RAPID7

WHITEPAPER

Leveraging Inter-chip Communication Analysis for Examining End-to-End Security within IoT Technology

Deral Heiland - Principal Security Researcher (IoT)
Matthew Kienow - Senior Software Engineer
Pearce Barry - Manager, Engineering

When evaluating security of IoT technology, the one thing we want to see is communication leveraging industry-standard encryption. Encrypted communication to and from sensors and actuators is critical to maintaining proper security within a product's ecosystem; the same goes for encryption that bridges technology, cloud services, and APIs.

So, when we encounter devices leveraging proper encryption, we ask ourselves, "are there other methods for examining IoT technologies that could give us deeper insight into the end-to-end security of the product's ecosystem?" The answer is yes! The evaluation of an embedded product's end-to-end security can often be greatly enhanced by examining data transfer at the circuit level via inter-chip communication as data passes through an embedded device. At the circuit level, communication between microcontrollers (MCU) is rarely encrypted; this can be used as an effective testing point on bridging devices. These devices provide Internet connectivity and remote management capabilities for IoT sensors and actuators. They utilize non-internet routable communication protocols such as Bluetooth Low Energy (BLE). It is common to see multiple MCUs in use within bridging devices.

To explore inter-chip communication concepts further, we'll explore essential topics that include:

- **Circuit-board layouts and communication path tracing**
- **Capturing of inter-chip communication**
- **Decoding and analysis of inter-chip communication**

Before we start hacking an IoT device, the first step is to set up and configure the product to be fully functional. During this process, we also learn much about the device, including how it communicates and what protocols are used (BLE, Zwave, Ethernet, etc.). This will come in handy as we proceed through the process of conducting inter-chip communication testing and analysis in the following sections.

In this paper, we will use an example with a number of unique characteristics which you may encounter during examination of other IoT technologies. Unfortunately, we won't be able to cover every variation of communication, MCUs, and circuit-board layout you might encounter. Although, from a circuit-board layout, our test example will give you insight into issues that might come up when examining inter-chip communication.

Circuit board layout and analysis

The questions to answer when evaluating a circuit board for inter-chip communication analysis include:

1. What is the device and its purpose?
2. What is the end-game for this analysis?
3. What are the primary components of interest?
4. Are they Ball Grid Array (BGA) or non-BGA chips?
5. Is the circuit board multilayered or just 2 layers?

For our example and to answer the first question, we'll use a BLE bridge device, which can add remote capability to a consumer-based door lock. As mentioned previously, bridging devices are the most common target for this style of inter-chip communication analysis. First, they often have multiple communication protocols and paths like WiFi, Ethernet or some form of non-internet routable RF communication used to control IoT sensors and/or actuators. Second, with multiple paths you'll often find the bridge's circuit design can leverage different MCUs for each communication protocol, allowing for inter-chip communication analysis. Here are some common block-diagram examples that show typical communication flow:

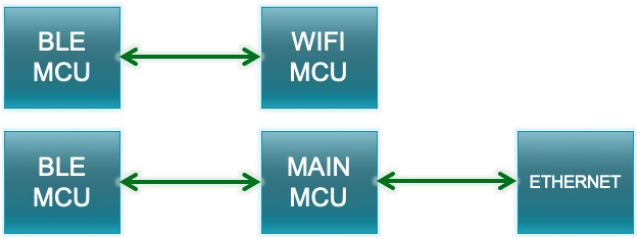


Figure 1: Communication flow block diagrams

Since each circuit board is different, it's impossible to cover every design scenario. We'll look at this particular bridge device because of the unique complexity within the circuit that we'll discuss later. We expect that most bridge-device circuit boards will have much less complexity, on average.

To answer the second question, what's our end-game? We want to gain a better understanding of this IoT product's authentication and validation mechanism. How does the

lock know when to open or close, and how is that mechanism secured within the end-to-end security of the product's ecosystem?

Moving on to question 3, we next open up the case on our target bridge device and identify all components of interest. Primarily, we need to identify what MCUs are being used by our bridge device. Since we know the device uses BLE for controlling the lock hardware and WiFi 802.11 for connecting to the Internet — learned during the product's initial setup — we next examine the hardware to see if these two protocols use different MCUs. If the same MCU is leveraged for both BLE and WiFi, then analysis of the communication becomes more problematic, if not impossible. However, finding bridging devices which use a single MCU for multiple communication protocols is becoming more common. In our example, each communication method uses a different MCU, as shown below:

Figure 2: MCU layout on targeted bridge device

Both BLE- and WiFi-communication MCUs are deployed as modules on this bridge device. Access to communication on both modules is done along the edge. This answers question 4, with regard to the MCU BGA. Exposed pinout on the modules will give us full access to the needed communication.

For BLE communication, this bridge device is using a module produced by Wistron NewWeb Corp (WNC). This was determined by using data printed on the module 'XRBH' and searching the Internet using Google. The search returned several results, but one of those was specifically for FCC information. On the FCC site, we are able to find internal photos of the module as well as the manufacturer's user manual. This contains module layout and pinout identification. If you encounter a BLE module and are not able to identify the manufacturer or its correct pinout, the next best method is to remove the shielding from the module. This will often reveal the actual MCU being leveraged and, by obtaining the datasheet for the actual MCU, you may be able to trace out the correct pinout on the module itself. For example, if we remove the shield on this module, we see that the BLE MCU being used is a Nordic nRF51822 (Figure 3). If you encounter a case where the shield needs to be removed, go slow and cautious to avoid damaging the device. Avoid placing too much stress on the shield-to-board solder joints; this could lead to tearing the ground plane and dislodging various components, rendering the module useless.

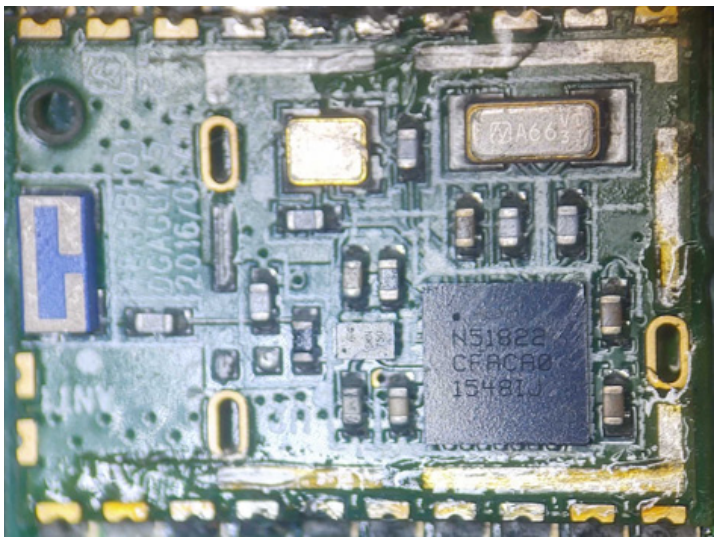


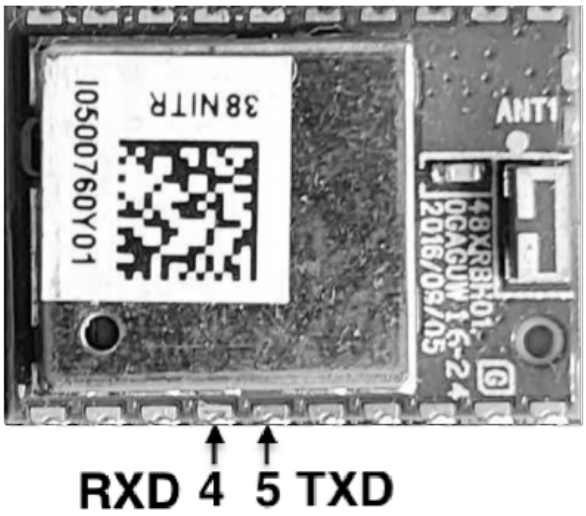
Figure 3: BLE module with shield removed

For network and Internet connectivity, the Espressif module 'ESP-WROOM-32D' on this device supports the WiFi 802.11 communication protocol. Espressif module information and development data is readily available from Espressif for identification of the pinouts along the edge of the module. Taking all of this information into account, we can relabel the block diagrams from above to show how this applies to our test example.

Figure 4: Communication flow block diagram

Our next step is to identify the inter-chip communication method being used and trace out the connection on the circuit board between the two MCUs. Host communication access to BLE MCUs historically is done via Universal Asynchronous Receiver Transmitter (UART). So, starting with that piece of knowledge, we next examine the user manual or datasheet for modules used in our example. Within the datasheets on most BLE MCUs, we should be able to identify the proper pinout and UART. In our example, we identify the UART pinout shown below.

PIN	DESCRIPTION
4	UART RXD
5	UART TXD



At this point, we attempted to trace the UART within our test example from UART RXD pin 4 and TXD pin 5 to the other MCU (in this case, that would be the Espressif ESP-WROOM-32D). Often the simplest way to do this is by using a multimeter that is set to 'continuity.' This setting often has a diode and audio symbol, as shown below.

When the leads of the meter are touched together, this creates a ringing sound, indicating circuit continuity. This makes it simple to trace the electrical connection across a circuit board.

In the case of the bridge device in this example, any attempt to trace pins 4 and 5 on the BLE module directly to the pins on the Espressif module will not work. Why? After close visual examination of the circuit board, we can see the designer placed resistors in series between pins 4 and 5 and the runs on the circuit board leading to the Espressif module. This is highlighted below in Figure 5, showing pins 4 and 5 connected to resistors. Once the electrical path goes through the resistors, it passes through a via to the other side of the circuit board.

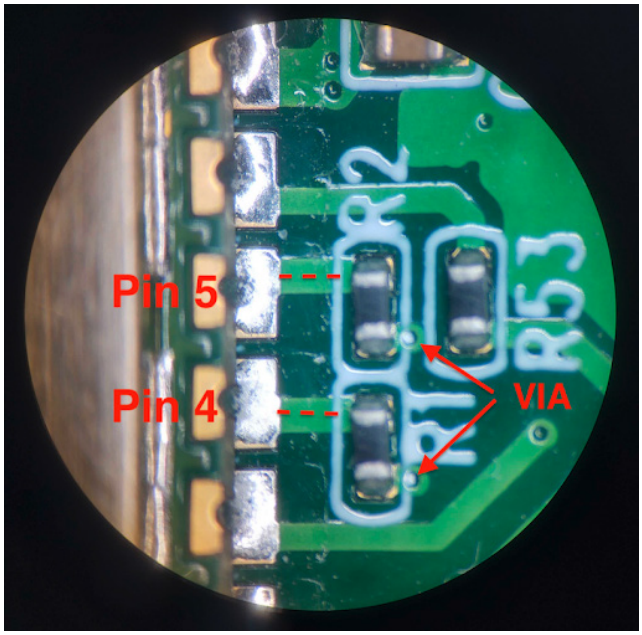
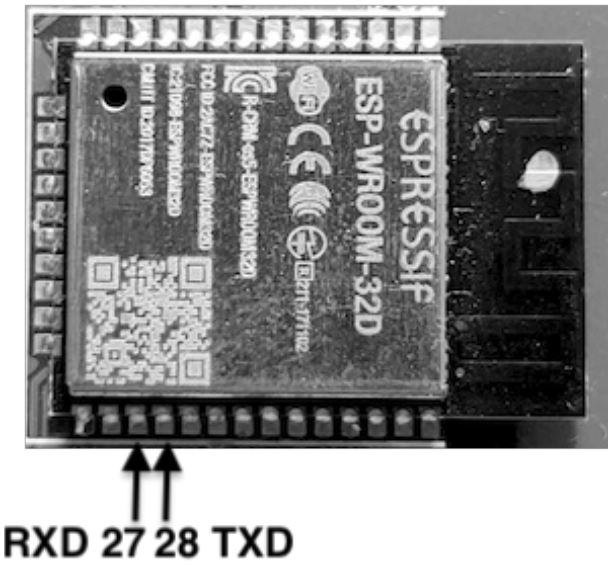


Figure 5: UART connections circuit

Although we don't often encounter UART communication circuits that have resistors in series between MCUs, it is not unheard of. The purpose of this is for impedance matching between the two modules (BLE and WiFi). Impedance matching is often needed to improve signal transfer or minimize signal reflection. We could continue tracing the circuit path visually, but often this, on its own, can become problematic as it passes between layers and under other components on the circuit board. So, using visual methods, combined with continuity tracing, is the most effective method. In this case, we continued tracing the circuit by using the multimeter set on continuity and placing one lead on the resistor at the via side near where it passes through the circuit board; and also touching the pins on the Espressif with the second lead and listening for the multimeter to ring.

Using this method, we were able to trace the following connection between the MCUs and then use the available datasheet to identify and validate the correct Input/Output

WNC BLE MCU		ESPRESSIF ESP-WROOM-32D MCU	
PIN	DESCRIPTION	PIN	DESCRIPTION
4	UART RXD	28	GPIO17, HS1_DATA5, U2TXD, EMAC_CLK_OUT_180
5	UART TXD	27	GPIO16, HS1_DATA4, U2RXD, EMAC_CLK_OUT



Using the datasheet available for the Espressif module, we identify that both pin 27 and 28 support multiple general-purpose IO capabilities. It is also important to note it is not uncommon to find multiple UARTs available on some MCUs, so tracing back from the BLE modules to the WiFi module was the most practical way to identify the UART port pinout on the WiFi MCU for the inter-chip communication. In cases where no data sheets are available for either MCU, it may be necessary to rely on a logic analyzer and a trial-and-error method to identify the proper UART communication channels between the MCUs.

Once the signal path has been traced out, connecting into the device for capturing and viewing data becomes simple. Although, if we are going to do more than just listen, we will need to expand our testing environment. This may include being able to do the following:

- **Capture communication**
- **Replay communication**
- **Fuzz communication**

The best way to accomplish this — once we trace out the inter-chip communication — will be to cut the circuit runs and reroute the communication flow off the circuit board for analysis. To help with this, we've created breakout boards which allow for communication to be routed off the primary circuit board onto the breakout board; and from there the communication can be more easily engaged for testing. Below is a sample of a breakout board. As can be seen in the following image (Figure 6), we have a screw terminal for connecting wires from the circuit board to the breakout board, headers for connecting test equipment such as FTDI devices and logic analyzers, and we also have inline on/off switches that can be used to open and close the circuit path flow, if needed.

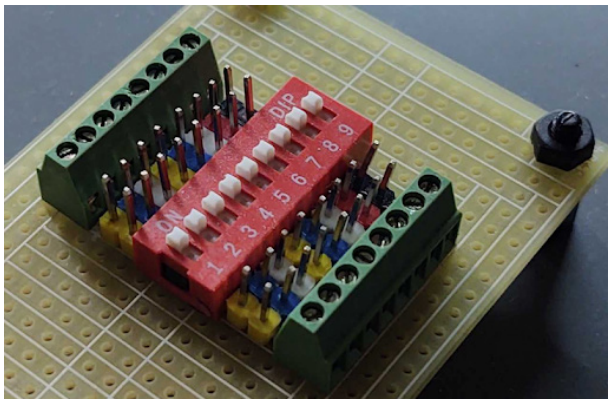


Figure 6: Breakout Board

As mentioned previously, once we've identified the communication path on the circuit board, we cut the runs on the circuit board and solder wires to each side of the severed connection and attach those wires to the breakout board shown above. This allows us to reconnect the circuit through the breakout board, making the breakout board an extension of the circuit board. The added length of the communication lines typically has no impact on performance or reliability when dealing with UART communication.

As mentioned earlier, the circuit board on our bridge device has some added complexity because of the inline resistors. Typically, we can cut the communication runs on the circuit board anywhere. But, in this case, we are challenged with cutting and redirecting the circuit on the proper side of the resistor. The answer to that

question is, when dealing with inline resistors between two modules communicating with UART, the circuit should be cut on the TXD side of the resistor, as shown in the following block diagram (Figure 7):

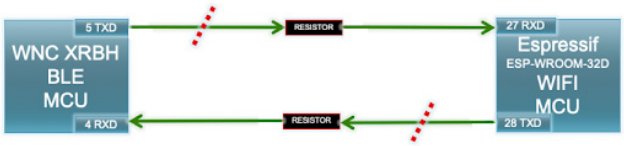


Figure 7: Circuit block diagram

This ensures that our interaction with the bridge devices UART inter-chip communication does not interfere with the impedance matching delivered by the inline resistors to the receiving module.

Also, in this example we run into even more complexity because the communication path from BLE MCU pin5 and WiFi MCU pin 27 has limited area accessible between the resistor and BLE MCU pin 5. The close proximity of the resistor to the module does not easily allow the circuit run to be cut or needed wires easily attached. This issue is shown below in Figure 8. I also added a U.S. penny to the image to give some size perspective.

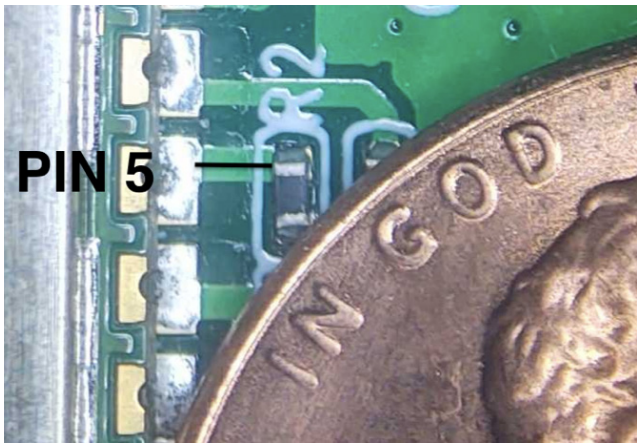


Figure 8: WNC BLE CPU Pin 5 to Resistor

It is possible to cut the run, then attach the wire directly to the resistor and to the module's Pin 5, but I found this to be more problematic. So, in this infrequent case, I found it easier to remove this resistor and then place the resistor closer to the WiFi MCU Pin 27. It is important to note that in some cases, like this one, prepping the device for effective and repeatable inter-chip communication testing and analysis may involve some simple modification to the device and often those are safe to do.

So, to move this resistor and modify the board to route inter-chip communication to our breakout board, we de-soldered the resistor and moved it to the back side of the board and then were able to attach our connecting wires where the resistor originally was installed, as shown in Figure 9:

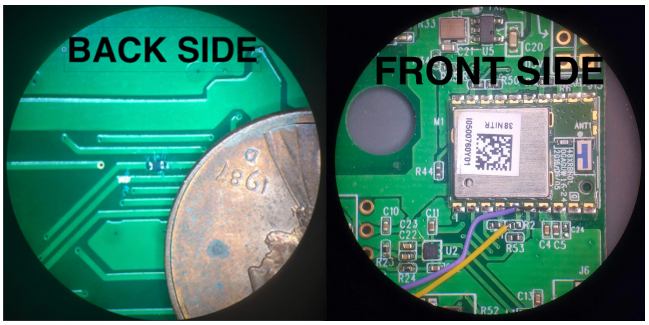


Figure 9: Circuit board modification

Once the inter-chip circuit communication runs for UART have been cut and the wires added and attached to the breakout board, we can move onto capturing, decoding and analysis of inter-chip communication.

Inter-chip Communication

Now with inter-chip communication routed out to a breakout board, we can start capturing the communication for analysis and testing. The best method to start with is to use a logic analyzer. A logic analyzer will give us a chance to evaluate the communication and determine the speed, settings, and general message structure.

Analysis with a logic analyzer using an Async Serial decoder will help in validating those settings. In the case with our example, again different from the norm we typically encounter, the baud rate was set to 9600, which is very slow for typical inter-chip communication. As stated previously, the logic analyzer will also help with determining the communication message structure. For example, if we connect a UART to USB FTDI device and begin capturing data immediately, we will find that inter-chip communication does not use end-of-line or carriage returns we typically see when connected to a standard UART console. So, in this case, the captured data will just wrap, making it more difficult to understand the message structure. An example of that is shown Figure 10.

```
0150   92 A4 7A 98 D6 C1 00 DC 00 FF 77 0F 00 BC CC C2   í§zò÷¡.‹.¨w..°A¬
0160   A0 1C 63 C4 EF 3C CD C3 A0 9C 62 C5 EF C1 77 20   †.cƒÔ<Õ√†úb≈Ô¡w
0170   00 BC CC C2 A0 1C 63 C4 EF 3C CD C3 A0 9C 62 C5   .°Ã¬†.cƒÔ<Õ√†úb≈
0180   EF D2 77 0D 00 9E 9E DE A8 32 13 CC AF 3C CD C3   Ô“w..ûûfi®2.ÃØ<Õ√
0190   A0 9C 62 C5 EF 25 77 0C 00 9E 9E DE A8 32 13 CC   †úb≈Ô%w..ûûfi®2.Ã
01A0   AF 3C CD C3 A0 9C 62 C5 EF 24 77 07 00 9E 9E 5E   Ø<Õ√†úb≈Ô$w..ûû^
01B0   28 32 13 4C AF 5F EA DF 92 A4 7A 98 D6 C7 77 01   (2.LØ_ÍflÍ§zò÷«w.
01C0   00 9E 9E 5E 28 32 13 4C AF 5F EA DF 92 A4 7A 98   .ûû^(2.LØ_ÍflÍ§zò
01D0   D6 C1 77 01 00 9E 9E 5E 28 32 13 4C AF 5F EA DF   ÷¡w..ûû^(2.LØ_Ífl
01E0   92 A4 7A 98 D6 C1 77 01 00 9E 9E 5E 28 32 13 4C   í§zò÷¡w..ûû^(2.L
01F0   AF 5F EA DF 92 A4 7A 98 D6 C1 77 01 00 9E 9E 5E   Ø_ÍflÍ§zò÷¡w..ûû^
0200   28 32 13 4C AF 5F EA DF 92 A4 7A 98 D6 C1 77 01   (2.LØ_ÍflÍ§zò÷¡w.
0210   00 9E 9E 5E 28 32 13 4C AF 5F EA DF 92 A4 7A 98   .ûû^(2.LØ_ÍflÍ§zò
0220   D6 C1 77 01 00 9E 9E 5E 28 32 13 4C AF 5F EA DF   ÷¡w..ûû^(2.LØ_Ífl
0230   92 A4 7A 98 D6 C1 77 01 00 9E 9E 5E 28 32 13 4C   í§zò÷¡w..ûû^(2.L
0240   AF 5F EA DF 92 A4 7A 98 D6 C1 77 01 00 9E 9E 5E   Ø_ÍflÍ§zò÷¡w..ûû^
0250   28 32 13 4C AF 5F EA DF 92 A4 7A 98 D6 C1 77 01   (2.LØ_ÍflÍ§zò÷¡w.
0260   00 9E 9E 5E 28 32 13 4C AF 5F EA DF 92 A4 7A 98   .ûû^(2.LØ_ÍflÍ§zò
0270   D6 C1                                             ÷¡
```

Figure 10: Typical UART Console Capture

But, with a logic analyzer, we can identify the beginning and end of the message structure used to send command and control packets between the MCUs based on timing, which is easier to break down using a high-speed logic analyzer. This allows us to delimit the message for decoding. An example of this is shown below in Figure 11. In the following data capture, we can see the message structure and identify the starting byte and message length. For example, the WiFi MCU message starts with 0x77, and the response message from the BLE MCU starts with 0x41, as shown in

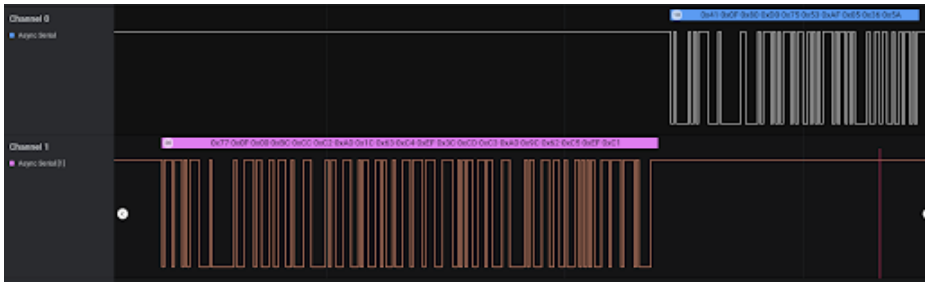


Figure 11: Logic Analyzer Capture

Once we identify starting delimiters, we can better parse the communication and start analyzing data. For this example, the process was simple due to the limited number of message types and starting delimiters we encountered, plus the amount of communication data is also limited. In some cases, you will encounter very verbose inter-chip communication with many different message types and different message starting delimiters. Also, in some cases, those message-start delimiters may be multibyte in length.

With 0x77 and 0x41 message-start bytes identified, we can next exercise the technology and capture the communication; then we'll start parsing and decoding the message structure and function. We start with both an open and close command message sent to the lock from the internet. This was captured passing from the WiFi MCU to the BLE MCU. The only difference between these 2 commands — open and close — is the second and the last byte shown in Figure 12. With the second byte (shown in red) we determined 00 is for open and 01 is for close. The last byte in blue is the checksum (CRC).

OPEN:
77 00 00 9E 9E 5E 28 32 13 4C AF 5F EA DF 92 A4 7A 98 D6 C0

CLOSE:
77 01 00 9E 9E 5E 28 32 13 4C AF 5F EA DF 92 A4 7A 98 D6 C1

Figure 12: Open and close inter-chip capture

A key observation from this captured data is the fact that nothing else changes. This seems to indicate that any security validation, such as key or packet signing, is fixed data. Often when reviewing communication data that originated from internet cloud services, we also see the MAC address of the target sensor or actuator. In this case, we expected to see the MAC address for the associated Bluetooth Low Energy (BLE) radio on the door lock, but did not. So, at this stage we don't know what the data between the open/close command and the CRC represent.

Since we have full access to the inter-chip communication, we have the luxury of making changes and replaying the data within the inter-chip communication.

Before we start conducting any replay or fuzzing type test, the first thing is to examine the response from BLE MCU and see what we can discern from that data. An example of this is shown in Figures 13 and 14. Each command sent to the BLE MCU resulted in 2 responses, each 10 bytes in length. The bytes shown in brown are the actual MAC address of the BLE radio on the

door lock. The purple byte is the response type. After some general analysis of these response messages, we concluded that the first 10-byte message, (for example: 41 00 03 C8 28 3A 64 8C 45 5A) acknowledged that an 'open' command was received and the second 10-byte message (41 E5 EA C8 28 3A 64 8C 45 5A) indicated the lock position status after the execution of the 'open' command.

OPEN:
41 00 03 C8 28 3A 64 8C 45 5A
41 E5 EA C8 28 3A 64 8C 45 5A

Figure 13: Open responses from BLE MCU

CLOSE:
41 01 06 C8 28 3A 64 8C 45 5A
41 E0 EA C8 28 3A 64 8C 45 5A

Figure 14: Close responses from BLE MCU

Using a captured open-and-close message, we started altering the data one byte at a time and replaying it. The purpose of this was to see if any of the data was associated with the BLE MAC address that was echoed back in the response message. Using this fuzzing exercise, we were successful in determining that the MAC address was encoded in the initial command message being sent from the Internet. After fuzzing through all the bytes, we determined the 8 bytes shown in green below were being used to encode the MAC address.

77 00 00 9E 9E 5E 28 32 13 4C AF 5F EA DF 92 A4 7A 98 D6 C0

Each time one of these bytes was changed, the responding message from the BLE MCU changed the MAC address within the response. The responding messages appeared to be an error response, as the MAC address did not match the MAC address of the configured BLE lock device. We also used this sample data to figure out the encoding method being used so it could be reproduced. To do this, we ran a number of tests across each of the encoded bytes in the command message. The following example data (Figure 15) shows information gathered from altering the encoded MAC address and replaying it in an open or close request message, followed by the actual MAC address returned in the response message. The red text in the left column shows the bytes altered, and the red text in the right column shows the changes to the returned response messages. The specific MAC address (CD:AA:85:CE:2F:6D) is one that was used in a second home BLE lock device we leveraged during testing.

Encoded MAC Address	Return Message
22 9e 26 a0 ea 19 70 85	= CD AA 85 CE 2F 6D
20 9e 26 a0 ea 19 70 85	= 4D AA 85 CE 2F 6D
22 9C 26 a0 ea 19 70 85	= 8D AA 85 CE 2F 6D
22 9e 24 a0 ea 19 70 85	= ED AA 85 CE 2F 6D
22 9e 26 9e ea 19 70 85	= DD BA 95 DE 3F 6D
22 9e 26 a0 ea 19 70 85	= C5 AA 85 CE 2F 6D
22 9e 24 a0 ea 17 70 85	= C9 AE 81 CE 2F 6D
22 9e 26 a0 ea 19 6e 85	= CF A8 87 CC 2F 6D
22 9e 26 a0 ea 19 70 83	= CC AB 85 CE 2F 6D

Figure 15: UART fuzzing response table

Once we gathered a block of data using this test method, we were able to take the data, convert the information into binary, and determined a rough encoding map being used. This provided us the ability to recreate an encoded 8-byte string from the known MAC address.

The purpose of using the second home BLE lock mentioned above was to see if we could target that lock from the example bridge device without configuring and syncing with the product's cloud service. In other words, could we weaponize a bridge to target endpoint BLE devices to which we didn't have access rights? The answer is no. The reason for this failure was that we didn't know the 8-byte key for that specific device to properly configure it and validate the message. Although, we were able to configure the test bridge to control 2 locks by capturing the device's original lock configuration setup via inter-chip communication for the separate locks. A factory reset was required after each setup to gain access to the key we mentioned previously. We then replayed both of the device setups via inter-chip communication — one after the other — tricking the test device's BLE MCU to start accepting inter-chip playback commands for both. The product manufacturer noted that a separate bridge device was required for each home door lock. Even though this double configuration trick worked, we were only able to control the lock from inter-chip playback and not from the remote cloud services.

Overall, the purpose of this exercise was to expand our understanding of inter-chip communication and gain deeper insight into methods for testing and understanding end-to-end security for devices like this BLE door lock. For this specific test case, we concluded that the following security

concerns existed and expect similar issues may apply to other IoT technologies:

- **The key received by the lock, and used to validate the message, is fixed and never changes.**
- **The length of that key is, at best, 8 bytes. This is based on the identified structure shown below in Figure 16.**
- **This key most likely consists of a shared secret, which is probably based on the registration number of the BLE door-lock device at the time it is set up with cloud services.**

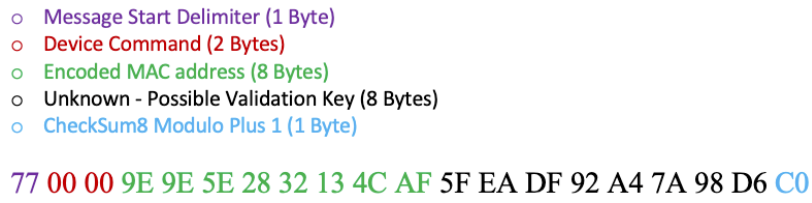


Figure 16: Identified message structure

With that information, we can better examine other aspects of the product's ecosystem security. We begin to realize that if a shared secret can be compromised or is predictable, then that information could have an adverse effect on the product's end-to-end security.

To better facilitate inter-chip communication analysis, the following section addresses a new UART Proxy tool. We are currently developing this tool to help security experts conduct inter-chip analysis on devices using UART for inter-chip communication.

Inter-chip UART Proxy Tool

In our initial inter-chip analysis projects, we used a very manual process for capturing and replaying data with typical UART console tools; also for outputting data to a file and then parsing that data using simple regex scripts. After several of those, we started digging into better ways to streamline inter-chip analysis and testing. First, we built a couple of standalone python scripts where we tested receiving and forwarding UART data in real time to simulate a man in the middle (MiTM) function. We also wrote another python script that allowed us to parse and display the data in real time based on starting/ending bytes or message length. This worked fine for a very basic proof-of-concept (PoC), but we had big plans to make a more robust solution for conducting UART inter-chip communication MiTM. At this stage we realized we needed help from real developers. We then reached out to teammates Matthew Kienow and Pearce Barry.

So, as a team, we sat down and decided that the first version (v0.1) would be simple beta PoC and contain the following basic options:

- **Allow user to set the incoming and outgoing serial connection.**
- **Allow the user to set the message start- or ending-byte. This also allows for a multibyte setting.**
- **Allow for display of UART data in real time as it passes through the application MiTM function.**
- **Allow for data to be captured to a file. During this capture, the data were tagged with direction, A->B, A<-B, and also messages were numbered in order received.**
- **Allow user to replay captured data using message direction and numbered messages.**
- **Allow users to do a byte identification and replace on data being replayed. For example, if you find AF E6 33 in the message, replace it with FF 01 D8 during replay of captured data. Additionally, support for simple message checksum update allows messages with replaced data to properly validate with the receiving device. This allowed for a simple fuzzing function using the captured data.**

Once we defined general project scope, we set out to create version 0.1 as a proof of concept, which we named Akheron Proxy. Akheron Proxy tool was named after the river Akheron from Greek mythology. The Akheron river is also known as the river of woe, one of the magical rivers that flows through the underworld. It is the only magical river that also flows through the mortal world.

To get the Akheron Proxy PoC tool hooked up and running in a real environment, the most critical first step is to properly hook up two USB to Serial FTDI devices to the breakout board. Before we start our introduction to this new proxy tool, we need to note some key rules when connecting devices to UART. TXD hooks to RXD and RXD hooks to TXD. Since we are connecting 2 FTDI devices, we also must remember that we will receive data on one FTDI, which we will call FTDI (A), and then retransmit that data out the other FTDI, which we will call FTDI (B). Data flowing the other direction will be received on FTDI (B) and retransmit out FTDI (A). The following image (Figure 17) shows these FTDI connections in reference to our test bridge devices (MCUs). When setting this up in a test environment, we also recommend that you use color coding, such as colored wiring, to help avoid confusion. Note, the grey dashed lines in this diagram represent the connectivity through the on/off switches on the breakout board, as was shown earlier in this paper in Figure 6.

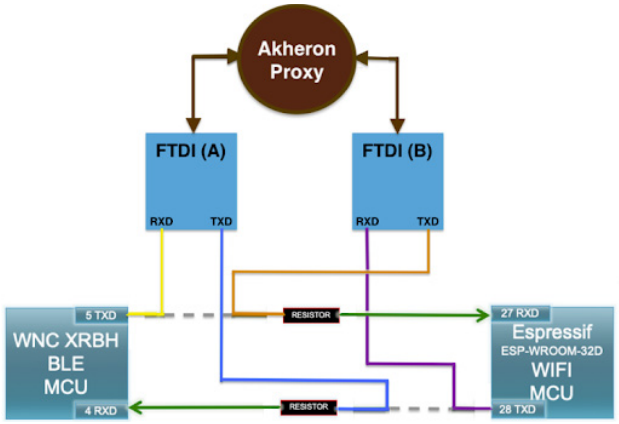


Figure 17: Connecting FTDI UART to USB devices

Once all wired up, you can download and run the UART proxy application Akheron Proxy. Once started, run the 'help' command to show the various commands available (Figure 18).

```
percx@MetaBox:~/tools/Kharon_proxy$ sudo ./kharon_proxy.py

########################
Welcome to the UART Proxy!
         v0.1
########################

> help

Documented commands (type help <topic>):
========================================
capturedump   checksumget  delimset  portget     replaceset  stop
capturestart  checksumset  help      portset     replay      watch
capturestop   delimget     list      replaceget  start

Undocumented commands:
======================
exit  quit  version

>
```

Figure 18: UART-Proxy Application

The next step is to set up the FTDI devices, which are typically ttyUSB0 and ttyUSB1. You can do this using the 'portset' command, as shown Figure 19:

```
> help portset

Description: apply UART port settings

Usage: portset <A|B> <device> <baud>

Example(s): portset A /dev/ttyUSB0 115200
            portset B /dev/ttyUSB1 115200

> portset A /dev/ttyUSB0 9600
> portset B /dev/ttyUSB1 9600
> portget
Port "A": device "/dev/ttyUSB0" at 9600
Port "B": device "/dev/ttyUSB1" at 9600
>
```

Figure 19: Setup FTDI devices

Once serial devices are connected within the application, you will next need to enable the delimiter to let the application know what the starting or ending byte(s) are for the communication messages. We discussed this earlier, and for our test device we determined the starting delimiters for the messages were 0x77 and 0x41. Figure 20 below shows how to use the 'delimset' command to set the delimiters:

```
> help delimset

Description: apply message parsing settings

Usage:  delimset <start|end> <hex byte pattern>[,<hex byte pattern>,...]

Example(s): delimset start 0x01 0x00, 0x01 0x04, 0x07
            delimset end 0x99

> delimset start 0x77, 0x41
> delimget
start delimiters: 0x77, 0x41
  end delimiters:
>
```

Figure 20: Set message delimiters

Once we set FTDI and delimiters, we can 'start' the MiTM passthrough and enable 'watch' so we can see the traffic passing through our MiTM proxy, as shown below in Figure 21:

Figure 21: Start MiTM Proxy

With this PoC, we also have enabled capture and replay. As mentioned earlier, capture will tag the messages with both direction and a sequential number. When you replay, you can select the message you wish to replay by specifying the number(s). Also, you will only be able to replay in one direction at a time. If you select more than a single line, the first message selected will determine replay direction and only the lines of message in that direction will be replayed. But first, let's look at capture. In the following images (Figure 22 and Figure 23) we show an example of the 'capture' commands in use.

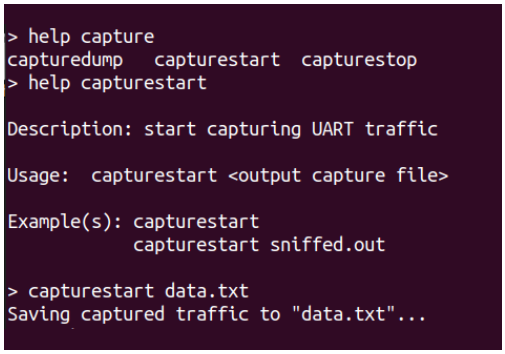


Figure 22: Starting capture

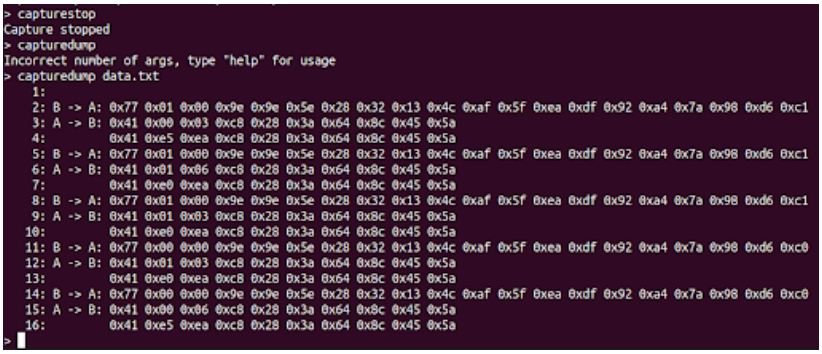


Figure 23: Stop and display capture

If you wish to see the data being captured in real time, you can restart the 'watch' function, which can be stopped by pressing 'control-c'.

To replay any of this captured data, you only need to use the 'replay' command, followed by the capture dump file name and the line number(s) of what you wish to replay; remembering that replay direction is determined by the direction of the first line selected. An example of this is shown below (Figure 24), where we replayed a close and then an open command and showed received responses from the BLE MCU, displaying command received and state of lock after the command was executed.

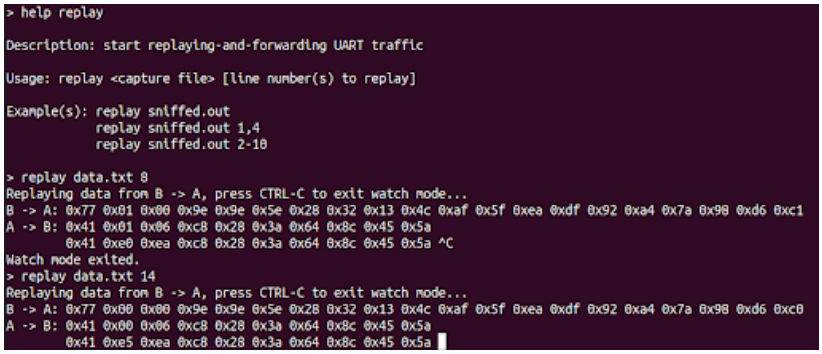


Figure 24: Message replay

The final feature of the PoC I we'll show is a simple replace on replay feature. This function allows us to alter things like MAC addresses or authentication tokens in an attempt to trigger adverse effects on targeted devices and services. Also, since any alteration of message data may require a recalculation of the CRC, a couple of CRC types were added (Figure 25). In this case, we only added a few 8-bit CRC options as PoC; we'll expand this in future releases, as needed.

```
> help checksumset

Description: set checksum recalculation used after message pattern replacement.
Note: This should be used with start delim patterns since the computed
checksum will be placed at the end of the message.

Usage:  checksumset <A|B> <checksum number or name>

Example(s): checksumset A 1
            checksumset B Checksum8Modulo256

Available Checksums:
  1: Checksum8Xor
  2: Checksum8Modulo256
  3: Checksum8Modulo256Plus1
  4: Checksum82sComplement

> checksumset B 3
> checksumget
No replace checksum specified on port A
Replace on port B using checksum 'Checksum8Modulo256Plus1'
>
```

Figure 25: Setting of CRC Checksum

In the following example (Figure 26), we show the use of this replace function by matching the pattern '0x77 0x01 0x00' on message being transmitted from PORT B and replacing it with '0x77 0x07 0x00' during a replay operation of captured data.

```
> replaceset B 0x77 0x01 0x00 -> 0x77 0x07 0x00
> replaceget
Replace port A pattern X -> pattern Y:
Replace port B pattern X -> pattern Y:
  0x77 0x1 0x0 -> 0x77 0x7 0x0
> replay data.txt 8
Replaying data from B -> A, press CTRL-C to exit watch mode...
B -> A: 0x77 0x07 0x00 0x9e 0x9e 0x5e 0x28 0x32 0x13 0x4c 0xaf 0x5f 0xea 0xdf 0x92 0xa4 0x7a 0x98 0xd6 0xc7
A -> B: 0x41 0x07 0x01 0xc8 0x28 0x3a 0x64 0x8c 0x45 0x5a
        0x41 0xe0 0xea 0xc8 0x28 0x3a 0x64 0x8c 0x45 0x5a
```

Figure 26: Replace during replay

So, what is the future of this proxy tool? We hope to expand its features and usability, perhaps adding a GUI somewhere down the road. Also, since inter-chip communication may include other protocols (SPI, I2C, USB), we want to structure the tool so that eventually we can plug other protocols' features into it. But for now, the plan is to make the tool a usable, key part of every embedded hardware-tester's arsenal.

Conclusion

In conclusion, we hope everyone can take away from this paper ideas and knowledge to help them understand the value in inter-chip analysis of embedded technology. Expand your understanding and get deeper insight into end-to-end security of IoT technology ecosystems.