

SWIFT Wireless Fire Alarm Pull Station Analysis

Donald Lawrence
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
dl@gatech.edu

George Kokinda
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
gkokinda3@gatech.edu

Garrett Brown
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
gbrown94@gatech.edu

Jaewon Jeung
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
jjeung6@gatech.edu

Chris M. Roberts
Advisor
Georgia Institute of Technology
Atlanta, Georgia, United States
chris.roberts@gatech.edu

Abstract—Building management and fire protection systems are relatively under-researched areas in the field of cybersecurity. A malicious actor could attack one of these systems in order to cause interruption of functionality, false alarms, and other dangers to the inhabitants of the building. Honeywell is a manufacturer of these building management systems, and this study investigates their Fire Alarm Pull Station and any vulnerabilities involving it and the overarching SWIFT system. This is accomplished through analyzing the protocol SWIFT devices use to communicate, and by finding exploits that a bad actor would want to take advantage of.

I. BACKGROUND

Fire protection systems are essential components of life-safety systems, which alert, protect, and assist evacuation of building populations in case of emergencies. A malicious actor threatening the occupants of the building may tamper with the safety systems. Some of the potential malicious activities that may occur are the following: manipulating logged temperature and triggering a false alarm, sending misinformation to the centralized monitoring system to distract a building security operator, effectively disarming the system without alarming the controller, and extracting a private key through a cryptography attack in order to gain privileged access to the system. Therefore, assessing the attack surface of the system and patching the vulnerabilities would potentially significantly disrupt the malicious actors' planned cyber kill chain.

A. The SWIFT System

Buildings require a centralized management system to operate and monitor numerous sensors and supporting devices. With the increase of a diverse usage of buildings such as datacenters, a Building Management System (BMS) that has automation capabilities to monitor various building systems became mission critical. Historically, management systems and their accompanying devices had to be wired together like any other network. However, over the past decade, there has been a rise in popularity of wireless management systems, allowing building owners to retrofit older buildings with newer systems, or to augment an existing wired system. Honeywell is

a company that produces building support technologies, and a part of their product suite is called “SWIFT” (Smart Wireless Integrated Fire Technology) [1]. The product line consists of a wireless smoke detector and fire alarm pull station, along with several devices designed to create an interface with other wired Honeywell products, such as their control panels or sirens [2].

B. Downsides of a Wireless System

The usage of a wireless system comes with drawbacks that are inherent in any network of wireless devices. Reliability is a key factor in a building management system, yet without specific design decisions, a wireless network tends to be less reliable than its wired counterpart. The SWIFT system operates through what is known as a “mesh network”, which is a decentralized method of connecting multiple devices over a large distance. In order to mitigate any issues with reliability, each device connects with multiple other devices in the network, providing multiple paths from a given device to the primary control panel (Figure 1). The system is also able to operate over multiple wireless frequencies, with the capability to dynamically switch between them as the surrounding environment changes [1]. Honeywell also provides software that allows for the control and management of SWIFT devices in a network, aptly called “SWIFT Tool”. With it, a building maintainer can view the status (battery level, connection strength, etc.) of various devices in the network, along with viewing and modifying the layout of the mesh itself.

II. SWIFT ADDRESSING METHOD

Placing devices throughout the entire building must be done strategically especially since identifying the location of the emergency is crucial. Multiple categories of fire alarm systems are created to address the issue. Fire alarm systems are categorized into four overlapping types: conventional, addressable, intelligent, and wireless [3]. Conventional systems set “zones” through devices connected by non-looping wires as a means to identify the location. In contrast, an addressable

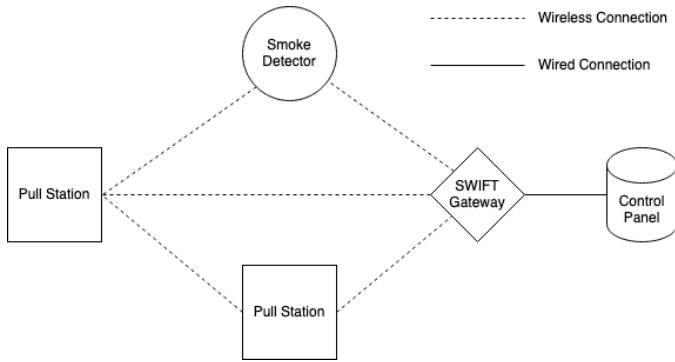


Fig. 1. The decentralized nature of the system ensures that as little disruption as possible occurs if one or more devices fail.

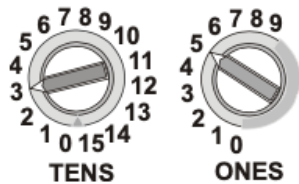


Fig. 2. Rotary switches used to set the address of SWIFT devices.

type system connects devices in a loop and assigns each device an “address” to identify them. Wireless systems incorporate the addressable system to enhance the flexibility of device placements. Therefore, the SWIFT system employs a wireless system and enforces specific addressing constraints.

A. Addressing Constraints

While the SWIFT addressable wireless system can support up to forty-nine devices with one gateway and allows multiple gateways, the supported address range is 1 to 159 [4]. The address is set by concatenating the resulting numbers of two rotary switches [5] (Figure 2). Therefore, to represent address 159, the “tens” switch will set to 15, and the “ones” switch will have the value of 9. The decimal values are converted to binary using the same concatenation logic—15 is 0b1111 and 9 is 0b1001, so 159 is 0b00011111001. The input address of 160 represented in binary is 0b000100000000, in which the address loops back to 0. This implies the number of bits allocated for addressing is eight. The limitation of the range of the address proves to be useful when analyzing the data the wireless devices emit since it decreases the scope of protocol analysis.

III. OTA PROTOCOL

The team has chosen to focus on the SWIFT Fire Alarm Pull Station in terms of analysis since it’s an easily accessible target for bad actors to attack. The hardware is relatively cheap to obtain and easy to maintain, and the SWIFT environment only requires one device to be able to communicate with SWIFT Tools.

Much of the team’s protocol analysis has been done using Universal Radio Hacker (URH). The goal of using URH is

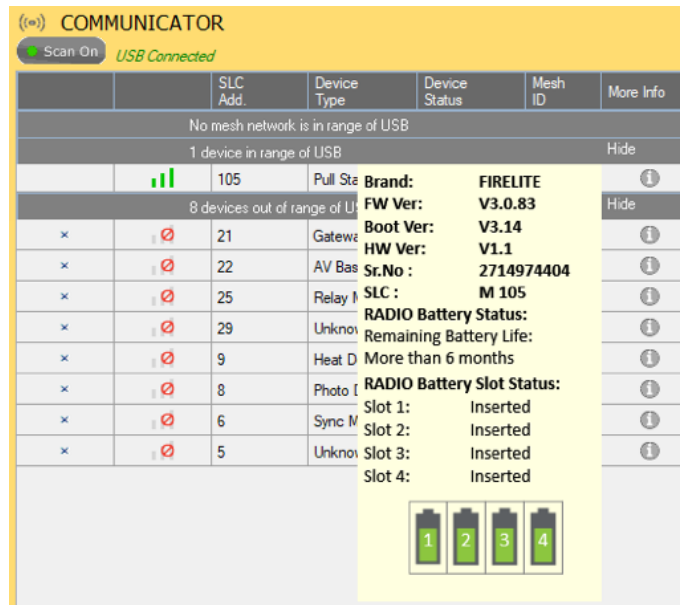


Fig. 3. The pull station’s information inside SWIFT Tools.

to capture and analyze some of the RF (Radio Frequency) signals being emitted from the pull station. The team found that a signal is broadcast over-the-air (OTA) from the pull station every 8 seconds. This signal contains a packet which holds data regarding the pull station. The data encoded by the OTA message holds the information seen inside SWIFT Tools (Figure 3). This includes data such as the address of the pull station, the serial number, battery information, and more. Due to its OTA nature, the protocol used to encode this data is referred to as the OTA protocol (or radio protocol).

A. Capturing Method

In order to capture the RF transmissions containing all of this data, the team has a software-defined radio (SDR) placed in the same room as the pull station. A SDR is a “radio communication system which uses software for the modulation and demodulation of radio signals” [6]. The specific SDR in use is a HackRF One which gives a user the ability to analyze signals on the radio spectrum. The pull station communicates with other devices in the mesh network via RF signals through its RF transceiver the SemTech SX1231. According to the SX1231 documentation provided by SemTech, their transceiver usually broadcast (in North America) over a frequency range of 902-928 MHz which is the ISM (industrial, scientific, and medical) radio band for North America [7]. The HackRF is capable of receiving and transmitting on a wide frequency range, but it was found that the pull station communicates at a frequency of 913 MHz (megahertz) which is accessible with the HackRF. Once the frequency is set to 913 MHz inside of URH, the HackRF captured the RF transmissions originating from the pull station. The SemTech SX1231 uses FSK (frequency-shift keying) modulation, and URH allows for this modulation scheme. A samples per second (sps) or sample rate of 35 is used to match the

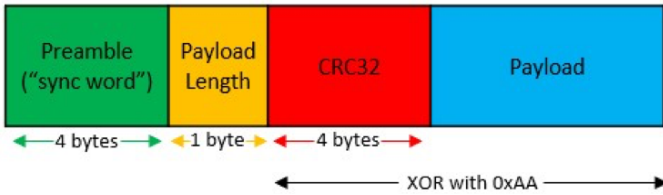


Fig. 4. Visual representation of the OTA message structure.

wavelength of the RF transmissions and ensure the signal is interpreted correctly (i.e., prevents aliasing). By demodulating the captured OTA traffic, a successful analysis of the signals has been produced using the tool set provided within URH.

B. Field Analysis

Looking at the raw binary data of the demodulated signals, the start of the OTA message is indicated by a binary two (0010). The sequence of 1's and 0's before the binary two and the sequence of 1's following the OTA message are both considered garbage data. That specific binary is discarded by cropping to the selection of data to be analyzed (i.e., the OTA message). By converting the newly cropped signals to hex, it can be seen that the OTA message indeed starts with a value of two because the sequence "21436587" is the preamble or "sync word" of the message. The general structure of the OTA protocol is visually represented in Figure 4.

Following the preamble and the payload length, all fields are exclusive-or'ed (XORed) with 0xAA which acts as a form data whitening per the SX1231 documentation [7]. The payload holds the data which populates SWIFT Tools as shown above in Figure 3. After the XOR operation is performed on the payload, some of the hex data can be deciphered by converting hexadecimal values to decimal values which are present inside the SWIFT Tools interface. For example, when the address of the pull station is set to 105 on the bottom row in Figure 5, the hex at nibble 35 and 36 show the address to be "0xC3", but after XORing this value with 0xAA, a value of "0x69" is found, and upon converting "0x69" from hexadecimal to decimal, the address of the pull station "105" is found (can be seen as SLC in Figure 3). A similar process is performed on the values of the other identified fields of the payload (seen in Figure 5). One exception to this is the serial number of the pull station. In between the XOR operation and converting to decimal, a change of endianness must also be performed on the serial number to find its value. Figure 5 highlights all the fields labeled inside of the analysis tab of URH that were found by executing the operations previously described.

C. OTA Protocol Breakdown

The remaining unidentified fields were decoded through a comparison of the OTA (radio) and USB protocols. The USB protocol will be examined thoroughly later in the paper. As it pertains to the OTA protocol, one of the RF transmissions collected from the pull station is the following:

Name	Color
<input checked="" type="checkbox"/> preamble	Yellow
<input checked="" type="checkbox"/> Payload Length	Red
<input checked="" type="checkbox"/> CRC	Green
<input type="checkbox"/> Unidentified	
<input checked="" type="checkbox"/> Serial Number	Cyan
<input type="checkbox"/> Unidentified	
<input checked="" type="checkbox"/> Address of Pull Station	Brown
<input type="checkbox"/> Unidentified	
<input checked="" type="checkbox"/> Sync Word	Blue
<input type="checkbox"/> Unidentified	
<input checked="" type="checkbox"/> Battery Position	Purple
<input type="checkbox"/> Unidentified	

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18				
2	1	4	3	6	5	8	7	2	1	c	d	4	a	c	3	4	5				
2	1	4	3	6	5	8	7	2	1	9	e	c	4	1	2	e	d				
19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36				
c	5	b	7	6	e	9	2	7	9	0	b	d	1	9	3	c	e				
c	5	b	7	6	e	9	2	7	9	0	b	d	1	9	3	c	3				
37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54				
a	a	a	3	b	2	5	5	a	b	8	a	7	f	b	d	f	5				
a	a	a	3	b	2	5	5	a	b	8	a	7	f	b	d	f	5				
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76
2	6	5	8	f	9	a	4	a	a	a	a	a	a	a	a	a	a	a	a	a	a
2	6	5	8	f	9	a	4	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Fig. 5. OTA message analysis of pull station for address 100 (top row) and address 105 (bottom row) inside URH.

Raw Hex Data From OTA Signals	
	Color
Preamble	Green
Payload Length	Yellow
CRC32 (part of payload)	Red
Payload	Blue
bit #	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
hex data	21436587 21 8 a c 1 4 1 2 2 c5b76e92790bd193c3aaa3b255ab8a7fbd52658f9a4aaaaaaaaaaaa

Fig. 6. General OTA message structure (with hex data from RF transmission).

```

21 43 65 87 21 9e c4 12 ed c5 b7
6e 92 79 0b d1 93 c3 aa a3 b2 55
ab 8a 7f bd f5 26 58 f9 a4 aa aa
aa aa aa aa aa

```

This hex data from the OTA message can be initially broken up into four primary parts: the preamble, the payload length, the CRC32 (Cyclic Redundancy Check), and the payload. The preamble is used to indicate the start of an OTA message. It acts as a "get ready" notification of sorts for all other devices that intend to receive the RF transmission. This establishes a synchronization of bits between two or more devices. The payload length indicates the size of the payload that follows. Despite being labeled its own field, the CRC32 is considered a part of (and thus the start of) the payload. It will be further defined later. All of this is visualized using the previous RF transmission in Figure 6.

The payload can be further broken down to reveal the information seen inside the SWIFT Tools application. In order

Payload After XOR 0xAA								
	CRC32	Unknown	Serial Number	Node Type	Bootloader Version and Node State	SLC Address	bScanResultPresent	Hardware (HW) Version
byte #	1 2 3 4	5 6	7 8 9 10	11	12	13	14	15
hex data	34 9e b8 47	0f 1d	c4 38 d3 a1	7b	39	69	00	09

Payload After XOR 0xAA							
	Software Release	Site Survey Address	Mesh ID	Sync Word	Fire Panel Brand	Batteries Inserted and Battery Status	Application Build Number
byte #	16	17	18	19 20 21 22	23	24	25
hex data	18	ff	01	20 d5 17 5f	8c	f2	53

Payload After XOR 0xAA								
	Bootloader Build Number	Link Test Result	Device State	Unknown	Unknown	Unknown	RF Scan Progress	Unknown
byte #	26	27	28	29	30	31	32	33
hex data	0e	00	00	00	00	00	00	00

Fig. 7. Full breakdown of payload from OTA message (emitted from pull station).

to more deeply identify the exact fields of the payload, every byte of hex data following the preamble and the payload length (i.e., the entire payload) must be exclusive-or'ed (XORed) with 0xAA as previously discussed. Following the exclusive-or operation, the hex data now appears as such:

```
21 43 65 87 21 34 6e b8 47 6f 1d
c4 38 d3 a1 7b 39 69 00 09 18 ff
01 20 d5 17 5f 8c f2 53 0e 00 00
00 00 00 00 00
```

The data of the payload starting at the first CRC byte 0x34 (which was 0x9e before) until the last byte 0x00 (which was 0xaa before) has undergone this operation. After the XOR operation, it's possible to further dissect the payload for the data used to populate the SWIFT Tools application. One last thing to acknowledge before decoding the payload is that the payload length is 0x21 which is equal to 33 in decimal. As such, the payload should contain 33 bytes of data. Figure 7 illustrates the breakdown and identification of all 33 bytes of data within the payload.

The first field of the payload is the CRC32 which makes up bytes 1 through 4. The CRC32 is used as a means of detecting data corruption or accidental changes to the OTA data. Byte 5 is believed to be the Message Type of the OTA message, but it remains labeled as "Unknown" as further analysis will be required to prove this. Currently, byte 6 also remains unknown. Following bytes 1 through 6, bytes 7 through 33 are identical to the data captured in the payload of the serial message. To avoid repetition, the remaining bytes will be analyzed in the USB Protocol section of the paper. Further, the differences between the two protocols will be properly examined.

IV. USB PROTOCOL

The wireless USB tool, referred to as the "W-USB" transceiver, is used as an interface between a PC and the wireless devices from Honeywell. By plugging in the W-USB into a computer, the wireless USB dongle can then communicate information about site survey data from all the devices in the mesh network by sending and receiving RF signals. The W-USB can retrieve data of devices within 20

feet of where the W-USB is connected to a PC. The OTA data that the W-USB collects is used to populate the SWIFT Tools software which is then stored into a database containing information about the mesh network and the various devices that make up the network.

A. Capturing Method

Since the W-USB can send and receive information to and from the wireless devices within the mesh network, the traffic can be captured and analyzed. Rather than collecting the OTA data, the W-USB offers the unique opportunity of analyzing the traffic over a serial connection. Using a software tool called Serial Port Monitor to analyze the COM 3 bus traffic between the W-USB and the SWIFT Tools application, it was discovered that the serial traffic is similar to the messages that were emitted over-the-air. The differences stem from the W-USB taking in the OTA data and slightly altering its format in a manner that's suitable for the SWIFT Tools software to parse. This differing set of rules that governs the communication between the W-USB and SWIFT Tools is regarded to as the USB Protocol.

B. Reverse Engineering SWIFT Tools



In order to decode the serial traffic (i.e., the USB Protocol) captured using Serial Port Monitor, a reasonable approach would be to analyze how SWIFT Tools is collecting the serial data and transforming it into readable data inside its interface. To that effect, reverse engineering SWIFT Tools was the logical step to take. The SWIFT Tools application is distributed as an unobfuscated .NET application and supporting dynamic-link libraries (DLLs). This makes the application easier to reverse engineer since all the symbols (imports, global variables, functions, etc.) are present. Through the utilization of an open-source .NET decompiler called ILSpy, a decompilation of the source code for the SWIFT Tools software was produced. The DLLs were decompiled to C which allowed the team to dig into exactly how SWIFT Tools is operating.

Much of the analysis of the decompilation took place in three dynamic-link library files. These files are in the installation directory of Swift Tools and are named: WirelessComm.dll, WirelessInterfaces.dll, and WirelessPlugin.dll. Each dll file contains code related to parsing and creating wireless USB messages. The team uncovered the general structure of all the W-USB packets. The message structure consists of two different packet frames an adapter frame and a node frame. This data is found in "Honeywell.WirelessTool.WirelessComm.ProtocolManager". The adapter frame is for controlling the USB adapter itself while the node frame is for sending and receiving information from devices within the mesh network. Regardless of the packet frame type, every message contains an open delimiter (7B = ''), message type (1 byte), payload length (1 byte), payload (variable length), CRC (XOR of the previous bytes), and a closing delimiter (7D = ''). The node frame has two extra fields namely the device type (1 byte) and device serial

USB Opening message				
bit #	Open Delimiter	Message Type	Payload Length	Device Level RSSI
1	0x7B	0x04	0x1C	0x27
2	0x7B	0x04	0x1C	0x41
3	0x7B	0x04	0xED	0x2A

Fig. 8. Node frame encoding start to USB messages.

Brand: FIRELITE	Brand: NOTIFIER
FW Ver: V3.0.83	FW Ver: V4.1.1
Boot Ver: V3.14	Boot Ver: V3.9
HW Ver: V1.1	HW Ver: V1.0
Sr.No : 2714974404	Sr.No : 3826548318
SLC : M 117	SLC : M 25
RADIO Battery Status: Remaining Battery Life: More than 6 months	RADIO Battery Status: Remaining Battery Life: 2 Years
RADIO Battery Slot Status: Slot 1: Inserted Slot 2: Inserted Slot 3: Inserted Slot 4: Inserted	RADIO Battery Slot Status: Slot 1: Inserted Slot 2: Inserted Slot 3: Inserted Slot 4: Inserted

Connection Type: Mesh	Login Time Remaining: UnAuthenticated
Brand: NOTIFIER	Profile Distribution: No
RF FW Ver: V3.0.83	Device Count Exceeded: No
RF Boot Ver: V3.6	Wireless Enabled Mode: WEP for SWIFT 2.0 panels
SLC FW Ver: V3.0.25	
SLC Boot Ver: V2.1	
HW Ver: V1.0	
Sr.No : 1398014036	
SLC : M 21	

Fig. 9. Pull station (left), gateway (bottom), and relay station (right) information in Swift Tools diagnostics at the time of message capture.

number (4 bytes). The team at this time has decoded several node frame packet types relating to various devices within the Honeywell SWIFT ecosystem.

C. Standard Devices

Using ILSpy and investigating the class "Honeywell.WirelessTool.WirelessPlugin.ScanForm" it was discovered that two methods in "ScanForm" namely "ScanForm.fillDevcieScanData" and "ScanForm.fillScanDataforDevices" were responsible for parsing the payload for a majority of the devices. By following along with the decompiled code, it is possible to identify all fields of the pull station, relay station, and AVBase payloads.

Bytes 0 through 3 of the payload consists of the device's serial number. This is taken and converted to decimal. This can be seen within the diagnostics section of the SWIFT Tools software. Each device has a serial number associated with the hardware. Byte 4 is the node type. This is an enum class which converts the value to decimal and selects the appropriate type. The node types consist of various device name such as but not limited to pull, gateway, relay module,

USB Message (Pull Station)											
Serial Number	Node Type	Bootloader/Version and Node State	SLC Address	IScanResultPresent	Hardware (HW) Version	Software Release	Site Survey Address	Mesh ID	Sync Word		
byte # 0 1 2 3	4	5	6	7	8	9	10	11	12 13 14 15		
hex data 7b 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f											

USB Message (Pull Station)											
Fire Panel Brand	Batteries Inserted and Battery Status	Application Build Number	Bootloader Build Number	Link Test Result	Device State	AVBase Only	AVBase Only	AVBase Only	RF Scan Progress	Unknown	
byte # 16	17	18	19	20	21	22	23	24	25	26	
hex data 0e 0f 10 11 12 13 14 15 16 17 18 19											

Fig. 10. Decoded and labeled pull station USB message.

etc. This value sets the device type which later is used for parsing the rest of the payload. Byte 5 consists of two different values which are broken up into their respective nibbles, boot loader version and state. Boot loader version is found in the diagnostics information. The exact specifics of state are unknown, however it assigned to an enum class called NodeState. The pull station's state at the time of capture for example was PROFILEASSIGNEDTAMPER. Byte 6 is the SLC address of the device. This can also be seen in diagnostics. Byte 7 is a Boolean value that checks if a device scan result is present. It is set to false if the value is 0 and true otherwise. Byte 8 is the hardware version and byte 9 is the software version. These both can be seen in diagnostics. The way the information is extracted is it takes the byte and converts it into 8 bits. Then it places a decimal after the second least significant bit (zero-indexed) for a total of 5 bits to the left of the decimal and 3 bits on the right. It then converts both sides to decimal. Byte 10 is the site survey address. At the time of capture all devices with the same payload structure had this value set to "NA". This value is set because the value within the payload is 0xFF. Byte 11 is the Mesh ID. This is a simple conversion of the payload hex value to decimal. Bytes 12 through 15 make up the device's sync word. This value is used in other functions when populating the mesh network. The exact specifics are unknown at the time. Byte 16 is the fire panel brand. This value is converted to decimal and compared to hardcoded decimal values corresponding to the different brands in the Honeywell fire alarm ecosystem. The brands captured were FIRELITE corresponding to the pull station and NOTIFIER corresponding to the gateway and relay station. Byte 17 makes up two different battery values. The most significant nibble pertains to battery inserted booleans. The devices can support up to four batteries and each bit is a boolean corresponding to if a battery is inserted into a slot or not. If a battery is inserted the bit becomes a 1 and if not, the bit is a 0. An example of this is the pull station at the time of serial capture. The pull station had all four batteries connected and so the value is 0xF. The next nibble is the battery life status. This value is converted into decimal and matched with an enum class that converts the decimal into a message that can be seen in Swift Tools Diagnostics. The pull station, for example, at the time of capture had a battery life of "More than 6 months." This message, in the enum class, corresponds to the decimal value 2. Bytes 18 and 19 are the application build number and the bootloader build number.

USB Message (Gateway)					
	Equivalent to other Devices	List Capacity and number of devices	Mesh SLC Addresses	Serial Numbers	Bit Vector 1
byte #	0 - 17	18	19 - 63	64 - 227	228
hex data	54 00 54 53 ... 82	7b	08 09 05 06 FF FF FF FF	2D F1 DF 8E ... 00 ... 00	36
USB Message (Gateway)					
	Bit Vector 2	Lock Time Remaining	RF Application Build Number	SLC Application Build Number	Equivalent to other Devices
byte #	229	230	231	232	233-236
hex data	B4	75	53	19	06 01 AD

Fig. 11. Decoded and labeled gateway USB message.

These values are directly converted to decimal and are seen the device information within diagnostics. The application build number is seen appended to the software release number, and the bootloader build number is seen appended to the boot version. Byte 20 is the link test result, and byte 21 is the device state. The team at this time has not uncovered what these two bytes represent. Bytes 22 through 24 are utilized by the AVBase only. These values are skipped (and thus arbitrarily set to 0x00) in the payload for all devices except the AVBase. For the AV Base, byte 22 and byte 23 remain 0x00 in the serial message which produces the message "No AV Device on Base" seen in the "More Info" portion of the AV Base inside Swift Tools. Further, byte 24 is set to 0xF5 inside the serial message for the AV Base. This data is then converted to binary to "11110101" and then divided up by its first five bits and last three bits to "11110.101". The first five bits produced the message "More than 8 Hours of Alarm Time Remaining" seen inside SWIFT Tools from an enum class. The last three bits produced the message "Good Batteries" seen inside SWIFT Tools from an enum class. Byte 25 is a value for RF scan progress. The exact specifics are unknown currently. Byte 26 is an unknown value. The team's captured messages all have skipped this value. The final byte 27 is the is the CRC. The CRC is formed by performing an XOR every byte after the start delimiter with every other byte before the CRC's position in the payload.

D. Gateway

The gateway's USB message as immediately seen it is much longer than the Pull Station or the Relay Station. The length of the payload is found at byte (3) of the message 0xED = 237 this is significantly longer than the payload sizes of the other devices which had a length of 28. The payload begins at byte (5) of the message like the other devices. Also like the other devices, it was discovered within the class "Honeywell.WirelessTool.WirelessPlugin.ScanForm" using the same method "ScanForm.fillDevicScanData." However, within Swift Tools, the gateway message is parsed with its own separate method, "ScanForm.fillScanDataforGateway."

As seen in figure 11, many of the gateway payload's fields are identical to the other device's payloads. This is because the gateway contains many of the same information such as serial number, device type, software version number, etc. as seen in Swift Tools diagnostics. At bytes 17 through 230, Swift Tools parses these bytes as gateway only attributes. Using ILSpy

and following the code within the method "fillScanDataforGateway" the gateway message can be deciphered.

Byte 17 is a "GatewayMeshAttribute" list capacity. This value creates several lists for different values at the capacity internal to Swift Tools. This value will be referenced later in parsing the payload because the value doubles as a variable representing the number of devices under the gateway which can be verified in diagnostics. Bytes 18 through 21 is the mesh SLC address. Converting this to decimal reveals a list of mesh SLC addresses, each one byte in length, from the devices connected to the gateway. At the time of capture there was four mesh SLC addresses. Bytes 22 through 63 are filled with a placeholder value 0xFF which denotes the ability for more devices connected to the gateway. These values are for more mesh SLC addresses if the gateway had a larger number of devices connected to it. Bytes 64 to 76 are a list of serial numbers also pertaining to the devices connected to the gateway. There are four in total and each are 4 bytes long. The serial numbers, when converted to decimal, are not equivalent to the ones found in the Swift Tools diagnostics. The team does not know why this is. Like the mesh SLC addresses, bytes 77 through 227 are placeholder values for more device serial numbers. The next couple bytes are bit vectors. These bytes are converted into binary and each bit or consecutive bits determine a value. Byte 228 is a bit vector that when converted to binary, bits 0 through 5 are boolean values. Starting from bit zero to five the boolean values are "isInterference," "isMaxGatewayTrouble," bits 3 and 4 are skipped, "bIsTrafficOn," and "bIsWeakLinkTroubleOn." The exact details of these values are unknown. Bits 6 and 7 is magnet lock status. These two bits are converted into decimal then set using an enum class for the magnet lock status. The lock status of the gateway at the time of capture is "MagnetVerified." This is apparent when you enter a password for the gateway in diagnostics. If you enter a correct password a magnet must physically be placed on the gateway in order to gain access to gateway information. Continuing, byte 229 is also another bit vector. Bits 0 through 2 correspond to an "M2MSyncStateMode" enum class. The values this class represents are not known at this time. Bit 3 corresponds "bIsSyncTroubleReportingOn." Bits 4 and 5 make up the value for the gateway's wireless enabled mode enum class. This value is seen in diagnostics as "WEP for SWIFT 2.0 panels." Bit 6 is a boolean pertaining to device count being exceeded. Bit 7 is the value "isProfileDistribution." Byte 230 is the lock time remaining. this is not converted into decimal and the string " minute(s)" is appended to it. Since the magnet lock status is "MagnetVerified," It checks a locking tracker list for a serial number that equals the gateways. It sets "UnAuthenticated" if it does not find one. This is shown as Login Time Remaining in the UI. Bytes 231 and 232 make up the RF application build number and SLC application build number, respectively. There are seen appended to their respective "FW Ver" in diagnostics. The next bytes 233 to 236 are equivalent to what was previously shown in the previous devices. These bytes make up the bootloader build numbers

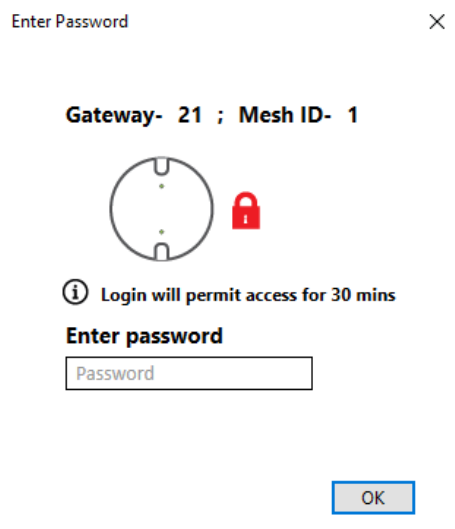


Fig. 12. Window asking for a gateway password in Swift Tools diagnostics.

and CRC in the same locations as the pull, relay, and AVBase messages.

E. Other Messages

The message type makes up the second byte of every serial message. The message types discussed for the devices are of the value 0x04 which gets translated, within the “MessageType” enum class, to “BackGroundScanResponse.” These messages are transmitted frequently and periodically through the COM 3 bus. This message generally contains data about the device information. Other message types have been found by the team.

One of these messages is “VerifyPasswordRequest.” This message is sent when one enters a password for the gateway in diagnostics see figure 12. This message follows the node frame packet structure and has been deconstructed by the team. Byte 0 contains the open delimiter. Byte 1 is the message type “VerifyPasswordRequest.” Bytes 2 and bytes 3 through 6 is the node type which is the gateway and the serial number for the gateway, respectively. Byte 7 is the payload length which is converted into decimal. Byte 8 is a constant value 0x20. This field is unknown however it does not show when a user inputs the maximum number of characters which is eight for the password. The next field is variable in length as matches the inputted password length. The password can be viewed with a simple conversion from hex to ASCII. The password is sent through the COM 3 bus in clear text. Then like the other messages you have a CRC and closing delimiter bytes.

Other messages have been captured by the team. These messages include “VerifyPasswordReply” and “NetworkTopologyStatus.” The “VerifyPasswordReply” message can be captured when a user enters a correct password, and the magnet is placed on the gateway. The message “NetworkTopologyStatus” is captured when the user successfully authenticates by entering the correct password to the gateway and views the mesh network shown in SWIFT Tools diagnostics. These

```
// Honeywell.WirelessTool.WirelessInterfaces.MessageType
public enum MessageType
{
    Error = 0,
    Ack = 1,
    Nack = 2,
    StartBackgroundScanOnly = 3,
    BackGroundScanResponse = 4,
    StopLiveEventsDataStreamWithBackgroundScan = 5,
    StopLiveEvents = 6,
    StopBackGroundScan = 7,
    AdapterCommandResponse = 8,
    AdapterErase = 9,
    AdapterEraseResponse = 10,
    AdapterDownload = 11,
    AdapterDownloadResponse = 12,
    AdapterLaunchRequest = 13,
    AdapterLaunchResponse = 14,
    PingAdapter = 19,
    PingAdapterReply = 20,
    VarifyPasswordRequest = 21,
    VarifyPasswordReply = 22,
    PasswordResetRequest = 23.
}
```

Fig. 13. Snippet of MessageType enum in ILSpy.

messages have yet to be analyzed and further research is needed. There are also many more message types that can be captured and analyzed. This just scratches the surface as there are many more enums in “MessageType.”

F. USB and OTA Protocol Comparison

The USB and OTA protocols are nearly identical in terms of payload. This similarity led to the decoding of the OTA protocol by simply comparing the hex data of the serial and OTA messages. A side-by-side comparison after the XOR’ing of the OTA data can be seen in Figure 14. The differences between the two protocols are highlighted in red, and the similarities are highlighted in yellow. Further, the data highlighted in blue (for USB) and green (for OTA) illustrates the differences within the same protocol when the pull station is set at different addresses (addresses highlighted in pink). The payload structure of the messages is the same starting at the 0xC4 byte all the way to the last 0x00 byte of data (i.e., 26 bytes are identical). This encompasses nearly all the fields within the OTA message’s payload that have been identified. This means that the W-USB sends most of the payload over serial (to populate SWIFT Tools) in the exact same format in which the devices on the mesh network send RF signals (OTA messages) to the other devices on the network (including the W-USB). The differences between the two protocols can really be boiled down to three things. One difference is the USB protocol’s use of a start delimiter instead of the preamble which the OTA protocol utilizes. Both the start delimiter and preamble serve the same purpose which is to indicate the start of a message (allows for the synchronization of bits). They just come in different formats. The second difference is the size and location of the CRC. The USB protocols includes a one-byte CRC at the end of the payload, while the OTA protocol includes a four-byte CRC at the start of the payload. Once again, they both serve the same purpose of ensuring the

After XOR 0xAA - Address #105 (0x69)	
USB Traffic (COM3 serial port)	OTA Traffic (RF transmissions)
7b 04 1c 29	21 43 65 87 21 34 6e b8 47 6f 1d
c4 38 d3 a1 7b 39 69 00 09 18 ff	c4 38 d3 a1 7b 39 69 00 09 18 ff
01 20 d5 17 5f 8c f2 53 0e 00 00	01 20 d5 17 5f 8c f2 53 0e 00 00
00 00 00 00 00 e5 7d ed ed ed	00 00 00 00 00
After XOR 0xAA - Address #100 (0x64)	
7b 04 1c 2b	21 43 65 87 21 67 e0 69 ef 6f 1d
c4 38 d3 a1 7b 39 64 00 09 18 ff	c4 38 d3 a1 7b 39 64 00 09 18 ff
01 20 d5 17 5f 8c f2 53 0e 00 00	01 20 d5 17 5f 8c f2 53 0e 00 00
00 00 00 00 00 ea 7d ed ed ed	00 00 00 00 00

Fig. 14. USB Traffic Compared to OTA Traffic.

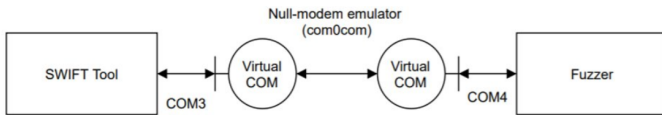


Fig. 15. Diagram of com0com virtual COM port connection layout.

data hasn't been corrupted during its transmission, but their appearances slightly differ. Thirdly, the USB protocol uses an end delimiter to indicate the end of the message. The OTA protocol hasn't been proven to use any type of end delimiter, although it's worth noting that following the OTA message there's a sequence of 0xFF's which is really 0x55's after the XOR operation takes place. Currently, no relation has been shown between these additional bytes and the OTA message.

V. PROTOCOL FUZZING

An effective method of analyzing an API or protocol is through the use of a "fuzzer", which is a program that provides random data for certain parameters in order to cause a crash or some other interesting behavior. This method is especially useful for testing the equipment and SWIFT Tools software, since unexpected behavior and interruptions of service would be the goal of a malicious actor. Because the previous semester's team focused on the OTA protocol through URH, the team decided that it would be best to target SWIFT Tools and attempt to send data from the W-USB adapter instead.

A. Serial Fuzzer

In order to send custom data to SWIFT Tools, the fuzzer must mimic the W-USB adapter's behavior, which uses a COM port for communication. To replicate W-USB's behavior, a Python module called 'pyserial' is used to develop the fuzzer. 'pyserial' can open, read from, and write to a COM port. However, the naïve approach of simply opening a port to communicate with SWIFT Tools does not work. SWIFT Tools also opens a port when it attempts to communicate, but once a port is opened by an application, another application cannot open the already-occupied port. This leads to a port conflict, which can be mitigated by com0com Null-modem emulator.

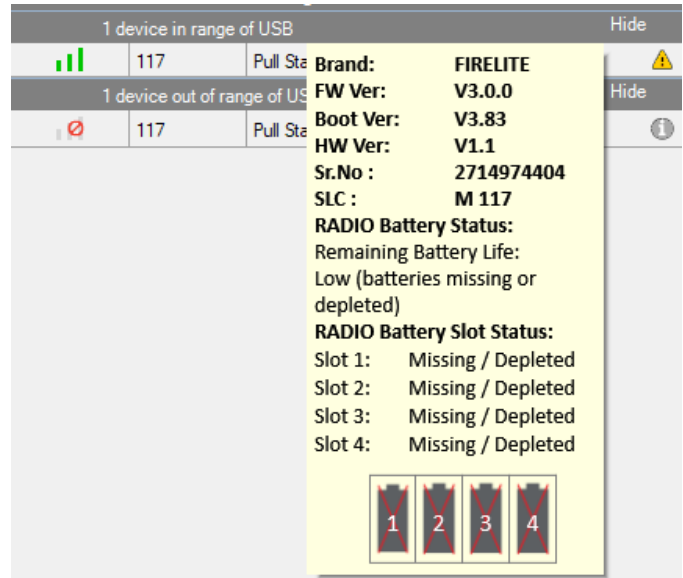


Fig. 16. A pull station without any battery sending an update.

A project called com0com Null-modem emulator is the software of choice to communicate directly to the SWIFT Tools. It can "create an unlimited number of virtual COM port pairs and use any pair to connect one COM port-based application to another. The output to one port is the input from other port and vice versa" [8]. Using the emulator as a virtual adapter, the Python fuzzer script can open a port while allowing SWIFT Tools to open a port concurrently. Python's serial module would be able to input data into one of the paired ports, and the other port would act as an interface between SWIFT tools and the Python script where the data is exchanged (Figure 15).

To find a device to communicate with, SWIFT Tools sends a 5-byte long conversation initiation message (7b 13 00 13 7d) on available COM ports sequentially. When an appropriate listener (e.g., W-USB adapter) receives the initiation message, it replies to SWIFT Tools with a 7-byte response message (00 00 7b 14 00 14 7d). Once SWIFT Tools receives a response from one of the ports, it proceeds to send an 11-byte long data (7b 49 46 00 00 00 00 01 00 0e 7d). Finally, the W-USB adapter responds with an at least 5-byte long acknowledgment message (7b 01 00 01 7d) to complete the handshake. Once the handshake ends, the W-USB adapter periodically sends updates regarding the fire alarm system devices' status. The conversation data was captured using Serial Port Monitor in order to analyze what kind of behavior the Python script should imitate to spoof the W-USB adapter. Since the byte sizes and the contents of the initiating handshake have been discovered, the Python script could utilize 'read(byte_num)' and 'write(bytes())' methods to replicate W-USB's behavior.

After the initial handshake, SWIFT Tools expects fire alarm system devices' status messages. At this stage, simply sending in a previously known status message will populate SWIFT Tools' device discovery GUI pane. For instance, modifying the



Fig. 17. Diagram of com0com virtual COM port connection layout for the MITM script.

message field indicating the battery status to 0x2 will cause SWIFT Tools to add a seemingly normal device but without any battery (Figure 16). This leads to a possibility where the script can spoof any device with any device configuration and condition. Changing a field incrementally manually, however, is a time-consuming process. Therefore, the script includes a mode where it can generate a combination of messages in which only a certain field differs. If the user wishes to modify the first byte of an example message, aabbcc, the user can provide a range of sub-messages such as 0x1 to 0xf. The script then generates a following combination of messages: 01bbcc, 02bbcc, 03bbcc... 0fbbcc. Then instead of repeatedly sending in a single message, the script cycles through the collection of messages and writes the data.

B. Serial Man-In-The-Middle

While the serial fuzzer script can inject various data into SWIFT Tools by spoofing the W-USB adapter, it is only useful in a local setup and cannot be used simultaneously with a W-USB adapter. To mitigate the problem, a man-in-the-middle (MITM) Python script has been developed. The MITM script setup also utilizes com0com Null-modem emulator to create virtual communication interfaces. Instead of using a pair of virtual COM ports like what the serial fuzzer script required, the MITM script communicates to two COM port interfaces, which means two pairs of virtual COM ports need to be set up. The SWIFT Tools communicates through a port that is connected to the MITM script, in which the messages are then passed to W-USB using another COM port (Figure 17).

The serial fuzzer only had the responsibility of reading and sending response messages, but the MITM script reads and forwards data to both sides. Currently, the MITM script can relay the initial handshake and the device status update messages from the W-USB side. During the forwarding process, it can read all the data that is sent between the SWIFT Tools and the W-USB adapter. This procedure can also be accomplished using Serial Port Monitor; however, the MITM tools' features can expand to handle more data-related operations. For instance, not only it can intercept the data, but it would also be able to modify it before forwarding it to either side. Also, the amount of data sent from the W-USB can be overwhelming if many devices are communicating to it. The MITM script would be able to filter out data to assist data analysis.

VI. CONCLUSIONS

The over-the-air (OTA) and USB (serial) communication protocols of Honeywell's SWIFT system have been analyzed. Further, Honeywell's user-side software tool called "SWIFT

Tools" has been reverse engineered which has led to substantial findings in SWIFT's communication protocols. The serial messages of multiple devices living on SWIFT's mesh network have been nearly fully decoded. Currently, only the pull station's RF transmissions have been captured and decoded through a comparison of the OTA and USB protocols. All work has been completed in the unencrypted mode of the SWIFT network. With or without encryption, understanding the communication protocol is essential to the vulnerability analysis of the system.

Future goals for the analysis of the SWIFT system include: the capturing and analysis of RF transmissions from other devices on the mesh network, the decoding of serial messages with different message types, an examination of the optional encryption mode of the SWIFT system, additional ability to inject and filter certain serial messages with the MITM script, and an exploration on bypassing the physical demands of the magnet in relation to the Gateway. With this information, a successful red team style cyber-attack against the SWIFT system is essential to substantiate any and all vulnerabilities.

REFERENCES

- [1] Honeywell. Frequently-asked questions about swift. [Online]. Available: <https://www.securityandfire.honeywell.com/notifier/en-us/latesttopics/frequently-asked-questions-about-swift>
- [2] —. Swift suite products page. [Online]. Available: <https://www.securityandfire.honeywell.com/notifier/en-us/browseallcategories/wireless/swift>
- [3] D. Crimmins. (2019, July) What is a fire alarm system? [Online]. Available: <https://realpars.com/fire-alarm-system/>
- [4] Honeywell, *NOTIFIER SWIFT intelligent wireless system Architectural and Engineering Specifications*. [Online]. Available: https://www.securityandfire.honeywell.com/en/textasciitilde/media/Files/Notifier/Engineering/%20Specs/SWIFT_specification_ENGLISH_docx
- [5] —, *NOTIFIER SWIFT intelligent wireless system Architectural and Engineering Specifications*. [Online]. Available: <https://fccid.io/AUBWFSAV/User-Manual/Exhibit-D-Users-Manual-per-2-1033-b3-3767262.pdf>
- [6] V. K. Garg, "Chapter 23 - fourth generation systems and new wireless technologies," in *Wireless Communications & Networking*, ser. The Morgan Kaufmann Series in Networking, V. K. Garg, Ed. Burlington: Morgan Kaufmann, 2007, pp. 23–1–23–22. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780123735805500570>
- [7] Semtech, *SX1231 Transceiver Datasheet*, June 2013. [Online]. Available: <https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/44000000MDrO/IWPNMeJCIEs8Zvyyu7AIDIKSyZqhYdVpQzFLVfUp.EXs>
- [8] F. Vyacheslav. Null-modem emulator (com0com). [Online]. Available: <http://com0com.sourceforge.net/>