NCCIC

National Cybersecurity and
Communications Integration Center

MALWARE ANALYSIS REPORT

MAR-17-352-01 HATMAN—SAFETY SYSTEM TARGETED MALWARE

December 18, 2017

SUMMARY

The HatMan malware affects Triconex controllers by modifying in-memory firmware to add additional programming. The extra functionality allows an attacker to read/modify memory contents and execute custom 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 can potentially result in physical consequences. This report discusses the components and capabilities of the malware and some potential mitigations. Media reporting also refers to this malware as both TRITON and TRISIS.

A NOTE ON THE ANALYSIS

This report will discuss the malware as though it is entirely functional. We are aware that the malware may currently have bugs—due to descriptions of how it is behaving—that prevent it from effecting its desired changes. Though this report presents a "worst case scenario," it should be considered accurate. We have no reason to suspect that the malware's creators have not fixed its bugs, or that a functional copy does not exist somewhere that we have not yet seen. We have and will continue to work with Schneider Electric—the manufacturer of the targeted safety controller—to test our hypotheses and the malware, and we will update the report when we can confirm any additional information.

ANALYSIS

HatMan follows Stuxnet and Industroyer/CrashOverride, but surpasses them with the ability to directly interact with, remotely control, **and compromise** a safety system—a nearly unprecedented feat. This report will discuss the malware's context, components, and capabilities.

CONTEXT: WHAT ARE SAFETY SYSTEMS?

Safety systems or PLCs are specialized hardware—similar to traditional PLCs—with a strong emphasis on reliability and predictable failure.[1] Unlike 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. They are normally used to provide a way for a process to safely shut down when it has encountered unsafe operating conditions, and provide a high degree of safety and reliability with important monitoring capabilities for process engineers.

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. All of these could potentially appear within the safety system environment, possibly in similar file system locations as TriStation TS1131 software installations.

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

- An executable that programs a Triconex device without the TriStation software,
- A native[2] shellcode program that injects a payload into the in-memory copy of the Triconex firmware,[3] and
- A native shellcode payload that performs malicious actions.

1 Safety systems are designed such that *if they were to fail*, they would fail in an entirely predictable manner, so that the worst case scenario is fully known.

2 The shellcode here is PowerPC, where newer Triconex devices (if targeted in other versions), would be ARM.

3 The persistent copy of the firmware is immutable, as it is stored in PROM.

REPROGRAMMING THE SAFETY PLC

In its current iteration, the component that programs the Triconex controllers is written entirely in Python.[4] 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`—that includes a Python environment.

This Python script communicates using four Python modules—`TsBase`, `TsLow`, `TsHi`, and `TS_cnames`—that collectively implement the TriStation network protocol ("TS", via UDP 1502); this is the protocol that the TriStation TS1131 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,[5] and could feasibly have been reverse engineered from knowing this, other manufacturers' documentation, and watching traffic between the programming workstation and safety PLC. In addition, this protocol does not require any authentication or encryption, although ACLs may be configured on the PLC.[6] 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 TriStation protocol, the Python modules also expose a set of methods to interact with the compromised safety PLC. These use a specific network command with some specially crafted data to pass messages to the implant in order to expose the functionality of the malicious modifications to an attacker on a computer on the safety network, regardless of key switch position.

The general execution flow of the Python script can be seen in the flow diagram below. It begins with connecting to a Triconex controller using an IP address provided as an argument. Once it has connected, it sets an argument for itself by appending a small program, running it, and then checking to ensure it succeeded. It then overwrites the program with a small dummy program if

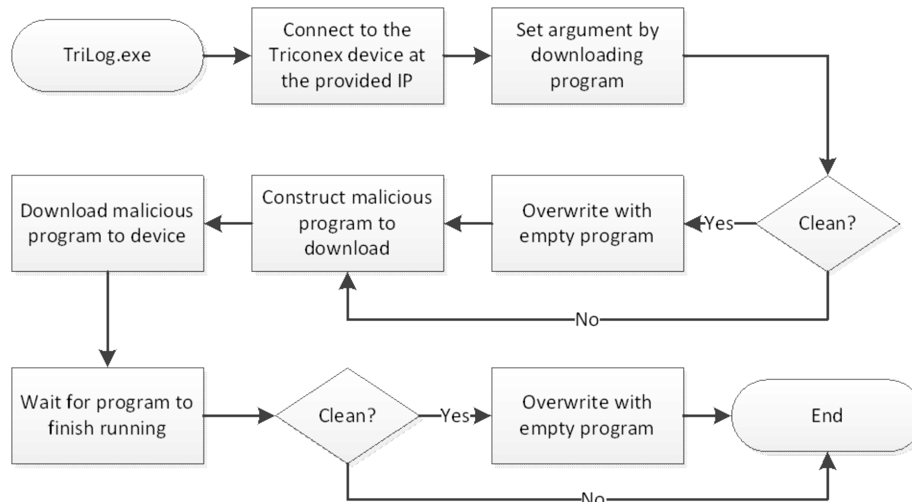
---

4 However, it would be relatively trivial to rewrite this component into any other language.

5 The TSAA protocol can be used for reading and writing data points and other user-application-level tasks from a third-party program, rather than using Schneider software.

6 That being said, ACLs are not an effective mitigation strategy, as it is solely based on IP address; thus, the attacker could still use the programming workstation to compromise the PLC. In addition, later Triconex devices have X.509 signing for programming—this is also not a bulletproof mitigation strategy, as it is very 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.

its “clean” flag is set. Following this, it builds the malicious payload and overwrites the same program slot with this new code. Then it runs the malicious payload, waits for it to finish, and verifies it succeeded. Finally, it overwrites the malicious program with the same dummy program if the same “clean” flag is set.



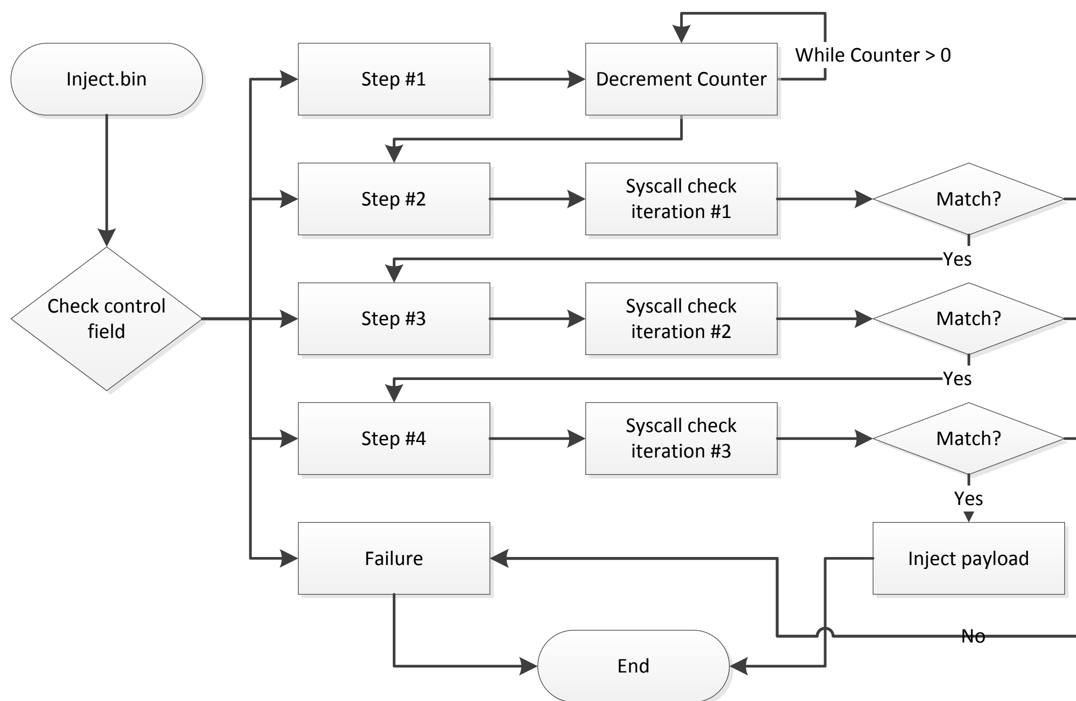
The script embedded in the PC component does not interact with the command modified by the malicious payload, but it is feasible and likely that a separate script was used to actually interact with the compromised safety controller as needed.

THE MALICIOUS PAYLOAD

The malicious shellcode is split into two separate pieces—`inject.bin` and `imain.bin`. The former is more generic code that handles injecting the payload into the running firmware, while the latter is the payload that actually performs the additional, malicious functionality. Both binary components are PowerPC bytecode—the same as the firmware and any applications compiled for and downloaded to the safety controller.

The injector masquerades as a normal, compiled PowerPC program for the Triconex device. It uses the argument value written by the first program downloaded by the Python script (described previously); however, unlike the earlier program that heuristically determines a location for the argument, it assumes that this value exists at a particular address. This address 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 and control execution progress; and as a field for writing debug information upon failure. This allows the attackers to monitor the status of and debug problems with the injector as it runs.

This shellcode component follows the general execution path shown in the flow chart below. During each cycle, the program runs and branches based on the control value stored in the control field. It begins by waiting a number of cycles based on the counter portion of the control field then proceeds to the later steps. The next two steps perform checks using a system call, matching an output value against an expected constant. The fourth step performs a similar system call check and then, if that check passes, copies the payload into memory, changes a function pointer to the address of the copied payload, and returns.



At the time of writing, research is ongoing to determine exactly what effects the intermediate system call checks and/or actions produce upon the device.

Once the injector has finished running, it will have modified a function pointer that is used in processing a specific network command ("get main processor diagnostic data") such that, when that command is received, the payload is executed first prior to normal processing.

The second component of the malicious program—the payload, `imain.bin`—is designed to take a TriStation protocol "get main processor diagnostic data" command, look for a specially crafted packet body, and perform custom actions on demand. It is able 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, it disables address translation,[7] writes the code at the provided address, flushes the instruction cache, and re-enables address translation. Based on our understanding of the Triconex device, this allows the malware to make changes to the running firmware; however, it appears these changes will be persistent only in memory.

IMPLICATIONS

Although by itself HatMan does not do anything catastrophic—safety systems do not directly control the process, so a degraded safety system will not cause a correctly functioning process to misbehave—it could be very damaging when combined with malware that impacts the process in tandem. Were both to be degraded simultaneously, physical harm could be effected on persons, property, or the environment.

It is safe to say that while HatMan would be a valuable tool for ICS reconnaissance, it is likely designed to degrade industrial processes or worse. Overall, the construction of the different components would indicate a significant knowledge about ICS environments—specifically Triconex controllers—and an extended development lifecycle to refine such an advanced attack.

DETECTION

NCCIC is working with Schneider Electric to develop an effective method for both detection and mitigation of the known samples in the short term. Schneider Electric is evaluating the possibility of longer-term strategies for detection and mitigation as well.

In addition, a YARA rule that matches the three binary components—`trilog.exe`, `inject.bin`, and `imain.bin`—is included as an appendix. This is not necessarily 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 Triconex controller itself; however, it could be useful for detection with agent-based detection systems or for scanning for artifacts.

7 This is also known as putting the processor into “real mode”.

MITIGATIONS

There are a number of possible mitigations that can reduce the chance of Triconex devices being compromised. Currently, none of these are complete solutions to the problem, as none will prevent the malware under all circumstances. The following are possible mitigations:

- *Ensure Triconex devices connect only to networks required for their proper function.*

  If possible, remove Triconex devices from any networks to which they do not need a persistent connection; however, if historians or other applications that rely on real-time information from safety controllers are needed, this might not be possible.

- *Only switch the key to "PROGRAM" when necessary.*

  When a Triconex device has its key set to "RUN" or "REMOTE", it is unable to be programmed; thus, only moving the key to "PROGRAM" whenever the device must be programmed will reduce the likelihood that it can be compromised. That being said, many processes may be in flux due to changes in the environment, so safety systems may need to be reprogrammed with some frequency; thus, it is rarely possible to never move the key from "RUN". This is not only a way to reduce the chances of being infected, but also best practice.

- *Avoid connecting TriStation workstations to a larger network, avoid using removable media to transfer programs, and follow best practices for updating workstations.*

  The malicious program must be transmitted to a machine—likely a TriStation workstation—that is connected to the same network as the Triconex device. These machines must be treated with caution to prevent malware spreading to them. If there is no path in, the malware we analyzed cannot jump through a workstation onto the safety PLC. This recommendation follows the best practices DHS NCCIC/ICS-CERT has previously detailed for control systems workstations in the "Defense in Depth" document.[8] Any other guidance provided in this document should also be considered.

8 This document may be found on the ICS-CERT web site.

Schneider Electric has published a security notification (SEVD-2017-347-01)[9] that recommends a variety of mitigations to decrease the chances of infection that are more tailored toward their specific systems. Several key points are mentioned here:

- Safety systems must always be deployed on isolated networks using zones and conduits as defined in IEC-62443;
- Physical controls should be in place so no unauthorized person has access to the plant, equipment rooms, safety controllers, safety peripheral equipment or the safety network;
- All controllers should reside in locked cabinets;
- All TriStation terminals (Triconex programming software) should be kept in locked cabinets and should never be connected to any network other than the safety network;
- Operator stations should be configured to display an alarm whenever the Tricon keyswitch is in the "Program Mode" with the key removed and secured; and
- Enhanced security features in TriStation, as well as the Triconex communication modules, should be enabled.

9. https://www.schneider-electric.com/en/download/document/SEVD-2017-347-01/

APPENDIX: YARA RULES

The following is a YARA rule that matches the binary components of the HatMan malware. The YARA rule is available at: https://ics-cert.us-cert.gov/sites/default/files/file_attach/MAR-17-352-01.yara

```
/*
 * 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
 */

/* Globally only look at small files. */

private global rule hatman_filesize : hatman {
    condition:
        filesize < 100KB
}

/* Private rules that are used at the end in the public rules. */

private rule hatman_setstatus : hatman {
    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:
        $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:
        $memcpy_be or $memcpy_le
}



private rule hatman_dividers : hatman {
    strings:
```

```
        $div1       = { 9a 78 56 00 }
        $div2       = { 34 12 00 00 }
    condition:
        $div1 and $div2
}
private rule hatman_nullsub : hatman {
    strings:
        $nullsub     = { ff ff 60 38  02 00 00 44  20 00 80 4e }
    condition:
        $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:
        $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:
        $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 }
        $mtmsr_le   = { 24 01 63 7c }
    condition:
        ($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:
        $loadoff_be or $loadoff_le
}
```

```
private rule hatman_injector_int : hatman {
    condition:
        hatman_memcpy and hatman_origaddr and hatman_loadoff
}
private rule hatman_payload_int : hatman {
    condition:
        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 : hatman {
    condition:
        hatman_injector_int and not hatman_payload_int
}
rule hatman_payload : hatman {
    condition:
        hatman_payload_int and not hatman_injector_int
}
rule hatman_combined : hatman {
    condition:
        hatman_injector_int and hatman_payload_int and hatman_dividers
}
rule hatman : hatman {
    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
}
```

ICS-CERT CONTACT

For any questions related to this report, please contact NCCIC Customer Service at:

U.S. Toll Free: (888) 282-0870
Email: ncciccustomerservice@hq.dhs.gov

Click this link for information on industrial control systems security information and incident reporting.

ICS-CERT continuously strives to improve its products and services. You can help by answering a short series of questions about this product on the Feedback page.