ARMIS®

WHITE PAPER

PwnedPiper

PWNEDPIPER

Uncovering Vulnerabilities in Critical Infrastructure of Healthcare Facilities

Barak Hadad
Ben Seri

Table of Contents

Introduction

Armis Labs discovered 9 vulnerabilities, dubbed PwnedPiper, affecting the Nexus Control Panel, which powers all current stations of the Translogic Pneumatic Tube System (PTS) by Swisslog Healthcare. Translogic is one of the most advanced PTS solutions in the market used by more than 80% of hospitals in North America and in over 3,000 hospitals worldwide.

A high-level overview of the discovered vulnerabilities and their impact can be found here. This document details the attack surface exposed by PTS systems, as well as the discovered vulnerabilities in the Nexus Control Panel. It also details the severe impact these vulnerabilities have if exploited on affected devices.

Who we are

Armis Labs is the Armis research team and is focused on mixing and splitting the atoms that comprise the IoT devices that surround us - be it a smart personal assistant, a benign-looking printer, a SCADA controller, or a life-supporting device such as a hospital bedside patient monitor.

Our previous research includes:

- Modipwn - Authentication bypass leads to remote-code-execution in Schneider Electric Modicon PLCs
- NAT Slipstreaming 2.0 - A NAT bypass technique that abuses support for VoIP protocols by NATs
- EtherOops - Exploit utilizing packet-in-packet attacks on ethernet cables to bypass firewalls & NATs.
- CDPwn - Five critical vulnerabilities in various implementations of the Cisco Discovery Protocol.
- URGENT/11 - 11 Zero-Day vulnerabilities impacting VxWorks, the most widely used Real-Time Operating System (RTOS).
- BLEEDINGBIT - Two chip-level vulnerabilities in Texas Instruments BLE chips, embedded in Enterprise-grade Access Points.
- BlueBorne - An attack vector targeting devices via RCE vulnerabilities in Bluetooth stacks used by over 5.3 Billion devices.

Pneumatic Tube System (PTS)

Pneumatic tube systems (PTS) are pressurized tube networks that transfer physical carriers through them. It might be surprising to learn that these systems are still relevant and commonly used today. However, certain applications that require the efficient transport of physical objects still exist today. Most of these applications transport packages for relatively short distances of up to a few kilometers.

The main industries using PTS today are:

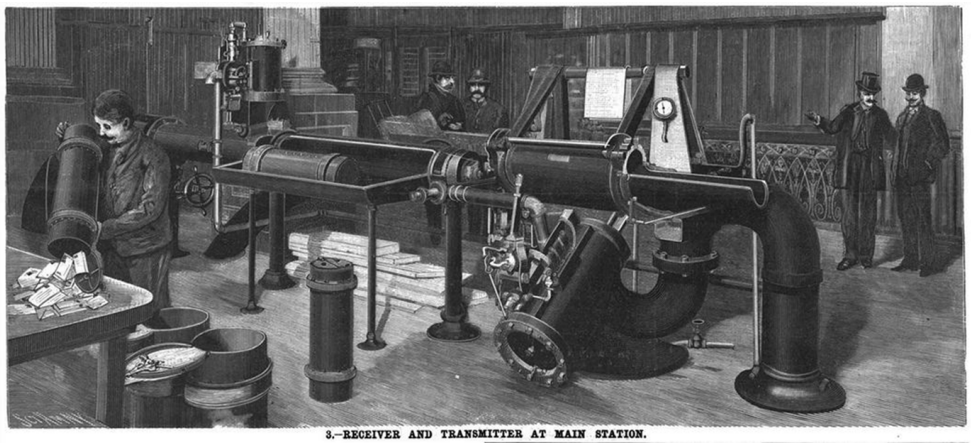
- **Medical facilities** use PTS to send various specimens, blood products, and medicine across different departments and even in and out of the emergency rooms.
- **Industrial production lines** and laboratories use PTS to send product samples from the factory line to the lab.
- **Banks and department stores** use PTS to transfer cash to centralized locations.

Modern tube systems reach speeds of ~7.5 m (25 ft) per second (27 KM/H), and powerful systems can transport items weighing up to 50Kg (110 lb)

While current applications of PTS solutions focus on transporting relatively small items, historically, PTS solutions were envisioned to be used for **Public Transportation**. An experimental pneumatic subway line was even built in 1867 under Broadway in Manhattan, but the technology never matured. Today, the idea to use PTS to deliver people across large distances has been resurrected by the *Hyperloop* project.

Physical Packet Switching

The concept of Pneumatic Tube Systems was first invented more than 200 years ago and is probably one of the earliest examples of packet switching networks. The New York Post Office system made use of the pneumatic tube, as seen in the following drawing from 1897:



3.—RECEIVER AND TRANSMITTER AT MAIN STATION.

In the 20th century, PTS solutions for inter-office communication became more and more prevalent, and an operator acting as a human router would transfer the capsules between the different tubes:

Taken from this article from Vox.

Over time, PTS solutions became fully automated - similar to the evolution that telecommunication systems had gone through. PTS systems became digitally connected, initially, through serial communication, and in modern solutions, all components are connected through Ethernet to the local IP network.

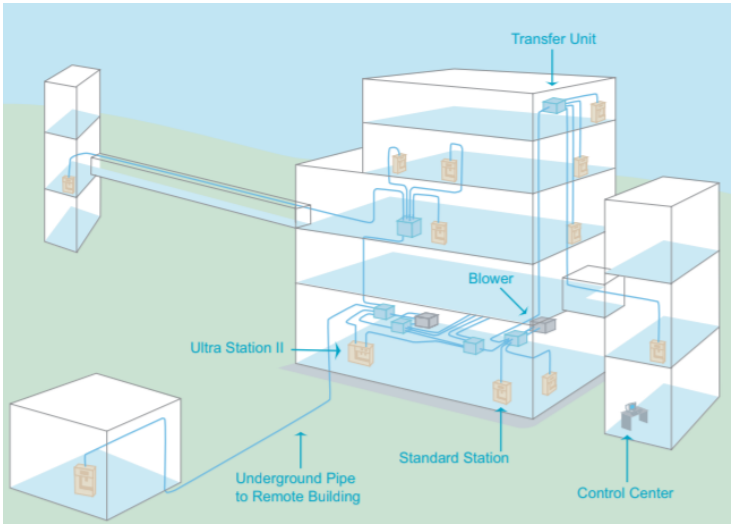
Pneumatic Tube Systems in Healthcare

The most prevalent use of PTS solutions currently is in Healthcare facilities. PTS solutions are vital to hospital operations as they automate logistics and the transport of materials throughout the hospital and are used for various applications:

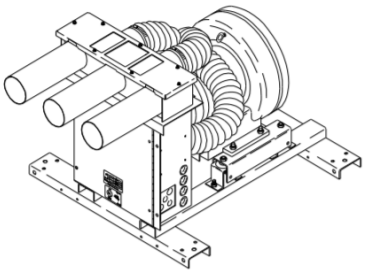
- Transferring various specimens from all departments of the hospital to centrally located laboratories, for testing.
- Distributing medicine from the hospital's pharmacy to all departments.
- Distributing blood units from the hospital's blood bank to operation rooms.

Components of a PTS network

PTS networks are built using various analog components that are all managed by a central server. The PTS system is based on blowers that maintain the required air pressure within tubes that interconnect using physical routers (diverters) which ultimately lead to the PTS stations.



Diverter



Blower

The blowers produce high pressured air that propels capsules through the tubes. The speed and direction a capsule travels within the tubes can be controlled by the speed and direction of the air pressure produced by the blowers. Certain items, such as blood products, might require slow transport with minimal acceleration. Advanced PTS solutions control blowers to allow for such transfers.

Diverters (also named transfer units) are the routers of the network. They are intersections between an uplink tube and multiple downlink tubes, allowing them to direct the carriers to the correct destination.

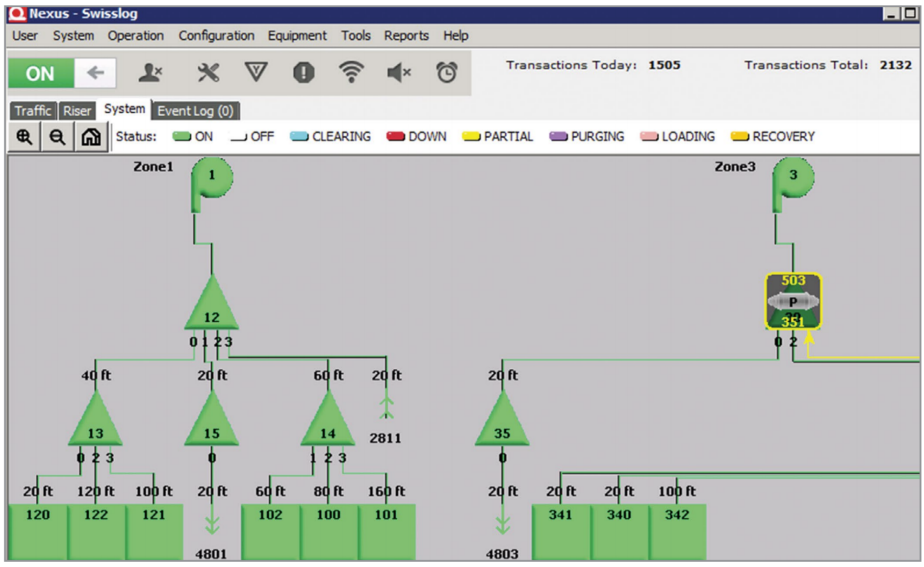
The endpoints of the PTS system are stations located throughout the hospital, where staff can send and receive carriers - the capsules that store the transferred items.

When a staff member wishes to send a carrier through the network, the central server will create a physical path for the carrier by instructing the various diverters to the correct paths, then turn on the blower with the correct air pressure, and whisk the carrier through the tubes to its destination.

PTS solutions offer different types of stations, with varying physical installation options and various advanced features. Generally speaking, however, the stations are the front-end of the system that allows operators to choose the destination of the carrier and queue a transaction.

Central management server

PTS systems use a star architecture in which all components (stations, blowers, diverters, etc.) are connected to a central management server that monitors and manages the overall system. The central server monitors the current state of the system and orchestrates the operation of all the components so that capsules are transferred efficiently throughout the system.



Screenshot from a Swisslog Translogic SCC (central management server), running Nexus software

Swisslog Translogic PTS

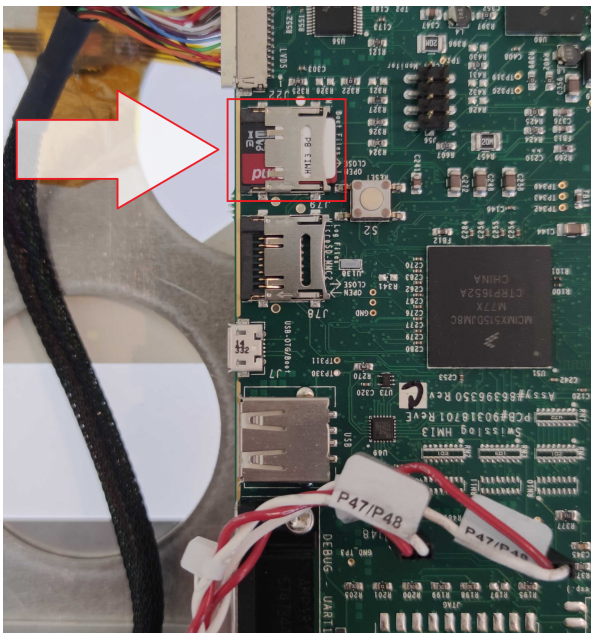
As noted above, Swisslog's Translogic PTS is one of the most popular PTS solutions in healthcare facilities and offers one of the most advanced PTS solutions in the market. It supports a LAN-connected network of stations, blowers, and diverters, all reporting to a central management server called **SCC**. The TransLogic system supports a variety of advanced features such as:

1. Secure transfers, with RFID and/or password-protected carriers.
2. Slow-speed transfers, for sensitive cargo.
3. Alert system, for user notifications via email/text/etc.
4. Remote system monitoring, for offloading the maintenance of the system to the vendor (managed in the Cloud).

The attack surface

To analyze the attack surface of any system installed within a hospital, we need to consider three types of potential attackers - an attacker that is physically within the hospital, an attacker that has gained access to the internal network, and an attacker on the Internet.

A physical attacker



The endpoints of PTS solutions, the stations, are often used by multiple users and are scattered throughout the hospital's various departments. A physical attacker that is able to interact with a station or attempt to physically access electronics within the station may be able to compromise it.

The current line of Swisslog Translogic stations are all powered by the Nexus Control Panel (more on that below), which has an SD card that holds the firmware run by the device. If an attacker is able to access this card, he can alter the code that runs on it and compromise the station.

Many Translogic stations (both current and legacy models) allow authentication of staff members using RFID cards. These modern stations also have touchscreen panels with a variety of menus and features. If vulnerabilities are found in these interfaces, a physical attacker may be able to compromise a station without the need to tinker with the electronics of a station.

An attacker within the LAN

Current models of the Translogic PTS components are IP-connected and communicate using unencrypted protocols. Performing man-in-the-middle attacks (such as ARP spoofing) can allow an attacker that has gained access to the internal IP network to intercept and alter packets used to control the operations of the PTS network. In addition, certain PTS components also have management ports for software updates and maintenance. Finding a vulnerability in any of these can allow an attacker with LAN access to remotely compromise such devices. More on this below.

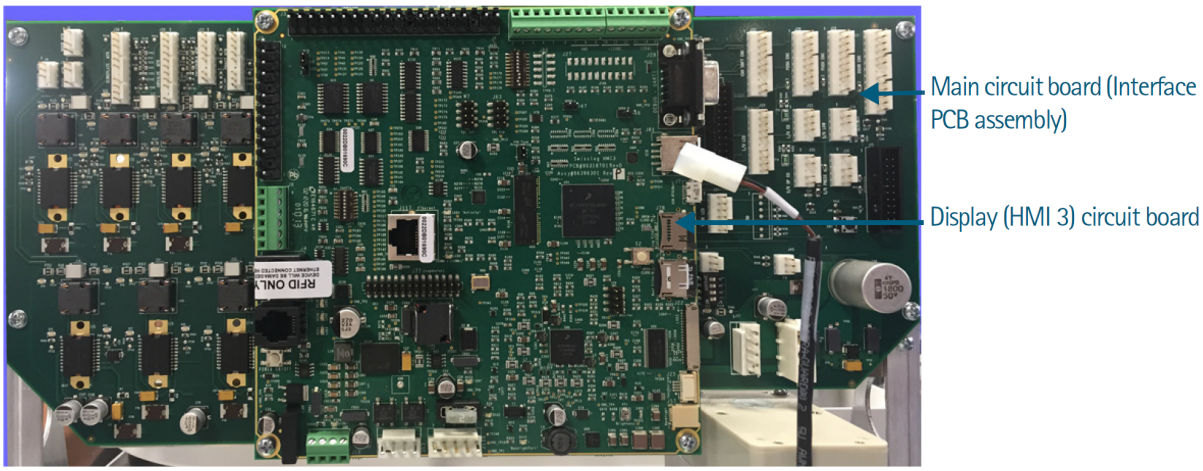
An Internet attacker

Most components of the Translogic PTS solution only communicate with the SCC (the central server), and do not require Internet connectivity. However, the SCC itself does require Internet connectivity to allow for various features. For example, The SCC can integrate with an *Alert System* that can notify individuals when a carrier has arrived in their station via email, SMS messages, and other methods. In addition, the SCC can be configured to use Swisslog's *Remote System Monitoring* solution, in which the server is maintained and monitored by Swisslog, from the Swisslog Cloud. The SCC server runs on a Windows server and is usually running a specific Windows version with which the Translogic software is compatible. For that reason, it is not uncommon for the SCC server to be using an outdated version of Windows.

Vulnerabilities in any Internet-based integration that the SCC server has, or in the underlying OS that powers it, may result in an Internet attacker being able to compromise the device from the Internet.

Swisslog PTS Station Research

Swisslog offers a variety of Translogic stations that vary in features, physical size, and types of installation. However, all current models of Translogic stations are powered by the Nexus Control Panel (which Swisslog also calls “HMI3 board”):



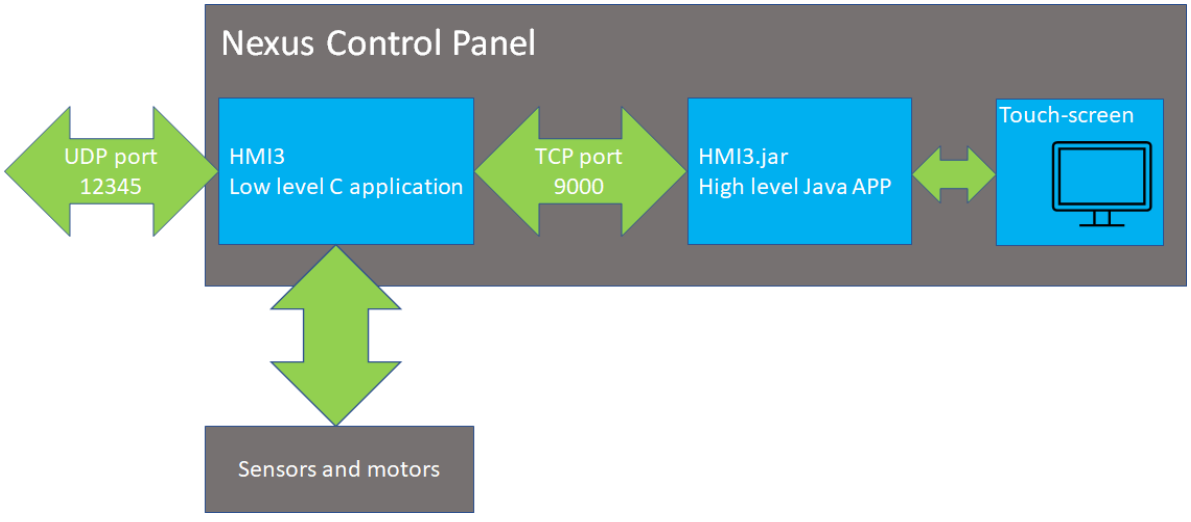
Circuit boards from the Nexus station (the latest Translogic station model);
The HMI3 board is the “brains” of all current Translogic stations

Nexus Control Panel (HMI3 board)

The Nexus Control Panel is a Linux based embedded device:

```
Linux freescale 2.6.35.3-433-g0fae922 #18 PREEMPT Fri Aug 2 15:34:08 MDT 2013 armv7l GNU/Linux
```

The Linux kernel used by this device is end-of-life, built on a version from 2010, which is likely susceptible to **many** known vulnerabilities. Here is a simplified design layout of the software components of the board:

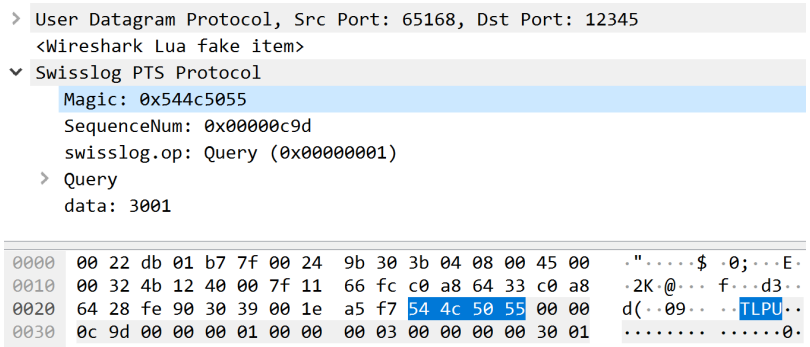


The Two main processes that operate the HMI3 board are:

- /root/HMI3: An ELF binary that is responsible for handling UDP communications with the SCC, and for controlling the station's sensors and motors. It communicates with a high-level Java app over a TCP port.
- /home/user/hmi/lib/HMI3.jar: A Java App that handles the high-level functions of the device, including the GUI of the station.

Translogic PTS Protocol

Translogic devices communicate with the central server using the TLP20 protocol - an unencrypted proprietary protocol that works over UDP port 12345. Each packet begins with the "TLPU" magic header (perhaps this stands for "Trans Logic Pneumatic UDP"), followed by a sequence number, a command identifier, and a payload of specific commands or response packets.



Wireshark capture of a TLP20 packet

By examining the firmware, it is possible to identify many interesting commands in this protocol, such as message broadcast, station door open/close (for releasing stored carriers), and firmware upgrade.

```
StationModelHandlers() {
  this.mMessageHandlers.put(MessageType.STATION_IS_RFID_EQUIPPED, new StationIsRFIDEquippedHandler());
  this.mMessageHandlers.put(MessageType.STATION_IS_NOT_RFID_EQUIPPED, new StationIsNotRFIDEquippedHandler());
  this.mMessageHandlers.put(MessageType.STATION_IS_RFID_OPERATIONAL, new StationIsRFIDOperationalHandler());
  this.mMessageHandlers.put(MessageType.STATION_IS_NOT_RFID_OPERATIONAL, new StationIsNotRFIDOperationalHandler());
  this.mMessageHandlers.put(MessageType.STATION_IS_DISPATCH_ONLY, new StationIsDispatchOnly(true));
  this.mMessageHandlers.put(MessageType.STATION_IS_RECEIVE_ONLY, new StationIsReceiveOnly(true));
  this.mMessageHandlers.put(MessageType.STATION_IS_NOT_DISPATCH_ONLY, new StationIsDispatchOnly(false));
  this.mMessageHandlers.put(MessageType.STATION_IS_NOT_RECEIVE_ONLY, new StationIsReceiveOnly(false));
  this.mMessageHandlers.put(MessageType.REQUEST_EMPTY_ENABLE_BUTTON, new EnableRequestEmptyHandler());
  this.mMessageHandlers.put(MessageType.REQUEST_EMPTY_DISABLE_BUTTON, new DisableRequestEmptyHandler());
  this.mMessageHandlers.put(MessageType.STATION_IS_EXPRESS, new StationIsExpress());
  this.mMessageHandlers.put(MessageType.STATION_ACCESS_CARD, new CardAccessStationHandler());
  this.mMessageHandlers.put(MessageType.STATION_ACCESS_PIN, new PinAccessStationHandler());
  this.mMessageHandlers.put(MessageType.STATION_ACCESS_REGULAR, new RegularAccessStationHandler());
  this.mMessageHandlers.put(MessageType.ACCESS_CODE_VALID, new AccessCodeValidHandler());
  this.mMessageHandlers.put(MessageType.ACCESS_CODE_INVALID, new AccessCodeInvalidHandler());
  this.mMessageHandlers.put(MessageType.INCOMING_CARRIERS, new IncomingCarrierHandler());
  this.mMessageHandlers.put(MessageType.SET_SECURE_CODE, new SetSecureCodeHandler());
```

Decompiled snippet from HMI3.jar

Some of these commands are handled by the low-level C application and some by the high-level Java application.

HMI3 low-level app architecture

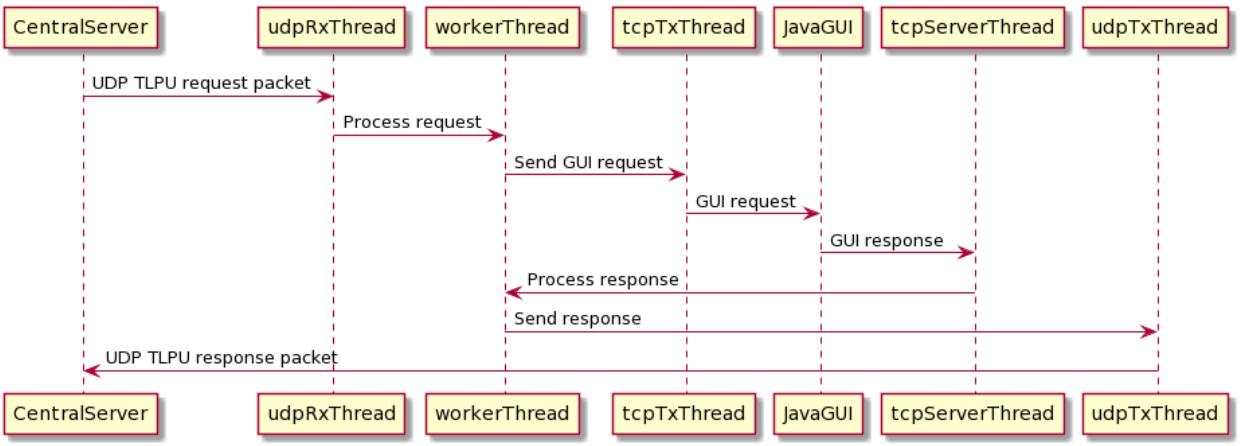
The low-level app runs multiple threads which are responsible for various tasks:

```
1  int startThreads()
2  {
3    int v0; // r0
4    int i; // [sp+4h] [bp-8h]
5
6    v0 = pthread_create(&ioThreadID, 0, ioThread, 0);
7    IO_safeDelay(v0);
8    if ( commMode == 1 )
9    {
10     pthread_create(&udpRxThreadID, 0, udpRxThread, 0);
11     pthread_create(&udpTxThreadID, 0, udpTxThread, 0);
12   }
13   else
14   {
15     pthread_create(&serialRxThreadID, 0, serialRxThread, 0);
16     pthread_create(&serialTxThreadID, 0, serialTxThread, 0);
17   }
18   pthread_create(&tcpServerThreadID, 0, tcpServerThread, 0);
19   pthread_create(&tcpTxThreadID, 0, tcpTxThread, 0);
20   pthread_create(&monitorHmiThreadID, 0, monitorHmiThread, 0);
21   pthread_create(&carrierSenseThreadID, 0, carrierSenseThread, 0);
22   for ( i = 0; worker_threads_num > i; ++i )
23   {
24     threadParams[i] = i + 1;
25     pthread_create(&workerThreadIDs[i], 0, workerThread, &threadParams[i]);
26   }
27   return pthread_create(&testThreadID, 0, testThread, 0);
28 }
```

The interesting threads are:

- *udpRxThread/udpTxThread* - Responsible for Rx/Tx communications over the TLP20 protocol with the SCC over UDP.
- tcpServerThread/tcpTxThread - Responsible for communicating with the Java App.
- *workerThreads* - Responsible for either handling incoming commands from the SCC, or forwarding them to the Java App.

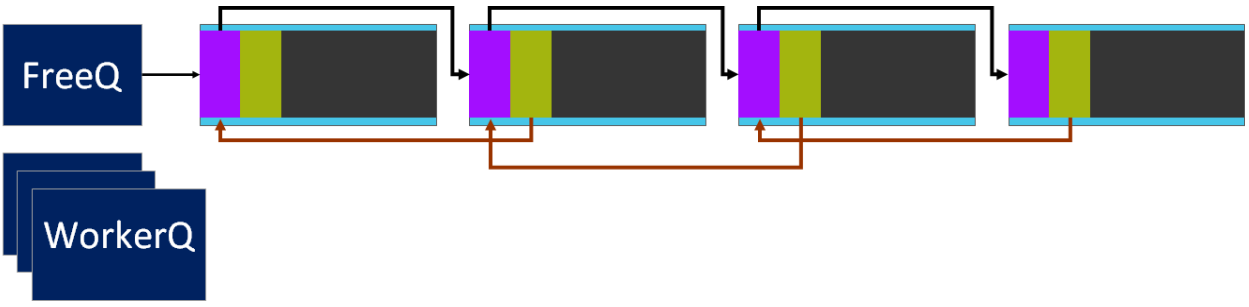
Most commands issued by the SCC go through this data flow path in the Nexus Control Panel:



An incoming message will involve multiple threads, making most of the threads part of a potential remote attack surface.

Inter thread queues

The inter-thread communication library in use is based on proprietary code that is most likely a relic of legacy code from pre-Linux Translogic systems. This library is based on queues which are doubly linked lists that are used to transfer messages between the various threads. The *FreeQ* is a queue of unused (free) messages. This queue is initialized with 59 fixed-size blocks, which are preallocated in the global section (.bss). Thus, this queue acts as a primitive heap implementation.



Overflowing one of the nodes in the queue means writing to the adjacent node's *next* and *prev* pointers since they are sequential in memory. Safe unlinking is not implemented, nor are any other heap overflow mitigations, making this structure highly exploitable when heap overflows are found. Furthermore, the payload of the nodes is not initialized between uses, which can allow an attacker simple heap manipulation and shaping that can be useful while exploiting various memory corruption bugs.

Discovered vulnerabilities

Vulnerabilities over Telnet

The device has an open Telnet server bound to the Ethernet interface. This server is not used in production and presumably facilitates diagnostics of the Nexus Control Panel, either at development stages and perhaps in production. Needless to say, use of Telnet in a modern application (and in critical infrastructure) is itself an insecure practice since the protocol is unencrypted and can allow attackers to obtain login credentials or alter commands issued to the device by establishing a man-in-the-middle position.

Hard-coded passwords (yeah, that old trick) - CVE-2021-37163

Unfortunately, the Telnet server on the device can be logged in using two users - *root* and *user*, which have hardcoded password hashes in the device's *shadow* file. Using common tools (such as *john the ripper*), attackers can easily brute force these hardcoded passwords, and these accounts can be used to log in to the devices over Telnet and abuse them for malicious intent.

Privilege escalation - CVE-2021-37167

While a device running on an old v2.6 Linux kernel is likely susceptible to a wide array of PE bugs, we were also able to find a very simple PE inherent to the boot sequence of the firmware.

The HMI3.jar file (the Java App) is designed to run with the credentials of *user*, and a bash script (at */home/user/hmi/run*) runs the Java app at boot. The bash script is owned by *user*, and can thus be edited by it. However, the *run* script is run as part of the boot sequence of the device by the *root* user. So simply by altering the content of the *run* script, a *user* can gain *root* privileges when the device reboots.

Design flaw

The Translogic central server (SCC) supports remote firmware updates of many of the PTS components it manages. The firmware are stored on the SCC, and are compatible with its version. The firmware files are not encrypted and not signed by Swisslog. In some PTS components, a certain **mode** needs to be set, via a physical switch, to allow the SCC to upgrade the firmware remotely, over the management connection (TLP20 over UDP, or other legacy protocols over serial connection):

Table 2.1 : S1 DIP Switch Modes

Mode	Function
Normal operation mode	Sets the station for normal operation. This is the default mode.
Reset ID mode	Clears all user-defined settings, including station ID, speed dials, audio level, and display contrast level. Resets the administration password to "1234" and enables the default user functions.
Download mode	Prepares the station to download program files when using the remote download kit or to download program files from the system control center.
Reset ID/download mode	Clears all user-defined settings and sets the station to download mode.

Various operation mode of the legacy IQ station, from it's manual

A firmware upgrade mechanism that does not require a secure cryptographic signature and/or encryption, can allow attackers to change the firmware en route, or on the SCC itself, which can lead to both remote-code-execution, but more importantly - persistence, on target devices.

Requiring a physical switch to initiate the upgrade offers some partial security (inadvertently) against remote attackers that wish to trigger the firmware upgrade process to compromise PTS components.

Unsecure firmware update - CVE-2021-37160

The Nexus Control Panel, however, does not have a physical switch that enables a *download mode*. It can be updated by the SCC over UDP using several download commands in the TLP20 protocol. The mechanism utilizes a command to initiate a firmware download, upload the firmware chunk-by-chunk over the UDP packets, and finally execute the new firmware. The process is as follows:

- The firmware is uploaded in chunks to */tmp/app_download*
- The md5 of /tmp/app_download is validated.
- The uploaded firmware is copied to the flash at */root/HMI3-new*
- When the system reboots, the loader script of the HMI3 program (*/root/run-ccp*) replaces the new firmware (*HMI3-new*) with the old firmware (*HMI3*), and the new firmware is then executed.

There is no cryptographic signature validation of any kind, and the firmware is not encrypted. An attacker can use this flaw to upload a malicious file and execute arbitrary code with root privileges. The lack of a secure boot mechanism can be used to alter the code persistently on the SD card, which will require manual re-imaging of the SD card to remove any malicious code. Since the TLP20 protocol over UDP is not encrypted, and does not use any authentication, attackers can abuse this design flaw to establish remote-code-execution with persistence without authentication.

Memory Corruption Vulnerabilities in TLP20

While the severity of previously described vulnerabilities is sufficiently critical, we wanted to investigate the security of the TLP20 protocol implementation itself. It's used both by the current Nexus Control Panel and older Translogic stations (such as the IQ station), which are still in use despite being end-of-life.

Underflow in udpRxThread - CVE-2021-37161

As described above, the *udpRxThread* thread is responsible for processing incoming UDP packets from the SCC over the TLP20 UDP protocol. Packets in the TLP20 protocol always start with a 20-byte header, which is processed and stripped by the *udpRxThread*. In the code snippet below, we see multiple underflow conditions that may occur when packets that are **shorter** than 20 bytes (the TLP20 header size) are received by this thread:

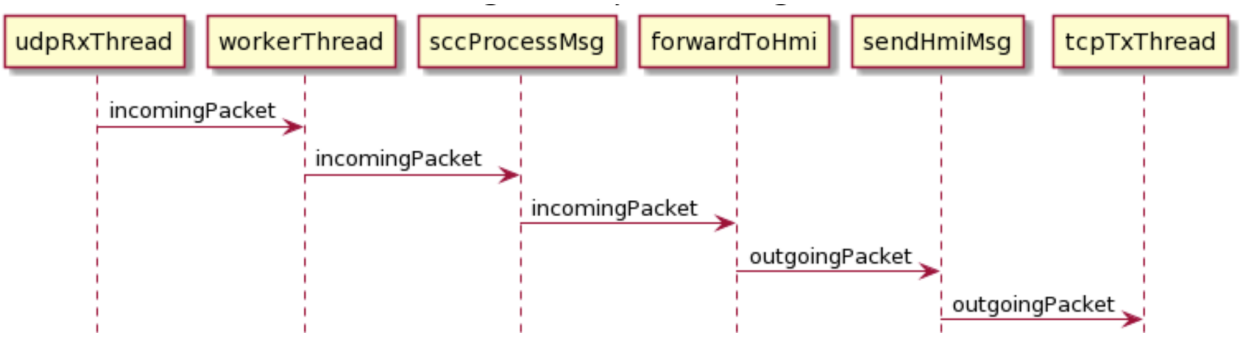
```
...
rec_len = recvfrom(udp_socket, buf, 370, 0, &addr, &addr_len);
op = *(_DWORD *)&buf[8]; // buf is the incoming UDP packet
// Keep alive
if (op == htonl(2u)) {
  ...
} else {
  ...
  // q_buf_1 is a node from the freeQ
  q_buf_1->data_len = rec_len - 20; // 16-bit underflow when rec_len < 20
  q_buf_1->should_process_using_hmi = 0;
  // 32-bit underflow when rec_len < 20
  memcpy(q_buf_1->data, &buf[20], rec_len - 20);
  ...
}
```

Decompiled code snippet from *udpRxThread*

If a short incoming packet that adheres to certain conditions (contain the required TLPU magic in the first 4 bytes, and **not** be a keep-alive command), the two underflow conditions highlighted above will occur. The incoming UDP packet is received to a stack buffer (*buf*) that is later copied to a message from the *freeQ* (the "heap"), and forwarded to the *workerThread* via a shared queue. The *data_len* field shown above is an unsigned 16-bit field, so when it underflows it will have a value of up to 0xFFFF. The second underflow is of *memcpy's* 32-bit unsigned size argument. In most cases, this type of underflow is likely to lead to an unexploitable denial-of-service vulnerability - since *memcpy* will attempt to copy MAX_UINT, which will lead, eventually, to overwriting the **entire** memory, or in other cases, to a page-fault that will halt execution.

However, in the case of the Nexus Control Panel, the call to *memcpy* doesn't lead to DoS, due to usage of an old version of **libc** by the HMI3 program, which has a bug in *memcpy* - CVE-2020-6096. This bug, *Signed comparison vulnerability in the ARMv7 memcpy*, leads a call to *memcpy* with a negative *size* argument to copy only a small number of bytes. Meaning, ultimately, that the only effect of the above underflow conditions is that the *data_len* field of the *q_buf_1* message is very large (up to 0xFFFF). Needless to say, this size doesn't represent the size of the incoming packet, and is far beyond the actual size of the *q_buf_1*.

When this message traverses through the various queues and threads within HMI3, it will ultimately be copied to another message from the *freeQ*, using the underflowed *data_len*. This will lead to an overflow of the "heap" of the inter-thread communication library. For example, when a certain short TLP20 packet is processed by the HMI3, it will eventually be forwarded to the Java App via the following flow:



In the *sendHmiMsg* function, the *incomingPacket* will be copied to an *outgoingPacket* and the "heap" corruption will occur. If an attacker is able to first shape the heap, prior to triggering the overflow, he may be able to control the payload of the overflow. Thus, this type of heap corruption can lead to remote-code-execution.

Overflow in sccProcessMsg - CVE-2021-37162

The vulnerability detailed above occurred when a packet shorter than 20 bytes was processed. The vulnerability shown below is triggered when an incoming packet of exactly 20 bytes is processed. After the TLP20 header is processed and stripped by *udpRxThread*, certain incoming packets will be processed by *sccProcessMsg*, and some of these will be forwarded to the Java App via the *sendHmiMsg* function. When this occurs, a new message (*q_buff*) is allocated from the *freeQ* and the incoming packet is copied to it starting at a 1 byte offset:

```
int __fastcall sccProcessMsg(q_buffer *a1)
{

  ...
  q_buffer *q_buff; // [sp+1Ch] [bp-8h]
  ...


  if ( a1->data[0]== 0x90 )
  {
    do
      q_buff = Q_remove_block((q_buffer *)&freeQ);
    while ( !q_buff );
    // if data_len is 0, it copies MAX_USHORT bytes
    q_buff->data_len = a1->data_len - 1;
    memcpy(q_buff->data, &a1->data[1], (unsigned __int16)q_buff->data_len);
    sendHmiMsg(q_buff);
    return 3;
  }
```

Decompiled code snippet from the *sccProcessMsg* function

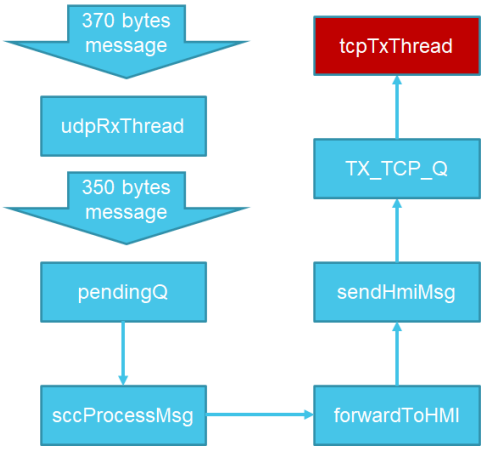
If an incoming packet of exactly 20 bytes is processed by *udpRxThread*, the *data_len* will equal zero when it reaches *sccProcessMsg*, and the new *data_len* (in *q_buff*) will underflow to 0xFFFF. In the code above, the underflowed *data_len* is then passed to *memcpy* leading to a similar overflow of the "heap", as shown in the previous vulnerability. Thus, this "heap" overflow can also lead to remote-code-execution (RCE).

Off-by-three stack overflow in tcpTxThread - CVE-2021-37164

In order to forward an incoming packet to the Java App, the *sendHmiMsg* function seen above will send the incoming packet to the *tcpTxThread*, where it will be sent through a TCP connection to the *HMI3.jar* Java App. When the largest possible incoming packet (370 bytes) is processed through the HMI3 program, it will create an off-by-3 condition in *tcpTxThread*, which will lead to a stack-overflow:

```
void __noreturn tcpTxThread()
{
  char buf[352]; // [sp+18h] [bp-17Ch]
  q_buffer *buffer_to_send; // [sp+178h] [bp-1Ch]
  ...
  while ( 1 )
  {
      ...
      // off-by-1 OOB write, since buf length 352, and buffer_to_send
      // can have 350 bytes written to &buf[3].
      memcpy(&buf[3], buffer_to_send->data,
             (unsigned __int16)buffer_to_send->data_len);
      // Writes two more bytes after the end of the buffer
      addCRC((int)&buf[1], buffer_to_send->data_len + 2);
      ...
    }
    Q_add_block(buffer_to_send, (q_buffer *)&freeQ);
  }
}
```

The incoming *buffer_to_send* message can be of a maximum size of 350 bytes, while it is copied at offset 3 to a buffer of size 352 bytes - meaning an off-by-one stack overflow will occur. Moreover, the *addCRC* function that follows that overflow, will add additional two bytes of CRC to the end of the buffer - meaning an off-by-three overflow will ultimately occur.



To get 350 bytes of payload to this function, an attacker would need to:

- Send an incoming UDP packet of length 370 which will be processed by *udpRxThread*.
- The *udpRxThread* will strip the first 20 bytes of the packet and send it to the *workerThread* via the *pendingQ* queue.
- The *workerThread* function calls *sccProcessMsg* which in turn calls *forwardToHMI*.
- *forwardToHMI* copies the buffer and calls *sendHmiMsg*.
- *sendHmiMsg* adds the buffer to the *TX_TCP_Q*
- The *tcpTxThread* pops the buffer from the *TX_TCP_Q* and parses it, triggering the overflow mentioned above.

This stack overflow shown above overwrites *buffer_to_send* - the pointer to the outgoing message that is being built. On one hand, this may make this overflow a bit tricky to exploit, since the *buffer_to_send* pointer is first overflowed with 1 byte (it's least-significant byte), and then the *addCRC* function will write the two CRC bytes by using the overflowed pointer. Nevertheless, as seen in the code snippet above, at the end of this flow, the malformed *buffer_to_send* pointer is then returned to the *freeQ*. So ultimately this stack overflow ultimately can also lead to a heap corruption, since a damaged pointer is being added back to the free list of the heap.

Despite these complexities both the stack overflow, and the resulting heap corruption can lead ultimately to remote-code-execution.

Inter-process socket hijack

As described above, the low-level application (HMI3) and the high-level application (HMI3.jar) communicate using a TCP socket.

GUI socket Denial Of Service (DoS) in tcpServerThread - CVE-2021-37166

At boot, the Java application connects to the low-level HMI3 process through the localhost on TCP port 9000. However, the HMI3 program mistakenly binds this port to all interfaces, and not only to the localhost interface -- exposing this socket to the network as well. If an attacker is able to reboot the device (for example, by using one of the vulnerabilities described above as denial-of-service, that leads to reboot), he may be able to connect to this TCP socket, from the network - hijacking the Java's connection to the HMI3 process:

```
while ( 1 )
{
  hmiCommStatus = 0;
  printLog(4u, 0, "Waiting for TCP connection...");
  c_socket = accept(fd, &addr, &addr_len);
  if ( c_socket < 0 )
  {
    perror("<1>accept()");
    v3 = _errno_location();
    printLog(8u, *v3, "<1>TCP accept().");
  }
  printLog(4u, 0, "CCP accepted TCP socket.");
```

Examining the log file of the boot sequence, we see that it takes ~30 seconds for the Java App to connect to the TCP socket, meaning that an attacker could easily win this race condition and hijack the socket connection when the device reboots. When this occurs the Java App will **not** be able to connect to the TCP socket, effectively disconnecting the GUI from controlling the station. This will also allow the attacker to command the low-level, as if he was physically there, controlling the GUI.

Underflow in hmiProcessMsg - CVE-2021-37165

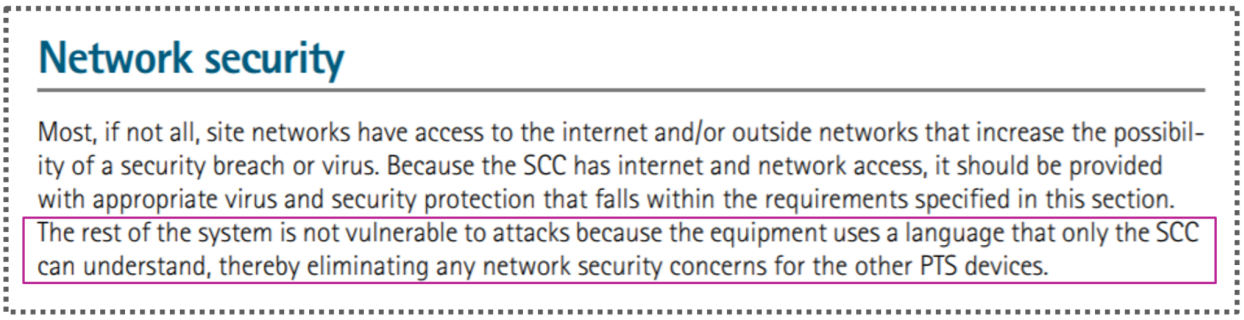
By hijacking the TCP socket mentioned above, an attacker can gain access to an additional attack surface available in the processing of packets received on the TCP connection between the Java App and the HMI3 process. When a message is sent to the HMI TCP socket (port 9000), it will ultimately be processed by the *hmiProcessMsg* function. The following code in *hmiProcessMsg* can experience an underflow condition similar to the one shown above in the *sccProcessMsg* function:

```
int __fastcall hmiProcessMsg(q_buffer *a1)
{
  q_buffer *v5; // [sp+14h] [bp-8h]
  ...
  If ( a1->data[0] == 0x33)
  {
    do
      v5 = Q_remove_block((q_buffer *)&freeQ);
    while ( !v5 );
    // Overflow when a1->data_len == 0, data len is an unsigned short
    v5->data_len = a1->data_len - 1;
    memcpy(v5->data, &a1->data[1], (unsigned __int16)v5->data_len);
    sendSccMsg(v5);
    return 3;
  }
  ...
}
```

If the received packet contains a *data_len* of size 0 - the outgoing *data_len*, of the packet that is to be forwarded to the SCC, will underflow to MAX_USHORT (0xFFFF), which will result in a overflow of the “heap” of the inter-thread library. As detailed above, this type of overflow can lead to remote-code-execution.

Final notes

The discovered vulnerabilities underline the lack of research that pneumatic tube systems, and specifically the Swisslog Translogic PTS solution, had received. This lack of research has led these systems to have more holes than Swiss cheese (pun very much intended). Part of this is a result of an era where security by obscurity was the gold standard. The following paragraph, taken from the Swisslog PTS Network Communication Deployment Guide, represents this era remarkably well:



Network security

Most, if not all, site networks have access to the internet and/or outside networks that increase the possibility of a security breach or virus. Because the SCC has internet and network access, it should be provided with appropriate virus and security protection that falls within the requirements specified in this section. The rest of the system is not vulnerable to attacks because the equipment uses a language that only the SCC can understand, thereby eliminating any network security concerns for the other PTS devices.

While the transition of analog systems to digital brings progress to all sorts of applications, including to a variety of infrastructure solutions used in healthcare, it is important to transition the security mindset of such systems in the process as well. When critical infrastructures, such as pneumatic tube systems that play a crucial role in providing patient care, are in mind, this requirement needs to be even more imperative.

About Armis

Armis is the leading unified asset visibility and security platform designed to address the new threat landscape that connected devices create. Fortune 1000 companies trust our real-time and continuous protection to see with full context all managed, unmanaged, and IoT devices, including medical devices (IoMT), operational technology (OT) and industrial control systems (ICS). Armis provides passive and unparalleled cybersecurity asset management, risk management, and automated enforcement. Armis is a privately held company and headquartered in Palo Alto, California.

armis.com

1.888.452.4011

20210801-3