A Totally Tubular Treatise on TRITON and TriStation

By by Steve Miller, Evan Reese

Published: 2018-06-07 · Archived: 2026-04-06 01:52:28 UTC

Threat Research

June 07, 2018 |

Introduction

In December 2017, FireEye's Mandiant discussed an incident response involving the TRITON framework. The TRITON attack and many of the publicly discussed ICS intrusions involved routine techniques where the threat actors used only what is necessary to succeed in their mission. For both INDUSTROYER and TRITON, the attackers moved from the IT network to the OT (operational technology) network through systems that were accessible to both environments. Traditional malware backdoors, Mimikatz distillates, remote desktop sessions, and other well-documented, easily-detected attack methods were used throughout these intrusions.

Despite the routine techniques employed to gain access to an OT environment, the threat actors behind the TRITON malware framework invested significant time learning about the Triconex Safety Instrumented System (SIS) controllers and TriStation, a proprietary network communications protocol. The investment and purpose of the Triconex SIS controllers leads Mandiant to assess the attacker's objective was likely to build the capability to cause physical consequences.

TriStation remains closed source and there is no official public information detailing the structure of the protocol, raising several questions about how the TRITON framework was developed. Did the actor have access to a Triconex controller and TriStation 1131 software suite? When did development first start? How did the threat actor reverse engineer the protocol, and to what extent? What is the protocol structure?

FireEye’s Advanced Practices Team was born to investigate adversary methodologies, and to answer these types of questions, so we started with a deeper look at the TRITON’s own Python scripts.

Glossary:

- TRITON – Malware framework designed to operate Triconex SIS controllers via the TriStation protocol.
- TriStation – UDP network protocol specific to Triconex controllers.
- TRITON threat actor – The human beings who developed, deployed and/or operated TRITON.

Diving into TRITON's Implementation of TriStation

TriStation is a proprietary network protocol and there is no public documentation detailing its structure or how to create software applications that use TriStation. The current TriStation UDP/IP protocol is little understood, but natively implemented through the TriStation 1131 software suite. TriStation operates by UDP over port 1502 and

allows for communications between designated masters (PCs with the software that are “engineering workstations”) and slaves (Triconex controllers with special communications modules) over a network.

To us, the Triconex systems, software and associated terminology sound foreign and complicated, and the TriStation protocol is no different. Attempting to understand the protocol from ground zero would take a considerable amount of time and reverse engineering effort – so why not learn from TRITON itself? With the TRITON framework containing TriStation communication functionality, we pursued studying the framework to better understand this mysterious protocol. Work smarter, not harder, amirite?

The TRITON framework has a multitude of functionalities, but we started with the basic components:

- TS_cnames.pyc # Compiled at: 2017-08-03 10:52:33
- TsBase.pyc # Compiled at: 2017-08-03 10:52:33
- TsHi.pyc # Compiled at: 2017-08-04 02:04:01
- TsLow.pyc # Compiled at: 2017-08-03 10:46:51

TsLow.pyc (Figure 1) contains several pieces of code for error handling, but these also present some cues to the protocol structure.

```
    def print_last_error(self):
        if self._uerror != 0:
            pdict('UDP Error:', self._uerror, NvTs_err)
        tcm_err = self.tcm_result()[0]
        if tcm_err >= 7:
            pdict('TCM Error ', tcm_err, TS_cnames.TS_cst)
        ts_err = self.ts_result()[0]
        if ts_err != 0:
            pdict('TS Error ', ts_err, TS_cnames.TS_names)
        if self._perror != 0:
            pdict('Parse Error:', self._perror, NvTs_err)
        print 'last data out:'
        sh.dump(self._ldout)
        print 'last data in:'
        sh.dump(self._ldin)
```

Figure 1: TsLow.pyc function print_last_error()

In the TsLow.pyc’s function for print_last_error we see error handling for “TCM Error”. This compares the TriStation packet value at offset 0 with a value in a corresponding array from TS_cnames.pyc (Figure 2), which is largely used as a “dictionary” for the protocol.

```
TS_cst = {1: 'CONNECT REQUEST',
    2: 'CONNECT REPLY',
    3: 'DISCONN REPLY',
    4: 'DISCONN REQUEST',
    5: 'COMMAND REPLY',
    6: 'PING',
    7: 'CONN LIMIT REACHED',
    8: 'NOT CONNECTED',
    9: 'MPS ARE DEAD',
    10: 'ACCESS DENIED',
    11: 'CONNECTION FAILED'
    }
```

Figure 2: TS_cnames.pyc TS_cst array

From this we can infer that offset 0 of the TriStation protocol contains message types. This is supported by an additional function, tcm_result, which declares type, size = struct.unpack('<HH', data_received[0:4]), stating that the first two bytes should be handled as integer type and the second two bytes are integer size of the TriStation message. This is our first glimpse into what the threat actor(s) understood about the TriStation protocol.

Since there are only 11 defined message types, it really doesn't matter much if the type is one byte or two because the second byte will always be 0x00.

We also have indications that message type 5 is for all Execution Command Requests and Responses, so it is curious to observe that the TRITON developers called this “Command Reply.” (We won’t understand this naming convention until later.)

Next we examine TsLow.pyc’s print_last_error function (Figure 3) to look at “TS Error” and “TS_names.” We begin by looking at the ts_err variable and see that it references ts_result.

```
def print_last_error(self):
    if self._uerror != 0:
        pdict('UDP Error:', self._uerror, NvTs_err)
    tcm_err = self.tcm_result()[0]
    if tcm_err >= 7:
        pdict('TCM Error ', tcm_err, TS_cnames.TS_cst)
    ts_err = self.ts_result()[0]
    if ts_err != 0:
        pdict('TS Error ', ts_err, TS_cnames.TS_names)
    if self._perror != 0:
        pdict('Parse Error:', self._perror, NvTs_err)
    print 'last data out:'
    sh.dump(self._ldout)
    print 'last data in:'
    sh.dump(self._ldin)
```

Figure 3: TsLow.pyc function print_last_error() with ts_err highlighted

We follow that thread to ts_result, which defines a few variables in the next 10 bytes (Figure 4): dir, cid, cmd, cnt, unk, cks, siz = struct.unpack('<, ts_packet[0:10]). Now things are heating up. What fun. There's a lot to unpack here, but the most interesting thing is how this piece script breaks down 10 bytes from ts_packet into different variables.

```
def ts_result(self):
    if self._ts_result != None:
        return self._ts_result
    else:
        self._ts_result = (-1, None, 0)
        self._perror = -1
        while True:
            tcm_reply = self.tcm_result()
            if tcm_reply[0] != 5:
                self._perror = 0
                break
            ts_packet = tcm_reply[1]
            if len(ts_packet) < 10:
                print 'bad ts size'
                self._perror = 7
                break
            dir, cid, cmd, cnt, unk, cks, siz = struct.unpack('<BBBBHHH', ts_packet[0:10])
            if cnt != self._lcnt:
                print 'bad ts cnt'
                self._perror = 9
                self.udp_flush()
                break
            if siz != len(ts_packet):
                print 'bad ts size'
                break
            packet_to_check = ts_packet[0:6] + struct.pack('<H', 0) + ts_packet[8:]
```

Figure 4: ts_result with ts_packet header variables highlighted

```
def tcm_result(self):
    if self._tcm_result != None:
        return self._tcm_result
    else:
        self._perror = -1
        data_received = self.udp_result()
        while True:
            self._tcm_result = (0, None)
            if data_received == None or len(data_received) < 6:
                print 'bad tcm size'
                self._perror = 10
                break
            type, size = struct.unpack('<HH', data_received[0:4])
            packet = data_received[4:-2]
```

Figure 5: tcm_result

Referencing tcm_result (Figure 5) we see that it defines type and size as the first four bytes (offset 0 – 3) and tcm_result returns the packet bytes 4:-2 (offset 4 to the end minus 2, because the last two bytes are the CRC-16 checksum). Now that we know where tcm_result leaves off, we know that the ts_reply "cmd" is a single byte at offset 6, and corresponds to the values in the TS_cnames.pyc array and TS_names (Figure 6). The TRITON script also tells us that any integer value over 100 is a likely "command reply." Sweet.

When looking back at the ts_result packet header definitions, we begin to see some gaps in the TRITON developer's knowledge: dir, cid, cmd, cnt, unk, cks, siz = struct.unpack('<, ts_packet[0:10]). We're clearly speculating based on naming conventions, but we get an impression that offsets 4, 5 and 6 could be "direction", "controller ID" and "command", respectively. Values such as "unk" show that the developer either did not know or did not care to identify this value. We suspect it is a constant, but this value is still unknown to us.

```
TS_names = {-1: 'Not set',
    0: 'Start download all',
    1: 'Start download change',
    2: 'Update configuration',
    3: 'Upload configuration',
    4: 'Set I/O addresses',
    5: 'Allocate network',
    6: 'Load vector table',
    7: 'Set calendar',
    8: 'Get calendar',
    9: 'Set scan time',
    10: 'End download all',
    11: 'End download change',
    12: 'Cancel download change',
    13: 'Attach TRICON',
    14: 'Set I/O address limits',
    15: 'Configure module',
    16: 'Set multiple point values',
    17: 'Enable all points',
    18: 'Upload vector table',
    19: 'Get CP status ',
    20: 'Run program',
    21: 'Halt program',
    22: 'Pause program',
    23: 'Do single scan',
    24: 'Get chassis status',
    25: 'Get minimum scan time',
    26: 'Set node number',
    27: 'Set I/O point values',
    28: 'Get I/O point values',
    29: 'Get MP status',
    30: 'Set retentive values',
    31: 'Adjust clock calendar',
    32: 'Clear module alarms',
    33: 'Get event log',
    34: 'Set SOE block',
    35: 'Record event log',
    36: 'Get SOE data',
    37: 'Enable OVD',
    38: 'Disable OVD',
    39: 'Enable all OVDs',
    40: 'Disable all OVDs',
    41: 'Process MODBUS',
    42: 'Upload network',
    43: 'Set lable',
    44: 'Configure system variables',
    45: 'Deconfigure module',
    46: 'Get system variables',
    47: 'Get module types',
    48: 'Begin conversion table download',
    49: 'Continue conversion table download',
    50: 'End conversion table download',
    51: 'Get conversion table',
    52: 'Set ICM status',
    53: 'Broadcast SOE data available',
    54: 'Get module versions',
    55: 'Allocate program',
    56: 'Allocate function',
    57: 'Clear retentives',
    58: 'Set initial values',
    59: 'Start TS2 program download',
    60: 'Set TS2 data area',
    61: 'Get TS2 data',
    62: 'Set TS2 data',
    63: 'Set program information',
    64: 'Get program information',
    65: 'Upload program',
    66: 'Upload function',
    67: 'Get point groups',
    68: 'Allocate symbol table',
    69: 'Get I/O address',
    70: 'Resend I/O address',
    71: 'Get program timing',
    72: 'Allocate multiple functions',
    73: 'Get node number',
    74: 'Get symbol table',
    75: 'Unk75',
    76: 'Unk76',
    77: 'Unk77',
    78: 'Unk78',
    79: 'Unk79',
    80: 'Go to DOWNLOAD mode',
    81: 'Unk81',
    83: 'Unk83',
    100: 'Command rejected',
    101: 'Download all permitted',
    102: 'Download change permitted',
    103: 'Modification accepted',
    104: 'Download cancelled',
    105: 'Program accepted',
    106: 'TRICON attached',
    107: 'I/O addresses set',
    108: 'Get CP status response',
    109: 'Program is running',
    110: 'Program is halted',
    111: 'Program is paused',
    112: 'End of single scan',
    113: 'Get chassis configuration response',
    114: 'Scan period modified',
```

Figure 6: Excerpt TS_cnames.pyc TS_names array, which contain TRITON actor's notes for execution command function codes

TriStation Protocol Packet Structure

The TRITON threat actor's knowledge and reverse engineering effort provides us a better understanding of the protocol. From here we can start to form a more complete picture and document the basic functionality of TriStation. We are primarily interested in message type 5, Execution Command, which best illustrates the overall structure of the protocol. Other, smaller message types will have varying structure.

```
00000000 05 00 28 00 00 00 37 be 00 00 73 09 28 00 03 00
00000010 01 00 05 00 00 00 05 00 ff ff 60 38 02 00 00 44
00000020 20 00 80 4e df a1 28 8d dc 4e 57 a0 49 f9
```

Offset	Length	Description	Notes
0x00	1	Message Type	0x05 for Execution Command
0x01	1	Constant	0x00 was seen in all tests.
0x02	2	Message Length	Length from offset 4 to the CRC
0x04	1	Message Source ID	0x00 from TriStation PC and 0x01 from Triconex Controller
0x05	1	Unknown ID	TRITON called this value "cid," perhaps Controller ID. We saw 0x00 for TriStation PC and !0x00 for the Triconex Controller.
0x06	1	Command Function Code	See Execution Command Function List in Appendix B
0x07	1	Command Counter	Increments by 1 from 0 to 255 then starts over
0x08	2	Unknown Constant	Unknown value, but 0x0000 was seen in all tests.
0x0A	2	Sum of Bytes Checksum	Sum except first two bytes, this field, and the CRC
0x0C	2	Message Length	Same as offset 0x02
0x0E	?	Rest of Payload Data	Variable Length
0x??	2	CRC Checksum	CRC-16 Checksum

Figure 7: Sample TriStation "Allocate Program" Execution Command, with color annotation and protocol legend

Corroborating the TriStation Analysis

Minute discrepancies aside, the TriStation structure detailed in Figure 7 is supported by other public analyses. Foremost, researchers from the Coordinated Science Laboratory (CSL) at University of Illinois at Urbana-Champaign published a 2017 paper titled "Attack Induced Common-Mode Failures on PLC-based Safety System in a Nuclear Power Plant". The CSL team mentions that they used the Triconex System Access Application (TSAA) protocol to reverse engineer elements of the TriStation protocol. TSAA is a protocol developed by the same company as TriStation. Unlike TriStation, the TSAA protocol structure is described within official documentation. CSL assessed similarities between the two protocols would exist and they leveraged TSAA to better understand TriStation. The team's overall research and analysis of the general packet structure aligns with our TRITON-sourced packet structure.

There are some awesome blog posts and whitepapers out there that support our findings in one way or another. Writeups by Midnight Blue Labs, Accenture, and US-CERT each explain how the TRITON framework relates to the TriStation protocol in superb detail.

TriStation's Reverse Engineering and TRITON's Development

When TRITON was discovered, we began to wonder how the TRITON actor reverse engineered TriStation and implemented it into the framework. We have a lot of theories, all of which seemed plausible: Did they build, buy,

borrow, or steal? Or some combination thereof?

Our initial theory was that the threat actor purchased a Triconex controller and software for their own testing and reverse engineering from the "ground up", although if this was the case we do not believe they had a controller with the exact vulnerable firmware version, else they would have had fewer problems with TRITON in practice at the victim site. They may have bought or used a demo version of the TriStation 1131 software, allowing them to reverse engineer enough of TriStation for the framework. They may have stolen TriStation Python libraries from ICS companies, subsidiaries or system integrators and used the stolen material as a base for TriStation and TRITON development. But then again, it is possible that they borrowed TriStation software, Triconex hardware and Python connectors from government-owned utility that was using them legitimately.

Looking at the raw TRITON code, some of the comments may appear oddly phrased, but we do get a sense that the developer is clearly using many of the right vernacular and acronyms, showing smarts on PLC programming. The TS_cnames.pyc script contains interesting typos such as 'Set lable', 'Alocate network accepted', 'Symbol table ccepted' and 'Set program information reponse'. These appear to be normal human error and reflect neither poor written English nor laziness in coding. The significant amount of annotation, cascading logic, and robust error handling throughout the code suggests thoughtful development and testing of the framework. This complicates the theory of "ground up" development, so did they base their code on something else?

While learning from the TriStation functionality within TRITON, we continued to explore legitimate TriStation software. We began our search for "TS1131.exe" and hit dead ends sorting through TriStation DLLs until we came across a variety of TriStation utilities in MSI form. We ultimately stumbled across a juicy archive containing "Trilog v4." Upon further inspection, this file installed "TriLog.exe," which the original TRITON executable mimicked, and a couple of supporting DLLs, all of which were timestamped around August 2006.

When we saw the DLL file description "Tricon Communications Interface" and original file name "TricCom.DLL", we knew we were in the right place. With a simple look at the file strings, "BAZINGA!" We struck gold.

File Name	**tr1com40.dll**
MD5	069247DF527A96A0E048732CA57E7D3D
Size	110592
Compile Date	2006-08-23
File Description	Tricon Communications Interface

Product Name	TricCom Dynamic Link Library
File Version	4.2.441
Original File Name	TricCom.DLL
Copyright	Copyright © 1993-2006 Triconex Corporation

The tr1com40.DLL is exactly what you would expect to see in a custom application package. It is a library that helps support the communications for a Triconex controller. If you've pored over TRITON as much as we have, the moment you look at strings you can see the obvious overlaps between the legitimate DLL and TRITON's own TS_cnames.pyc.

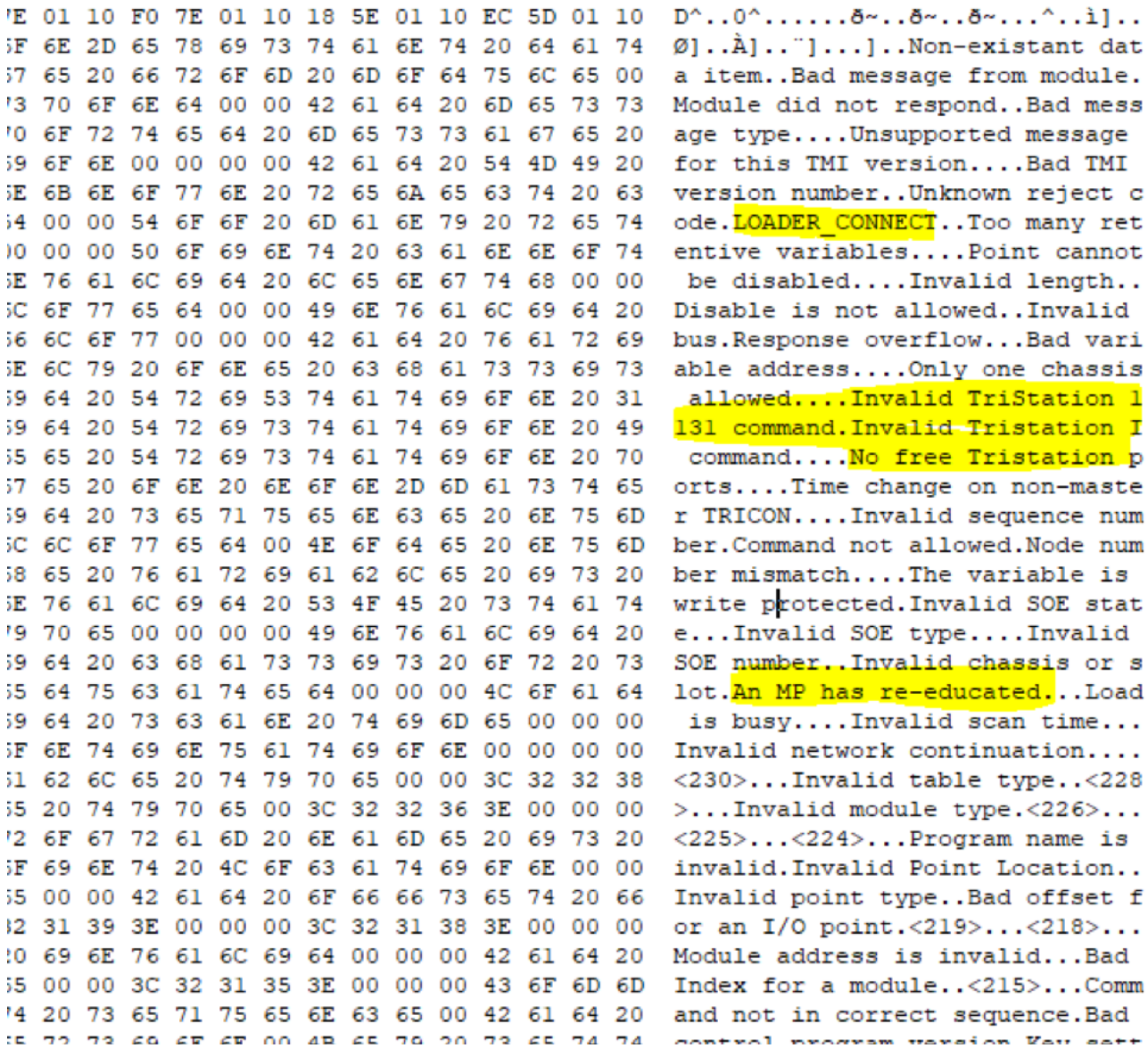

Figure 8: Strings excerpt from tr1com40.DLL

Each of the execution command "error codes" from TS_cnames.pyc are in the strings of tr1com40.DLL (Figure 8). We see "An MP has re-educated" and "Invalid Tristation I command". Even misspelled command strings verbatim such as "Non-existant data item" and "Alocate network accepted". We also see many of the same unknown values. What is obvious from this discovery is that some of the strings in TRITON are likely based on code used in communications libraries for Trident and Tricon controllers.

In our brief survey of the legitimate Triconex Corporation binaries, we observed a few samples with related string tables.

Pe:dllname	**Compile Date**	**Reference CPP Strings Code**

Lagcom40.dll	2004/11/19	$Workfile: LAGSTRS.CPP $ $Modtime: Jul 21 1999 17:17:26 $ $Revision: 1.0
Tr1com40.dll	2006/08/23	$Workfile: TR1STRS.CPP $ $Modtime: May 16 2006 09:55:20 $ $Revision: 1.4
Tridcom.dll	2008/07/23	$Workfile: LAGSTRS.CPP $ $Modtime: Jul 21 1999 17:17:26 $ $Revision: 1.0
Triccom.dll	2008/07/23	$Workfile: TR1STRS.CPP $ $Modtime: May 16 2006 09:55:20 $ $Revision: 1.4
Tridcom.dll	2010/09/29	$Workfile: LAGSTRS.CPP $ $Modtime: Jul 21 1999 17:17:26 $ $Revision: 1.0
Tr1com.dll	2011/04/27	$Workfile: TR1STRS.CPP $ $Modtime: May 16 2006 09:55:20 $ $Revision: 1.4
Lagcom.dll	2011/04/27	$Workfile: LAGSTRS.CPP $ $Modtime: Jul 21 1999 17:17:26 $ $Revision: 1.0
Triccom.dll	2011/04/27	$Workfile: TR1STRS.CPP $ $Modtime: May 16 2006 09:55:20 $ $Revision: 1.4

We extracted the CPP string tables in TR1STRS and LAGSTRS and the TS_cnames.pyc TS_names array from TRITON, and compared the 210, 204, and 212 relevant strings from each respective file.

TS_cnames.pyc TS_names and tr1com40.dll share 202 of 220 combined table strings. The remaining strings are unique to each, as seen here:

TS_cnames.TS_names (2017 pyc)	**Tr1com40.dll (2006 CPP)**
Go to DOWNLOAD mode	<200>

Not set	<209>
Unk75	Bad message from module
Unk76	Bad message type
Unk77	Bad TMI version number
Unk78	Module did not respond
Unk79	Open Connection: Invalid SAP %d
Unk81	Unsupported message for this TMI version
Unk83	
Wrong command	

TS_cnames.pyc TS_names and Tridcom.dll (1999 CPP) shared only 151 of 268 combined table strings, showing a much smaller overlap with the seemingly older CPP library. This makes sense based on the context that Tridcom.dll is meant for a Trident controller, not a Tricon controller. It does seem as though Tr1com40.dll and TR1STRS.CPP code was based on older work.

We are not shocked to find that the threat actor reversed legitimate code to bolster development of the TRITON framework. They want to work smarter, not harder, too. But after reverse engineering legitimate software and implementing the basics of the TriStation, the threat actors still had an incomplete understanding of the protocol. In TRITON's TS_cnames.pyc we saw "Unk75", "Unk76", "Unk83" and other values that were not present in the tr1com40.DLL strings, indicating that the TRITON threat actor may have explored the protocol and annotated their findings beyond what they reverse engineered from the DLL. The gaps in TriStation implementation show us why the actors encountered problems interacting with the Triconex controllers when using TRITON in the wild.

You can see more of the Trilog and Triconex DLL files on VirusTotal.

Item Name	MD5	Description
Tr1com40.dll	069247df527a96a0e048732ca57e7d3d	Tricom Communcations DLL
Data1.cab	e6a3c93a6d433cbaf6f573b6c09d76c4	Parent of Tr1com40.dll
Trilog v4.1.360R	13a3b83ba2c4236ca59aba679941c8a5	RAR Archive of TriLog
TridCom.dll	5c2ed617fdec4779cb33c89082a43100	Trident Communications DLL

Afterthoughts

Seeing Triconex systems targeted with malicious intent was new to the world six months ago. Moving forward it would be reasonable to anticipate additional frameworks, such as TRITON, designed for usage against other SIS controllers and associated technologies. If Triconex was within scope, we may see similar attacker methodologies affecting the dominant industrial safety technologies.

Basic security measures do little to thwart truly persistent threat actors and monitoring only IT networks is not an ideal situation. Visibility into both the IT and OT environments is critical for detecting the various stages of an ICS intrusion. Simple detection concepts such as baseline deviation can provide insight into abnormal activity.

While the TRITON framework was actively in use, how many traditional ICS “alarms” were set off while the actors tested their exploits and backdoors on the Triconex controller? How many times did the TriStation protocol, as implemented in their Python scripts, fail or cause errors because of non-standard traffic? How many TriStation UDP pings were sent and how many Connection Requests? How did these statistics compare to the baseline for TriStation traffic? There are no answers to these questions for now. We believe that we can identify these anomalies in the long run if we strive for increased visibility into ICS technologies.

We hope that by holding public discussions about ICS technologies, the Infosec community can cultivate closer relationships with ICS vendors and give the world better insight into how attackers move from the IT to the OT space. We want to foster more conversations like this and generally share good techniques for finding evil. Since most of all ICS attacks involve standard IT intrusions, we should probably come together to invent and improve any guidelines for how to monitor PCs and engineering workstations that bridge the IT and OT networks. We envision a world where attacking or disrupting ICS operations costs the threat actor their cover, their toolkits, their time, and their freedom. It's an ideal world, but something nice to shoot for.

Thanks and Future Work

There is still much to do for TRITON and TriStation. There are many more sub-message types and nuances for parsing out the nitty gritty details, which is hard to do without a controller of our own. And although we've published much of what we learned about the TriStation here on the blog, our work will continue as we continue our study of the protocol.

Thanks to everyone who did so much public research on TRITON and TriStation. We have cited a few individuals in this blog post, but there is a lot more community-sourced information that gave us clues and leads for our research and testing of the framework and protocol. We also have to acknowledge the research performed by the TRITON attackers. We borrowed a lot of your knowledge about TriStation from the TRITON framework itself.

Finally, remember that we're here to collaborate. We think most of our research is right, but if you notice any errors or omissions, or have ideas for improvements, please ~~spear phish~~ contact: smiller@fireeye.com.

Recommended Reading

- Attackers Deploy New ICS Attack Framework "TRITON" and Cause Operational Disruption to Critical Infrastructure
- Attack Induced Common-Mode Failures on PLC-Based Safety System in a Nuclear Power Plant: Practical Experience Report
- Analyzing the TRITON industrial malware
- Repository containting original and decompiled files of TRISIS/TRITON/HATMAN malware
- TRISIS Malware Analysis of Safety System Targeted Malware

Appendix A: TriStation Message Type Codes

The following table consists of hex values at offset 0 in the TriStation UDP packets and the associated dictionary definitions, extracted verbatim from the TRITON framework in library TS_cnames.pyc.

Value at 0x0	**Message Type**
1	Connection Request
2	Connection Response
3	Disconnect Request
4	Disconnect Response

5	Execution Command
6	Ping Command
7	Connection Limit Reached
8	Not Connected
9	MPS Are Dead
10	Access Denied
11	Connection Failed

Appendix B: TriStation Execution Command Function Codes

The following table consists of hex values at offset 6 in the TriStation UDP packets and the associated dictionary definitions, extracted verbatim from the TRITON framework in library TS_cnames.pyc.

Value at 0x6	**TS_cnames String**
0	0: 'Start download all',
1	1: 'Start download change',
2	2: 'Update configuration',
3	3: 'Upload configuration',
4	4: 'Set I/O addresses',

5	5: 'Allocate network',
6	6: 'Load vector table',
7	7: 'Set calendar',
8	8: 'Get calendar',
9	9: 'Set scan time',
A	10: 'End download all',
B	11: 'End download change',
C	12: 'Cancel download change',
D	13: 'Attach TRICON',
E	14: 'Set I/O address limits',
F	15: 'Configure module',
10	16: 'Set multiple point values',
11	17: 'Enable all points',
12	18: 'Upload vector table',
13	19: 'Get CP status ',

14	20: 'Run program',
15	21: 'Halt program',
16	22: 'Pause program',
17	23: 'Do single scan',
18	24: 'Get chassis status',
19	25: 'Get minimum scan time',
1A	26: 'Set node number',
1B	27: 'Set I/O point values',
1C	28: 'Get I/O point values',
1D	29: 'Get MP status',
1E	30: 'Set retentive values',
1F	31: 'Adjust clock calendar',
20	32: 'Clear module alarms',
21	33: 'Get event log',
22	34: 'Set SOE block',

23	35: 'Record event log',
24	36: 'Get SOE data',
25	37: 'Enable OVD',
26	38: 'Disable OVD',
27	39: 'Enable all OVDs',
28	40: 'Disable all OVDs',
29	41: 'Process MODBUS',
2A	42: 'Upload network',
2B	43: 'Set lable',
2C	44: 'Configure system variables',
2D	45: 'Deconfigure module',
2E	46: 'Get system variables',
2F	47: 'Get module types',
30	48: 'Begin conversion table download',
31	49: 'Continue conversion table download',

32	50: 'End conversion table download',
33	51: 'Get conversion table',
34	52: 'Set ICM status',
35	53: 'Broadcast SOE data available',
36	54: 'Get module versions',
37	55: 'Allocate program',
38	56: 'Allocate function',
39	57: 'Clear retentives',
3A	58: 'Set initial values',
3B	59: 'Start TS2 program download',
3C	60: 'Set TS2 data area',
3D	61: 'Get TS2 data',
3E	62: 'Set TS2 data',
3F	63: 'Set program information',
40	64: 'Get program information',

41	65: 'Upload program',
42	66: 'Upload function',
43	67: 'Get point groups',
44	68: 'Allocate symbol table',
45	69: 'Get I/O address',
46	70: 'Resend I/O address',
47	71: 'Get program timing',
48	72: 'Allocate multiple functions',
49	73: 'Get node number',
4A	74: 'Get symbol table',
4B	75: 'Unk75',
4C	76: 'Unk76',
4D	77: 'Unk77',
4E	78: 'Unk78',
4F	79: 'Unk79',

50	80: 'Go to DOWNLOAD mode',
51	81: 'Unk81',
52	
53	83: 'Unk83',
54	
55	
56	
57	
58	
59	
5A	
5B	
5C	
5D	
5E	

5F	
60	
61	
62	
63	
64	100: 'Command rejected',
65	101: 'Download all permitted',
66	102: 'Download change permitted',
67	103: 'Modification accepted',
68	104: 'Download cancelled',
69	105: 'Program accepted',
6A	106: 'TRICON attached',
6B	107: 'I/O addresses set',
6C	108: 'Get CP status response',
6D	109: 'Program is running',

6E	110: 'Program is halted',
6F	111: 'Program is paused',
70	112: 'End of single scan',
71	113: 'Get chassis configuration response',
72	114: 'Scan period modified',
73	115: '<115>',
74	116: '<116>',
75	117: 'Module configured',
76	118: '<118>',
77	119: 'Get chassis status response',
78	120: 'Vectors response',
79	121: 'Get I/O point values response',
7A	122: 'Calendar changed',
7B	123: 'Configuration updated',
7C	124: 'Get minimum scan time response',

7D	125: '<125>',
7E	126: 'Node number set',
7F	127: 'Get MP status response',
80	128: 'Retentive values set',
81	129: 'SOE block set',
82	130: 'Module alarms cleared',
83	131: 'Get event log response',
84	132: 'Symbol table ccepted',
85	133: 'OVD enable accepted',
86	134: 'OVD disable accepted',
87	135: 'Record event log response',
88	136: 'Upload network response',
89	137: 'Get SOE data response',
8A	138: 'Alocate network accepted',
8B	139: 'Load vector table accepted',

8C	140: 'Get calendar response',
8D	141: 'Label set',
8E	142: 'Get module types response',
8F	143: 'System variables configured',
90	144: 'Module deconfigured',
91	145: '<145>',
92	146: '<146>',
93	147: 'Get conversion table response',
94	148: 'ICM print data sent',
95	149: 'Set ICM status response',
96	150: 'Get system variables response',
97	151: 'Get module versions response',
98	152: 'Process MODBUS response',
99	153: 'Allocate program response',
9A	154: 'Allocate function response',

9B	155: 'Clear retentives response',
9C	156: 'Set initial values response',
9D	157: 'Set TS2 data area response',
9E	158: 'Get TS2 data response',
9F	159: 'Set TS2 data response',
A0	160: 'Set program information reponse',
A1	161: 'Get program information response',
A2	162: 'Upload program response',
A3	163: 'Upload function response',
A4	164: 'Get point groups response',
A5	165: 'Allocate symbol table response',
A6	166: 'Program timing response',
A7	167: 'Disable points full',
A8	168: 'Allocate multiple functions response',
A9	169: 'Get node number response',

AA	170: 'Symbol table response',
AB	
AC	
AD	
AE	
AF	
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	

B9	
BA	
BB	
BC	
BD	
BE	
BF	
C0	
C1	
C2	
C3	
C4	
C5	
C6	
C7	

C8	200: 'Wrong command',
C9	201: 'Load is in progress',
CA	202: 'Bad clock calendar data',
CB	203: 'Control program not halted',
CC	204: 'Control program checksum error',
CD	205: 'No memory available',
CE	206: 'Control program not valid',
CF	207: 'Not loading a control program',
D0	208: 'Network is out of range',
D1	209: 'Not enough arguments',
D2	210: 'A Network is missing',
D3	211: 'The download time mismatches',
D4	212: 'Key setting prohibits this operation',
D5	213: 'Bad control program version',
D6	214: 'Command not in correct sequence',

D7	215: '<215>',
D8	216: 'Bad Index for a module',
D9	217: 'Module address is invalid',
DA	218: '<218>',
DB	219: '<219>',
DC	220: 'Bad offset for an I/O point',
DD	221: 'Invalid point type',
DE	222: 'Invalid Point Location',
DF	223: 'Program name is invalid',
E0	224: '<224>',
E1	225: '<225>',
E2	226: '<226>',
E3	227: 'Invalid module type',
E4	228: '<228>',
E5	229: 'Invalid table type',

E6	230: '<230>',
E7	231: 'Invalid network continuation',
E8	232: 'Invalid scan time',
E9	233: 'Load is busy',
EA	234: 'An MP has re-educated',
EB	235: 'Invalid chassis or slot',
EC	236: 'Invalid SOE number',
ED	237: 'Invalid SOE type',
EE	238: 'Invalid SOE state',
EF	239: 'The variable is write protected',
F0	240: 'Node number mismatch',
F1	241: 'Command not allowed',
F2	242: 'Invalid sequence number',
F3	243: 'Time change on non-master TRICON',
F4	244: 'No free Tristation ports',

F5	245: 'Invalid Tristation I command',
F6	246: 'Invalid TriStation 1131 command',
F7	247: 'Only one chassis allowed',
F8	248: 'Bad variable address',
F9	249: 'Response overflow',
FA	250: 'Invalid bus',
FB	251: 'Disable is not allowed',
FC	252: 'Invalid length',
FD	253: 'Point cannot be disabled',
FE	254: 'Too many retentive variables',
FF	255: 'LOADER_CONNECT',
	256: 'Unknown reject code'
