Bluetooth Attacks on Commercial-Grade Electronic Locks

Somerset Recon

December 12th 2016

Version 1.0

Table of Contents

Introduction

Internet of Things (IoT) has become ubiquitous and today's consumer electronic industry has been focused on creating smarter and more automated devices with little standardization on how they should be implemented. A common pattern that many electronics companies[1] are using is applying modern interfaces, like web servers and Bluetooth, on top of pre-existing systems. The SecuRam ProLogic 0601A-B01 safe lock is a prime example of this. Instead of creating a secure safe container from the ground up, SecuRam designed a Bluetooth Low-Energy electronic lock that could be retrofitted on pre-existing safe containers. And while this is convenient, improper implementation of wireless security has lead to vulnerabilities[2] in recent residential-grade Bluetooth locks.

Instead of residential door and padlocks, we chose to focus our efforts on analyzing the security implementation of wireless commercial-grade products. We audited the SecuRam ProLogic system composed of the ProLogic 0601A-B01 (Prologic B01) entry pad and the lock body versions EL-0701 and EL-0601. Each of these devices are designed to secure both safes and facilities. The ProLogic B01 is a Bluetooth Low Energy (BLE) enabled entry pad that can be used with a mobile application. When a successful 6-digit number (PIN code) is entered into the entry pad or transmitted using the mobile application, it can send a signal to the lock body to pull the bolt and open the safe. This process relies on hardware and software to perform security functions for users, of which we discovered security vulnerabilities and design weaknesses.

This research required analysis of the low-level BLE packet protocol as well as the mobile application. We discovered that the mobile application had not been protected with anti-reversing safeguards and the wireless communication was not adequately protected. This ultimately allowed us to create a cheap and practical attack that could locate a Prologic B01 nearly 100 yards away, notify when a lock is opened over BLE, and pull the PIN directly out of the air. This leads us to believe that some modern BLE locks, regardless of commercial-grading, lack proper protocol standards and currently should be avoided.

[1] August Smart Lock, Belkin Wemo products like the Wemo Switch and Mr. Coffee® Smart Optimal Brew

[2] https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Rose-Ramsey-Picking-Bluetooth-Low-Energy-Locks-UPDATED.pdf

Device Overview

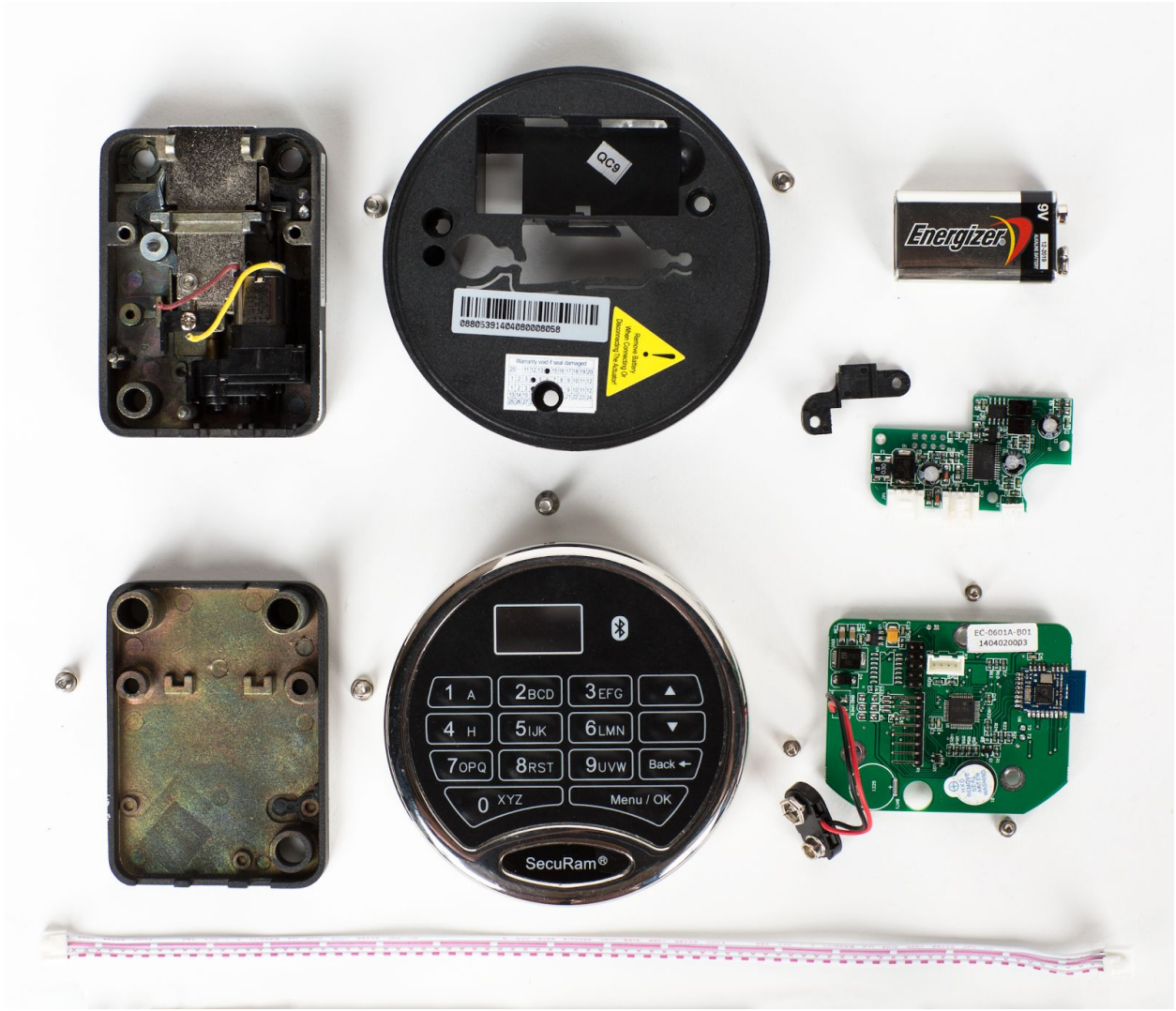


Figure 1: SecuRam lock body and entry pad teardown

The SecuRam electronic lock setup is composed of two main devices, the entry pad and the lock body. The entry pad is the front-facing device with which a user interacts; it is detachable and mounts onto a safe door via two bolts, making it easy to service the 9V battery that powers it. The lock body is the physical mechanism that pulls the lock bolt and mounts on the inside of a safe door. Both devices are connected via a cable the feeds through the spindle hole in the safe door. This 4-wire cable provides power to the lock body and a 2-wire communication channel for both devices. Once connected, the keypad and the lock body must be paired. This is done by holding the reset button on the lock body, connecting it to the entry pad via the 4-wire cable, and then pressing a button on it. If the devices are out of sync, they will refuse to connect to each other. For more information on the items in the Figure 1 teardown refer to Appendix A.

The entry pad functions both over BLE with a mobile application or physically via the 10-digit keypad. The device is programmed with default PINs for a manager user (111111) and regular user (123456). Each account is associated to a six-digit PIN used to issue an unlock command. The manager account can unlock a safe, add users, and delete users. As a means of protection, entering four wrong PINs in a row will result in a four-minute lockout. Once unlocked, the entry pad exposes auxiliary options such as viewing the lock version, firmware version, and lock recovery code[3].

Mobile Application Analysis

The ProLogic lock can be controlled from the “SecuRam Access” app on Android and the “Securam Access +” on iOS, where its primary function is to unlock the safe and manage user PIN codes. We decided to analyze the Android application because the decompilation came out fairly clean and there did not seem to be obfuscation or anti-root protections. We used the CFR decompiler[4] and a set of custom scripts to automate the process.

```
1 if ("com.easthouse.bluetoothlock.adapter.ACTION_DEVICE_UNLOCK".equals(string2)) {
2
3   BleLock bleLock =
4       (BleLock)
5           intent.getSerializableExtra(
6               "com.easthouse.bluetoothlock.adapter.EXTRA_DATA_CLICK_DEVICE");
7   String string3 =
8       intent.getStringExtra("com.easthouse.bluetoothlock.adapter.EXTRA_DATA_UNLOCK_PASSWORD");
9   UL uL = new UL();
10  uL.setOpenSecond(bleLock.getOpenSecond());
11  uL.setSendMacAddress(BluetoothService.this.mLocalDeviceAddress);
12  uL.setReceiveMacAddress(bleLock.getMacAddr());
13  uL.setOpenCode(Long.parseLong(string3));
14  uL.setYsDisplay((byte) bleLock.getScreenState());
15  uL.setYsVoice((byte) bleLock.getOpenVoice());
16  BluetoothSession bluetoothSession =
17      BluetoothService.this.FindOrCreateBluetoothSession(bleLock.getMacAddr());
18  bluetoothSession.sendAC();
19  bluetoothSession.sendCMD(uL);
20  bluetoothSession.setCharacteristicNotification(
21      bluetoothSession.getBluetoothGattCharacteristicRead(), true);
22  bluetoothSession.readCharacteristic(bluetoothSession.getBluetoothGattCharacteristicRead());
23  return;
24 }
```

Figure 2: BluetoothService.java

The code that handles the Bluetooth services also handles user intents. Meaning, when a user performs an unlock action on the Android app, it gets passed into a nested if statement in BluetoothService.java that digests the intent.

[3] https://www.youtube.com/watch?v=8AbZVjVgDCs
[4] http://www.benf.org/other/cfr/

```
1 @Override
2 public byte[] compose() {
3   if (this.openCode > 65535 && this.openCode <= 0xFFFFFFFFL) {
4     this.setOcLength(4);
5   } else if (this.openCode > 0xFFFFFFFFL && this.openCode <= 18446744073709551591L) {
6     this.setOcLength(8);
7   }
8
9   int n = 13;
10  byte[] arr = this.BuildPackage(4 + this.ocLength);
11
12  arr[10] = this.getCmdName().getBytes()[0];
13  arr[11] = this.getCmdName().getBytes()[1];
14  arr[12] =
15      (byte)
16          ((byte) ((byte) ((byte) ((byte) (0 | this.ysDisplay) << 2) | this.ysVoice) << 4)
17              | this.ocLength);
18
19  byte[] arrCode = HexByte.LongToHex(this.openCode, this.ocLength);
20  int arrCodeLen = arrCode.length - 1;
21  while (true) {
22    if (arrCodeLen < 0) {
23      arr[n++] = (byte) this.openSecond;
24      arr[n++] = this.CRC(arr, 10, -13 + arr.length);
25      arr[n] = (byte) (-1 ^ arr[n - 1]);
26      return arr;
27    }
28
29    arr[n] = arrCode[arrCodeLen--];
30    n++;
31  }
32 }
```

Figure 3: UL.java refactored

The UL classes compose() function gets called via the bluetoothSession.sendCMD(uL). Figure 3 shows a refactored version of this code for readability purposes. The main flaws exist in lines 19-31 where it fills in the open PIN code, open time, and CRC. In particular, lines 19, 29 and 30 show that the open PIN code is written out as HEX converted from a LONG INT converted from an ASCII String representation of the open PIN code, and finally written in reverse order. It also shows that there is not encryption, scrambling, XORing, or encoding done over the array, which means that the PIN can be pulled directly out of the packet.

```java
@Override
public byte[] compose() {
  byte[] arrby = new byte[16];
  System.arraycopy("largeapple".getBytes(), 0, arrby, 0, 10);
  String[] arrstring = this.deviceAddress.split(":");
  byte[] arrby2 = new byte[6];
  int n = 0;
  do {
    if (n >= 6) {
      System.arraycopy(arrby2, 0, arrby, 10, 6);
      byte[] arrby3 = new byte[8];
      System.arraycopy(arrby2, 0, arrby3, 2, 6);
      arrby3[0] = "E".getBytes()[0];
      arrby3[1] = "H".getBytes()[0];
      this.encreptData = new byte[8];
      WirelessEncrept.encrept(arrby3, arrby, 8, this.encreptData);
      byte[] arrby4 = new byte[10];
      arrby4[0] = "P".getBytes()[0];
      arrby4[1] = "D".getBytes()[0];
      System.arraycopy(this.encreptData, 0, arrby4, 2, 8);
      return arrby4;
    }
    arrby2[n] = HexByte.hexStringToByte(arrstring[n]);
    ++n;
  } while (true);
}
```

Figure 4: PD.java

Figure 4 shows that there is an artifact of a compose function using “encrept”-ion, which appears to be a self-implemented version of the Tiny Encryption Algorithm (TEA)[5]. While it never actually gets called, we noted that there was at least some level of development effort made for application-level encryption.

Wireless Analysis

Bluetooth Low Energy Protocol

Bluetooth Low Energy (BLE) is a wireless protocol that is built for low power usage and was introduced in the Bluetooth 4.0 specification[6]. Some BLE devices use so little power that they can operate for long periods of time off of a coin-cell battery[7] [8]. Bluetooth low energy devices save power by transmitting on certain intervals, transmitting less frequently in comparison to other wireless protocols, and entering low-power states when not in use.

[5] https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm
[6]https://www.bluetooth.org/docman/handlers/downloaddoc.ashx/doc_id=229737&usg=AFQjCNFY1IFeFAAWwimnoaWMsIRZQvPDSw&bvm=bv.135974163,d.cGw&cad=rja
[7] https://en.wikipedia.org/wiki/Bluetooth_low_energy
[8] http://www.ti.com/lit/an/swra347a/swra347a.pdf

The BLE protocol typically follows a common set of messages. The BLE device (server) continuously sends out advertising packets to let clients know its device name and the BLE address. A client can then request the server to reply with a list of services and characteristics are available. This list is referred to as the Generic Attributes (GATT) Profile. The two devices can then synchronize connection information, detailing how often to hop channels and the distance to hop. This hopping sequence allows the client to synchronize with the BLE device and transmit data. Three primary types of messages can be sent with or without an acknowledgement: read, write and notify. Figure 5 illustrates an initial set of BLE communications.

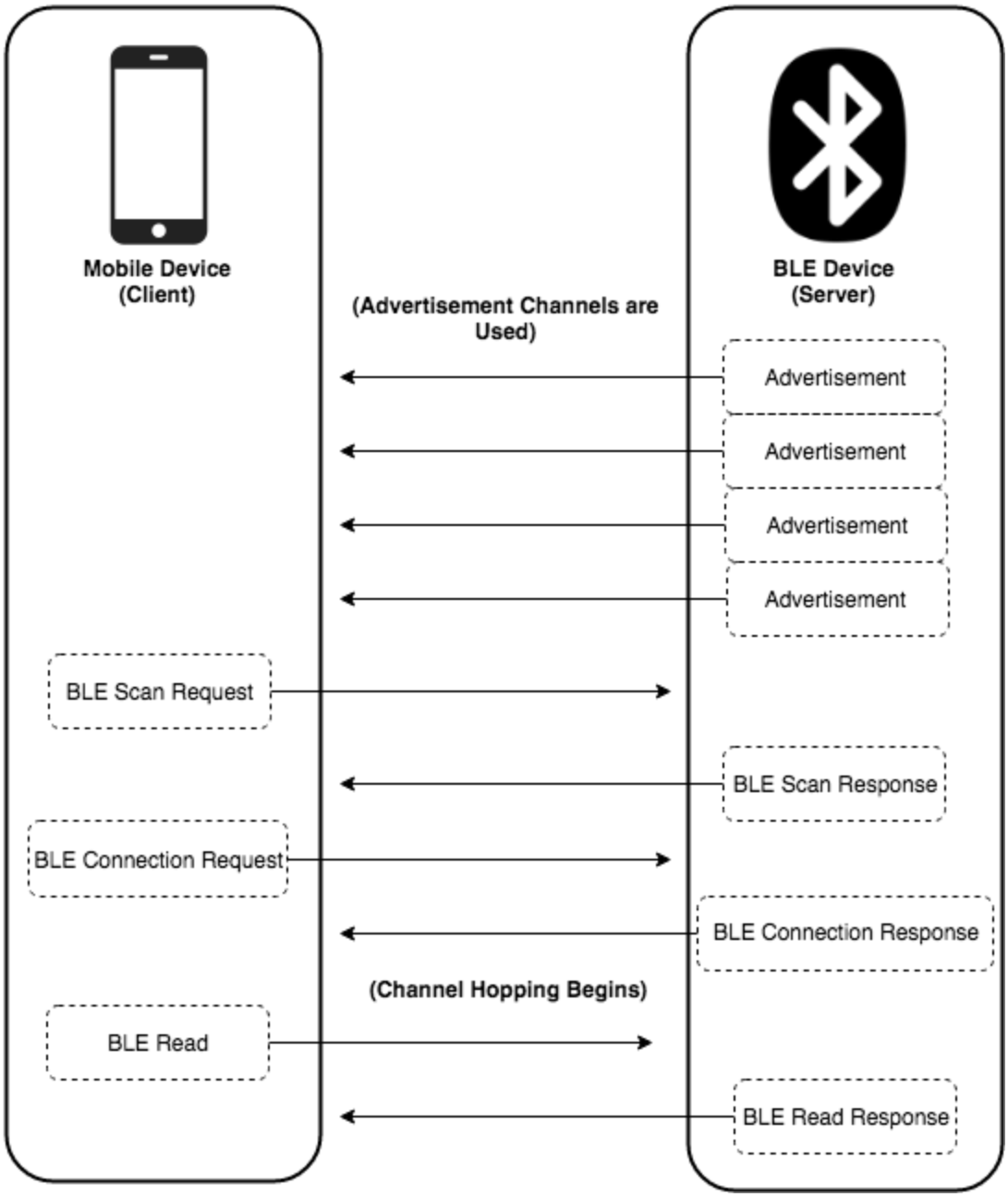
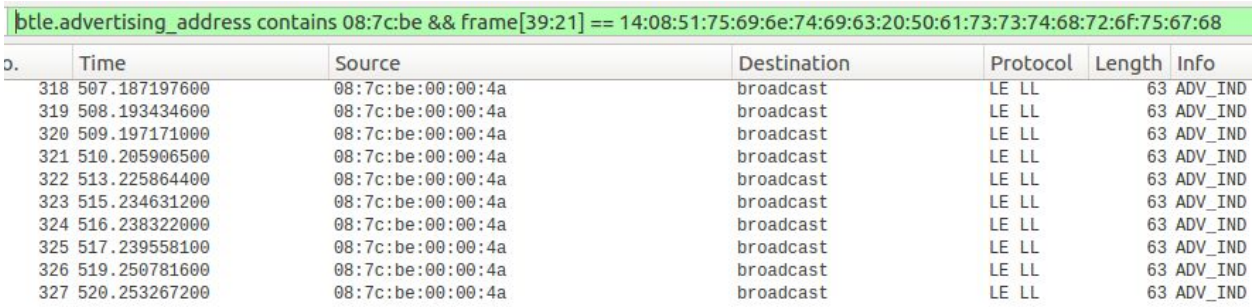


Figure 5: Bluetooth Low Energy basic communication example

The BLE protocol actually supports several encryption options such as Just Works, Passkey Entry, and Out of Band (OOB). OOB being the only good option, as the other two have known attacks that fully break the confidentiality and integrity of the encryption protocol[9] [10]. However, there are tradeoffs as it is difficult to devise a scheme to load a unique private key into a lock and mobile application that is cost-effective, fast, convenient, and secure. It is potentially for these reasons that many IoT companies choose not to implement encryption.

Wardriving

BLE traffic can be collected using an array of commodity software defined radios (SDRs) and wireless development devices. We used the Ubertooth One and Wireshark to capture BLE advertisements, scan requests, and scan response packets. This allowed us to find unique characteristics about the packet data such as the Device Name (“Quintic Passthrough”), Organization Unique Identifier (OUI) of the Bluetooth address (“08:7c:be”), packet header flags, and advertising data flags. The BLE scan response data includes additional unique service and characteristic information that uniquely identifies the Prologic B01. All of this information could be used to identify Prologic B01 BLE devices and the location of safes wirelessly.


btle.advertising_address contains 08:7c:be && frame[39:21] == 14:08:51:75:69:6e:74:69:63:20:50:61:73:73:74:68:72:6f:75:67:68

o.	Time	Source	Destination	Protocol	Length	Info
318	507.187197600	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
319	508.193434600	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
320	509.197171000	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
321	510.205906500	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
322	513.225864400	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
323	515.234631200	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
324	516.238322000	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
325	517.239558100	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
326	519.250781600	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND
327	520.253267200	08:7c:be:00:00:4a	broadcast	LE LL	63	ADV_IND

Figure 6: Ubertooth One and Wireshark wardriving filter

Figure 6 shows a filter being used to display BLE devices using a MAC address that starts with “08:7c:be”, certain flags are set, and the advertising packet contains “Quintic Passthrough”. This passive sniffing technique should give a relatively strong indication that the results are Prologic B01 devices. Sniffing can also provide a passive method of capturing scan responses containing unique characteristics. If that fails, performing an active scan can provide more targeted information. These techniques would allow an attacker to wardrive for Prologic B01s.

When testing the wardriving technique, we observed that the Ubertooth One and a 5dBi antenna could detect a Prologic B01 at a range of approximately 100 yards away. This test was performed where the listening device had line of site to the Prologic B01.

[9] https://www.usenix.org/system/files/conference/woot13/woot13-ryan.pdf
[10] https://lacklustre.net/projects/crackle/

Capturing and Analyzing BLE Traffic

We were interested in capturing and analyzing the BLE traffic of an unlock command. For this we used the Texas Instruments CC2540 and BLE Sniffer tool[11] because the Ubertooth One failed to follow the Prologic B01's BLE hopping sequence. From a BLE perspective, the unlock command is a characteristic write command. The first write has a static value of 0xFFFFFFFFFFFFFFFFFFFF5A5A5A5A5A5A5A5A5A, which could be used to passively identify an unlock request.

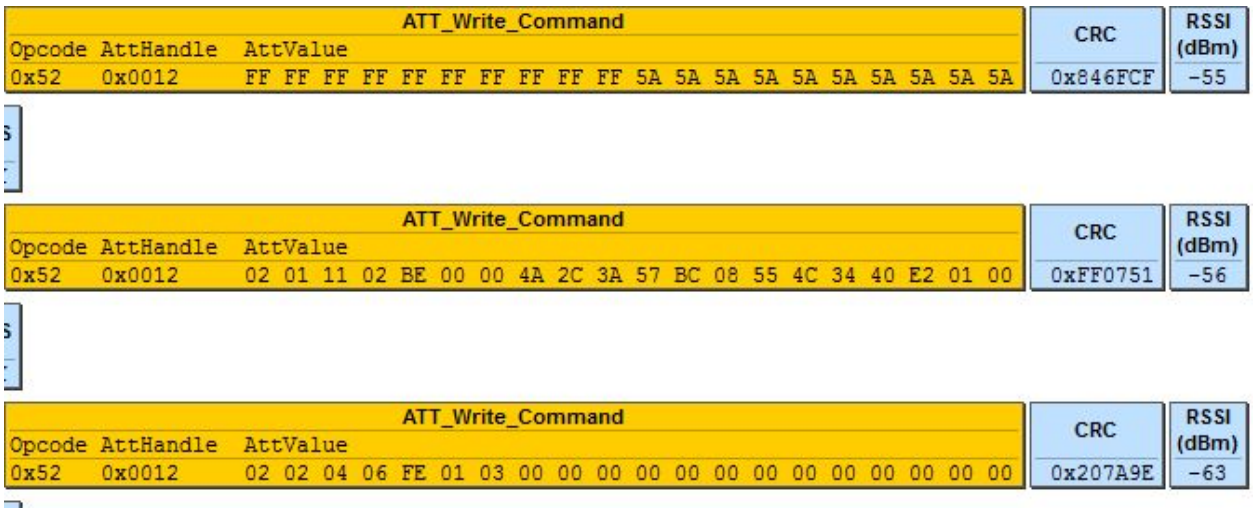


Figure 7: BLE Unlock Request 1, PIN: 123456

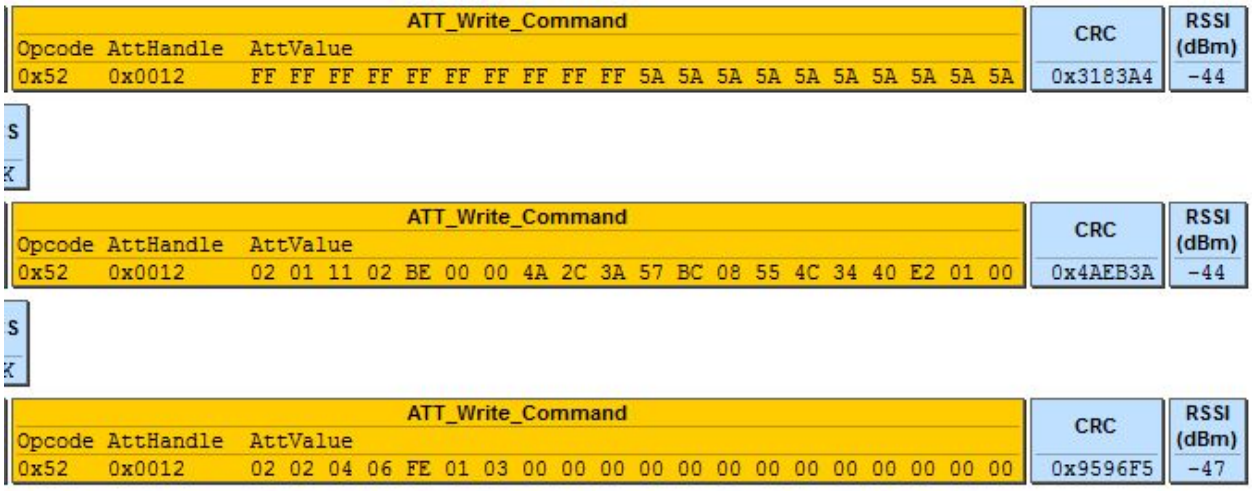


Figure 8: BLE Unlock Request 2, PIN: 123456

[11] http://processors.wiki.ti.com/index.php/BLE_sniffer_guide

Comparing the packets in Figures 7 and 8, it is clear to see that both packet payloads are the same. This means that the Prologic B01 is also vulnerable to a replay attack. If an attacker were to successfully sniff the unlock command they could replay the message to unlock the lock.

ATT_Write_Command		
Opcode	AttHandle	AttValue
0x52	0x0012	02 01 11 02 BE 00 00 4A 2C 3A 57 BC 08 55 4C 34 40 E2 01 00

ATT_Write_Command		
Opcode	AttHandle	AttValue
0x52	0x0012	02 02 04 06 FE 01 03 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 9: Application payload

```
>>> hex(long('123456'))
'0x1e240L'
```

Figure 10: PIN in hexadecimal format

The packet payload is prefixed with three byte headers, so the interesting data starts at the fourth byte of each message. The last four bytes of the receiver's (pink) and sender's (cyan) MAC address is included. The PIN (green) is parsed as a Long type and is sent in reverse order, which is illustrated in Figure 10. Finally, the open time (blue) is included that specifies how many seconds the lock should stay open for.

The BLE Sniffer tool has the ability to broadcast out packet data on a UDP port for further processing, which we configured to port 5000. We then wrote a Python service to listen on that port and process the packet data. We were able to grab unique packet signatures that would notify us when a Prologic B01 was being opened, how long the lock was open for, and the PIN that was used to open it.

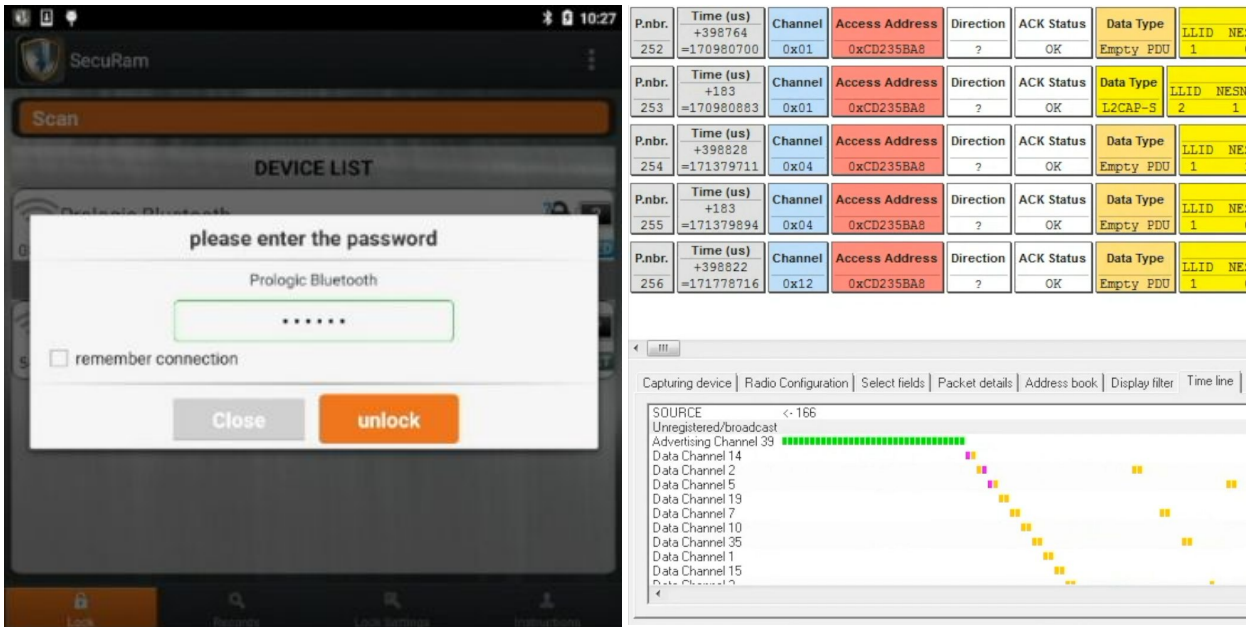


Figure 11: SecuRam Access Android application unlock command

Figure 12: TI BLE Sniffer tool capturing an unlock command

Figure 13: SecuRam Prologic B01 unlocked via SecuRam Access app

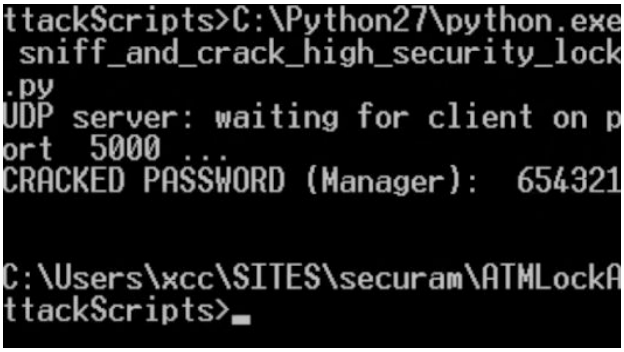


Figure 14: PIN extraction Python service processing data from TI BLE Sniffer tool

The images above illustrate the attack in chronological order. After wardriving for all Prologic B01 protected safes in an area, an attacker starts by capturing traffic for a specific target using the TI BLE Sniffer tool. A victim then uses the mobile application to unlock their lock (Figure 11), at which point the sniffer tool follows and captures the conversation (Figure 12). At around the same time the victim's safe unlocks (Figure 13) the Python service listening on port 5000 would have extracted the unlock time and PIN (Figure 14) and sent it to the attacker.

Conclusion

The consumer electronics industry has recently been pushing on creating wireless electronic locks. Being a new concept, there has been little to no standardization causing the security of these devices to suffer. While this has been a known issue for many residential-grade BLE locks, our findings show that similar vulnerabilities extend to commercial-grade BLE locks, which are intended to secure highly-sensitive property.

The SecuRam ProLogic B01 wireless protocol is insecure. The design weaknesses and security vulnerabilities outlined throughout this paper show that attackers can execute cheap and practical attacks to locate and map these devices, know when they are unlocked over BLE, and extract the PIN with which they were unlocked. This leads us to believe that some BLE locks, regardless of commercial-grading, are not enforced to meet standards that properly protect their consumers from wireless attacks, and should be avoided.

Appendix A.

Hardware Overview

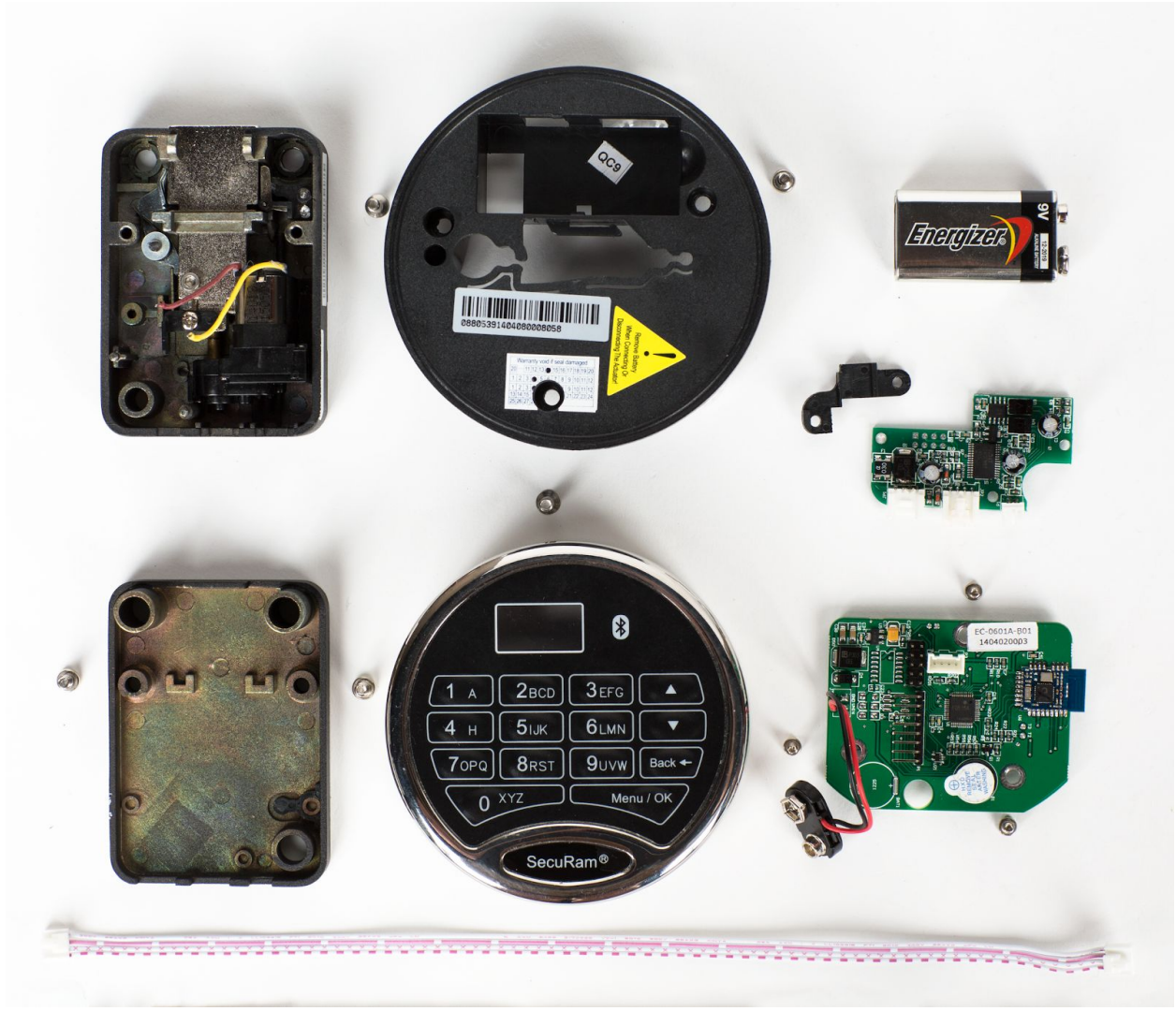


Figure 15: Entry Pad and Lock Body teardown

The image above shows the full breakdown of a SecuRam system. Pieces (a) and (b) make up the lock body. The DC motor in piece (a) is controlled by the lock body logic board (f). This logic board then is supplied powered and communicates to the entry pad logic board (g) via the four-pin communication cable (h).

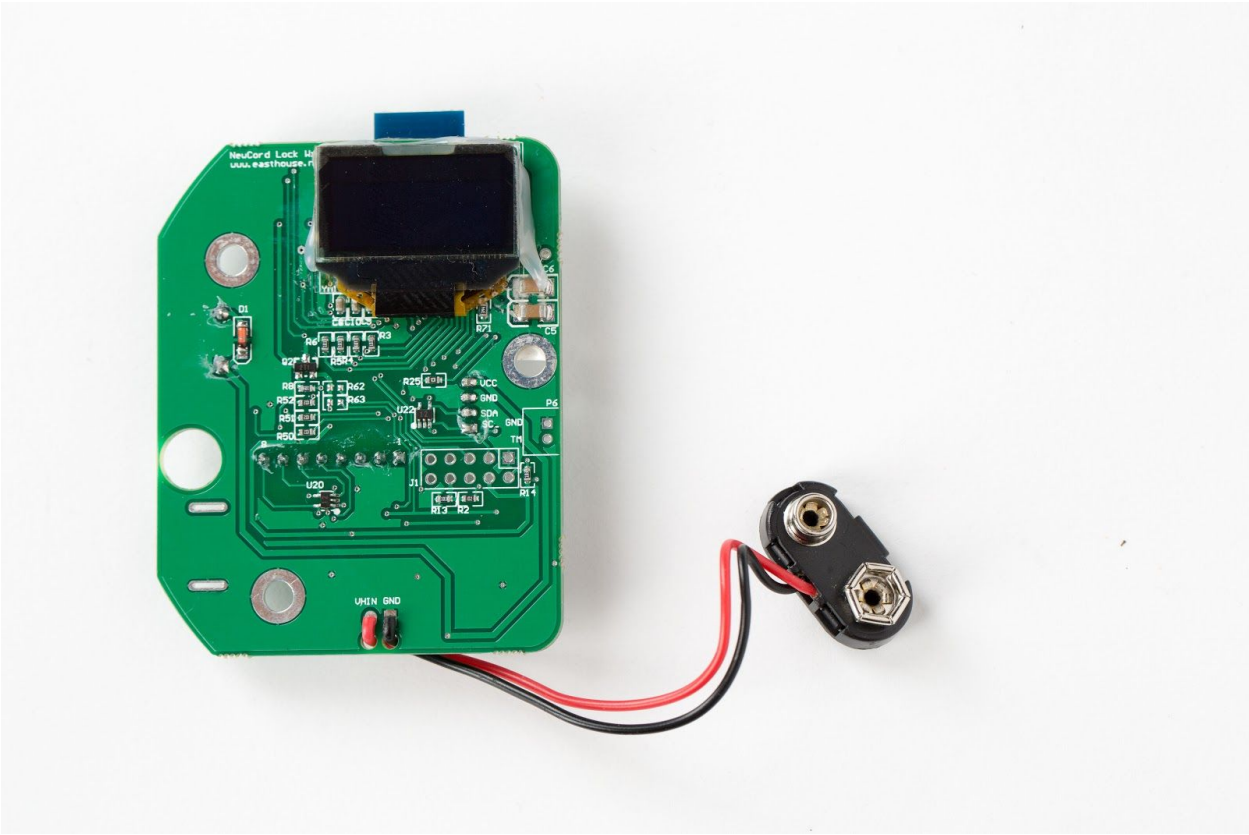


Figure 16: Front side of entry pad PCB

Front side of the entry pad circuit board shows power connections for a 9V battery and a 10-pin debugging interface (J1).

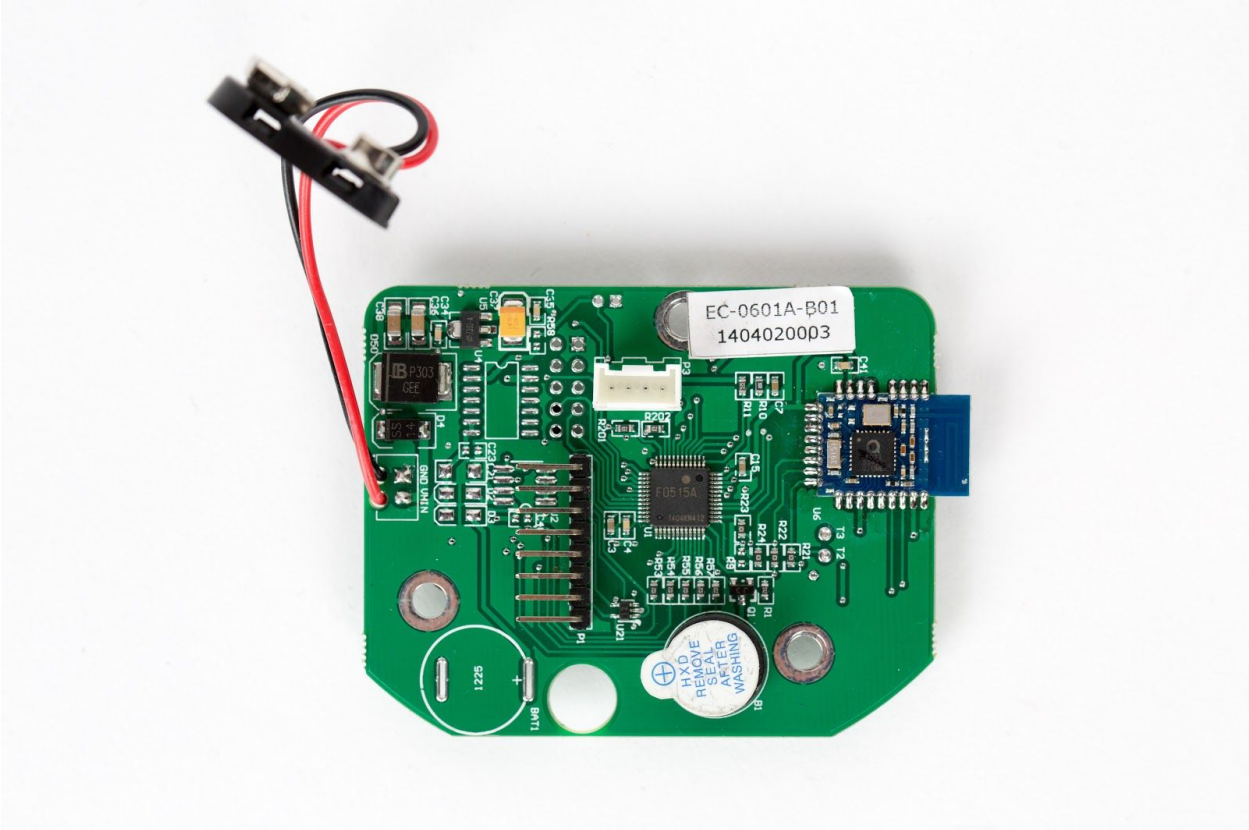


Figure 17: Back side of entry pad PCB

Back side of the entry pad is where the board's main MCU, a Renesas µPD78F0515A (U1), can be found. It's accompanied with an NXP QN902X SoC (U6) for BLE communication. Peripherals, such as the 8-pin keypad header (P1) and 4-pin serial interface (P3; ref above) to the lock-body are also located on this side. This serial interface is used for communication and to carry power to the lock body. There are unpopulated footprints (U2, U4, BAT1), which may have been used in previous revisions of this board, for debugging purposes, or other models of this entry pad.

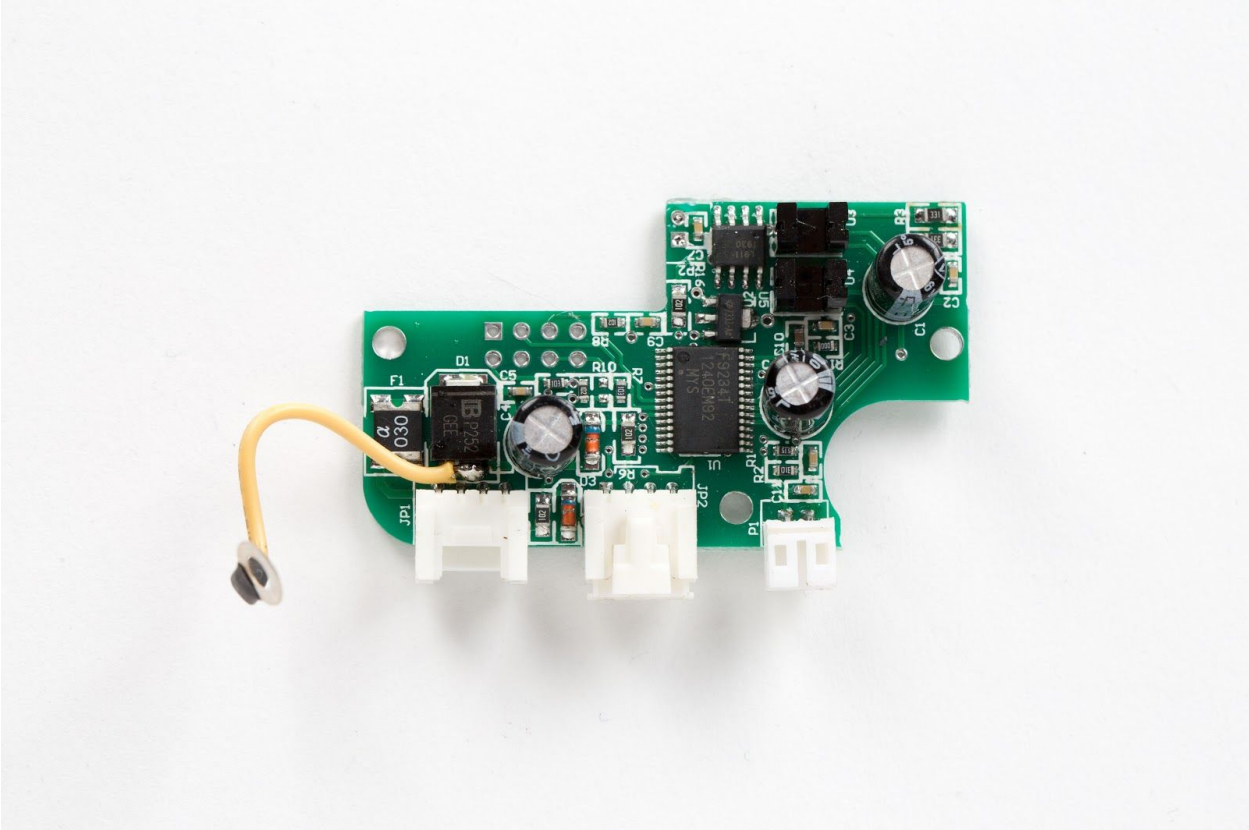


Figure 18: Front side of lock body PCB board

Front side of the lock body circuit board uses a less featureful Renesas μPD78F9234 MCU (U1). It is only known to communicate with a entry pad over the wired 4-pin serial communication interface (JP1), but there is another serial interface (JP2) adjacent to it that is hidden by the lock body cover. Additionally hidden is a 2-pin connector (P1) that is tied to pin 14 on the MCU. Its purpose is still undetermined.

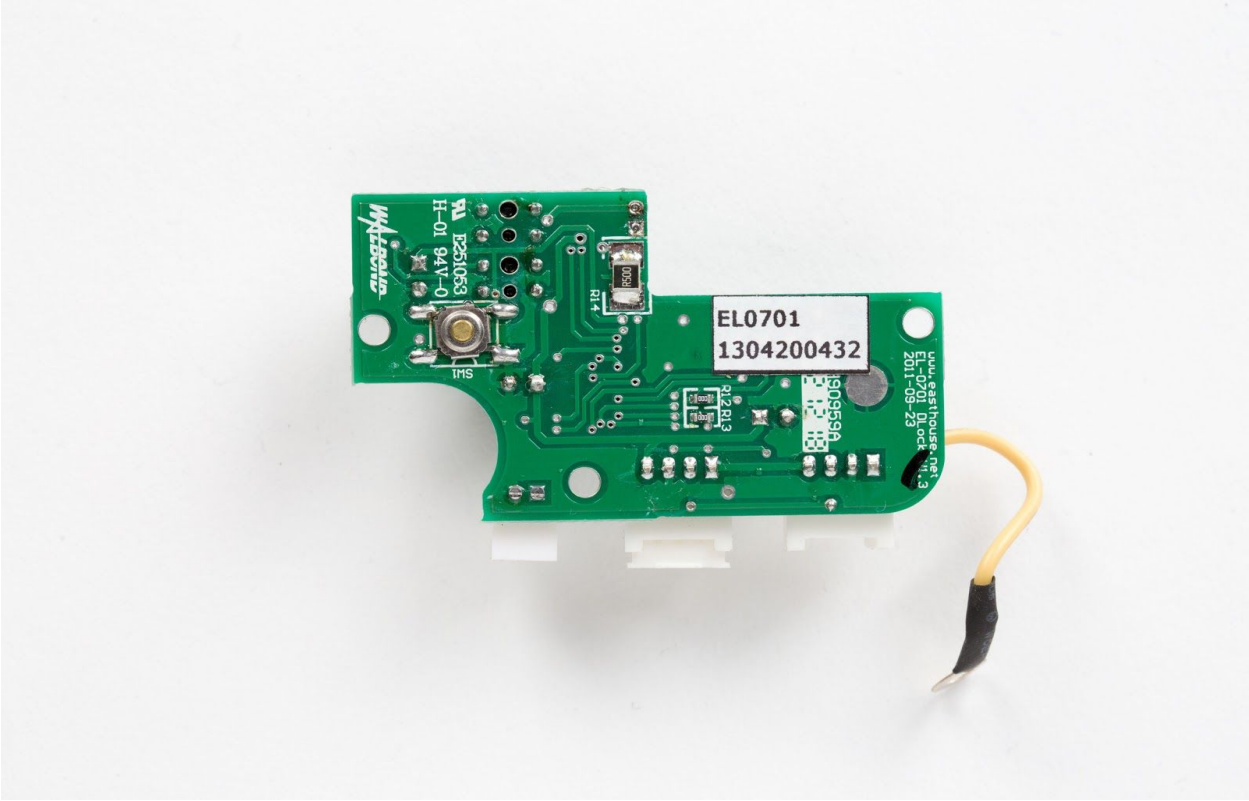


Figure 19: Back side of lock body PCB board

The back side of the lock body reveals another 8-pin debugging interface (J1; under sticker) and the hardware reset button (SW1). The reset button allows the lock body and entry pad to if the devices were to somehow fall out of sync with each other. The reset button is also used when connecting a preconfigured lock body with a new entry pad.

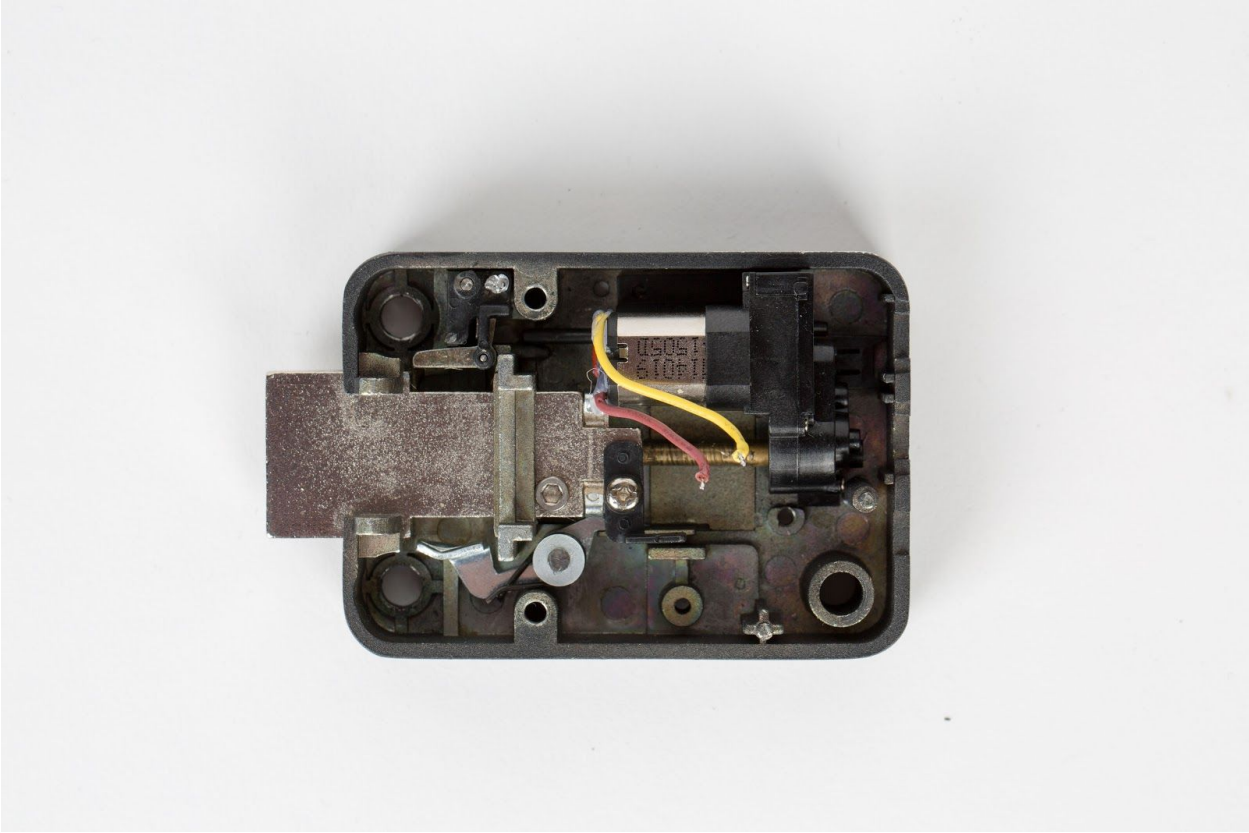


Figure 20: Disassembled lock body without PCB

The lock body is the only mechanical part of the system. It is composed of a DC motor and bolt. If 5 volts is applied across the red and yellow wires shown, the DC motor will retract the bolt and allow the security container to be opened.