance software. Nonetheless, the fundamental architecture is quite similar: the PLC manages its inputs and outputs using internal variables and logic. Programming for the PLC is done on an engineering workstation using the vendor's specific control language. This code is compiled into an executable format and then transferred to the PLC. The PLCs are monitored and controlled through dedicated systems running Human Machine Interface (HMI) software. While modern PLCs, HMIs, and engineering stations all use TCP/IP for communication, they generally operate with proprietary higher-level protocols.

Our focus in this paper is Schneider Electric's line of Modicon Programmable Logic Controllers (PLCs), which are ranked among the top ten most popular PLCs by their market share, with estimated sales revenue of around one billion US dollars from their PLC business [24]. Along with the PLCs, Schneider Electric's product line features the 'EcoStruxure Control Expert' software platform. This platform serves as both an engineering station and, optionally, as a Human-Machine Interface (HMI). The Control Expert, or the HMI, communicates with the PLCs using the UMAS network protocol, which is an extension to the popular Modbus protocol. The latest versions of the UMAS protocol incorporate cryptographic measures to secure communication, including cryptographic signing of UMAS messages through dynamic nonces exchanged during the authentication process. Our objective is to gain access to a reserved session with the capability to

sign privileged messages, ultimately allowing us to inject a rootkit shellcode and remotely execute it via the UMAS network. Our use-case for achieving our objectives is the Modicon M340 PLC product line.

1.1 Related Work

The Modbus/TCP protocol [1] is well understood, and has been studied by the research community for many years, cf. [7, 13]. However, the proprietary UMAS extensions over Modbus, are less well understood. What is known was discovered via protocol reverse engineering. We can point out the research of [11] that was among the first to uncover the parsing of the UMAS protocol. Subsequent researches [16, 17, 23] furthered the understanding and updated the parsing to reflect the protocol patches introduced in firmware updates (such as appending signature to reserved UMAS messages).

Insights into the authentication process and its evolution through firmware updates until version 3.50 are detailed in [3, 17, 23, 27].

Studies on exploiting vulnerabilities and implementing remote code execution in Schneider Electric Modicon PLCs are detailed in [3, 27, 16, 14, 21].

We can also point out the very useful UMAS wireshark [28] dissector [4]. However, in our research we discovered that this dissector has some shortcomings and is unable to parse certain UMAS messages, hence we needed to update it. We plan to release our revised dissector to the community.

The flavor of work described here in is analogous to the work of Biham et al. [5] which analyzed the S7 protocol used between the Siemens TIA management system and Siemens S1200/S1500 PLCs.

1.2 Contributions

Prior research [17] demonstrated a pass-the-hash attack against Modicon PLCs (CVE-2021-22779). In response, Schneider Electric released firmware v3.60, which claims to mitigate the attack. Against this backdrop our contributions are as follows:

- By reverse engineering the PLC we identified that the main change in v3.60 is the introduction of a Diffie-Hellman key exchange, and the elimination of the cleartext exchange of the password hash (pwd_{hash}).
- We discovered a new vulnerability through which the adversary can steal the password hash by sniffing the project upload or download messages (CVE-2024-8933). With pwd_{hash} in hand we are able to revive the pass-the-hash attack, i.e., we bypass the mitigations introduced against CVE-2021-22779 [3].
- We observed that even if the project is encrypted, the transmission occurs in cleartext during upload or download. Additionally, we discovered that the encryption password hash and its salt are visibly transmitted, similar to the pwd_{hash}.
- The Diffie-Hellman exchange used in v3.60 is a plain-vanilla exchange. so it is vulnerable to a Man-in-the-Middle (MITM) attack. We demonstrate such an attack that is able to steal the management unit and PLC nonces, use them to calculate the signature for authenticating reserved UMAS messages—and thereby achieve the ability to inject arbitrary *signed* UMAS messages to the PLC (CVE-2024-8935).
- We identify a new vulnerability in which via the MITM attack, and using a signed message, we are able to modify a critical variable in the PLC memory—the *addressReadLimiter* (CVE-2024-8936). Once it is overwritten the attacker can read all the restricted memory areas using public *unsigned* read requests sent over UMAS traffic, without needing to execute the MITM attack again, until the PLC is rebooted: i.e., we bypass the mitigations against CVE-2020-7537 [3]. This allows reading the pwd_{hash} and other cryptographic data whenever the attacker wishes.
- Modicon M340 PLCs are built on ARMv4T processors which do not have a "Non-Execute" bit over memory regions. Hence by using signed messages we are able to inject shellcode into unused memory

regions. We identify two new vulnerabilities through-which we can trigger the injected shellcode and achieve code execution, at will, using unsigned messages (CVE-2024-8937 and CVE-2024-8938).

The content of this paper was disclosed to Schneider Electric prior to publication. Schneider officially confirmed our findings on November 12, 2024 and registered them under the five CVE numbers mentioned above: CVE-2024-8933, CVE-2024-8935, CVE-2024-8936, CVE-2024-8937, and CVE-2024-8938. In parallel to the registration of the CVEs Schneider Electric released firmware update 3.65 to address the *addressReadLimiter* overwrite and the two code-execution vulnerabilities. They also published mitigations against the other vulnerabilities.

2 Basics

2.1 EcoStruxure Control Expert

EcoStruxure Control Expert (previously named Unity Pro) is a software platform developed by Schneider Electric as part of their EcoStruxure architecture. It is designed for programming and managing Modicon PLCs and PACs (Programmable Automation Controllers). It provides a set of tools for creating, simulating, debugging, and maintaining control applications.

2.1.1 Control Expert Project and Application Project Binary

As reported by [27], all the information pertaining to the control logic running in a given PLC is organized in an *application project*. The application project is created and compiled in the Control Expert software, and is stored as an STU file: the STU file is a zipped archive that contains the application project binary file, APX, and other data files. The APX files are binary files structured into sections with multiple subsections. Each section has a header with details such as the section type, ID, offset, and size, followed by the actual data subsection. These subsections can vary in content. The data subsection of each section, is constructed as memory blocks different types (such as data/executable/constants etc.) and are loaded into the PLC memory. The blocks' metadata is stored in a Relocation Table (RT) in the APX file, which specifies the address, size, and attributes for each block ([27, 25]). Each block is identified by a 1-byte Block ID. As we shall see, the contents in certain APX blocks is readable (over the network) by specifying their Block ID and an offset within the block.

2.1.2 Control Expert Project User-Defined Protection

When creating a new project, the software suggests the owner to set application and file encryption passwords. Both passwords are set only once when creating the project and they remain unchanged even if the project is modified.

2.1.2.1 Project Password

The project password (*pwd*) is used to establish a secured connection between the management unit using the Control Expert software and the PLC. Moreover, it is required in order open the project file in the Control Expert software.

2.1.2.2 Project Encryption Password

As observed by [27], the project encryption password (*enc_pwd*) is used to encrypt (AES-CBC-256) the project STU file (2.1.1) in order to prevent malicious file corruption or theft of the intellectual property. The file encryption password is not required for openning the STU file or for establishing a secured UMAS connection.

2.2 Modicon M340 Firmware

Modicon M340 firmware files can be downloaded from the Schneider Electric website as LDX files. An LDX file is a zipped archive containing the firmware BIN file and other ftp commands and web files. Schneider also

UMAS header	*Session key*	*UMAS function code*	*Data*
0x5A	0x00	...	...
1 Byte	1 Byte	1 Byte	Variable size

Table 1: The structure of non-reserved UMAS message request

UMAS header	*Session key*	*ACK/NACK code*	*Data*
0x5A	*rsvID*	0xFE/0xFD	...
1 Byte	1 Byte	1 Byte	Variable size

Table 2: The structure of non-reserved UMAS message response

allows to download older firmware versions (as it is possible to downgrade the PLC firmware). In our study we refer our findings to firmware versions from v3.01 and above, in which Schneider started implementing a password protection mechanism over UMAS protocol. In particular we show undiscovered findings and approaches to handle with the recent firmware version v3.60.

3 The UMAS Protocol

UMAS (Unified Messaging Application Services) is a specialized Schneider Electric protocol designed for the configuration and monitoring of Schneider Electric PLCs by the Control Expert software. It is embedded within the Modbus protocol, and uses a propriety Modbus RTU function code: '0x5A' (in the scope of this article we named it the 'UMAS header'). In other words the UMAS messages are transmitted over the Modbus/TCP protocol, using TCP port 502 [1]. Modbus is a request-response protocol: every message exchange is initiated by the management unit (the Control Expert) which sends a request, and the PLC sends a response to that request. The PLC never initiates a transmission. The UMAS extension of Modbus retains this basic behavior.

A UMAS communication session between the management unit and a PLC can be in one of two states. After the management unit authenticates to the PLC, the session becomes *reserved*. A reserved session has a unique reservation ID, and UMAS messages in a reserved session include a signature derived from the message data and session-specific cryptographic data. Reserved signed UMAS messages are required for privileged PLC operations (such as writing to memory, running control logic, performing application management tasks etc.). Before authentication the session is called *non-reserved*, and messages sent in such a session are called *public*. Public messages can be sent either on reserved on non-reserved sessions. Note that there can only be a single reserved session established with a PLC at any point in time.

As reported by [16, 17, 23, 11], from the PLC's point of view, UMAS messages can have one of the following structures, which are detailed in subsections below:

- Non-reserved UMAS request messages.
- Non-reserved UMAS response messages.
- Reserved UMAS request messages.
- Reserved UMAS response messages.
- Signature error UMAS response messages.

3.1 Non-Reserved UMAS Request Messages

Such messages can be sent from the management unit to the PLC without authentication, regardless of whether the PLC is reserved or not. Table 1 shows the general structure of non-reserved public UMAS

UMAS header	*Session key*	*UMAS sign function code*	*Magic*	*Signature*	*UMAS*	*Session key*	*UMAS function code*	*Data*
0x5A	$rsvID$	0x38	0x01	...	0x5A	$rsvID$	...	...
1 Byte	1 Byte	1 Byte	1 Byte	32 Bytes	1 Byte	1 Byte	1 Byte	Variable size

Table 3: The structure of reserved UMAS message request

UMAS header	*Session key*	*ACK code*	*Magic*	*Signature*	*UMAS header*	*Session*	*ACK/NACK code*	*Data*
0x5A	$rsvID$	0xFE	0x01	...	0x5A	$rsvID$	0xFE/0xFD	...
1 Byte	1 Byte	1 Byte	1 Byte	32 Bytes	1 Byte	1 Byte	1 Byte	Variable size

Table 4: The structure of reserved UMAS message response

messages. The messages include: the 'UMAS header', a propriety Modbus function byte used in all UMAS messages; The 'Session key', a byte that carries the reservation ID. If the PLC is not reserved, then the 'Session key' value is '0x00'. Then there is the 'UMAS function code', and variable length 'Data' that corresponds to the function type.

3.2 Non-Reserved UMAS Response Messages

Table 1 shows the structure of non-reserved response messages, which are sent by the PLC in response to messages from the management unit. They include an 'ACK/NACK' field, and an optional response 'Data' field.

3.3 Reserved UMAS Request Messages

Reserved messages that are sent during a reserved session include additional fields, see Table 3. Beyond the fields present in non-reserved public messages, reserved signed messages have a non-zero 'Session key', a special 'UMAS sign function code' and a cryptographic 'Signature' fields. The signature calculation is described in Section 4.4, and it requires the nonces exchanged in the authentication messages (see Section 4.2). Only correctly signed messages will be accepted by the PLC. Note that the reserved message is effectively formatted as two concatenated UMAS messages, one with the 'sign' function code 0x38, and the other with whatever function code is being sent. The 'Session key' is duplicated to retain the format.

3.4 Reserved UMAS Response Messages

Table 4 shows the structure of reserved UMAS response messages. They have a structure similar to that of the non-reserved UMAS message response (Table 1), but with the added signature, following the format shown in Table 3 for reserved UMAS request messages.

3.5 Signature Error UMAS Response Messages

In our research we discovered the structure of the (undocumented) UMAS message response sent by the PLC when it receives a reserved UMAS message with an incorrect signature, see Table 5. It only includes a 1-byte 'UMAS error header' with value 0xDA to indicate a signature error, and a 1-byte constant 'Error data' field with value 0x04. This response is independent of the content of the UMAS request message from the management content and is solely related to the incorrect signature.

4 UMAS Authentication

A project in the management unit's Control Expert software is password-protected. As described by [16], when such a project is created, the Control Expert prompts the user to select a password (pwd), and generates

UMAS error header	*Error data*
0xDA	0x04
1 Byte	1 Byte

Table 5: The structure of UMAS message response for incorrectly signed UMAS message request

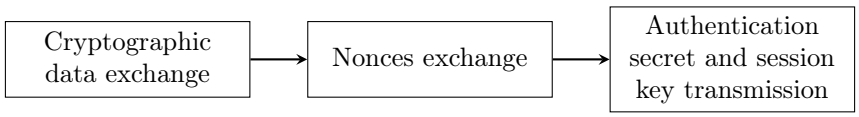

Figure 1: The authentication process stages

a 16 bytes salt value (pwd_{salt}). It then calculates the password hash (pwd_{hash}) as follows:

$$\begin{aligned} pwd_{salt} &= randomBytes(16) \\ pwd_{hash} &= \text{SHA256}(pwd_{salt}||pwd) \end{aligned} \tag{1}$$

Both pwd_{salt} and pwd_{hash} are saved in the APX binary in Base64 encoding. The plaintext password pwd itself is not stored.

Figure 1 shows a high-level view of the the authentication handshake that occurs whenever the management unit establishes a session with the PLC. The handshake assumes that the APX binary is already present on the PLC. The handshake consists of three stages: a cryptographic data exchange, nonces exchange, and authentication secret and session key transmission.

The general structure of the handshake has not changed by the firmware updates since v3.01. However, Schneider did enhance the protection of the stages' implementation on each firmware update to make it successively harder to attack. In the following sections we will go into more detail for each stage, and discuss the changes in their implementation for the different firmware versions.

4.1 Cryptographic Data Transmission

In the first stage of the handshake the management unit requests cryptographic data from the PLC regarding the project password. As mentioned above, the pwd_{hash} and pwd_{salt} are stored in the APX binary in Base64 encoding, but the plaintext password is not. The management unit uses the non-reserved UMAS function 'ReadMemoryBlock' (0x20) to read the values it needs for the handshake from an APX block, see Figures 6 and 7 in the Appendix.

4.1.1 Versions v3.01 to v3.50

As described in CVE-2021-22779 [17], in versions v3.01 up to v3.50, one could send a 'ReadMemoryBlock' UMAS message and obtain both the pwd_{hash} and pwd_{salt} by reading offsets within block 0x14 of the APX (recall Section 2.1.1). Figure 2) shows the UMAS messages used to obtain the pwd_{hash} and pwd_{salt} from the PLC.

Thus the management unit prompts the user for pwd, recalculates pwd_{hash} as in equation (1) using the pwd_{salt} it read from the PLC, and compares the result with the pwd_{hash} it read from the PLC. From the PLC's perspective, in this stage of the handshake the management unit is not authenticated and the session is not reserved yet.

4.1.2 Versions v3.50 and above

Starting from version v3.50, the pwd_{hash} is no longer stored in publicly-readable block 0x14. However, as observed by [17], block 0x14 still stores the pwd_{salt}. By reversing the firmware binary we discovered that pwd_{hash} is still stored in the APX file, but now it is located in read-protected block X (detail omitted). Block

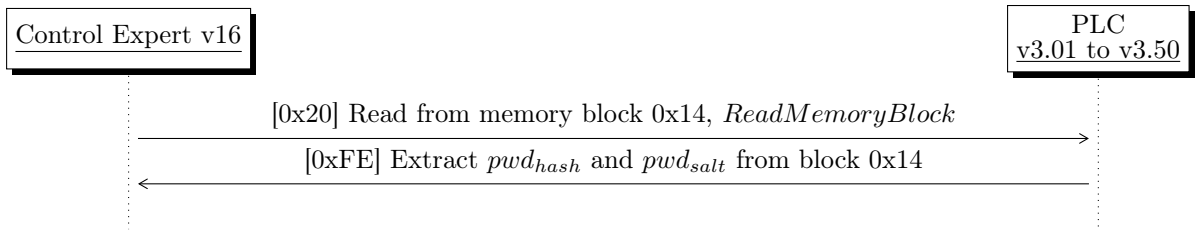

Figure 2: Reading pwd_{hash} and pwd_{salt} using 'ReadMemoryBlock' UMAS message

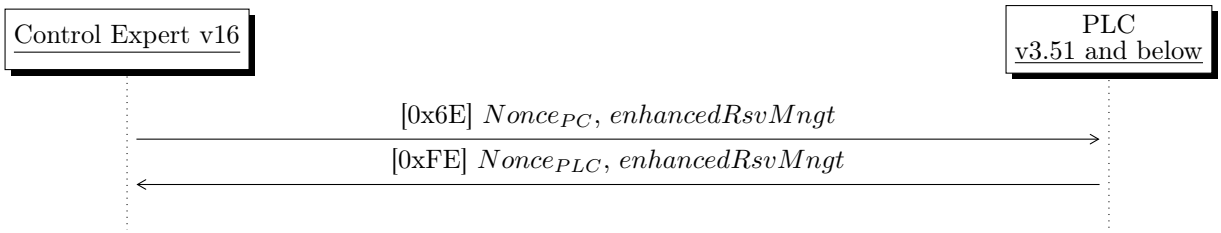

Figure 3: Exchanging nonces between management unit and PLC using 'enhancedRsvMngt' UMAS message

X data is inaccessible via the 'ReadMemoryBlock' UMAS message: the firmware code denies this request and replies with a NACK UMAS message.

Hence in the v3.50 and above, the management unit just calculates pwd_{hash} based on the user-provided pwd and the pwd_{salt} extracted from the UMAS traffic. Note that at this stage the software cannot know whether the pwd given by the user is correct or not since it does not have the expected pwd_{hash} to compare with.

4.2 Nonces Exchange

At this stage, the management unit sends a request for nonces exchange via UMAS message 0x6E to the PLC. Two nonces are exchanged, one sent by the PC and one by the PLC. As we shall see below, these nonces participate in the calculation of the authentication secret.

4.2.1 Version v3.51 and Below

Figure 3 shows the nonces exchange over UMAS traffic as implemented up to version 3.51: the two nonces, $Nonce_{PC}$ and $Nonce_{PLC}$, are exchanged using the 'enhancedRsvMngt' UMAS message. The packet structures of request and response 'enhancedRsvMngt' UMAS messages are shown in Tables 8 and 9 in the Appendix. The 'Mode' field [0x02] indicates that the handshake function in use is 'enhancedRsvMngt'. The 'Reserver ID' field uniquely identifies the reserver: i.e., the management unit.

In the PLC response (Table 9) we also observe the 'ACK/NACK code' [0xFE] and a 'Magic' [0xAAAA] which indicates that the $Nonce_{PC}$ was received properly in the PLC. (recall Section 3.3). Note that in these software versions v3.01 to v3.51 the two nonces are sent in plaintext.

By the end of this stage both the management unit and the PLC share both $Nonce_{PC}$ and $Nonce_{PLC}$.

4.2.2 Version v3.60

In versions up to v3.51 the nonces were exchanged in plaintext, and as such were accessible to a passive eavesdropping adversary. Schneider Electric addressed this vulnerability with version v3.60. However, details of the changes they introduced were not published until now. In our research we shed light on the new nonce-exchange procedure.

By reversing the PLC firmware v3.60, we established that the main modification to the nonce exchange

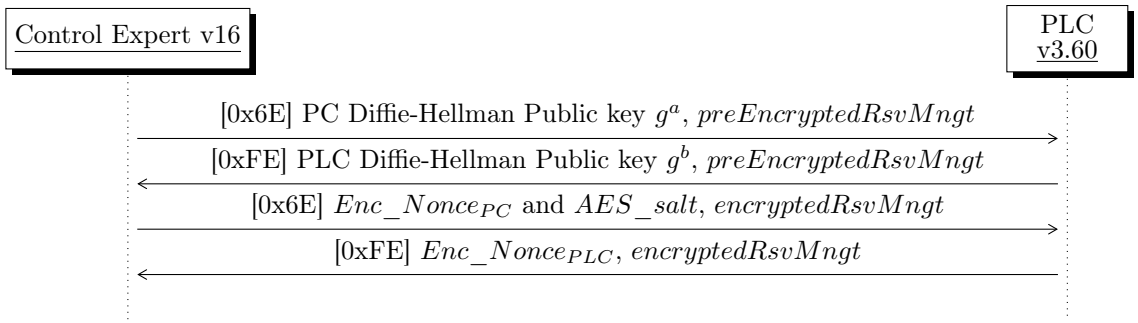

Figure 4: Exchanging nonces between management unit and PLC using v3.60 two-stage UMAS nonces handshake messages

is the introduction of a textbook Diffie-Hellman key exchange to generate an ephemeral shared value ('DH_shared'), from which a salted shared ephemeral AES key ('AES_secret') is derived. The nonces are transmitted encrypted with AES-CBC-256 using 'AES_secret' as the key. Figure 4 shows the message flow.

We identified that the cryptographic computations in the PLC are based on the mbedtls_dhm library, a component of the mbedtls cryptographic open-source library [22]. The modular exponential (MODP) group used for Diffie-Hellman calculations is the RFC-3526 [18] 2048-bit MODP group:

$$\begin{aligned} g &= 2 \\ p &= 2^{2048} - 2^{1984} - 1 + 2^{64} \cdot [(2^{1918}\pi) + 124476] \end{aligned} \tag{2}$$

Tables 10 and 11 in the Appendix describe the structure of the two Diffie-Hellman ('preEncryptedRsvMngt') messages exchanging the public DH values $g^a \pmod p$ and $g^b \pmod p$. Note that in the message from the management unit to the PLC (Table 10) there is a 'Reserver ID' field, a 4 bytes that uniquely identifies the management unit, that is sent in plaintext. At this point, both parties calculate the Diffie-Hellman shared key

$$DH\_shared = g^{ab} \pmod p. \tag{3}$$

The management unit now generates a random 16-byte AES_salt, and derives a shared AES_secret by

$$AES\_secret = \text{SHA256}(AES\_salt || DH\_shared) \tag{4}$$

The management unit encrypts its $Nonce_{PC}$ using AES-CBC-256 (using a fixed $IV = 0$), with AES_secret as the key:

$$Enc\_Nonce_{PC} = \text{AES-CBC-256}_{AES\_secret}(Nonce_{PC})$$

and sends Enc_Nonce_{PC} together with AES_salt as an 'encryptedRsvMngt' message to the PLC. With the received AES_salt the PLC also derives AES_secret according to equation (4), encrypts its own $Nonce_{PLC}$, and sends it to the management unit. Tables 12 and 13 in the Appendix show the structure of the two 'encryptedRsvMngt' messages. At the end of this exchange both sides decrypt the other party's nonce.

4.3 Authentication Secret and Session Key Transmission

Regardless of the firmware version, at this stage both sides have pwd_{hash} (equation (1)), pwd_{salt}, and both nonces. This is all the information that is required to elevate the session to a 'reserved' state and calculate its Session Key. Both sides calculate the $auth\_secret$, as discussed by [17][16] (CVE-2021-22779):

$$auth\_secret = \text{SHA256}\big(Nonce_{PLC} || Base64(pwd_{salt}) || '\backslash r \backslash n' || \text{Base64}(pwd_{hash}) || '\backslash r \backslash n' || Nonce_{PC}\big) \tag{5}$$

The management unit encodes *auth_secret* with UTF-8 and sends it to the PLC using the 'TryReserve' UMAS function (0x10). If the received *auth_secret* is equal to the value calculated by the PLC, it responds with a reservation session key. Tables 14 and 15 in the Appendix show the structure of these two messages.

Through reverse engineering the code, we discovered how the PLC generates the 1-byte Session Key for the reserved session: if the PLC was not reserved before then it picks a random value for the Session Key. Otherwise in increments the previous value by 1 (modulo 256).

We note that having a 1-byte Session Key seems to be weak—however, this value is exchanged in plaintext in all subsequent messages, and knowing it seems not to provide much value to an attacker since there can only be one reserved session at a time.

4.4 Signing Reserved UMAS Messages

As described in Section 3, reserved UMAS messages are signed. The signature algorithm was been researched in prior firmware versions [16, 27]. We can confirm that the signature algorithm remains unchanged up to the latest firmware updates, and is done as follows. Let

$$Msg = \text{'UMAS header'}||\text{'Session key'}||\text{'UMAS function code'}||\text{'Data'}$$

be the request UMAS message content to be signed (for response UMAS message, the 'UMAS function code' is replaced by the 'ACK/NACK code'); it is a concatenation of all "non-reserved" fields in a UMAS message. The signature computation is symmetric. To sign Msg, the signing party uses the two nonces from Section 4.2, hashes each nonce with the PLC_{ID}, and calculates the signature as follows:

$$\begin{aligned} Nonce_{PC_{ID}} &= \text{SHA256}(PLC_{ID}||Nonce_{PC}) \\ Nonce_{PLC_{ID}} &= \text{SHA256}(PLC_{ID}||Nonce_{PLC}) \\ Signature &= \text{SHA256}\Big(Nonce_{PC_{ID}}||Msg||Nonce_{PLC_{ID}}\Big) \end{aligned} \tag{6}$$

The PLC_{ID} is a 4-byte identifier for the PLC that can be obtained by public UMAS messages.

Interestingly, this unusual signature algorithm does not use *auth_secret* (equation (5)), and in fact does not depend on the project password *pwd* nor its hash pwd_{hash} in any way. The algorithm has remained the same in all firmware updates since version v3.01. We argue that this is a significant design weakness: an attacker that learns the two nonces and some public or easy-to-sniff values is able to sign messages at will, without needing to know the project password. As we shall see in Sections 6 and 7, this weakness is an enabler to our advanced attacks leading to code execution.

In our opinion, using a standard symmetric signature like HMAC [19], using a shared key derived from *auth_secret*, would be much stronger than the current method, and would mitigate some aspects of our attacks.

5 Attacks: Reaching a Reserved Session

In this session we describe two new network-level attacks on the latest firmware version v3.60 and later, despite the new Diffie-Hellman addition to the UMAS authentication handshake. The attacks allow the attacker to steal the password hash, mount a pass-the-hash attack to achieve a reserved session, steal the nonces, and allow signing messages at will. One of the attacks is based on a new vulnerability we found; the other is based on the known susceptibility of Diffie-Hellman key exchange to Man-in-the-Middle (MITM) attacks. We implemented both attacks and demonstrated their effectiveness.

5.1 Getting the Password Hash over UMAS

In this section, we show how to capture the pwd_{hash} through passive sniffing of UMAS traffic. This attack is based on the following:

Vulnerability 1 (CVE-2024-8933): The pwd_{hash} is transmitted in plaintext during project upload from or download to the PLC.

UMAS includes function codes to download a project from the management unit to the PLC, and to upload a project from the PLC. Both are reserved commands that need to be signed. However, our research shows that the upload or download traffic transfers the full APX binary, and *is not encrypted*. As described in Section 4, the APX binary includes both the pwd_{hash} and pwd_{salt}. Therefore, a passive attacker can capture these two parameters by sniffing the traffic when a legitimate upload or download process is occurring.

When the management unit performs a project upload from the PLC, it sends an 'UploadPacket' reserved UMAS message (0x34). The structure of this message and the PLC's response appear in Tables 16 and 17 in the Appendix.

Similarly, UMAS supports the project upload, in the opposite direction: The management unit sends a 'DownloadPacket' reserved UMAS message (0x31). The structure of this message and the PLC's response appear in Tables 18 and 19 in the Appendix.

An immediate implication of stealing the pwd_{hash} and pwd_{salt} from the upload/download UMAS traffic is that cracking the password from the password hash becomes possible: As an example, we modified the popular hashcat [15] password cracker to support the hash calculation of Equation (1) and successfully cracked our own projects' password.

5.1.1 Notes Regarding Encrypted Projects

In addition to a project password, the Control Expert software also supports a project encryption password *enc_pwd* (recall Section 2.1.2.2). We discovered two important points worth mentioning regarding encrypted projects:

- Even if the project file is encrypted on the PC, it is transferred (both downloaded and uploaded) *in cleartext* over the UMAS traffic. The project is only kept in encrypted form on the PC in the STU file. Thus by sniffing the traffic during project upload or download, one can retrieve the complete APX binary, including pwd_{hash} and pwd_{salt}, even if the project is encrypted.
- The SHA256 hash of the *enc_pwd* (2.1.2.2) and its corresponding salt are also located in the APX binary alongside pwd_{hash} and pwd_{salt}.

5.2 Pass-the-Hash and Creating a Reserved Session

Once the attacker successfully acquires the pwd_{hash} and pwd_{salt} it can establish an authenticated reserved session via a "pass-the-hash" attack. This was demonstrated against versions 3.51 and below in CVE-2021-22779 ([3, 17, 16]). We demonstrate that this is possible also against the updated handshake in v3.60 by following the steps described in Section 4.2. The flow transmission diagram of the attack is shown in Figure 5.

First, the attacker selects an arbitrary 4-byte $reserverID$ and a 32-byte private z, and computes its corresponding $g^z \pmod p$, constructs and sends the 'preEncryptedRsvMngt' message (Table 10 in the Appendix). The PLC responds with its $g^b \pmod p$ 'preEncryptedRsvMngt' message. The attacker can now calculate DH_shared according to equation (3). Next it selects an arbitrary 16-byte AES_salt and a 32-byte $Nonce_{Attacker}$ and derives the AES_secret according to equation (4). It encrypts $Nonce_{Attacker}$ using AES_secret and sends it to the PLC along with the $reserverID$ and AES_salt, using the 'encryptedRsvMngt' message (Table 12 in the Appendix). Once it receives the encrypted nonce Enc_Nonce_{PLC} from the PLC it decrypts it to $Nonce_{PLC}$ with the AES_secret. At this point, with the stolen pwd_{hash} and pwd_{salt} the attacker has all the information to calculate the $auth\_secret$ according to equation (5). The attacker sends the $auth\_secret$ using the 'TryReserve' message alongside an arbitrary PC_name and the

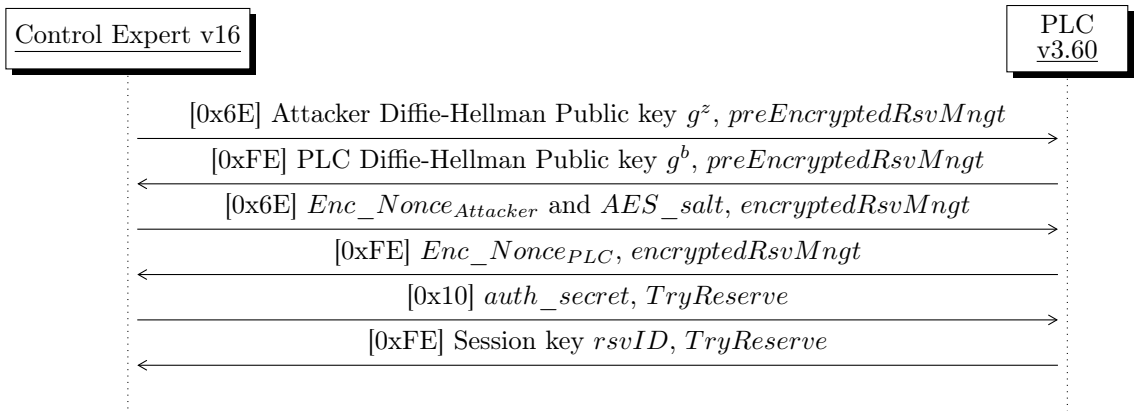

Figure 5: Authenticated reservation process initiated by an attacker (firmware v3.60)

$reserverID$ (Table 14). Upon completion of the process, the attacker establishes an authenticated reserved session with the PLC.

We implemented this attack in python and demonstrated that it works successfully against our PLC.

5.3 Man-in-the-Middle Attack against Diffie-Hellman

Independently of whether the attacker obtained pwd_{hash} or not, if it can attain a Man-in-the-Middle (MITM) network position, it can steal the nonces due to the well-known vulnerability of unprotected Diffie-Hellman:

Vulnerability 2 (CVE-2024-8935): The nonce exchange is vulnerable to a Man-in-the-Middle attack.

We can mount a man-in-the-middle (MITM) attack against the Diffie-Hellman nonce exchange to gain access to a reserved session in firmware v3.60. Our objective is to obtain the decrypted nonces exchanged during the process and utilize them to sign reserved messages within the session. We base on the authentication process theory shown in Section 4.

A MITM network position can be reached in various ways depending on the attackers level of access, including taking over a network router on the path, injecting interception software into a virtualization platform hypervisor, DNS poisoning, etc. In our demonstration we chose to achieve a MITM position using ARP spoofing, assuming that the management unit PC and the PLC are on the same IP subnet and that the attacker has network access to that same subnet. As in [2] we used the ettercap [12] package for this purpose. The message flow of the attack is shown in Figure 6.

First, the attacker mounts an ARP poisoning attack to establish a MITM network position. Then the attacker intercepts the 'preEncryptedRsvMsg' message transmitted from the management unit to the PLC as a trigger. It implements two separate DH exchanges, one with the management unit and the other with the PLC, establishing two DH_shared values ($g^{az} \pmod p$ and $g^{zb} \pmod p$ following the notation in Figure 6).

It then intercepts the management unit's 'encryptedRsvMsg' message, extracts the AES_salt, computes the AES_secret_{PC} towards the management unit, and decrypts the $Nonce_{PC}$.

The attacker then reuses the AES_salt it received with g^{zb} to derive AES_secret_{PLC} toward the PLC and uses it to encrypt the $Nonce_{PC}$. The attacker transmits $Enc_{PLC}\_Nonce_{PC}$ alongside AES_salt to the PLC. Then, the PLC replies with $Enc_{PLC}\_Nonce_{PLC}$. The attacker decrypts it using AES_secret_{PLC}, and encrypts it to $Enc_{PC}\_Nonce_{PLC}$ using AES_secret_{PC}. Finally, the attacker transmits to the management unit the $Enc_{PC}\_Nonce_{PLC}$.

By the end of this stage, all three sides have the same nonces, $Nonce_{PC}$ and $Nonce_{PLC}$. Note that the reuse of the AES_salt and $Nonce_{PC}$ is not mandatory, the attacker can select its own salt and nonce toward

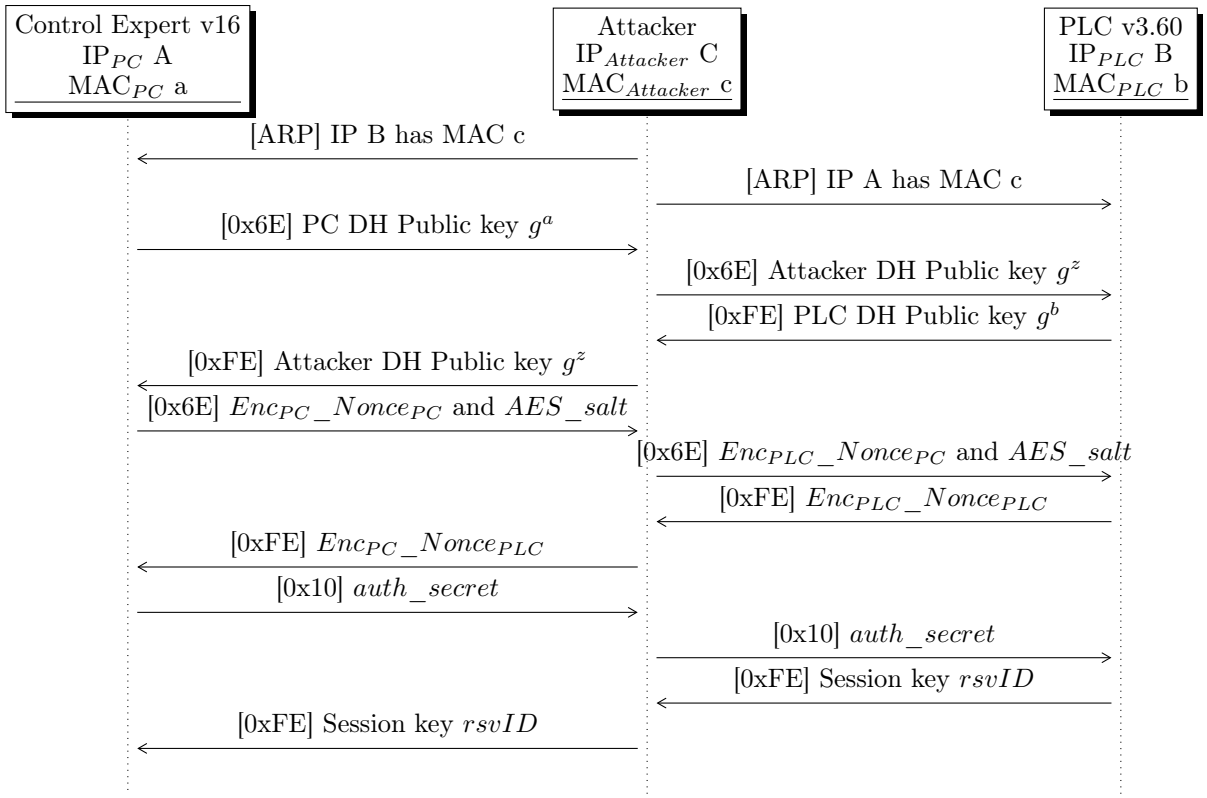

Figure 6: A MITM attack using ARP spoofing against the nonce exchange

the PLC, but reusing the values sent by the management unit allows the attacker to avoid re-signing all subsequent messages.

In the last 2 'TryReserve' messages the attacker simply forwards the messages between the management unit and the PLC, leaving the packets unchanged.

The implication of this attack is that now the attacker has both nonces of an active reserved session, and is able to inject arbitrary signed messages into the session. In and of itself this attack does not provide the pwd_{hash}, since it is not exchanged in the handshake—however, as we already noted above, only the nonces are needed to forge reserved message signatures.

6 From a Reserved Session to Full Memory Read Access

Given the ability to inject signed reserved messages into a session, we now elevate the attack privileges in two ways:

- Read all memory over UMAS messages: in other words, bypassing the mitigations to CVE-2020-7537 introduced in v3.30 ([8]).
- Write to all the PLC data memory via UMAS messages: i.e., exploit CVE-2019-6829, which remains unmitigated in v3.60

We do so using two UMAS messages: the non-reserved public message 'ReadPhysicalAddress' (0x28) to read memory, and the reserved signed message 'WritePhysicalAddress' (0x29) to modify memory.

By default 'ReadPhysicalAddress' message is restricted and allows access only to the low addresses of the data memory. This limitation was introduced in v3.30 to mitigate an information leak CVE-2020-7537 (See

Schneider's security notification [8]). We exploit a new vulnerability to remove the limitation and allow full memory access.

As for modifying memory, 'WritePhysicalAddress' can be modify any writable address. This property, noted in CVE-2019-6829 by Talos ([26]), was exploited to achieve RCE by Armis ([3]). Nonetheless the 'WritePhysicalAddress' logic was not changed in v3.60 and it is still capable of modifying any writable address, which we exploit. As we shall see in Section 7 we also exploit 'WritePhysicalAddress' to inject code to memory in the same manner as Armis did. For this we need to introduce a some additional background material.

6.1 Background on Modicon M340 Memory Protection

The operating system running on the PLC, VxWorks, includes 'vmLib' - an architecture-independent library for managing virtual memory access permissions. This library provides the 'vmStateSet' function, which is used to modify the state of a page in the processor's virtual memory. There are three types of states available:

- 'VM_STATE_VALID' or 'VM_STATE_VALID_NOT' - used to specify that a memory page is accessible or not.
- 'VM_STATE_WRITABLE' or 'VM_STATE_WRITABLE_NOT' - used to specify that a memory page is writable or write-protected.
- 'VM_STATE_CACHEABLE' or 'VM_STATE_CACHEABLE_NOT' - used to specify that a memory page is cacheable or not-cacheable.

In our case, we focus on two categories of memory regions: executable code memory and data memory. The executable code memory regions (firmware code area and executable application blocks) are protected against writing ('VM_STATE_WRITABLE_NOT' state), while the data memory regions are not protected ('VM_STATE_WRITABLE' state) ([27, 14]). The write protection operates at the internal processor level, meaning even privileged reserved UMAS messages cannot modify pages protected by 'vmLib'.

The ARMv4T processor in the PLC does not have the 'Non-Execute' (NX) bit, a feature meant to restrict code execution from unintended memory pages.

The implications of these hardware and OS functionalities are twofold:

- PLC data memory regions are writeable;
- PLC data memory regions are executable.

6.2 Memory Read over UMAS

Reading from the PLC memory over UMAS is available using the non-reserved 'ReadPhysicalAddress' (0x28) message. The packet structure sent from the management unit is shown in table 20 in the Appendix, and the packet structure received from the PLC is shown in table 21. However, the read access using this function is limited to a specific set of permitted addresses. This restriction is due to the firmware code implementation, and is not enforced by 'vmLib' (recall Section 6.1). Through reverse engineering, we discovered the algorithm used to set the read limit. There is a variable, which we call $addressReadLimiter$, located in the PLC data memory, which is used in conjunction with two parameters we call $Base$ and $factor$ to set the highest address readable by 'ReadPhysicalAddress', as follows:

$$maxAddressAllowed \quad = \quad Base + Factor \times addressReadLimiter$$

In our research we discovered the precise value of $addressReadLimiter$ and established that it remains stable across reboots, and we also have the exact values of the Base and Factor (details are omitted).

However, we discovered the following vulnerability:

Vulnerability 3 (CVE-2024-8936): addressReadLimiter is writeable over UMAS.

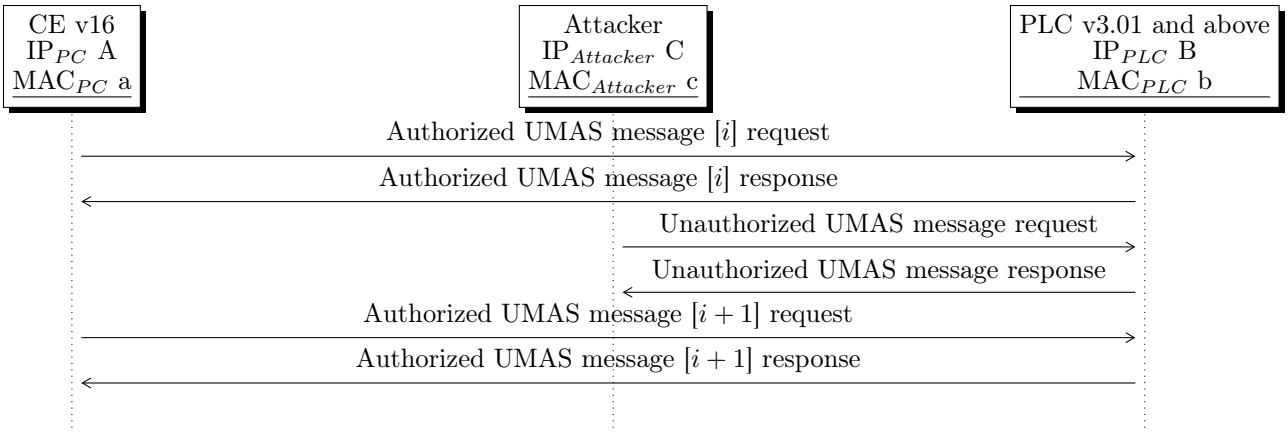

Figure 7: An Injection of unauthorized UMAS message during authorized UMAS session

This is since *addressReadLimiter* is stored in a *writable* data memory segment ('VM_STATE_WRITABLE' state) and as such it can be modified using a properly signed 'WritePhysicalAddress' message.

Modifying the PLC memory over UMAS is available using reserved 'WritePhysicalAddress' (0x29) message, see Tables 22, and 23 in the Appendix. Given that we can sign a reserved 'WritePhysicalAddress' message, and inject it to the PLC, and since the *addressReadLimiter* is writable—then we can modify its value as we desire. By doing so we can configure *maxAddressAllowed* to be the top of memory.

This attack has the following critical implications:

- It is now possible to read the entire memory over UMAS using the public non-reserved 'ReadPhysical-Address' (0x28) message.
- In particular we can extract the pwd_{hash} from the PLC memory (details omitted).
- We can read $Nonce_{PLC_{ID}}$ and $Nonce_{PC_{ID}}$ from a parallel session (details omitted).
- The new value of *addressReadLimiter* persists until the PLC reboots.

In other words: the attacker only needs to implement the MITM attack of Section 5.3 once, and within the hijacked and reserved session, update the *addressReadLimiter*, and leave. It can now read the pwd_{hash}, and recreate the pass-the-hash attack of Section 5.2 at will. Furthermore, if when the attacker wishes to connect there is an active reserved session, it can read that session's nonces from PLC memory, and is able to sign reserved messages.

6.2.1 Injecting Unauthorized UMAS Messages

If the attacker stole the nonces from a legitimate active reserved session, it is able to sign messages. Our research revealed that in order to send such reserved messages to the PLC it is not strictly required to inject them into the active session. Instead, the attacker can open a separate connection on TCP/502, and send the signed messages through it. The PLC accepts such UMAS messages as long as the data and the signatures within the packets are valid. This applies to both non-reserved and reserved sessions. Figure 7 illustrates an injection UMAS message from an unauthorized and unreserved TCP connection, during an active session between the PLC and the authorized management unit.

This behavior may possibly be reasonable for public non-reserved messages, but it makes the attacker's task much easier since injecting messages from a separate TCP connection does not require tracking and compensating the TCP sequence numbers. We argue that accepting *signed* messages outside the reserved session is redundant—there should be at most one reserved session at any time.

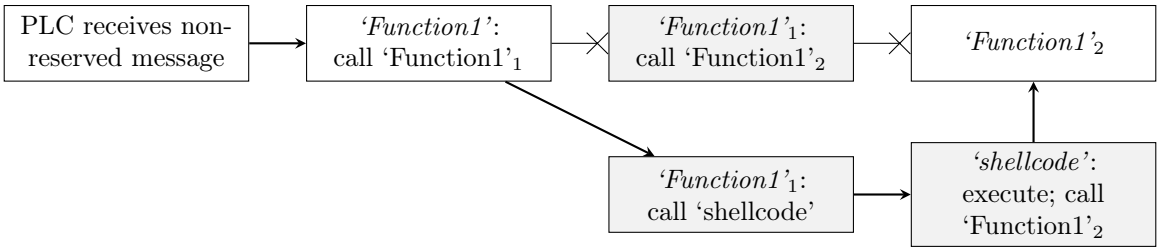

Figure 8: Two-step function pointer redirection of 'Function1'. White rectangles are actions within code memory regions, and grey rectangles are actions within data memory regions.

7 Attacks: Reaching Remote-Code Execution

As noted in Section 6.1, all the executable memory areas in the PLC firmware are write-protected at the internal processor level. As a result, even valid reserved UMAS write messages cannot inject code into these area. However, all the PLC data memory is executable, thus injecting shellcode over the network using UMAS 'WritePhysicalAddress' is possible (once the attacker is able to sign messages): we only need to locate a free writable memory region to store the shellcode. The only remaining challenge to achieve code execution is the ability to trigger a jump to the shellcode.

Our reverse engineering efforts revealed *two vulnerable indirect function calls* where the function pointers are located outside the write-protected code area and are writable: i.e., we found two function pointers in the code memory that call a secondary function pointer in the writable area, which in turn point to functions situated in the code memory area. These vulnerabilities are:

Vulnerability 4 (CVE-2024-8937): 'Function1'[1] secondary function pointer overwrite leads to code execution over UMAS.

Vulnerability 5 (CVE-2024-8938): 'Function2' secondary function pointer overwrite leads to code execution over UMAS.

Both function pointers can be triggered with non-reserved UMAS messages. The exploitation requires overwriting the secondary function pointers to point to the shellcode (via a reserved signed UMAS 'WritePhysicalAddress' command), and then sending a triggering non-reserved UMAS message (which varies for each case).

7.1 Redirecting the Secondary Function Pointers

We discovered that these function are called through a two-step function pointer, where the first pointer ('Function1'$_1$/'Function2'$_1$) is write-protected but the secondary pointer ('Function1'$_2$/'Function2'$_2$) is writable. Details are omitted.

To exploit this, we modify the secondary function pointers ('Function1'$_2$ and/or 'Function2'$_2$) to execute our shellcode, which then calls the original target function 'Function1' and/or 'Function2' respectively. A flow diagram is shown in Figure 8 for 'Function1' (the same applies for 'Function2').

It is crucial to note that this function is in use in all firmware versions since v3.01. Therefore, the vulnerability associated with exploiting calls to this function affects all these firmware versions.

8 Conclusions

In this paper showed that version v3.60 of the UMAS protocols, which was the most recent at the time of writing, was vulnerable. We identified vulnerabilities through which we were able to masquerade as the EcoStruxure Control Expert to the PLC, cryptographically sign messages, and inject any messages favourable

[1]The real function names in the firmware code are redacted.

to the attacker. Moreover, we identified additional vulnerabilities in the M340 PLC's memory management, which in conjunction with the protocol vulnerabilities could be exploited to create read-anywhere and write-anywhere primitives over the network. We demonstrated that these primitives could lead to remote code execution. In many cases we also provided suggestions on how the vulnerabilities we identified might be mitigated. We disclosed our findings to the vendor.

8.1 Schneider Electric Response

Schneider Electric verified all five reported vulnerabilities (and assigned the CVE numbers) and confirmed the accuracy of the paper's content. The company released firmware update 3.65 in November 2024 to address the read-limit bypass and the two code-execution vulnerabilities, effectively disrupting the attack chain. Furthermore, Schneider Electric recommend several mitigations: activation of memory protection on the PLC, blocking unauthorized access to port Modbus/TCP and implementing a VPN, to defend against the two other vulnerabilities. Two security notifications detailing our findings and the mitigations are published on the Schneider Electric website: SEVD-2024-317-02 [9] and SEVD-2024-317-03 [10].

We sincerely appreciate Scheneider Electric's collaboration handling the disclosure these vulnerabilities. This partnership ensured that the disclosure process was handled responsibly, with a focus on protecting the systems and the customers.

References

[1] *Modbus Application Protocol V1.1b*. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf.

[2] Nichole Anne. Packet modification attack on PLC with ARP spoofing (MITM attack). *Medium*, 2020. https://medium.com/@npcole/packet-modification-attack-on-plc-with-arp-spoofing-mitm-attack-f0c4d58e3e83.

[3] Armis. The Vulnerability Can Lead to Native Remote-Code-Execution on Vulnerable PLCs, 2020. Armis, https://www.armis.com/research/modipwn/.

[4] biero-el corridor. Wireshark umas modicon m340 protocol dissector github, 2022. https://github.com/biero-el-corridor/Wireshark-UMAS-Modicon-M340-protocol/blob/main/modbus-umas-schneider.lua.

[5] E. Biham, S. Bitan, A. Carmel, A. Dankner, U. Malin, and A. Wool. Rogue7: Rogue engineering-station attacks on S7 Simatic PLCs. In *Black Hat Briefings*, Las Vegas, August 2019.

[6] T.M. Chen. Stuxnet, the real start of cyber warfare? *IEEE Network*, 24(6):2–3, 2010.

[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] Schneider Electric. Schneider Electric security notification - Modicon M580, Modicon M340, legacy controllers Modicon Quantum & Modicon Premium, December 2020. Schneider Electric, https://download.schneider-electric.com/files?p_Doc_Ref=SEVD-2020-343-08.

[9] Schneider Electric. Schneider Electric security notification - Modicon M340, Momentum & MC80, Novmeber 2024. Schneider Electric, https://download.schneider-electric.com/doc/SEVD-2024-317-02/SEVD-2024-317-02.pdf.

[10] Schneider Electric. Schneider Electric security notification - Modicon M340, Momentum & MC80, Novmeber 2024. Schneider Electric, https://download.schneider-electric.com/doc/SEVD-2024-317-02/SEVD-2024-317-03.pdf.

[11] Liras en la red. The Unity (UMAS) protocol, 2017. http://lirasenlared.blogspot.com/2017/08/the-unity-umas-protocol-part-i.html.

[12] Ettercap home page. https://www.ettercap-project.org/.

[13] 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.

[14] Barak Hadad, Gal Kauffman, and Ben Seri. Exploring and exploiting programmable logic controllers with URGENT/11 vulnerabilities, 2020. Armis, https://info.armis.com/rs/645-PDC-047/images/Armis-URGENT11-on-OT-WP.pdf.

[15] hashcat. Hashcat advanced password recovery, 2019. http://hashcat.net.

[16] Gao Jian. Going deeper into Schneider Modicon PAC security. In *HITB Security Conference*, 2021. https://conference.hitb.org/hitbsecconf2021sin/materials/D1T2%20-%20Going%20Deeper%20into%20Schneider%20Modicon%20PAC%20Security%20-%20Gao%20Jian.pdf.

[17] Kaspersky. The secrets of Schneider Electric's UMAS protocol, 2022. https://ics-cert.kaspersky.com/publications/reports/2022/09/29/the-secrets-of-schneider-electrics-umas-protocol/.

[18] T. Kivinen and M. Kojo. RFC 3526: More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE), May 2023.

[19] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. RFC 2104: Hmac: Keyed-hashing for message authentication. Technical report, 1997.

[20] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.

[21] Seok Min Lim. Attacking SCADA: Vulnerabilities in Schneider Electric SoMachine and M221 PLC (CVE-2017-6034 and CVE-2020-7489). *Trustwave SPIDERLABS BLOG*, May 2020. https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/vulnerabilities-in-schneider-electric-somachine-and-m221-plc/.

[22] Mbed-TLS / mbedtls. https://github.com/Mbed-TLS/mbedtls.

[23] Nicholas Miles. Examining crypto and bypassing authentication in Schneider Electric PLCs (M340/M580). *Medium Tenable Techblog*, 2021. https://medium.com/tenable-techblog/examining-crypto-and-bypassing-authentication-in-schneider-electric-plcs-m340-m580-f37cf9f3ff34.

[24] Thrive Reports. Programmable Logic Controller (PLC) Market Size, Growth and Forecast from 2023 - 2030, 2023. https://www.linkedin.com/pulse/programmable-logic-controller-plc-market-size-growth-forecast-v8ure/.

[25] Mashav Sapir, Uri Katz, Noam Moshe, Sharon Brizinov, and Amir Preminger. Evil PLC attack: Weaponizing PLCs, 2022. Claroty, https://web-assets.claroty.com/resource-downloads/team82-evil-plc-attack-research-paper-1661285586.pdf.

[26] Talos. Schneider Electric Modicon M580 UMAS Function Code 0x29 Denial of Service Vulnerability, August 2019. https://talosintelligence.com/vulnerability_reports/TALOS-2019-0807.

[27] Jos Wetzels. Nakatomi space - lateral movement as L1 post-exploitation in OT. In *Black Hat Asia*, 2023. https://www.classcentral.com/course/youtube-nakatomi-space-lateral-movement-as-l1-post-exploitation-in-ot-228451.

[28] Wireshark - network protocol analyzer. www.wireshark.org.

A Message Formats

UMAS header	*Session key*	*UMAS function code*	*Block ID*	*Address offset*	*Size*
0x5A	0x00	0x20	...	...	...
1 Byte	1 Byte	1 Byte	1 Byte	4 Bytes	2 Bytes

Table 6: The structure of 'ReadMemoryBlock' UMAS message request structure

UMAS header	*Session key*	*ACK/NACK code*	*Data*
0x5A	0x00	0xFE	...
1 Byte	1 Byte	1 Byte	Variable size

Table 7: The structure of 'ReadMemoryBlock' UMAS message response

UMAS header	*Session key*	*UMAS function code*	*Mode*	*Reserver ID*	$Nonce_{PC}$	*Padding*
0x5A	0x00	0x6E	0x02	$reserverID$	...	0x0000
1 Byte	1 Byte	1 Byte	1 Byte	4 Bytes	32 Bytes	2 Bytes

Table 8: The structure of 'enhancedRsvMngt' UMAS message

UMAS header	*Session key*	*ACK/NACK code*	*Magic*	$Nonce_{PLC}$
0x5A	0x00	0xFE	0xAAAA	...
1 Byte	1 Byte	1 Byte	2 Bytes	32 Bytes

Table 9: The structure of 'enhancedRsvMngt' UMAS message response

UMAS header	*Session key*	*UMAS function code*	*Mode*	*Reserver ID*	DH_public_{PC}
0x5A	0x00	0x6E	0x03	$reserverID$	...
1 Byte	1 Byte	1 Byte	1 Byte	4 Bytes	256 Bytes

Table 10: The structure of 'preEncryptedRsvMngt' UMAS message request

UMAS header	*Session key*	*ACK/NACK code*	*Magic*	DH_public_{PLC}
0x5A	0x00	0xFE	0xAAAA	...
1 Byte	1 Byte	1 Byte	2 Bytes	256 Bytes

Table 11: The structure of 'preEncryptedRsvMngt' UMAS message response

UMAS header	*Session key*	*UMAS function code*	*Mode*	*Reserver ID*	Enc_Nonce_{PC}	AES_salt
0x5A	0x00	0x6E	0x04	$reserverID$	...	...
1 Byte	1 Byte	1 Byte	1 Byte	4 Bytes	32 Bytes	16 Bytes

Table 12: The structure of 'encryptedRsvMngt' UMAS message request

UMAS header	*Session key*	*ACK/NACK code*	*Magic*	Enc_Nonce_{PLC}
0x5A	0x00	0xFE	0xAAAA	...
1 Byte	1 Byte	1 Byte	2 Bytes	32 Bytes

Table 13: The structure of 'encryptedRsvMngt' UMAS message response

UMAS header	*Session key*	*UMAS function code*	*Reserver ID*	*PC_name and auth_secret length*	*PC name*	*Pad*	*Authentication secret*
0x5A	0x00	0x10	$reserverID$	...	PC_name	0x00	$auth\_secret$
1 Byte	1 Byte	1 Byte	4 Bytes	1 Byte	Up to 64 Bytes	1 Byte	64 Bytes

Table 14: The structure of the 'TryReserve' UMAS message respone

UMAS header	*Session key*	*ACK/NACK code*	*Generated session key*
0x5A	0x00	0xFE	$rsvID$
1 Byte	1 Byte	1 Byte	1 Byte

Table 15: The structure of the 'TryReserve' UMAS message reponse

UMAS header	*Session key*	*UMAS function code*	*Magic*	*Buffer number*
0x5A	$rsvID$	0x34	0x0001	...
1 Byte	1 Byte	1 Byte	2 Bytes	2 Bytes

Table 16: The structure of the non-reserved part of the 'UploadPacket' UMAS reserved message request

UMAS header	*Session key*	*ACK/NACK code*	*Buffer size*	*Buffer data*
0x5A	$rsvID$	0xFE	...	...
1 Byte	1 Byte	1 Byte	2 Bytes	Variable size

Table 17: The structure of the non-reserved part of the 'UploadPacket' UMAS reserved message response

UMAS header	*Session key*	*UMAS function code*	*Magic*	*Buffer number*	*Buffer size*	*Buffer data*
0x5A	$rsvID$	0x31	0x0001	...	...	...
1 Byte	1 Byte	1 Byte	2 Bytes	2 Bytes	2 Bytes	Variable size

Table 18: The structure of the non-reserved part of the 'DownloadPacket' UMAS reserved message request

UMAS header	*Session key*	*ACK/NACK code*	*Magic*
0x5A	$rsvID$	0xFE	0x00
1 Byte	1 Byte	1 Byte	1 Byte

Table 19: The structure of the non-reserved part of the 'DownloadPacket' UMAS reserved message response

UMAS header	*Session key*	*UMAS function code*	*Address*	*Size*
0x5A	0x00	0x28	...	...
1 Byte	1 Byte	1 Byte	4 Byte	2 Byte

Table 20: The structure of the non-reserved 'ReadPhysicalAddress' UMAS message request

UMAS header	*Session key*	*ACK/NACK code*	*Size*	*Data*
0x5A	0x00	0xFE	...	...
1 Byte	1 Byte	1 Byte	2 Byte	Variable size

Table 21: The structure of the non-reserved 'ReadPhysicalAddress' UMAS message response

UMAS header	*Session key*	*UMAS function code*	*Address*	*Size*	*Data*
0x5A	$rsvID$	0x29	...	...	...
1 Byte	1 Byte	1 Byte	4 Byte	2 Byte	Variable size

Table 22: The structure of the non-reserved part of 'WritePhysicalAddress' UMAS message request

UMAS header	*Session key*	*ACK/NACK code*
0x5A	$rsvID$	0xFE
1 Byte	1 Byte	1 Byte

Table 23: The structure of the non-reserved part of 'WritePhysicalAddress' UMAS message response