MAR-17-352-01 HatMan—Safety System Targeted Malware (Update B)

CONTENTS

KEY TAKEAWAYS

- The report discusses the components and capabilities of HatMan malware, also known as TRITON and TRISIS. This report also provides information for industrial control systems owner and operators about detection and recommendations for potential mitigations.
- This report is an update to the HatMan–Safety System Targeted Malware (Update A) published on April 10, 2018.

DISCLAIMER: This report is provided "as is" for informational purposes only. The Department of Homeland Security (DHS) does not provide any warranties of any kind regarding any information within. DHS does not endorse any commercial product or service referenced in this advisory or otherwise. This document is distributed as TLP:WHITE: Disclosure is not limited. Sources may use TLP:WHITE when information carries minimal or no foreseeable risk of misuse, in accordance with applicable rules and procedures for public release. Subject to standard copyright rules, TLP:WHITE information may be distributed without restriction. For more information on the Traffic Light Protocol, see https://www.us-cert.gov/tlp.

OVERVIEW

This malware analysis report is an update to the report titled MAR-17-352-01 HatMan – Safety System Targeted Malware (Update A) that was published April 10, 2018, on the Cybersecurity and Infrastructure Security Agency's (CISA) ICS-CERT website. This report, MAR-17-352-01 HatMan – Safety System Targeted Malware (Update B), contains an updated YARA signature to identify a custom, Windows-based remote deployment tool that threat actors may have used.

The HatMan malware, also known as TRITON and TRISIS, affects Triconex Tricon safety controllers by modifying in-memory firmware to add additional programming. The extra functionality allows an attacker to read/modify memory contents and execute arbitrary code on demand through receiving specially-crafted network packets. HatMan consists of two pieces: a PC-based component to communicate with the safety controller and a malicious binary component that is downloaded to the controller. Safety controllers are used in a large number of environments, and the capacity to disable, inhibit, or modify the ability of a process to fail safely could result in physical consequences. This report discusses the components and capabilities of the malware and potential mitigations.

ACKNOWLEDGEMENTS

This Analysis Report is the product of collaboration between several groups, including Schneider Electric, the producer of Triconex Tricon safety controllers.

A NOTE ON TERMS

This document uses the terms "Triconex" and "Tricon" frequently throughout. For clarity, "Triconex" refers to the overall product line produced by Schneider Electric, whereas "Tricon" refers to the actual safety system hardware.

ANALYSIS

HatMan follows Stuxnet and Industroyer/CrashOverride in specifically targeting devices found in industrial control system (ICS) environments, but the malware surpasses both forerunners with the ability to directly interact with, remotely control, and compromise a safety system—a nearly unprecedented feat. This section will discuss the malware's context, components, and capabilities at a reasonably high level. The Technical Details section provides a deeper dive into the inner workings of the malware.

Vulnerable Systems

Triconex MP3008 main processor modules running firmware versions 10.0–10.4 are vulnerable to HatMan. Based on testing, versions earlier or later than this are not vulnerable to the analyzed malware sample as is; however, it is not known whether adjustments to the malware or exploitation of a different vulnerability might lead to a successful compromise of other versions of the firmware. This version of the hardware uses an MPC860 PowerPC processor, whereas newer Triconex safety systems are ARM-based. This means that a different version of the malware would be required to target newer Tricons.

Context: What are Safety Systems?

Safety systems—also known as safety instrumented systems (SIS) or safety programmable logic controllers (PLCs)—are specialized hardware, similar to traditional PLCs, with a strong emphasis on reliability and predictable failure. Unlike traditional PLCs, safety PLCs often have redundant components such as multiple main processors, watchdog capabilities to self-diagnose anomalies, and robust failure detection on inputs and outputs. Safety PLCs are used to provide a way for a process to safely shut down when it has encountered unsafe operating conditions and to provide a high degree of safety and reliability, with important monitoring capabilities for process engineers. Though designed never to fail, safety PLCs are also designed such that, were one to fail, it would fail in a predictable manner so that the worst-case scenario is known and planned for ahead of time.

Overview of Operation

Prior to discussing the individual components of the HatMan malware, it is worthwhile to provide a brief, high-level overview of how an attacker would utilize HatMan to compromise a safety controller. Figure 1 shows the overall sequence of events of these two operations.

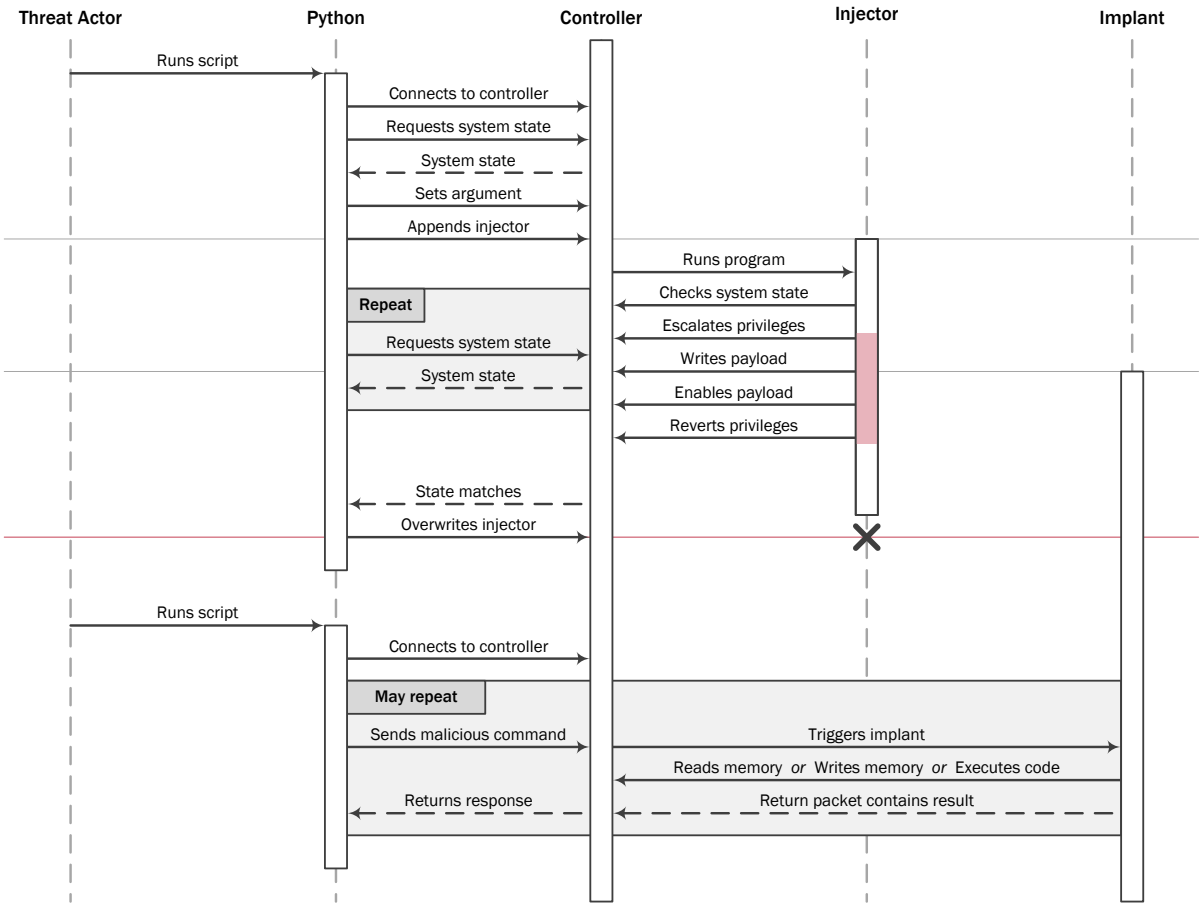

Figure 1: Overall flow of HatMan malware

The threat actor's first step—after having compromised a computer within the safety network—is to execute the main HatMan Python script (`script_test.py`, compiled into `trilog.exe`) that leverages a custom implementation of an internal TriStation (TS) protocol (`library.zip`). This script, in turn, connects to the controller, gathers some information about the system state, then begins the attack. It first sets an argument for the injector, then downloads a combination of the injector and implant to the device as a new program for the controller to run. The script then periodically checks the system state to determine whether the injector has completed.

Concurrently, the injector begins executing automatically on the controller. It begins by verifying that the controller looks like it is able to be compromised through several tests that exercise the vulnerability the injector leverages. Once it has done enough testing, the injector uses the vulnerability to escalate its privileges on the device, write the implant into the in-memory firmware region, enable the implant, and revert its privileges. The injector then reports that it has finished.

The Python script, seeing that the injector has completed its execution, overwrites the program slot that it had used for the injector with a "dummy" program and exits. At this point, the controller has been fully compromised.

The threat actor can now exercise the capabilities of the HatMan implant—a remote access Trojan (RAT) capable of reading and writing memory and executing arbitrary code. Because the same Python implementation of the TS protocol includes functions for utilizing the HatMan implant, the attacker would only need to use a similar script to connect to the controller and manipulate it. By utilizing the three functions HatMan provides as building blocks, the attacker can then freely modify the controller's programming.

Components

HatMan consists of two parts: a more traditional PC-based component that interacts with the safety PLCs and a binary component that compromises the end device when downloaded. Files related to these components could appear on a workstation or similar device and might mimic legitimate TS software paths and filenames.

The PC-based component consists of three pieces in the form observed:

- An executable that programs a Tricon device without the TS software,
- A native shellcode program that injects a payload into the in-memory copy of the Tricon firmware, and
- A native shellcode payload that performs malicious actions.

Reprogramming the Safety PLC

In its current iteration, the HatMan component that programs a Tricon is written entirely in Python, although nothing would preclude these being written in a different language. The modules that implement the communication protocol and other supporting components are found in a separate file—`library.zip`—while the main script that employs this functionality is compiled into a standalone Windows executable—`trilog.exe`.

This Python script communicates using four Python modules—`TsBase`, `TsLow`, `TsHi`, and `TS_cnames`—that collectively implement the TS network protocol via UDP 1502; this is the

protocol that the TS software uses to communicate with Triconex safety PLCs. Although this protocol is undocumented, it is similar to the officially documented user application Triconex System Access Application (TSAA) protocol, used for reading and writing data points from a third-party program. Due to this, it could feasibly have been reverse engineered from knowledge of the TSAA protocol, other manufacturers' documentation, and studying traffic between the programming workstation and safety PLC. In addition, this protocol does not require any authentication or encryption, although access control lists (ACLs) may be configured on the PLC. The Python script is also capable of autodetecting Triconex controllers on the network by sending a specific UDP broadcast packet over port 1502.

In addition to their implementation of the TS protocol, the Python modules expose a set of methods to interact with the compromised safety PLC. These send specially-crafted network commands to pass messages to the implant in order to read and write memory or execute arbitrary code. These commands may be sent from any device on the safety network and will be accepted by the compromised controller, regardless of key switch position.

The script embedded in `trilog.exe` connects to a TCM (Tricon Communication Module) using an IP address provided as an argument. Once it establishes a connection, it tries to determine if the Tricon could be compromised. This is accomplished by checking the current state of the device, then downloading a small PowerPC program (`PresetStatus`) to the Tricon that, when executed, sets an argument or "control value" in the Tricon's memory. If the control value was successfully set, the script constructs a malicious program—comprised of the injector (`inject.bin`) and implant (`imain.bin`)—that it downloads to the controller. Once this program has finished running, the script checks whether or not the injector succeeded. Regardless of outcome, the script cleans up after itself before finishing.

This script does not interact with the command modified by the malicious payload, but it is feasible and likely that a separate script was used to actually utilize the RAT functionality of the compromised safety controller as needed.

The Malicious Payload

The malicious shellcode is split into two separate pieces—`inject.bin` and `imain.bin`. The former is a less specific generic code that handles injecting a payload into the in-memory running copy of the firmware, while the latter is the payload that actually provides the additional malicious functionality. Both binary components are PowerPC machine code—the same as the controller's firmware and any user programs.

The injector masquerades as a standard compiled PowerPC program for the Tricon. It uses the control value written by the `PresetStatus` program in several ways: as an input argument; as a step counter to track execution progress; and as a field for writing debug information.

During each cycle, the injector is run, branching based on the step value stored in the control field. It begins by waiting a number of cycles (or seconds, since each cycle is usually a second) based on the control value. It then checks to ensure the vulnerability it intends on exploiting is available. The final step exploits the vulnerability to gain supervisor permissions, then copies the

payload into memory, patches a RAM/ROM consistency check, changes the jump table entry for a specific TS protocol command to the address of the copied payload, and returns.

Once the injector has finished running, it will have modified the address of the handler for a specific TS protocol command such that, when that command is received, the payload may be executed instead of normal processing.

The second component of the malicious program—the payload, `imain.bin`—is designed to take a specific TS protocol command, look for a specially-crafted packet body, and perform custom actions on demand. A threat actor can use `imain.bin` to read and write memory on the safety controller and execute code at an arbitrary address within the firmware. In addition, if the memory address it writes to is within the firmware region, the malicious payload disables address translation, writes the code at the provided address, flushes the instruction cache, and re-enables address translation. This allows the malware to change the running firmware; however, changes will be persistent only in memory and will be lost when the device is reset fully.

TECHNICAL DETAILS

This section presents a deeper analysis of the HatMan malware, providing a much more extensive look into the technical details. This malware is highly sophisticated and involves a number of distinct components. Several of these components have already been extensively discussed elsewhere, but other components have not received the same amount of consideration. Table 1 provides the set of components that will be discussed, the relationship between them, and the associated SHA-1 hash for each. These components are described in the following sections.

Table 1: SHA-1 Hashes

Filename	Description	SHA-1 Hash
library.zip	Module archive	1dd89871c4f8eca7a42642bf4c5ec2aa7688fd5c
TsLow.pyc	Protocol impl.	a6357a8792e68b05690a9736bc3051cba4b43227
TsBase.pyc	Protocol impl.	d6e997a4b6a54d1aeedb646731f3b0893aee4b82
TsHi.pyc	Protocol impl.	66d39af5d61507cf7ea29e4b213f8d7dc9598bed
TS_cnames.pyc	Protocol impl.	97e785e92b416638c3a584ffbfce9f8f0434a5fd
crc.pyc	Support module	2262362200aa28b0eead1348cb6fda3b6c83ae01
sh.pyc	Support module	25dd6785b941ffe6085dd5b4dbded37e1077e222
trilog.exe	Compiled Python	dc81f383624955e0c0441734f9f1dabfe03f373c
PresetStatus	PPC Tricon program	78265509956028b34a9cb44d8df1fcc7d0690be2
dummy	PPC Tricon program	1c7769053cfd6dd3466b69988744353b3abee013
inject.bin	PPC Tricon program	f403292f6cb315c84f84f6c51490e2e8cd03c686
imain.bin	PPC shellcode	b47ad4840089247b058121e95732beb82e6311d0

Module Archive

`library.zip` contains a number of compiled Python modules (.pyc files); these are generated during normal execution of the Python interpreter. Use of the compiled modules, instead of the source code, may have been to help obscure its purpose. The large majority of the files contained within this archive are standard Python libraries, but there are a few exceptions (see table 1 above). Collectively, these can be treated as the TS protocol implementation.

TriStation Protocol Implementation

The four compiled Python files beginning with "TS" collectively implement the TS protocol. `TsLow.pyc`, `TsBase.pyc`, and `TsHi.pyc` each implement successively higher-level portions.

`TsLow` implements the lowest-level functionality—UDP, TCM, and TS packet building, sending, receiving, and parsing. The rest of the functionality is built upon this base.

`TsBase` uses the TS packet capabilities of `TsLow` to perform individual actions on the controllers, such as downloading and uploading programs, retrieving device status, and running programs.

`TsHi` abstracts the individual actions of `TsBase` to provide simple ways of performing complicated tasks, such as appending a program, uploading one or more programs or functions, retrieving the program table, and interpreting returned status structures.

The final module—`TS_cnames`—provides string representations of a number of different features of the TS protocol, including message and error codes, key position states, and other values returned by the status functions

Cyclic Redundancy Check Support Module

`crc.pyc` implements or imports a number of standard Cyclic Redundancy Check (CRC) functions and includes input and polynomial pairs for several different standards, including Modbus and XMODEM. The inclusion of the extra CRC functions is interesting, since the TS protocol does not use them.

Assorted Support Module

`sh.pyc` provides a few utility functions for flipping endianness and printing out binary data with a hexadecimal representation. It is not especially interesting, but is a custom module that can be tied to the HatMan malware.

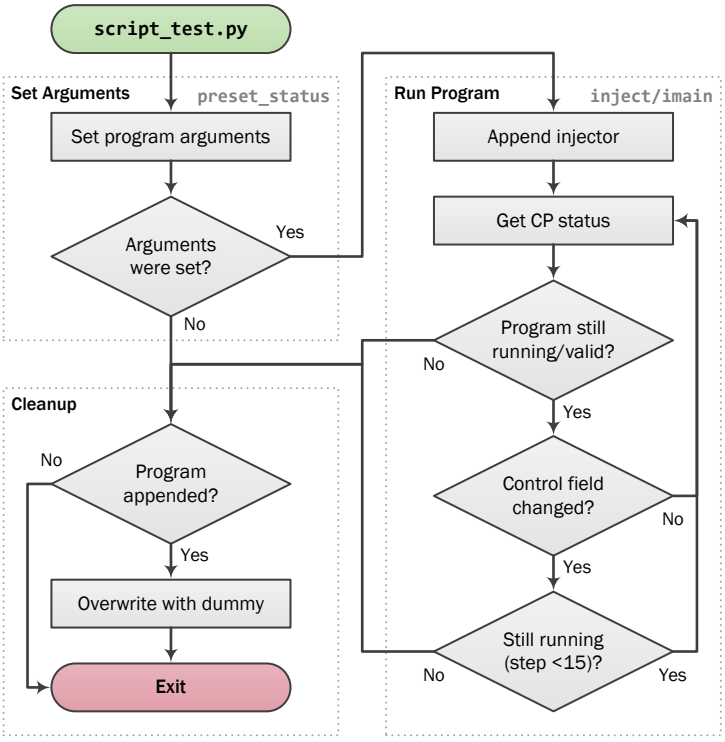

Figure 2: Python driver operation

Python Executable

`trilog.exe` uses the functionality provided by `library.zip` to communicate with and infect a Tricon controller. This Windows executable is a Python script—`script_test.py`—compiled to run standalone. The diagram in figure 2 shows the normal operation of this script.

When the script begins, it attempts to append (essentially download) an embedded PowerPC blob (`PresetStatus`, discussed in Control Value Setting section) to the target Tricon's program list. This program, when executed by the Tricon, writes a hard-coded value into the control program memory region; this value serves both as the argument to and status for the program that is uploaded later. Once the program has been appended and executed, the script attempts to see if the program ran successfully by retrieving the value it wrote to confirm that it changed to the new value; if it does not match, the script terminates, as it is unable to operate on this controller. If the value matches, execution moves on to the malicious section.

Upon success, the script builds the combined payload (discussed in the Appending Programs section), appends it to the controller, and checks to see if execution was successful. This check consists of verifying that the program still flags as valid and is running, checking to see if the control field has changed, and, if so, checking that the recorded step is less than 15 (`0xF`)—the "stopped" value. At most, the script will check 4,096 times without the control field changing before exiting.

Once the script has finished running the malicious payload, it attempts to clean up after itself by overwriting the malicious payload with a "dummy" program that does, quite literally, nothing.

Combined Payload

The malicious program that is downloaded to the Tricon consists of the following: the injector (`inject.bin`); the length of the payload (the length of `imain.bin` plus eight); the first marker (`0x00001234`); the payload (`imain.bin`); the same length again; and the second marker (`0x0056789A`). Figure 3 shows the overall structure of the program.

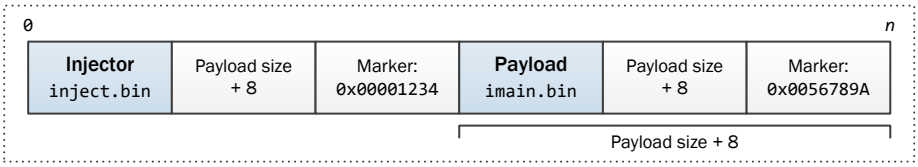

Figure 3: Layout of payload

Appending Programs

This section refers to the concept of "appending" a program to the controller. This is a very simplified term for the much more complicated sequence of actions the Python modules take to add a program to a running controller—more complicated than just allocating a new program and writing its contents. The diagram in figure 4 provides an in-depth look at how this is accomplished.

There are two ways of providing new programming to a Tricon—either via a "download all" or a "download changes." The former is used to download all of the user control application to the Tricon, whereas the latter allows a number of changes to be pushed without requiring the entirety of the application to be redownloaded. There are several differences worth noting:

- A "download all" requires the application on the controller to be stopped, while a "download changes" may be executed while the application is running.
- Programs may only be deleted via a "download all," although they may be fully overwritten during a "download changes."

When the "append" action is called, the Python function first checks to ensure that the controller is in a state that it knows can have a program appended to it; this includes checking the key state, ensuring the current programming is valid, and seeing if the user-provided logic on the Tricon is running. The function then checks if any programs are currently being downloaded and, if desired, can cancel them.

The Python function then counts the number of programs and functions on the controller and tries to retrieve the final program in the program list. If this is successful, execution continues with attempting to allocate or write the program slot using "download changes."

The function attempts to append the program three times, stopping once it has succeeded; however, during testing, any more than a single attempt to write the program resulted in at least one main processor faulting. Each attempt consists of trying to start downloading changes, checking to see if the final program has a custom CRC appended to it (allocating a new program slot if it does not), and trying to write the program. If the write is successful, execution continues.

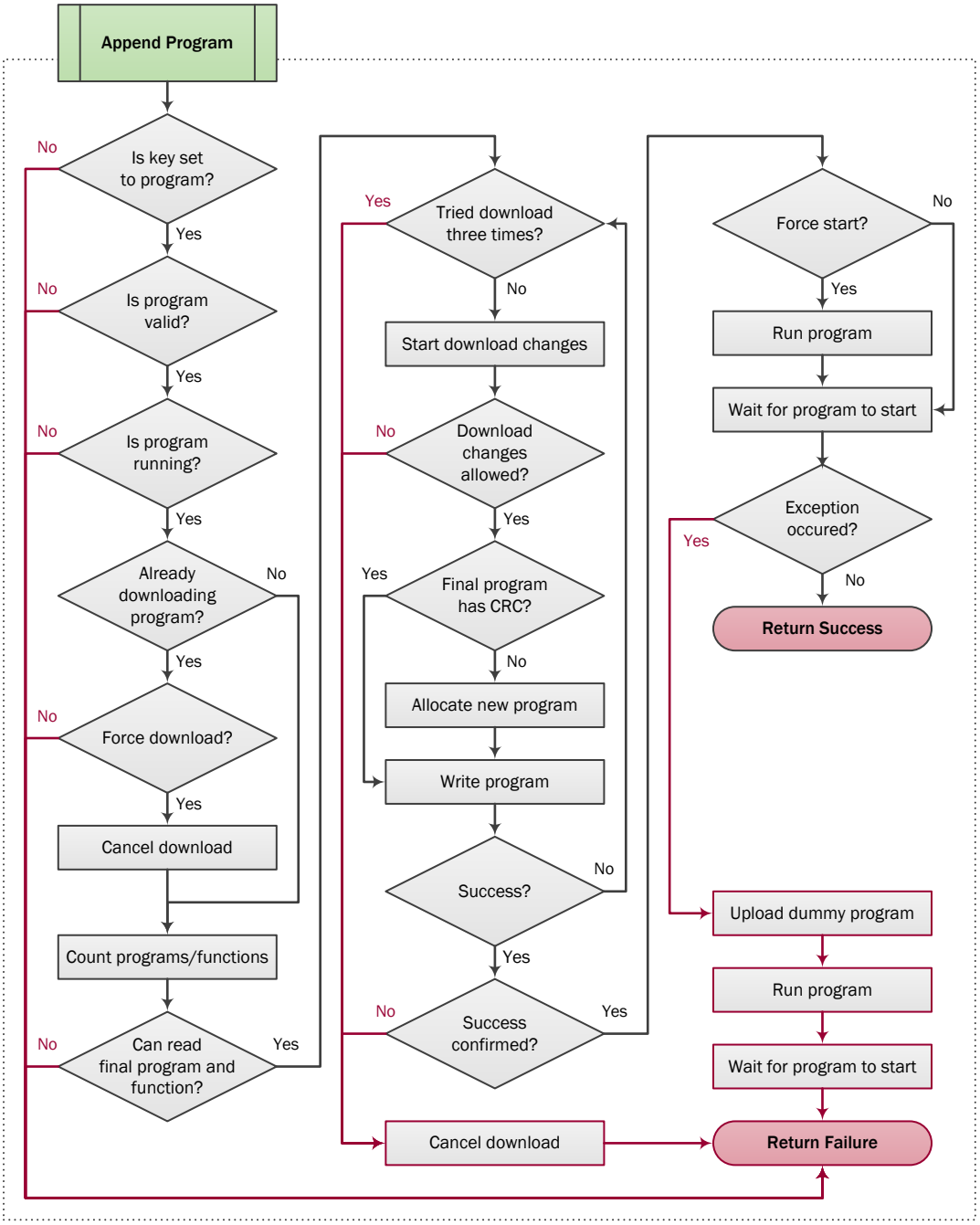

Figure 4: Complete append operation

If the program was appended and successfully run—with a "run" command if desired, otherwise by waiting for the changes to take effect—the function returns success. If the append was attempted but failed, the function attempts to cancel the download then returns failure. If the append was successful, but an exception occurred when it ran, the function overwrites the program slot with the "dummy" program and returns failure.

It is worth noting that generally the same program slot is always used by the Python component of the malware—it is appended the first time a malicious program is downloaded and overwritten

any subsequent times. Specifically, this is to say that so long as the final program slot is marked by the custom CRC appended to the program, it will continue to be used by the malware. If another program were to be appended without this checksum (such as via the TS software), the malware would allocate a new program slot to use.

In addition, it is also important to note that all of these actions—first appending, then overwriting—do not require a “download all” to occur, only a “download changes”; however, deleting a program requires a “download all” action. This is likely the reason why the script overwrites its program slot with a dummy program, rather than deleting it altogether.

Control Value Setting

The program control value is set by a small PowerPC program that searches memory for two known values; this is also known as “egg-hunting.”

Figure 5 details the operation of this component. The program is run each cycle as it is added to the program table on the Tricon; this has the side effect that, so long as this program is resident on the controller, the field will be set repeatedly until the Python component overwrites that program slot. When the program runs, it starts at the beginning of the control program region of memory and walks, 4 bytes at a time, until it either reaches the egg (two consecutive, constant values) or the end of the search region. If the program finds the values, it writes the hard-coded control value into an address at a constant offset after the location of the egg.

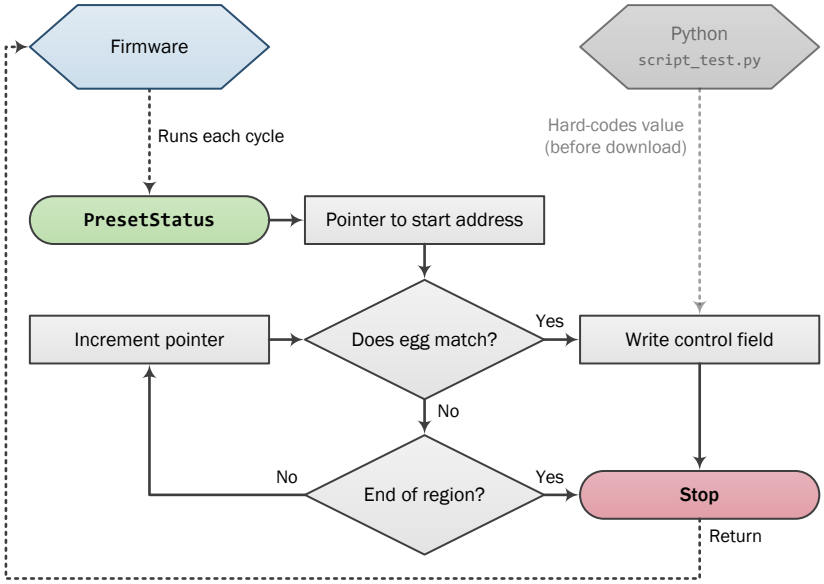

Figure 5: Control value setting program

Interestingly, the actual injector refers to this address by a constant address rather than egg-hunting; it is not clear why `PresetStatus` searches for this address rather than just referencing it directly. This could possibly be an additional check to see whether or not the system can be infected—if the control value is not set correctly, the implant would not work. It also could indicate that the `PresetStatus` program was written at an earlier stage of development when it was not yet known that that value was always at the same offset in vulnerable firmware.

Injector/Implant

The combined program that the `trilog.exe` sends to the Tricon controller is a custom PowerPC injection program that exploits a vulnerability in the device firmware to escalate privileges then disables a firmware RAM/ROM consistency check, injects a payload (`imain.bin`) into the firmware memory region, and changes a jump table entry to point to the added code. Each of these—the injector, the vulnerability, and the implant—will be discussed in this section.

The end result of the injector executing is that the functionality of the payload will be available via a compromised network command (part of the TS protocol), providing the functionality of a rudimentary RAT—reading and writing memory and executing arbitrary code—to an attacker on any device on the safety network, regardless of key switch position.

Control Value

Throughout this section, there will be repeated references to a "control value." This is the value stored by the `PresetStatus` PowerPC program and essentially controls the execution flow of the injector. The address of this value is within the structure handed back from the TS protocol "get control program status" command. As the injector runs, it uses this control field several ways: as an input argument that specifies the number of cycles to idle before attempting to inject the payload; as a step counter to track/control execution progress; and as a field for writing debug information upon failure. This allows an attacker to monitor and debug the injector as it runs.

Injector

The diagram in figure 6 shows the overall operation of the injector, `inject.bin`; the actual operation is somewhat more complex, but this provides a simplified discussion that covers the important points. This diagram also shows how the different components of the injector interact with the control field that was set previously.

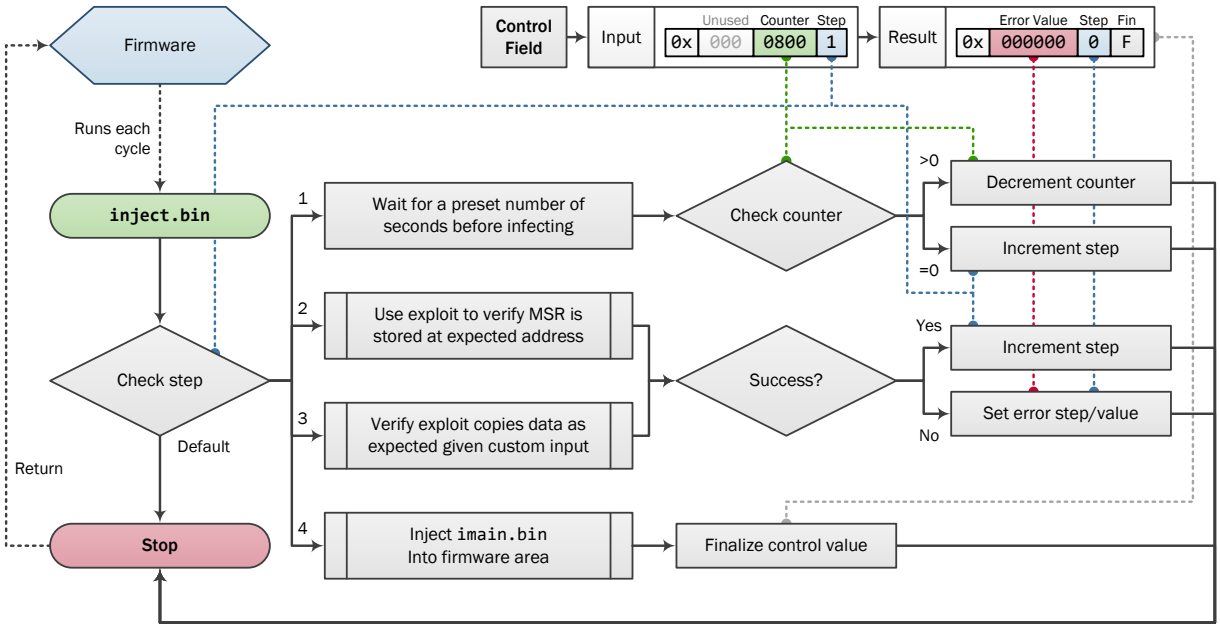

Figure 6: Operation of the injector

Much like the `PresetStatus` program, the injector is executed by the firmware once each cycle. The injector is written in a manner conducive to that—the current step of its execution is saved into the final nibble of the control value and it branches based on the current step.

The first step simply waits a number of cycles (or seconds, as one cycle is normally equivalent to one second), decrementing the counter passed in as part of the argument set by `PresetStatus`. Once this counter reaches zero, it increments the step value.

The second step uses a partial implementation of the exploit to verify that the value of the machine state register (MSR) is stored at the expected address within the context of a system call. MSR controls privileges, endianness, address translation, and other low-level processor features. This has been shown to fail on non-vulnerable versions of the MP3008 firmware. If it is successful, it increments the step value again; otherwise, it marks failure and saves the value that was copied into the control value for debugging purposes.

The third step again uses a similar partial implementation of the exploit to verify that the vulnerable system call behaves as expected—verifying that both input and output structures may be controlled by putting a known, 2-byte value into the input and checking to see if it is copied into the output structure. On success, it increments the step value, otherwise it marks a failure.

The fourth step actually performs the injection of the payload (`imain.bin`) into the firmware region, using the full version of the exploit. This will be discussed in more depth later. Once this has completed, the control value will have been finalized.

Injection Process

The diagram in figure 7 shows the operation of the actual injection process. While this does not specifically cover every single instruction, it shows the general flow of operation in enough detail to provide an accurate description of the checks and actions taken.

This function is executed as part of the injector (step 4 in figure 6). Once this stage has been reached, the vulnerability of the firmware has been confirmed and the control value should indicate that the current step is four and no error has occurred. The “steps” indicated in the diagram correspond to the value stored into the second-to-lowest nibble (mask `0xF0`), where the lowest nibble is set to 15 (`0x0F`) throughout this function.

- **Step 5**—the first step—uses the exploit to enable supervisor privileges and disable instruction and data caching via MSR. This allows the rest of the code to function, including reading from and writing to the firmware region.
- **Step 6** disables external interrupts, likely to prevent any other code from interrupting the changes that are being made to the system, then checks that an instruction that branches on the result of a RAM/ROM consistency check may be patched (the patching will occur during **step 12**). Assuming success, it will continue.
- **Step 7** verifies that the jump table entry that will be patched in **step 12** has the expected value (the address of the default case branch).
- **Step 8** verifies that the appended payload (see Figure 3 for how the injector and payload are combined) appears to be in place by checking for the boundary markers.

- **Step 9** first determines where to place the payload within the firmware region based on a constant value stored as part of the firmware, then checks to see if a previous payload exists leading up to that location in memory; if one does, it is zeroed out.
- **Step 10** ensures that the payload has content, and that the area to put the payload (the area leading up to the target address) is empty of any non-zero bytes. If it is empty, this is where the payload will be placed, and execution will continue to **step 11**; otherwise, execution branches to step thirteen.
- **Step 13** checks to see if the payload will fit at the target address rather than leading up to the target address; if it will, the new destination is at the target address and execution continues at **step 11**.
- **Step 11** actually copies the payload into the in-memory firmware region at the previously determined destination.
- **Step 12** patches the jump table entry for the overridden network command, patches a memory consistency check, then flushes the instruction cache so that changes made to firmware code immediately take effect.
- Finally, the function re-enables external interrupts and restores the system state to how it was prior to the exploit having been triggered.

Once this code executes successfully, the implant (`imain.bin`) will have been copied to an area within the in-memory firmware region and patched such that a specific network command will trigger the malicious code being called rather than the default behavior.

In addition, a specific RAM/ROM consistency check will have been patched (during step 12) that prevents a fault from occurring when the firmware region does not match the ROM image that was loaded. Without patching this check, the injector would not be able to write the payload into the firmware region or modify the jump table to point to it without faulting the device.

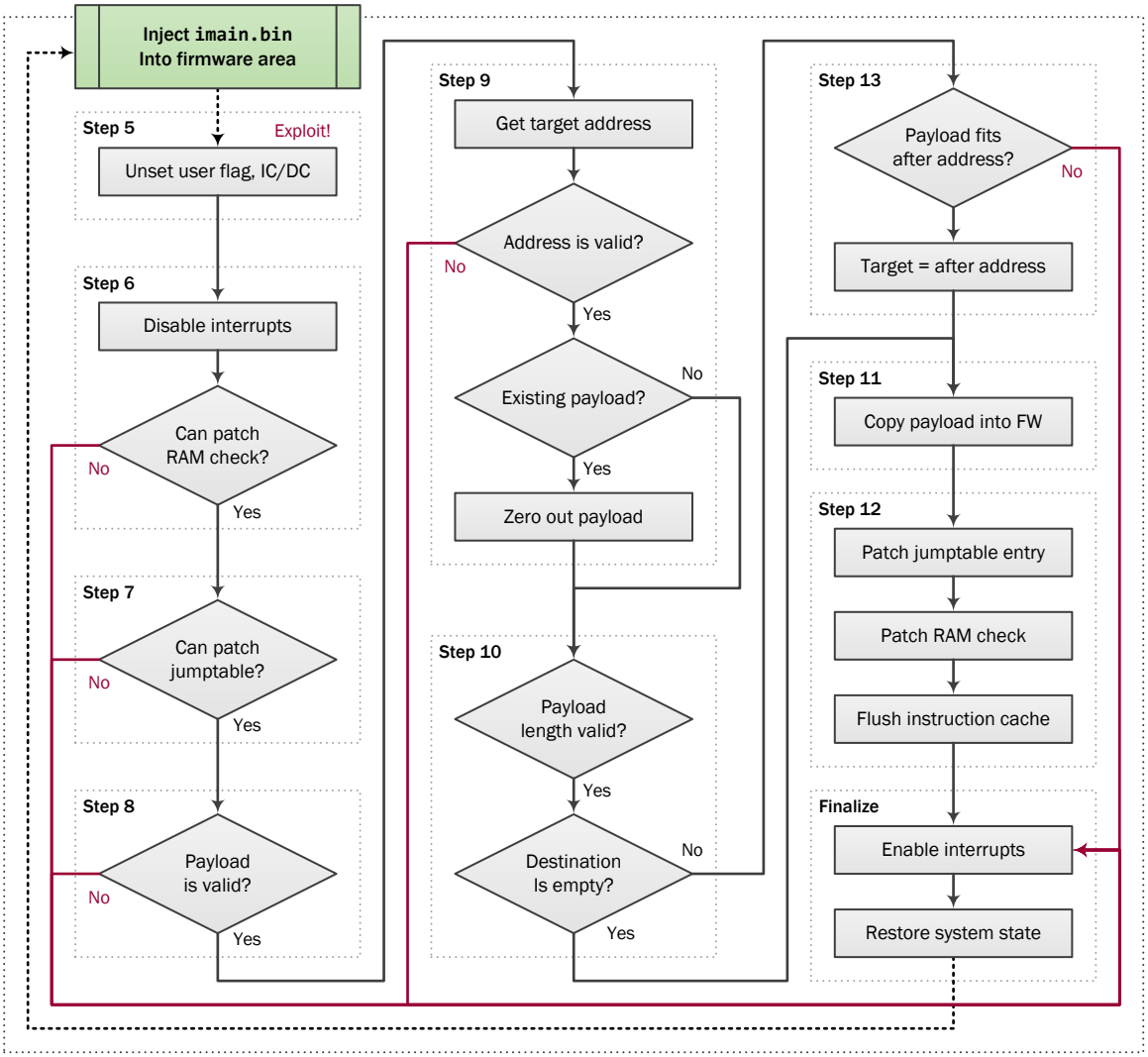

Figure 7: Full injection process

Vulnerability

The previously-unknown vulnerability affecting Tricon MP3008 firmware versions 10.0–10.4 allows an insecurely-written system call to be exploited to achieve an arbitrary 2-byte write primitive, which is then used to gain supervisor privileges. There are a number of factors that contribute to this system call being exploitable:

- The system call directly reads from three memory addresses from the control program area without any verification. Because these are userspace addresses, they may be written by a hand-crafted PowerPC program, such as the injector.
- The system call has few side-effects; it is designed to return information about the state of a firmware-level feature. This means that the userspace pointers may be changed for the duration of the system call such that they still appear “valid” (do not cause any out-of-bounds accesses) and will not fault the device.

- The input values to the system call may be controlled such that the value that is written to the output structure is the same as that passed in. This allows a specific 2-byte value to be copied from one of the input structures into one of the output pointers.
- No checking is performed on the output addresses to ensure the pointers do not refer to the firmware region or other protected areas. This allows for data to be written to normally immutable and privileged regions.

When the system call is triggered (via the `sc` instruction), the processor automatically saves the current MSR contents—which includes the user/supervisor flag, among a variety of other things—into the SRR1 register. The firmware implementation of the system call then saves this SRR1 register into a predictable address. When the system call returns, this stored SRR1 register is restored, moved back into the MSR, and execution resumes at the instruction after the `sc` instruction.

Exploiting the vulnerability allows the attacker to write 2 bytes into the location of the stored SRR1 register, replacing it with another valid but different MSR value. When the system call returns, the modified MSR is restored, giving the attacker supervisor access and disabling the instruction and data caches. Once the attacker has finished their desired actions, the system state is restored by performing a “manual” system call—jumping to the address of the system call handler, rather than using the `sc` instruction.

Implant

The implant, in many ways, is far more straightforward than the injector. This code is run when the compromised TS protocol command is received and provides RAT-like functionality. Most importantly, it allows an actor to read and write memory—including within the in-memory firmware region—and execute arbitrary code regardless of the key switch position, including “RUN.” This allows an actor to effect changes on the controller while it is in full operation, not just while it is being reprogrammed. Figure 8 shows the control flow of this component.

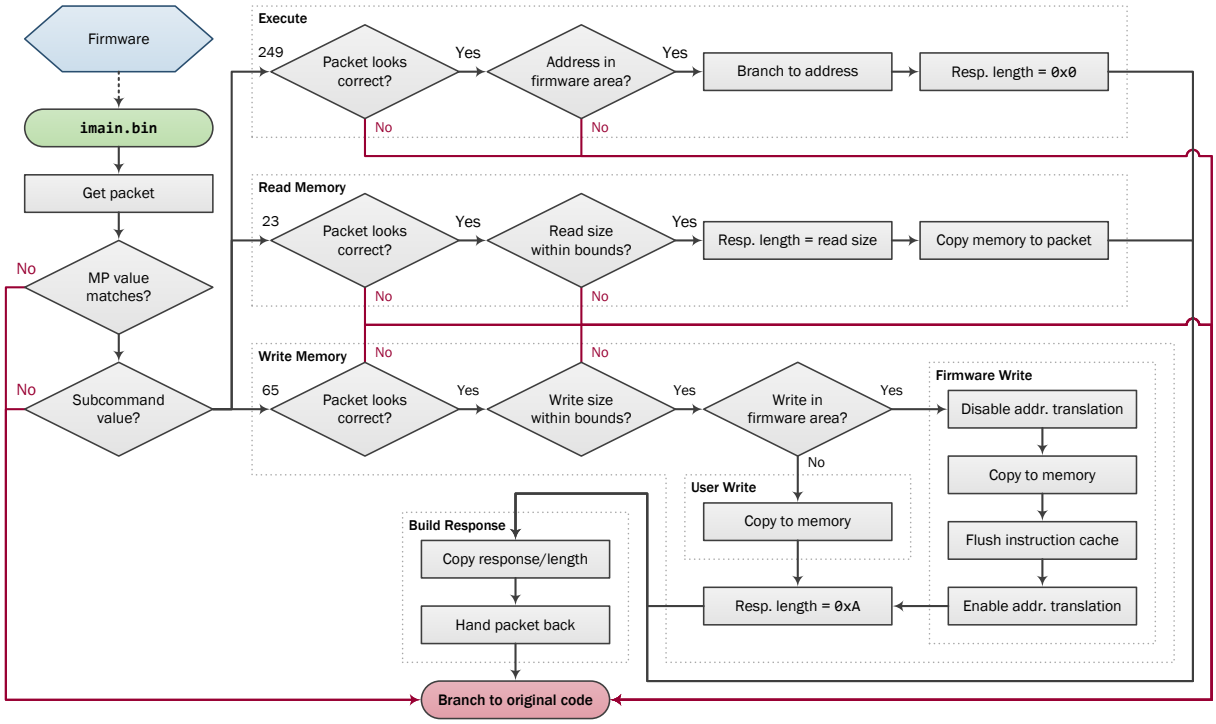

Figure 8: Operation of implant

Structurally, this component is not laid out like a function; instead, it is written to replace a branch in a jump table. This means that it has a few differences from the rest of the code, such as it not having the same initial setup code (register/stack saving, etc.) and it branching to the default case at the end of execution without setting a return address. This also means that some registers are set prior to execution, as they would be for the other branches of the jump table—again reinforcing that this is not an entire function.

When execution is transferred from the firmware, the implant begins by getting the address of the packet currently being processed. It then checks to see whether the "MP" is either 255 (`0xFF`), or equal to a byte stored in memory. It is not clear what "MP" means in this context—this is how the Python module names the optional argument that gets written into the packet. If the value matches, the implant then checks to see if the subcommand value stored in the packet is a known value—one of **23**, **65**, or **249**, telling it to read memory, write memory, or execute.

- The subcommand **23** (read memory) takes two arguments: the memory address to read from and the size of the read. When it is triggered, the implant performs some checks on the rest of the packet to ensure that the two arguments it needs are present, verifies that the read size is greater than 0 and less than or equal to 1,024, then copies memory into the response packet based on the input arguments. The packet response size is based on the amount of data read.
- The subcommand **65** (write memory) takes two arguments: the memory address to write to and the data to write. This command is the most complicated of the three. When it is triggered, the implant checks to make sure the arguments are present, ensures that there

is a non-zero amount of data to write, then checks to see whether the write is within the firmware memory region.
    - If the write is in a userspace region, the implant simply writes the provided data to the memory address specified.
    - If the write is in the firmware region, the implant goes through a more complicated process to ensure that the code being written has no chance of being executed during the write, and that the changes take effect immediately. To accomplish this, it disables address translation/caching and external interrupts, copies the provided data to the address specified, and flushes the instruction cache, then re-enables address translation/caching. It is worth noting that this is very similar to what the exploit does to write to the firmware region.
- The subcommand **249** (execute) takes a single argument: the address to call. When this command is triggered, the implant checks that the packet contains an address and that the provided address is within the firmware region, then calls that address.

Once the subcommand has finished executing, the implant builds the response packet using a length determined by the subcommand and a fixed response code, and branches back to the original default jump table case, finishing its execution.

Although this is a fairly simplistic RAT, an attacker can use these primitives to build much more complicated actions. For example, an attacker could execute arbitrary code on a Tricon by reading memory to figure out where to place the custom function and return value, writing the custom code to an empty location, executing at the address of the shellcode, storing a return value elsewhere in memory; and reading memory to extract the return value. This was proven during testing with a modified version of the implant that had known bugs fixed, providing first-hand evidence that this capability is very real.

IMPLICATIONS

While it is safe to say that HatMan is a valuable tool for ICS reconnaissance, it is likely designed as part of a multi-pronged attack that collectively would degrade industrial processes, or worse. Were both the process and the safety systems to be degraded simultaneously, persons, property, and/or the environment could suffer physical harm—barring the presence of additional safety mechanisms.

Due to the unique nature of each facility, there is no way for CISA to assess the impact of this malware on an individual plant. Thus, CISA strongly advises that individual asset owners assess the impact of a compromise on their safety systems. Facility owners and operators should discuss the impact of a safety system compromise and consider adding contingencies to their continuity of operations planning for impacts associated with such a compromise at their facility.

The construction of the different HatMan components indicates significant knowledge about ICS environments—specifically Tricon controllers—and an extended development lifecycle to refine such an advanced attack. In addition, it is very likely that an additional component or a separate piece of malware has been developed to impact a control system in tandem with a HatMan attack on the safety system. Although there may be theories as to what this might look like—

considering the areas in which Triconex equipment is used—this piece of the puzzle has not yet been revealed.

It is also worth considering the possibility of other threat actors moving into this attack space. Because the HatMan samples have been made public—some files are on VirusTotal and many have been made available on other sites—it is very likely that both researchers and other threat actors alike are doing their own analysis. In particular, the components made available could allow another party to build a similar attack, or to use it as a basis for attacks on other systems. To this end, the security of all safety systems, not just Triconex controllers, should be considered.

One of the library files—`crc.pyc`—also poses an interesting conundrum. Based on its contents, it appears that this file was either built for interaction with other ICS systems, or part of an unknown project that dealt with ICS equipment. Although CISA cannot substantiate such a claim without hard evidence, one could, in the worst case, interpret this as proof of development of another "prong" of the HatMan attack on both control and safety systems.

DETECTION AND MITIGATION

Schneider Electric has provided an updated security bulletin[1] describing a method for detecting—and removing—the HatMan malware.

In addition, a YARA rule that matches the three binary components—`trilog.exe`, `inject.bin`, and `imain.bin`—is included in Appendix A. This is not a reliable method for detection, as the files may or may not be present on any workstation, and such a rule cannot be used on a Tricon controller itself; however, it could be useful for detection with agent-based detection systems or for scanning for artifacts.

A number of vendors that provide solutions for detecting anomalies through passive network scanning have added the capability to detect the network traffic generated by the HatMan malware. Although this may not specifically prevent an attack, it would allow for an early warning that the malware might exist on a particular network or safety system.

It is worth noting that the onboard security features of Triconex hardware do not serve as effective prevention/mitigation. The ACLs that are available are solely based on IP address, meaning that an attacker could still use the programming workstation to compromise the safety device. Later Triconex devices have X.509 signing for programming, but this is also not a bulletproof mitigation strategy, as it is entirely feasible for the authors to update their script to employ these certificates—resident on the programming workstation—to sign any updates they push, circumventing the measure. At best, this would be a stop-gap measure.

Ultimately, the best mitigation strategy for this malware—and others of the same sort—is to employ defense in depth and follow any relevant best practices. Rather than solely attempting to protect vulnerable targets—such as the Triconex devices targeted by HatMan—one prevents an attacker from ever reaching them.

[1] https://www.schneider-electric.com/ww/en/download/document/SEVD-2017-347-01

CONTACT INFORMATION

Recipients of this report are encouraged to contribute any additional information that they may have related to this threat. For any questions related to this report, please reach out to:

- Phone: +1-703-235-8832
- Email: NCCICCustomerService@hq.dhs.gov

FEEDBACK

DHS strives to make this report a valuable tool for our partners and welcomes feedback on how this publication could be improved. You can help by answering a few short questions about this report at the following URL: https://www.us-cert.gov/forms/feedback.

APPENDIX A: YARA SIGNATURE

The following is a YARA rule that matches the binary components of the HatMan malware. This rule is also available on the ICS-CERT website.

```
/*
* DESCRIPTION: Yara rules to match the known binary components of the HatMan
* malware targeting Triconex safety controllers. Any matching
* components should hit using the "hatman" rule in addition to a
* more specific "hatman_*" rule.
* AUTHOR: DHS/NCCIC/ICS-CERT
*/

/* Private rules that are used at the end in the public rules. */
private rule hatman_filesize {
    condition:
        filesize < 350KB
}
private rule hatman_setstatus : APT unknown_attribution hatman RAT {
    strings:
        $preset = { 80 00 40 3c 00 00 62 80 40 00 80 3c 40 20 03 7c
                    ?? ?? 82 40 04 00 62 80 60 00 80 3c 40 20 03 7c
                    ?? ?? 82 40 ?? ?? 42 38                         }
    condition:
        hatman_filesize and $preset
}
private rule hatman_memcpy : hatman {
    strings:
        $memcpy_be = { 7c a9 03 a6 38 84 ff ff 38 63 ff ff 8c a4 00 01
                       9c a3 00 01 42 00 ff f8 4e 80 00 20             }
        $memcpy_le = { a6 03 a9 7c ff ff 84 38 ff ff 63 38 01 00 a4 8c
                       01 00 a3 9c f8 ff 00 42 20 00 80 4e             }
    condition:
        hatman_filesize and ($memcpy_be or $memcpy_le)
}
private rule hatman_dividers : hatman {
    strings:
        $div1 = { 9a 78 56 00 }
        $div2 = { 34 12 00 00 }
    condition:
        hatman_filesize and $div1 and $div2
}
private rule hatman_nullsub : hatman {
    strings:
        $nullsub = { ff ff 60 38 02 00 00 44 20 00 80 4e }
    condition:
        hatman_filesize and $nullsub
}
```

```
private rule hatman_origaddr : hatman {
    strings:
        $oaddr_be = { 3c 60 00 03 60 63 96 f4 4e 80 00 20 }
        $oaddr_le = { 03 00 60 3c f4 96 63 60 20 00 80 4e }
    condition:
        hatman_filesize and ($oaddr_be or $oaddr_le)
}
private rule hatman_origcode : hatman {
    strings:
        $ocode_be = { 3c 00 00 03 60 00 a0 b0 7c 09 03 a6 4e 80 04 20 }
        $ocode_le = { 03 00 00 3c b0 a0 00 60 a6 03 09 7c 20 04 80 4e }
    condition:
        hatman_filesize and ($ocode_be or $ocode_le)
}
private rule hatman_mftmsr : hatman {
    strings:
        $mfmsr_be = { 7c 63 00 a6 }
        $mfmsr_le = { a6 00 63 7c }
        $mtmsr_be = { 7c 63 01 24 }
        $mtmsr_le = { 24 01 63 7c }
    condition:
        hatman_filesize and (($mfmsr_be and $mtmsr_be) or ($mfmsr_le and $mtmsr_le))
}
private rule hatman_loadoff : hatman {
    strings:
        $loadoff_be = { 80 60 00 04 48 00 ?? ?? 70 60 ff ff 28 00 00 00
                        40 82 ?? ?? 28 03 00 00 41 82 ?? ??             }
        $loadoff_le = { 04 00 60 80 ?? ?? 00 48 ff ff 60 70 00 00 00 28
                        ?? ?? 82 40 00 00 03 28 ?? ?? 82 41             }
    condition:
        hatman_filesize and ($loadoff_be or $loadoff_le)
}
private rule hatman_injector_int : hatman {
    condition:
        hatman_filesize and hatman_memcpy and hatman_origaddr and hatman_loadoff
}
private rule hatman_payload_int : hatman {
    condition:
        hatman_filesize and hatman_memcpy and hatman_origcode and hatman_mftmsr
}

/* Actual public rules to match using the private rules. */
rule hatman_compiled_python : hatman {
    condition:
        hatman_nullsub and hatman_setstatus and hatman_dividers
}

rule hatman_injector : APT unknown_attribution hatman RAT {
```

```
    condition:
        hatman_injector_int and not hatman_payload_int
}
rule hatman_payload : APT unknown_attribution hatman RAT {
    condition:
        hatman_payload_int and not hatman_injector_int
}
rule hatman_combined : APT unknown_attribution hatman RAT {
    condition:
        hatman_injector_int and hatman_payload_int and hatman_dividers
}
rule hatman : APT unknown_attribution hatman RAT {
    meta:
        author = "DHS/NCCIC/ICS-CERT"
        description = "Matches the known samples of the HatMan malware."
    condition:
        hatman_compiled_python or hatman_injector or
                hatman_payload or hatman_combined
}
rule hatman_netexec : hatman {
    strings:
        $a1 = "=signal" wide
        $a2 = { 43 00 6f 00  6d 00 53 00  70 00 65 00  63 00 00 00
                20 00 3e 00  3e 00 20 00  4e 00 55 00  4c 00 00 00 }
        $a3 = { 6a 00 6a 00 6a 00 6a 0a 6a 32 e8 [4] 85 c0 }
        $c  = { 6a 0d 6a 36 [0-2] e8}
    condition:
       hatman_filesize and all of ($a*) and #c > 3
}
```