Firmware modification attacks on programmable logic controllers

Zachry Basnight, Jonathan Butts*, Juan Lopez Jr., Thomas Dube

Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio 45433, USA

ABSTRACT

Recent attacks on industrial control systems, such as the highly publicized Stuxnet malware, have intensified a “race to the bottom” where lower-level attacks have a tactical advantage. Programmable logic controller (PLC) firmware, which provides a software-driven interface between system inputs and physical outputs, can be easily modified at the user level. Efforts directed at protecting against firmware modification are hindered by the lack of foundational research about attack development and implementation. This paper examines the vulnerability of PLCs to intentional firmware modifications in order to obtain a better understanding of the threats posed by PLC firmware modification attacks and the feasibility of these attacks. A general firmware analysis methodology is presented, and a proof-of-concept experiment is used to demonstrate how legitimate firmware can be updated and uploaded to an Allen-Bradley ControlLogix L61 PLC.

Published by Elsevier B.V.

1. Introduction

Modern industrial applications employ advanced automation and management networks that are collectively referred to as industrial control systems. These systems are responsible for the precise and consistent operation of critical infrastructure assets. The reliance of industrial control systems on modern information technology solutions, including IP-based networking and embedded computing, has raised serious security concerns [19]. The inexorable amalgamation of technologies from two traditionally distinct cultures has created a schism with regard to the security capabilities of information technology and industrial control systems. Indeed, industrial control system security is well behind information technology system security in terms of the sophistication and scale of security policies, techniques and tools.

Meanwhile, cyber attacks on industrial control systems are increasing in intensity [20]. Examples such as Stuxnet provide insight into future cyber threats on industrial control systems [7]. Like traditional attacks on information technology systems, attacks on industrial control systems are targeting lower-level control to allow for more powerful and flexible system manipulation. The allure of industrial control system attacks – and the ultimate goal of malicious manipulation – is the ability to elicit physical manifestations through cyber means. As the final link between cyber and physical components of industrial control systems, programmable logic controllers (PLCs) are critical to the operation of critical infrastructure assets. PLCs are embedded devices that are programmed to manage and control physical components based on system inputs and requirements. The lowest programming abstraction layer of a PLC is the firmware. Malicious modification or counterfeiting of PLC firmware can provide an adversary with complete control over an industrial control device and any physical system components that come under its purview.

This paper examines the feasibility of firmware modification attacks on PLCs. Specifically, it investigates and assesses the...
vulnerability of a common PLC to counterfeit firmware updates. The PLC vulnerability is investigated by first using reverse engineering techniques to infer the firmware update validation method. The firmware update validation method is then analyzed for weaknesses that facilitate firmware modification and counterfeiting. The weaknesses are subsequently exploited to create a counterfeit firmware sample that is uploaded and executed on a PLC. A case study involving the popular Allen-Bradley ControlLogix L61 controller is described. The results demonstrate that PLCs can be highly vulnerable to firmware modification attacks as a result of design weaknesses associated with their firmware update validation methods.

2. Background

PLCs are embedded computer systems that are specifically designed to control and to some extent independently monitor the physical system components under their supervision. PLCs are commonly used in distributed control systems and very often as field devices in supervisory control and data acquisition (SCADA) systems. PLCs enable customized control of system components by providing a user-programmable interface between physical inputs and outputs. Proprietary software installed on a standard computer, typically running Microsoft Windows, is required to program a PLC. Examples of programming software include the Rockwell RSLogix series for managing Allen-Bradley controllers and Siemens Simatic Step 7 for Simatic controllers. Control engineers commonly utilize a graphical programming language such as ladder logic to specify how the controller should respond to inputs. By creating a virtual project in the programming application, an engineer can view the logic currently running on a PLC as well as write new logic for the device.

McMinn et al. [14] have delineated three PLC operational layers: (i) programming layer; (ii) firmware layer; and (iii) hardware layer. Fig. 1 illustrates the three operational layers. This section discusses the three layers and the methods used to validate PLC firmware.

2.1. Programming layer

The programming layer is the main channel of interaction between control system operators and a PLC. This layer is also used to provide the device with the logic that it uses to perform control operations. A variety of languages, including versions of traditional languages such as C or BASIC, are used in the programming layer [11]. However, most PLCs are programmed using a graphical language called ladder logic. Ladder logic provides engineers, who may be unfamiliar with traditional programming languages, an intuitive interface to the controller. Programming software such as RSLogix is often used to develop the graphical ladder logic code, compile it to low-level code for execution and load the compiled code on the device. The code loaded on a PLC dictates how the controller responds to inputs.

Because the loading of PLC code is managed by programming software, the loaded code can be easily compared with the original code residing on the device.

2.2. Firmware layer

The firmware layer bridges the programming layer and the hardware layer. Firmware is the low-level software that executes on a device to support higher level operations. For this reason, firmware is commonly referred to as the operating system of an embedded device. In a broader sense, however, firmware also includes lower-level functionality such as bootloader code that initializes and loads the operating system.

The firmware layer controls the basic behavior of a device, including its communications with management systems and the execution of compiled user-level programs loaded on the device. The firmware handles all interactions between the user and the device hardware, including physical inputs and outputs. The functional analogy of operating system firmware is further extended when discussing potential threats to a PLC. A rootkit on a traditional computer system typically exploits kernel-level processes to gain privileged access to operating system functionality. Using this access, the rootkit is able to modify the underlying behavior of the operating system, including hiding the fact that it exists. In the same way, an attacker with access to the firmware on a PLC has potentially unlimited control over the device, including the ability to covertly alter device behavior.

In some embedded devices, the firmware is installed at the factory and the devices cannot be reprogrammed by users; this firmware is relatively safe from modification attacks. However, modern embedded devices, including PLCs, are commonly designed with a firmware update feature. This feature enables vendors to patch bugs and upgrade firmware without requiring physical changes to the hardware. For a device with programmable firmware, the task of updating the firmware is usually the responsibility of the user and is performed using a software update package. Since the user must have the appropriate access to a device in order to update its firmware, the opportunity exists for an attacker to leverage this access to upload malicious firmware to the device.

2.3. Hardware layer

The hardware layer comprises microchips and other electronic components. The key components include microprocessors, volatile memory and non-volatile storage.

In general, three main vectors are available to attack the hardware layer: physical manipulation of the hardware, software exploitation of hardware design flaws and supply chain attacks. However, hardware is relatively safe from modification attacks. Hardware is the least accessible layer and is typically physically isolated from the rest of the system. This isolation makes it difficult for an attacker to gain access to the hardware. However, even with this isolation, hardware is susceptible to attacks such as reverse engineering and physical manipulation. Reverse engineering involves analyzing the hardware to infer its design and functionality. Physical manipulation involves physically altering the hardware to modify its behavior. These attacks can be difficult to detect and may require specialized tools and expertise.

Fig. 1 – Operational layers of a programmable logic controller.
compromise. Of these, physical manipulation is the least likely in an operational scenario as it would require the attacker to have intimate access to the device, possibly for an extended period of time. Such an attack would typically involve a malicious insider, in which case, more straightforward methods of attack such as direct malicious reprogramming are possible. The second vector, involving the exploitation of hardware flaws via software running on the device, requires compromises within the programming layer or the firmware layer. The third attack vector, which is most insidious, is a supply chain compromise. In this case, the attacker compromises the manufacturing process itself to create vulnerabilities or backdoors. Detecting a supply chain compromise is a difficult and costly endeavor. This reinforces the need to implement strict security and quality control throughout every phase of the supply chain.

2.4. Checksum algorithms

A checksum algorithm typically uses a hash function to verify data or code integrity. Such an algorithm is commonly used by an embedded device to validate firmware update payloads before installation and execution. Because embedded devices have limited memory and computational power, they commonly employ simple checksum algorithms rather than strong cryptographic hash functions [13]. However, a trade-off exists between the simplicity of an algorithm and its ability to accurately detect modifications. Thus, embedded systems commonly use modular summation or CRC algorithms [10,15]. Some embedded devices engage proprietary checksum algorithms [13]. A proprietary checksum algorithm makes it much more difficult for an attacker to implant malicious firmware. Indeed, the only option is to extract and reverse engineer the firmware validation code to uncover the proprietary algorithm.

3. Firmware reverse engineering process

Reverse engineering firmware is by no means a new concept. Skochinsky [18] has provided an excellent overview of the general process. In the area of industrial control systems, Santamarta [17] and Peck et al. [15] have made important contributions to reverse engineering PLCs in order to discover backdoors, fuzz testing for vulnerabilities and determining the validation algorithm used by a ControlLogix Ethernet module. These efforts constitute the basis of the reverse engineering approach presented in this paper, including the determination of the validation algorithm used by an Allen-Bradley ControlLogix L61 controller module.

Fig. 2 presents an overview of the reverse engineering process. The steps include: (i) firmware sample acquisition; (ii) binary analysis; (iii) firmware disassembly; and (iv) derivation of the firmware update validation method. While the process described in this paper builds on the work of several researchers (e.g., [15,17,18]), reverse engineering is still as much an art as it is a science. Reverse engineering requires intuition and experience, and the success and effectiveness of each step often rely on the human factor. Nevertheless, the process described in this paper can serve as a roadmap for reverse engineering PLCs and other embedded devices.

Firmware samples were obtained directly from the vendor website in the form of firmware update packages. The firmware binaries were then extracted from the update packages. Following sample acquisition, binary analysis of the firmware files helped determine likely image formats and identify sections of interest related to validation (e.g., header information and candidate checksum fields).

The next step involving firmware disassembly required the determination of the target processor architecture, disassembly of the binary code to assembly code, identification of assembly functions, determination of the firmware base address, string analysis, and the rebuilding of function names in the disassembly.

The derivation of the firmware update validation method involved disassembly analysis, in which strings and recovered function names that are relevant to validation were identified. Following this, black box analysis used the characteristics of common validation algorithms to narrow the search space and brute force techniques were used to attack the firmware update validation method. Hardware debugging

![Fig. 2 – Firmware reverse engineering process.](image-url)
was also used to establish physical connections to the PLC, enabling direct access to the execution path of the processor and device memory, including the executive loader, which is typically inaccessible to users. These techniques resulted in the determination of a candidate firmware update validation method. This firmware update validation method was then confirmed, analyzed and exploited to the extent possible.

4. Experimental evaluation

The environment for conducting the reverse engineering experiments included an Allen-Bradley ControlLogix 1756-L61, Series B, Standard Controller Module manufactured by Rockwell Automation. Standard firmware obtained from the manufacturer was applied to the PLC using Rockwell’s ControlFlash update software. Since the scope of the evaluation was confined to the PLC hardware and firmware layers with standard update procedures, the test environment was equivalent to the intended manufacturer configuration. The ControlLogix L61 controller module was used with a standard ControlLogix 1756-PA72/C power supply, 1756-A7 chassis/backplane and 1756-ENBT Ethernet communications module to support operations and testing. However, these three components were not considered in the evaluation. Thus, the results of the experimental evaluation are accurate for any standard ControlLogix L61 controller.

Additional tools used in the evaluation included the Notepad ++ text editor and the HxD binary file editor. The binary analysis step of the process engaged the Visual Binary Diff (VBinDiff) [12] binary file difference tool to perform binary file comparisons, and the binwalk static binary analysis tool to perform embedded file and filesystem analysis. Firmware disassembly employed the IDA tool [8]. Brute forcing techniques applied during black box testing used the CRC RevEng tool [5]. Finally, the hardware debugging techniques employed an ARM RealView ICE device along with the ARM Development Studio 5 (DS-5) debugging software [3].

4.1. Firmware sample acquisition

ControlLogix L61 firmware update packages are readily available from Rockwell Automation’s website. Furthermore, it is straightforward to access the firmware image from within the update packages. These circumstances facilitated the procurement of a range of firmware samples with little time, cost and effort.

4.2. Binary code analysis

This section describes the three steps involved in analyzing binary code. The steps involve code inspection, code comparison, and embedded file and filesystem analysis.

4.2.1. Manual code inspection

Manual code inspection engages a tool such as HxD to examine the firmware image files downloaded from the vendor’s website. The general structures of the binary files are assessed, and any encrypted or compressed sections are identified. If such sections exist, appropriate decryption or decompression techniques are applied to obtain the original code. A marked lack of randomness in the bytes is a good indicator of the absence of encrypted and compressed sections.

4.2.2. Code comparison

Binary comparison of firmware can accurately identify the dynamic and static fields in an image. Important advantages of binary comparison are the identification of a candidate checksum value field and the collection of information about the image header. However, binary comparison is limited by the availability of multiple firmware sample images; the effectiveness and accuracy of a code comparison is directly proportional to the number of samples available.

To identify dynamic sections of firmware, it is useful to have two similar firmware revision numbers (FRNs). Consecutive firmware versions generally have many similarities, so the procedure involves comparing the similarities in two consecutive firmware versions working backwards from the most recent FRNs available. In the case of the Allen-Bradley ControlLogix L61 PLC, the two most similar firmware versions discovered following this method were FRN 16.081 and FRN 16.057. These two firmware versions were the only ones with the same length. Furthermore, only 14 byte differences were identified between all the files belonging to the two versions.

The comparison of dissimilar firmware versions can help identify the static fields. Optimal dissimilarity is defined as the greatest possible number of byte differences between two valid firmware images of given lengths. For two dissimilar images, the bytes that are unchanged between the two images represent static data that is likely to be constant across many firmware versions. However, optimal dissimilarity is not mandatory for this comparison. The comparison procedure can be simplified if the newest and oldest available firmware versions are dissimilar enough to identify the static fields. At the time of conducting our experiments, the newest and oldest versions for the ControlLogix L61 PLC were FRN 20.013 and FRN 13.071, respectively. Loading these two files into the Visual Binary Diff (VBinDiff) revealed that major portions of their headers were the same.

Noticeable patterns were seen when comparing the ControlLogix L61 firmware images. For example, the third, fourth and seventh words of the headers were observed to remain constant. In addition, the second word was identified as containing the firmware version number of each image. The first byte of this word represents the major revision number, the second is the minor revision number and the third is the subrevision number found in the accompanying .nvs file. However, none of the header values directly related to the file length. A thorough review of the dynamic fields uncovered upon analyzing FRN 16.081 and FRN 16.057 revealed that the last eight bytes of every firmware image differ in an apparently random fashion. This is strong evidence that the trailing eight bytes represent a validation value.

4.2.3. Embedded file and filesystem analysis

After the manual inspection and comparative analysis are performed, the target firmware must be analyzed for embedded files or filesystems by searching for byte signatures that match known files or filesystems (e.g., zlib,
gzip or LZMA compressed files and the cramfs, SquashFS, JFFS2 or YAFFS filesystems. From this point forward, FRN 16.081 was designated the targeted firmware image. The designation was made based on its similarity to another version (FRN 16.057) as well as the fact that, at over 1 MB smaller than the most recent firmware, FRN 16.081 had significantly less data to analyze. The underlying assumption was that the size advantage comes with no significant variation regarding implementation of the validation method or other firmware features critical to operation.

The firmware image FRN 16.081 was then analyzed using the binwalk tool to identify known files and filesystems. The results revealed six gzip file candidates and approximately 170 zlib file candidates. Extraction of the zlib containers yielded no significant information. Since no filesystems were identified, the results implied that, except for the identified gzip files, the firmware image likely contained raw binary code and data.

4.3. Binary code disassembly

This section describes the steps involved in disassembling the binary code.

4.3.1. Processor determination
Before the firmware image can be passed to an automated disassembly tool such as IDA, it is necessary to determine the target processor. This is accomplished by comparing different target processor code samples. In the case of the ControlLogix L61 PLC, the comparison of instruction signatures indicated that the firmware was targeted for an ARM processor. The physical examination of the PLC hardware also confirmed an ARM target processor.

The IDA tool provides two ARM targets, one based on little-endian byte ordering and the other based on big-endian byte ordering. The little-endian target processor was selected first; it resulted in the automatic disassembly of several functions in the binary code, indicating that the little-endian target processor was the correct choice for the ControlLogix L61 PLC firmware. This result was confirmed because IDA failed to disassemble any code when the big-endian target option was selected.

4.3.2. Function identification
The IDA tool could not completely disassemble the firmware binary. In fact, a significant portion of the binary image remained unexplored by the automated tool. The identified functions from the code that IDA natively disassembled were examined to determine their prologue signatures. All the functions that were automatically identified by IDA began with a Store Multiple with Full Descending Stack Address Mode (STMFD) instruction that pushes current register values on the stack. The exact registers pushed on the stack varied, but the two most significant bytes of the instruction remained constant (0xE9 0x2D). A review of the ARM Procedure Call Standard specified in the ARM Software Development Toolkit Reference Guide [1] confirmed that this instruction is a standard prologue for ARM functions. Furthermore, a review of the ARM Instruction Encoding Standard [2] confirmed that, for this particular instruction, the most significant two bytes are always 0xE9 0x2D. The identification of the specified bytes enabled the extraction of the vast majority of functions in the binary.

4.3.3. Rebasing the disassembly
The next step is to determine the base address. The “load immediate” technique was initially applied because the disassembled code had immediate address references. A search of all Load Register (LDR) instructions that referenced immediate values revealed that the majority of the references consisted of addresses in the range 0x10000 to 0x30000. We assumed that these addresses referenced locations in the firmware, which implied a base address of zero. However, we discovered that the firmware was loaded with a base address of zero and that no immediate addresses referenced the beginning of functions or strings. This meant that the addresses did not reference sections of the firmware.

After the complete search of immediate values in LDR instructions was performed, other common address bases were identified. The address bases in descending order of occurrence were: 0x08000000, 0x00C00000, 0x00B00000, 0x00B00000 and 0x60000000. The starting location referenced in the .nvs file was 0x0B160000. Given that this location was titled as “starting location” and fell within the range of several immediate values in the general 0x0B000000 base, it was incorporated as the true base address.

The IDA tool was used to rebase firmware at the new address. No significant changes in the disassembled code were immediately visible. However, many instructions in the firmware used relative addresses that were independent of the base address, so significant changes were not expected. For the same reason, proceeding with an incorrect base address did not hinder the reversing process. Indeed, the majority of the instructions utilized relative addressing, so function interactions and operations remained consistent regardless of the base address. However, knowledge of the true base address was still critical in understanding the firmware as a whole. While many of the function references were correct, working with an incorrect base address could lead to inaccurate interpretations of segments referenced by immediate addresses.

4.3.4. String inspection
In order to gain as much information as possible from the firmware image, all the ASCII strings contained in the image were inspected. The IDA tool provides a subview for listing all strings in the loaded binary; the UNIX strings command also provides equivalent functionality, but without the support of IDA disassembly. A manual inspection of the list revealed several XML strings. A review of the official documentation from Rockwell Automation determined that these strings were related to the use of CompactFlash memory cards as storage for project files [16].

Strings were discovered that were indicative of the ARM compiler versions used to build the firmware. These strings indicated that Rockwell Automation used standard ARM tools for the original firmware development and also provided time period of the original firmware development. For example, a search of the most recent firmware (FRN 20.0130) for strings containing “ARM” produced one result for “ARM ADS1.2 [Build
Another string in the firmware referred to a multiple-precision math library. A similar string in the associated .nvs file referred to this library using the name BigDigits. Since official documentation for BigDigits specifically describes its use in cryptographic applications, the fact that this string was found indicated that the firmware may have some cryptographic functionality [6]. The same string was also found in FRN 20.013.

4.3.5. Symbol rebuilding

Two basic requirements exist when rebuilding symbols in the firmware—symbol names and a way to associate the names with the functions to which they belong. String inspection revealed that a number of source code filenames existed in the firmware. Further inspection of these strings, however, revealed that they were not located in one common section of the binary. Instead of being located in a symbol table, the source filenames were distributed throughout the firmware image. While this discovery hampered the reverse engineering effort, the fact that filenames were available was significant.

During disassembly, IDA automatically creates cross references between string addresses and the addresses of instructions that reference them. Using these cross references, instructions that referred to several of the source filename strings were investigated and compared. The investigation revealed that all the strings were used in the same manner: they were passed in as parameters to a common function. A brief exploration of this common function identified a series of calls that apparently never returned. When multiple functions that make exception calls are contained in a single source file, the filename arguments passed are identical. Thus, there was not a one-to-one mapping, but a one-to-many mapping of names to functions. As a consequence, the names were less descriptive and specific than the individual function names provided by a symbol table.

Since the source filename strings were distributed throughout the firmware, the process of naming their related functions was less straightforward, but it could still be automated using a script. Every function that referenced each filename string was identified using the IDA cross reference database. The script then renamed the functions according to the referenced filename. Duplicates were commonly encountered during this process and were handled by adding generic numerical suffixes to each function name.

4.4. Derivation of firmware update validation method

This section describes the procedure used to derive the method used by the ControlLogix L61 PLC to validate firmware updates.

4.4.1. Disassembly analysis

The reverse engineering of the validation method from disassembled code began by considering the general functionality of validation algorithms in order to determine relevant patterns and structures in the disassembled code. For instance, validation algorithms are known to inherently perform computations over the contents of the firmware; therefore, the code that calculates the validation values would likely contain a loop that performs an operation over a range of memory addresses. Furthermore, the specific operations performed in the loop would be dependent on the type of algorithm used for validation.

Given these considerations, the list of firmware functions was searched for names relevant to a checksum algorithm used for validation. After an apparent relevant function was identified, the surrounding disassembly was examined for operational flow indicative of such an algorithm. An example function discovered in this manner was identified by the cmCS.c source filename. Since the “CS” abbreviation in the filename may indicate “checksum,” the function was explored. However, an analysis of the disassembled code revealed no operational flows indicative of a checksum calculation, so the function was deemed to be not significant. Several other strings were discovered via this process, including the binary file extension .bin, a set of functions referencing the Encryption.c source filename, a large set of “up” functions, including the source filenames upexec.c and upprog.c, and the ReUpLockForUpdate.c function set. Again, these functions were examined because of the possible relevance of their names, but while general information regarding their operation was gained, no specific information regarding the firmware update validation method was discovered.

During this process, the first word in the firmware image was discovered to contain a branch instruction. IDA was instructed to treat such a location as the beginning of a function; this revealed initialization code at the branch target location. The initialization code was followed and analyzed for more than 1000 instructions, but no code related to the functionality of the validation method was discovered. Since a CRC code, if present, would likely use an exclusive or instruction (“EOR” in ARM assembly), we directly searched the disassembly for all EOR instructions. A total of 263 occurrences were found, which were too numerous to analyze exhaustively. Nevertheless, several of the results containing loop structures were explored, but no code relevant to the validation method was discovered. Since the validation method still had to be located from among the disassembled functions of the operating system firmware, other approaches were explored.

4.4.2. Black box testing

In black box testing, incremental alterations to firmware were tested on the device to gain information regarding their effects on device operation. Since the true firmware update validation method was implemented in the device, the success of a particular tactic could only be reliably determined by modifying the firmware, uploading it to the device, and assessing the result. When modifications are made at such a low level, there is a significant risk of “bricking” the device. Although it may be possible to reset the device using hardware debugging methods, they are not guaranteed to work. This hazard presents a challenge to reverse engineering efforts by restricting the modifications that can be safely attempted while simultaneously minimizing the risk of harm to the device. Using multiple devices in black box testing can be prohibitively expensive; therefore, it is best to avoid damage.
4.4.3. Hardware debugging

Since JTAG is the common standard for hardware debugging interfaces, the device was physically examined for possible JTAG test access ports (TAPs) [9]. After physically disassembling the device to access the circuit board, unknown and unused connectors were targeted first for investigation. The ControlLogix L61 PLC had no unused connectors on the board, so the search moved on to empty solder pads and test points. A number of unused solder pads were present, but in order to identify them as JTAG TAPs, their corresponding signals had to be verified. Since the ControlLogix L61 PLC is ARM-based, a search was conducted for standard ARM JTAG pinouts. Two common ARM JTAG configurations are the 14-pin and 20-pin layouts. One apparently empty connector pad on the controller board had 14 pins, so it was a prime candidate for a JTAG interface.

An initial visual inspection found no signs that any of the solder pads of the candidate connector traced to any non-integrated-circuit components other than a resistor. A multimeter was used to test the pinout signals of the candidate connector. Ground and power signals are the most straightforward to identify, so the pins were tested against ground and power while their layout was compared with the 14-pin ARM JTAG reference pinout. All eight combined power and ground pins corresponded to the reference pinout and the candidate connector pads, providing initial evidence that the empty connector was used for JTAG. The other candidate pins also had to be confirmed using Breeuwsma’s method [4], but this was a difficult task. When the controller was fully assembled for operation as required by Breeuwsma’s method, the candidate TAPs were inaccessible to manual probing. To access them, lead wires had to be soldered directly to the candidate points.

An ARM RealView ICE device and the ARM Development Studio 5 (DS-5) software were used to debug the controller. Using the soldered connector, the ControlLogix L61 PLC was connected to the ICE hardware. With the debugger configured, the first attempts were made to connect to the target. The connections succeeded, but the device could not be halted by debugging. When a stop command was issued, the controller entered a fault state and became unresponsive, requiring a manual restart. An attempt to connect immediately after the manual reset determined that controller execution could only be stopped before the operating system finished booting. This implied that the debug connection should be made as soon as the device was powered on before the operating system had time to boot. The cause of this anomaly could not be confirmed, but a possible explanation is that halting the processor while the operating system is running triggers a fail-safe response from a watchdog timer. While the processor is halted, the watchdog timer is not properly fed, so a fault is triggered when the timer expires in an attempt to prevent unsafe operation.

After the device was connected to the debugger and halted, the device stopped executing in its pre-boot state. At this point, the debugger was used to perform a memory dump of the device. Since no information was available about the proper mapping of device memory, all of the 32-bit address space that could be accessed by the debugger was dumped. Manual analysis of the acquired memory dump using HxD revealed that much of the dumped address space repeated itself. After accounting for duplicated memory, several sections of binary code were identified by searching for common ARM instruction patterns. The operating system firmware was found in two distinct locations in these code segments: one starting at address 0x0B1A0000 and again at 0x0D000000, and the other a large code segment starting at address 0x0B020000. The beginning of this segment contained a header of the same format as the operating system.
firmware. Inspection of the version number field revealed the segment to be FRN 1.010, the base firmware.

In addition to these firmware images, two other short and unfamiliar code segments were found: one at address 0x0A000000 and the other at address 0x80000000. The two unknown code segments were extracted from the dump. Analysis of the segments revealed multiple source filename strings akin to the operating system firmware. Two interesting filenames were ExecLoader.s and hw_setup.s. Reverse engineering of the function referencing ExecLoader.s using IDA confirmed that the segment was the executive loader based on its functionality. Extensive analysis of the code revealed the following candidate firmware update validation method.

After performing hardware initialization, ExecLoader follows a process to validate and load the firmware into memory before handing over execution.

ExecLoader begins by determining what, if any, firmware is present. To accomplish this, ExecLoader first verifies the existence of base firmware. The base address of the base firmware was hardcoded as 0x08020000. Using this base address, ExecLoader compares the third and fourth words of the presumed base firmware header. If these two values are bitwise inverses of each other, ExecLoader accepts that a base firmware is present. If no base firmware is detected, execution on the device is terminated in an infinite loop. However, if the presence of base firmware is confirmed, ExecLoader continues by determining if operating system firmware is present. Given the operating system firmware base address, ExecLoader performs the same check as above on the third and fourth words of the presumed operating system firmware header. Again, if the two values are bitwise inverses, ExecLoader accepts that operating system firmware is present.

ExecLoader then proceeds to validate the firmware. If operating system firmware is present, the validation is performed on it; otherwise, ExecLoader defaults to validating the base firmware. The algorithm used for validation is a variation of modular summation. In the event operating system firmware is present but does not pass the validation, ExecLoader proceeds to validate the base firmware. If no firmware passes validation, execution is terminated in an infinite loop.

After the successful validation of firmware, ExecLoader loads the firmware from flash memory RAM. To determine the address in RAM where the firmware is loaded, ExecLoader references the fifth word of the firmware header. The firmware is copied to this address. Finally, execution jumps from ExecLoader to the base address of the loaded firmware.

Based on the firmware modification goal, the disassembly was searched for potentially relevant strings and function names. The goal in this case was to counterfeit the firmware version number FRN 16.081 to appear as FRN 20.66.99, a value higher than any currently available version number from Rockwell Automation.

First, the firmware was searched for locations that referenced the version number. IDA detected no code references to the version number bytes in the header, but there had to be a reference to the version number in the code because the device reports a version number to ControlFlash. Since FRN 16.081 only differed from FRN 16.057 by 14 bytes, the binary files of the two versions were compared. Inspection of the disassembly surrounding the differences in the versions revealed that one difference was in a function that returned a hard-coded version number. The immediate values of the instructions were modified using the HxD tool to represent the target version number 20.66.99. The version number bytes in the firmware header were also modified appropriately. The correct checksum values were calculated and updated in the new firmware binary file. The utility then revalidated the binary to confirm that the checksum values were correctly updated.

5.2. Device exploitation

Adjustments were made to the .nvs configuration file in order to use ControlFlash to upload the counterfeit firmware. The firmware version number was specified several times in the configuration file, so these values were modified to reflect the counterfeited version number. This made the modified firmware appear in the ControlFlash firmware catalog list as version 20.66.99. The ControlLogix L61 PLC was first updated with the legitimate FRN 20.013 firmware from Rockwell Automation in order to test the effectiveness of a counterfeit update on the newest major revision. ControlFlash was then used to update the device with the counterfeit FRN 20.66.99. The update successfully completed, the device restarted and the new 20.66.99 revision number was reported to ControlFlash, indicating a successful update as shown in Fig. 3.

While ControlFlash was used in the experiment, a custom utility could also be developed to perform the same operation. This is recommended because ControlFlash introduces some problems when attempting a complex firmware modification. Specifically, in order to process the catalog of available updates, ControlFlash references a restriction file (.RES extension) in

5. Experimental demonstration

This section describes the results of an experiment involving firmware modification and exploitation of the ControlLogix L61 PLC

5.1. Firmware modification

The process for firmware modification began by reversing the firmware binary to obtain named functions in the disassembly.
addition to the configuration script and firmware image. This .RES file is used to determine if the associated firmware has any usage restrictions. In addition, the .RES file contains another type of checksum value for the firmware image. The associated checksum is highly prone to collisions, so minor modifications are not detected, but extensive modifications, including changes in image length, require the checksum value to be recalculated. Since .RES file processing is handled exclusively by ControlFlash, a custom update utility would avoid this validation. However, because the modifications involved in the experiment were minor, the .RES file value did not have to be recalculated.

6. Conclusions

Attacks such as Stuxnet have demonstrated the serious threats posed by malware on industrial control systems and the critical infrastructure assets they operate. This paper demonstrates the relative ease with which malware can be implanted in industrial control systems. More importantly, the general firmware analysis methodology presented in the paper provides a foundation for developing forensic techniques as well as for designing defensive methods to combat firmware modification attacks. We hope that this research will stimulate new efforts focused on detecting attacks that target industrial control system firmware and on developing sophisticated mitigation strategies.

Note that the views expressed in this paper are those of the authors and do not reflect the official policy or position of the U.S. Air Force, U.S. Department of Defense or the U.S. Government.

REFERENCES