2018

ChargePoint Home security research

Dmitry Sklyar, @d_skljar

Kaspersky Lab Security Services, @kl_secservices

Contents

1. Introduction

Home electric vehicle (EV) charging stations are a relatively new class of electronic device designed for installation at private locations, such as homes, private parking lots, etc.

Typically, the hardware and software design of these devices is based on a simplified version of public charging stations, as they don't have to deal with charge payments and advanced power grid management.

As the industry develops, manufacturers are adding new features to home EV charging stations, such as remote control of the charging process – something that can make these devices vulnerable to different types of attack.

Due to the comparative novelty of the industry, not much research has been conducted in this area. We found the paper 'OCPP Protocol: Security Threats and Challenges' by Christina Alcaraz, Javier Lopez and Stephen Wolthusen, and the 'Ladeinfrastruktur für Elektroautos: Ausbau statt Sicherheit' talk by Mathias Dalheimer at the Chaos Communication Congress 2017.

The OCPP Protocol: Security Threats and Challenges paper is devoted to the basic properties of one of the industry protocols, but does not include any specific device research. Mathias Dalheimer's talk is mostly devoted to security aspects of public stations, such as security of billing transactions and weaknesses in the RFID card data storage format.

2. Research target

This paper is devoted to research on the ChargePoint Home charging station (see Figure 1).

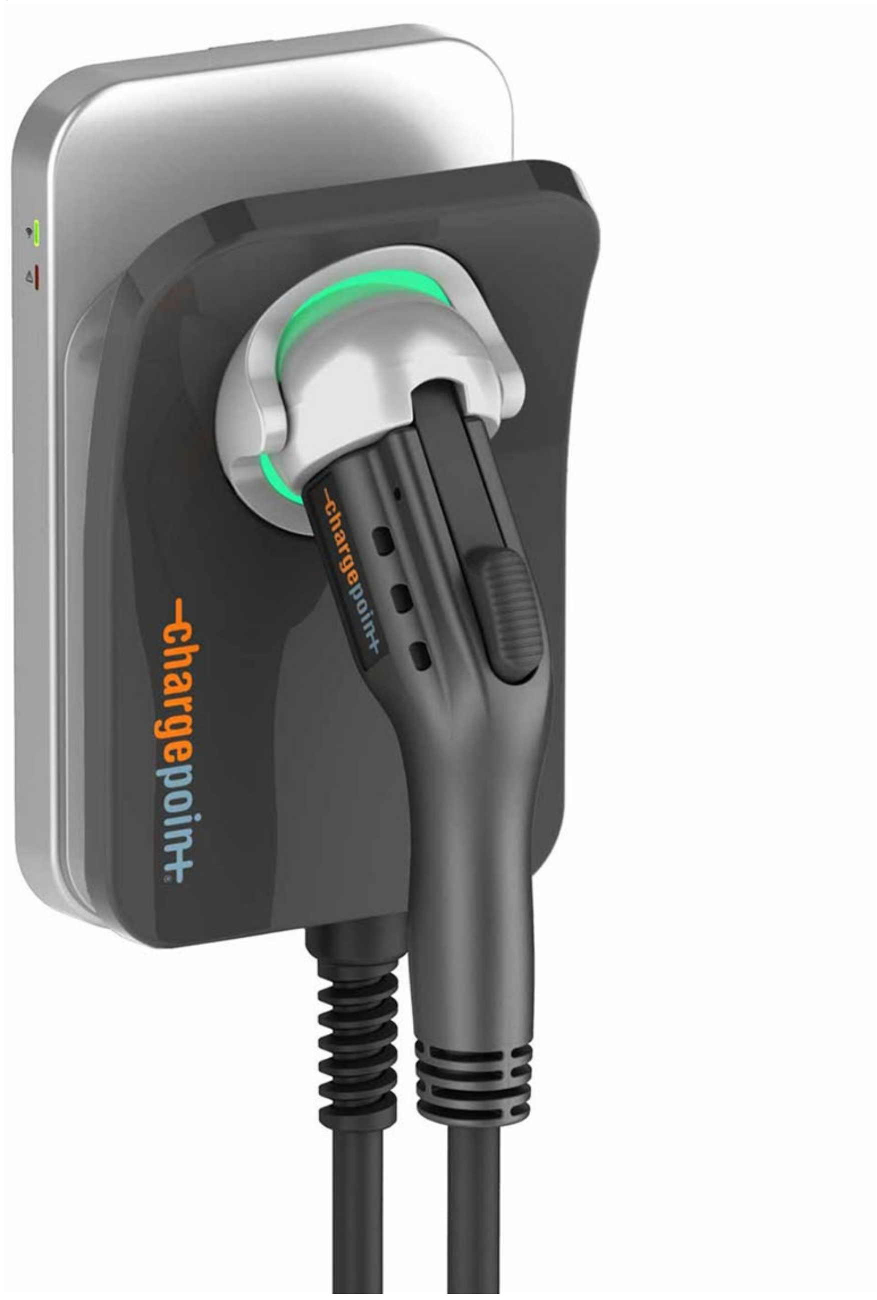


Figure 1.ChargePoint Home

The main technical characteristics of this charging station are:

- Supports Wi-Fi and Bluetooth protocols
- J1772 EV socket type
- Two output power levels – 16 Amp and 32 Amp
- Offers remote start, scheduling, reminder, energy tracking and other remote features through the ChargePoint mobile app

3. Mobile application analysis

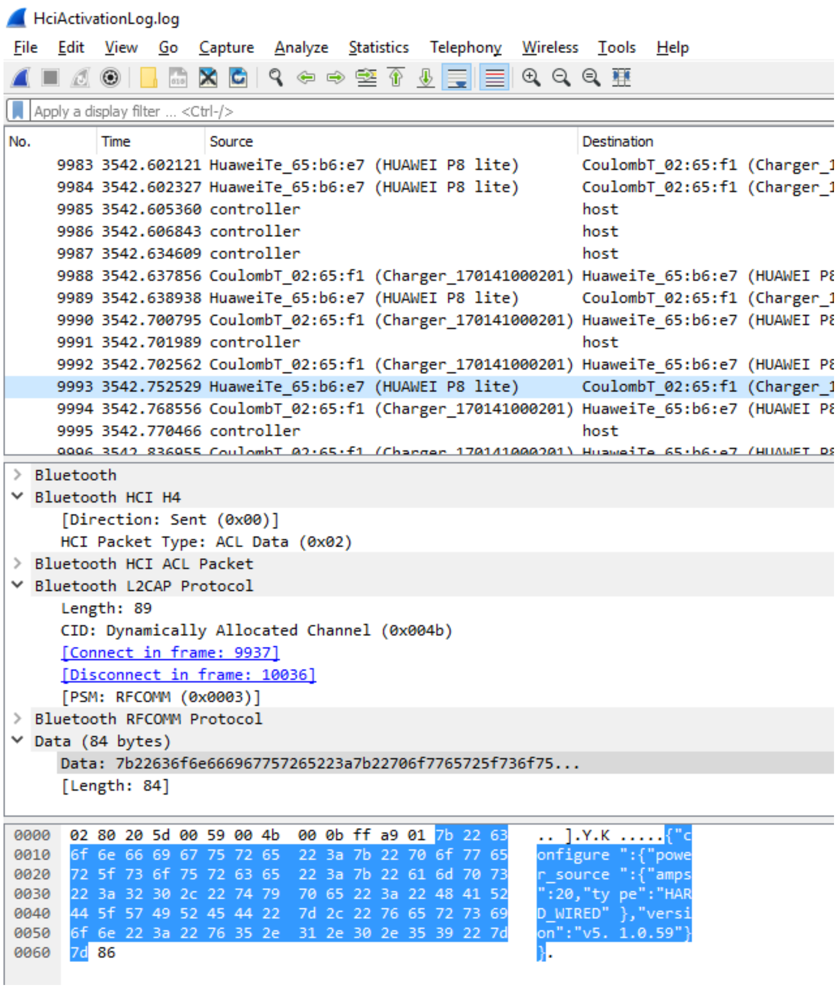
There is a mobile application available for both iOS and Android platforms, which was developed for managing the charging process in the ChargePoint ecosystem. This application is described as having two main functions:

1. Account management for public charging stations.
2. Registration and control of home charging stations.

In the case of station registration, the smartphone with the installed application connects to the station via Bluetooth, sets the station’s maximum consumable current and binds the station’s serial number to the application’s user account. After successful registration the station connects to the remote backend server, which translates the mobile application’s commands to station proprietary protocol commands and sends them to the station.

To explore registration data flows in more detail, we used a rooted smartphone with the hcidump application installed. With this application, we were able to make a traffic dump of the whole registration process, which can later be analyzed using Wireshark.

The Wireshark view of the dumped commands and responses is shown in Figure 2.


HciActivationLog.log
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.       Time        Source                                     Destination
     9983 3542.602121 HuaweiTe_65:b6:e7 (HUAWEI P8 lite)         CoulombT_02:65:f1 (Charger_1
     9984 3542.602327 HuaweiTe_65:b6:e7 (HUAWEI P8 lite)         CoulombT_02:65:f1 (Charger_1
     9985 3542.605360 controller                                 host
     9986 3542.606843 controller                                 host
     9987 3542.634609 controller                                 host
     9988 3542.637856 CoulombT_02:65:f1 (Charger_170141000201) HuaweiTe_65:b6:e7 (HUAWEI P8
     9989 3542.638938 HuaweiTe_65:b6:e7 (HUAWEI P8 lite)         CoulombT_02:65:f1 (Charger_1
     9990 3542.700795 CoulombT_02:65:f1 (Charger_170141000201) HuaweiTe_65:b6:e7 (HUAWEI P8
     9991 3542.701989 controller                                 host
     9992 3542.702562 CoulombT_02:65:f1 (Charger_170141000201) HuaweiTe_65:b6:e7 (HUAWEI P8
     9993 3542.752529 HuaweiTe_65:b6:e7 (HUAWEI P8 lite)         CoulombT_02:65:f1 (Charger_1
     9994 3542.768556 CoulombT_02:65:f1 (Charger_170141000201) HuaweiTe_65:b6:e7 (HUAWEI P8
     9995 3542.770466 controller                                 host
     9996 3542.836955 CoulombT_02:65:f1 (Charger_170141000201) HuaweiTe_65:b6:e7 (HUAWEI P8

> Bluetooth
v Bluetooth HCI H4
    [Direction: Sent (0x00)]
    HCI Packet Type: ACL Data (0x02)
> Bluetooth HCI ACL Packet
v Bluetooth L2CAP Protocol
    Length: 89
    CID: Dynamically Allocated Channel (0x004b)
    [Connect in frame: 9937]
    [Disconnect in frame: 10036]
    [PSM: RFCOMM (0x0003)]
> Bluetooth RFCOMM Protocol
v Data (84 bytes)
    Data: 7b22636f6e666967757265223a7b22706f7765725f736f75...
    [Length: 84]

0000  02 80 20 5d 00 59 00 4b  00 0b ff a9 01 7b 22 63   .. ].Y.K .....{"c
0010  6f 6e 66 69 67 75 72 65  22 3a 7b 22 70 6f 77 65   onfigure ":{"powe
0020  72 5f 73 6f 75 72 63 65  22 3a 7b 22 61 6d 70 73   r_source ":{"amps
0030  22 3a 32 30 2c 22 74 79  70 65 22 3a 22 48 41 52   ":20,"ty pe":"HAR
0040  44 5f 57 49 52 45 44 22  7d 2c 22 76 65 72 73 69   D_WIRED" },"versi
0050  6f 6e 22 3a 22 76 35 2e  31 2e 30 2e 35 39 22 7d   on":"v5. 1.0.59"}
0060  7d 86                                              }.
```

Figure 2. Wireshark view of the HCI dump

It turns out that the device supports the following Bluetooth commands:

- Get_version – returns the software version
- Configure – sets maximum charging current and device power supply type (plug-in or hardwired)
- Get_wifi_networks – returns a list of visible Wi-Fi networks with their signal strength and security type
- Connect_to_wifi – connects to the selected Wi-Fi network
- Register_with_nos – commands the device to send information about the smartphone's coordinates and the mobile application account ID to the remote backend server
- Shutdown_Bluetooth – disables the Bluetooth service

Further Bluetooth app analysis showed that this is the full list of supported commands. All commands are send to the device in a JSON packed format.

The application is written in Java, and can be easily decompiled.

During analysis of the application's activities, we noticed one with the rather intriguing name ResetToFactoryDefaultsFlashSequenceActivity. This activity's decompiled pseudocode is shown on Listing 1.

Listing 1. ResetToFactoryDefaultsFlashSequenceActivity pseudocode

```
private void a() {

this.a = new FlashSequence();

if (PermissionUtil.requestCameraPermission(this, true)) {

return;

}

try {

var1_1 = this.a.a();
```

This code requests camera permissions for some reason. We were able to run the activity independently on a rooted smartphone with the adb shell command (see Listing 2).

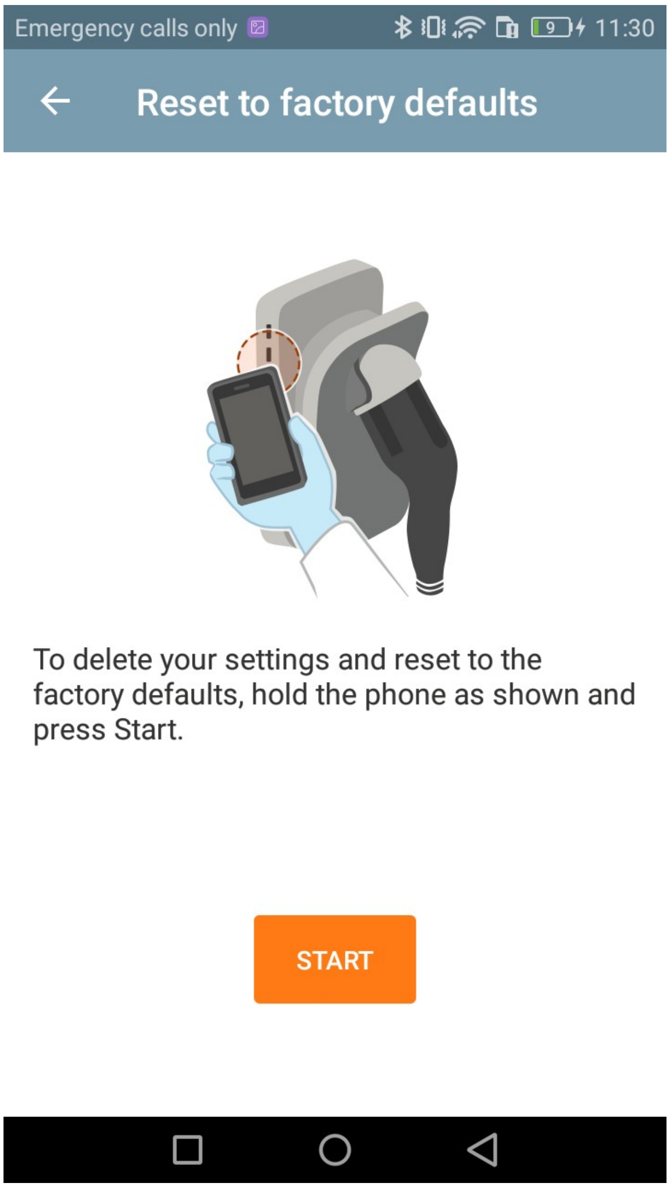
Listing 2. Activity invocation

```
adb -d shell

$ su

$ am start -n com.coulombtech/com.cp.ui.activity.homecharger.settings.reset.ResetToFactoryDefaultsActivity
```

Running this activity leads to the control screen shown in Figure 3.


Emergency calls only 11:30

Reset to factory defaults

To delete your settings and reset to the factory defaults, hold the phone as shown and press Start.

Figure 3. “Reset to factory defaults” screen

When the “START” button is pressed, the flash of the smartphone’s camera starts playing a special blinking pattern. There is a small photodiode window located on the bottom of the device. If the flash is pointed towards that window while playing the pattern, the device will perform a factory reset after the next reboot. This results in the Wi-Fi and user account settings being wiped. Our subsequent investigations show that there is only one pattern that can be recognized by the device, so no additional commands could be received using the photodiode.

4. Hardware revision

After unscrewing and removing the front panel, we found that the device consists of two separate PCBs (see Figure 4).

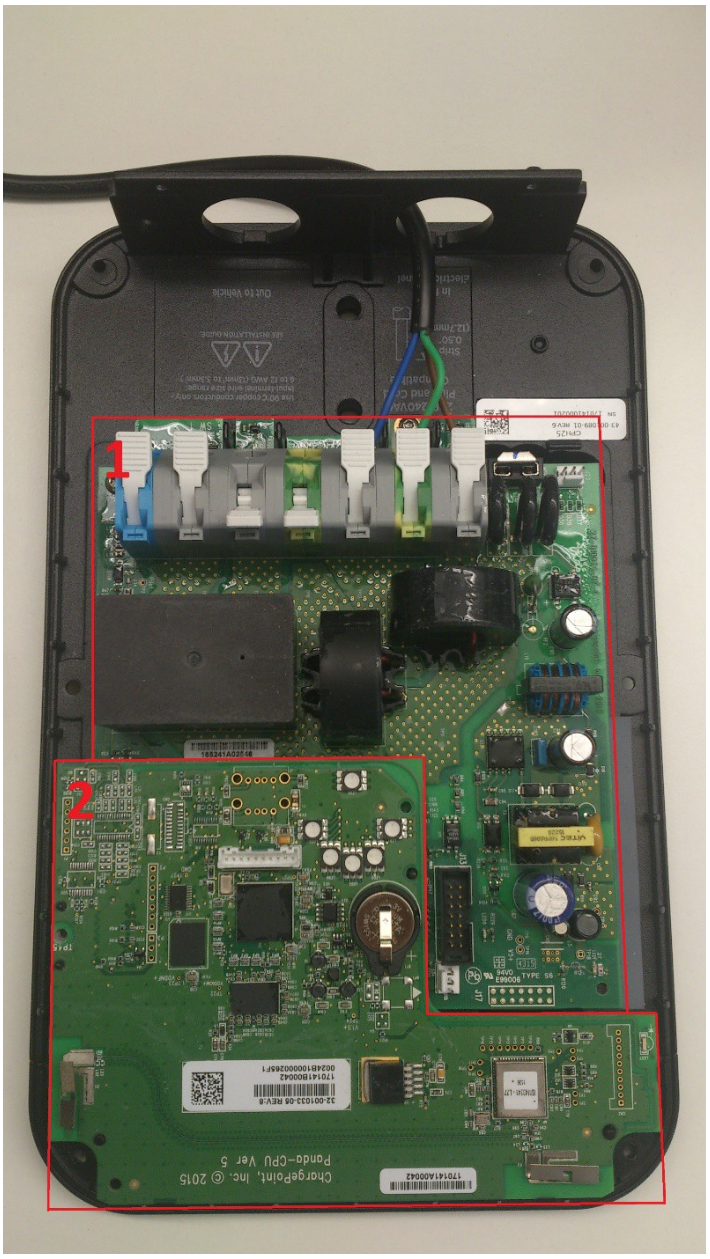


Figure 4. ChargePoint Home with the front panel removed

The PCBs are connected to each other with a proprietary connector. For the sake of clarity, we will refer to the PCB denoted by “1” in Figure 4 as the Power board, and to the PCB denoted by “2” as the Panda board. The name Panda board is written in silkscreen on the top side of the second board.

The Power board with its main components is shown in Figure 5.

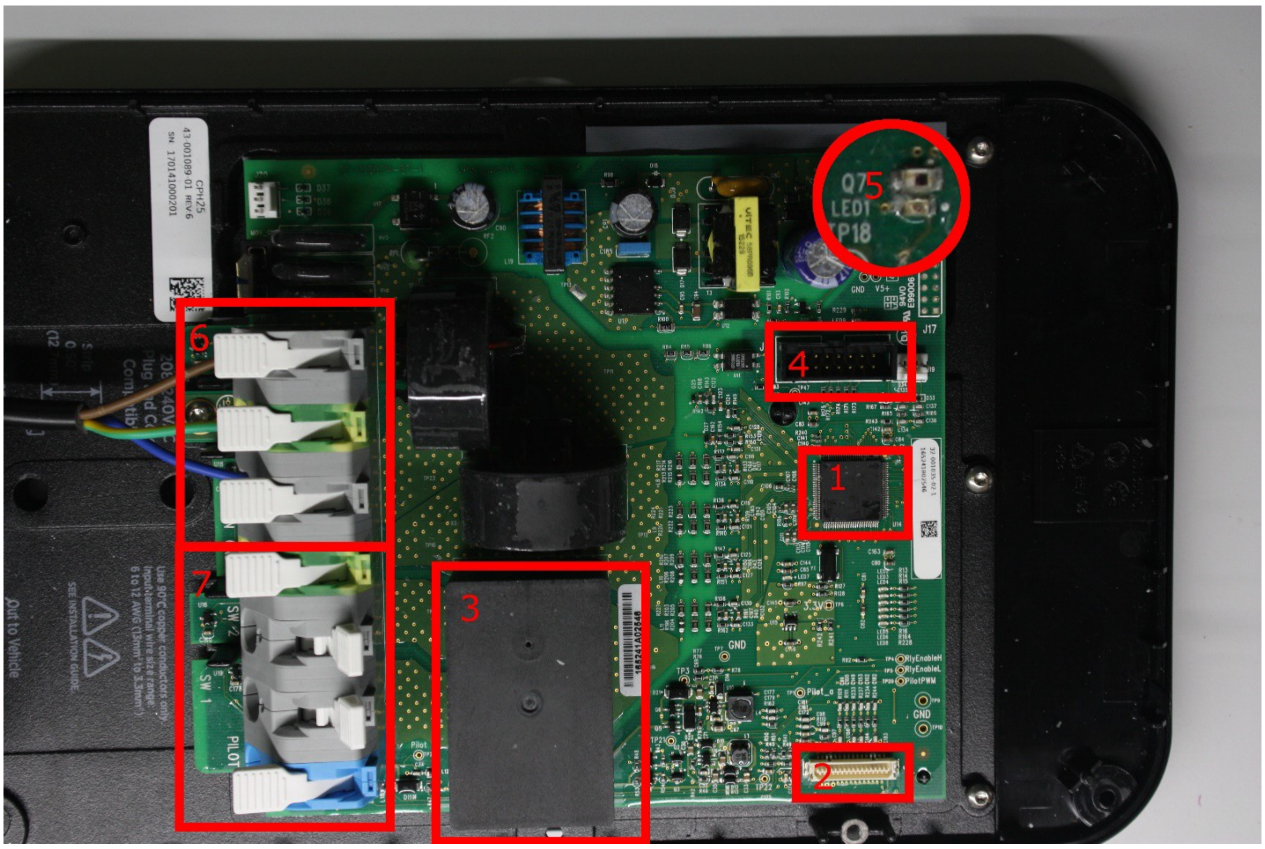


Figure 5. The Power board and its components

The following components are denoted by numbers in Figure 5.

1. MCU TI 6BATG4MSP430 F67691
2. Connection socket
3. Mechanical relay TE T92S7D12-12
4. Debug socket
5. LED and photodiode
6. Power plug terminal strip
7. Vehicle outlet terminal strip

This board is mainly used for controlling current commutation between an electrical network and a vehicle's outlet. Photodiode pattern recognition is also performed by this board. If the pattern is recognized, one dedicated GPIO pin in the connection socket will be turned on. This is how a factory reset is triggered. We didn't spend much time on this board during the research, so it may be a subject for further investigations.

The Power board is based on an MCU developed by Texas Instruments, which has the MSP430 architecture. This architecture is mainly used for electricity management and control applications and has open documentation, i.e., all specifications are available on the Texas Instruments website.

The Panda board with its main components is shown in Figure 6.

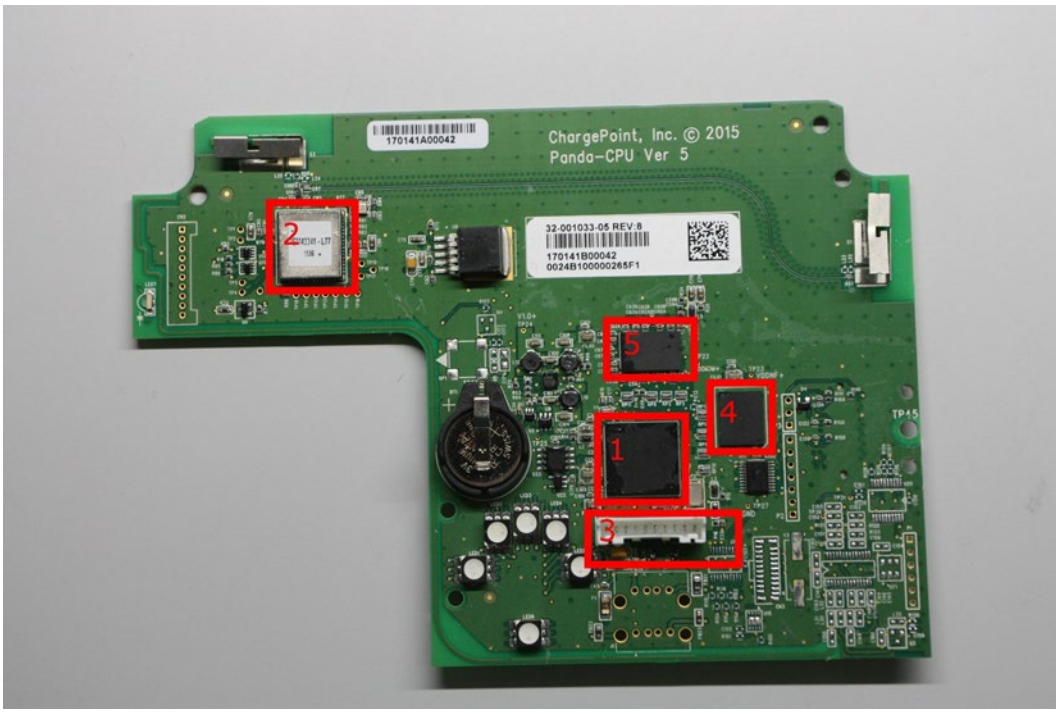


Figure 6. The Panda board and its components

The following components are denoted by numbers in Figure 6.

1. MPU Atmel AT91SAM9N12
2. Wireless communication module ISM43341-L77
3. JTAG socket
4. External DDR RAM 1 GB Micron 6WM17 D9RZT
5. NAND FLASH 512 MB Micron 4XD12 NW196

This board is based on an MPU with ARM architecture, which is designed for external firmware storage and RAM connection. The firmware is stored on the Flash NAND chip designed by Micron. All wireless communications are provided by the ISM communications module. It is designed on the basis of a Cypress chip that supports Wi-Fi, Bluetooth and NFC protocols. The NFC protocol is used for communications with payment cards. We didn't find any software components designed for that in the ChargePoint Home station, so this protocol may be used in other ChargePoint Inc. products that utilize the same hardware design.

This Board has a debug socket. We found a JTAG interface on this socket with the JTAGulator board. The socket pinout is shown in Figure 7.

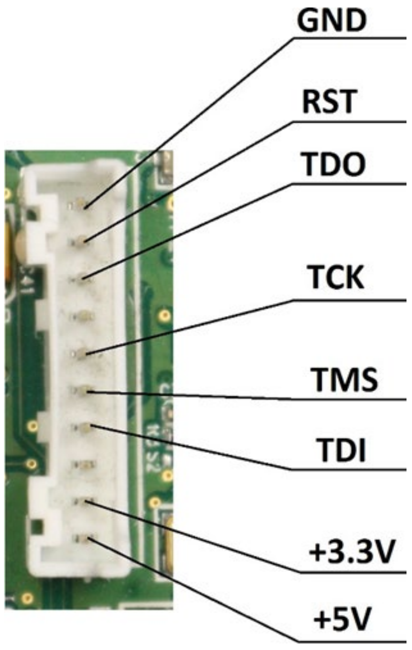


Figure 7. Debug socket pinout

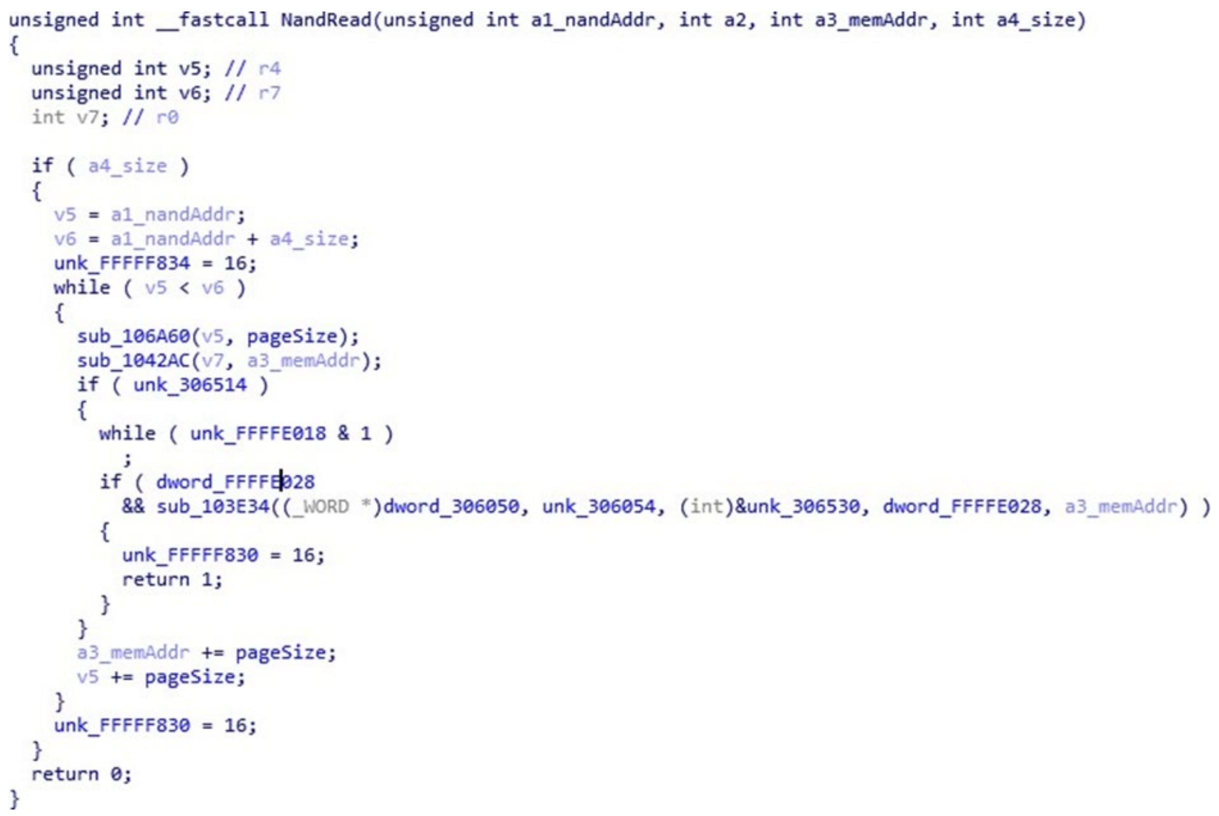
5. NAND image downloading

The NAND content can be read and written by OpenOCD scripts. A collection of such scripts designed for different processor families is included in the OpenOCD distribution. There are several scripts for Atmel processors, but none of them supports our device.

As the Panda board is based on an AT91 MPU, we assumed it uses the standard boot sequence, which is proposed by Atmel. This sequence consists of four stages:

1. AT91 BootROM. This is a small bootloader that is burned into an on-chip mask ROM. It can only work with an internal SRAM.
2. AT91Bootstrap. This is a second stage open-source bootloader that can initialize an external RAM and thus load bigger images.
3. U-Boot. This is a well-known Linux bootloader.
4. Linux kernel.

We decided to use the AT91 BootROM code to read the NAND memory content, as this method is simple and universal. It can later be applied to a large number of devices based on the same MPU family. AT91 BootROM isn't an open source bootloader, so we read it via JTAG and analyzed it. As a result of our analysis, we found the address of the NAND read procedure. The pseudocode of this procedure is shown in Figure 8.



```
unsigned int __fastcall NandRead(unsigned int a1_nandAddr, int a2, int a3_memAddr, int a4_size)
{
  unsigned int v5; // r4
  unsigned int v6; // r7
  int v7; // r0

  if ( a4_size )
  {
    v5 = a1_nandAddr;
    v6 = a1_nandAddr + a4_size;
    unk_FFFFF834 = 16;
    while ( v5 < v6 )
    {
      sub_106A60(v5, pageSize);
      sub_1042AC(v7, a3_memAddr);
      if ( unk_306514 )
      {
        while ( unk_FFFFE018 & 1 )
          ;
        if ( dword_FFFFE028
          && sub_103E34((_WORD *)dword_306050, unk_306054, (int)&unk_306530, dword_FFFFE028, a3_memAddr) )
        {
          unk_FFFFF830 = 16;
          return 1;
        }
      }
      a3_memAddr += pageSize;
      v5 += pageSize;
    }
    unk_FFFFF830 = 16;
  }
  return 0;
}
```

Figure 8. NandRead procedure pseudocode

This procedure takes four arguments:

1. a1_nand_addr – address on the NAND chip.
2. a3_memAddr – address of the buffer for incoming data in the internal SRAM.
3. a4_size – amount of bytes to read.
4. a2 – dummy argument.

To read the whole content of the NAND chip, we need to set a breakpoint after the NandRead procedure's call in the AT91 BootROM code and wait until this breakpoint is hit. After that, we need to cyclically pass the execution flow to the NandRead procedure with the appropriate arguments' values set, and dump the procedure's output buffer to the host.

5.1. NAND image structure

We analyzed the dumped image with the binwalk tool, and it showed that, among other things, the image contains a UBI file system. In further analysis, we manually parsed the UBI header's data and discovered that the image contains five UBI volumes. Two of these volumes seem to be Linux root file system volumes. We mounted them to a Linux system as part of the emulated NAND chip to analyze their content, and found mounting scripts containing information about the image partitioning (Table 1).

Table 1. NAND image partitioning

⬜	-	Second stage bootloader
🟦	-	Linux image 1
🟧	-	Linux image 2
🟨	-	Additional UBI volumes
🟩	-	Additional partitions

0x00000000 – 0x00003940	AT91-bootstrap
0x00280000 – 0x002e9ee0	U-boot
0x00380000 - 0x003a0000	Kernel args section
0x00480000 – 0x007a0000	Kernel v.3.10.0
0x00c80000 – 0x08c80000	UBI rootfs volume
0x08c80000 – 0x08e80000	Parameter section

0x08e80000 – 0x0ae80000	UBI opt volume
0x0ae80000 – 0x0ee80000	UBI data volume
0x0ee80000 – 0x0ef80000	U-boot
0x0ef80000 – 0x0f080000	Kernel args section
0x0f080000 – 0x0f880000	Kernel v.3.10.0
0x0f880000 – 0x17880000	UBI rootfs volume
0x17880000 – 0x17a80000	Parameter section
0x17a80000 – 0x1fa80000	UBI otavdata volume
0x1fa80000 – 0x20000000	SSH key recovery partition

There are two full Linux images located at offsets 0x280000 and 0xee80000. Each of them consists of five areas: U-Boot bootloader image, kernel, arguments section, proprietary parameters section and root file system. The device can switch to the alternate image at the next boot if it decides that something went wrong. There are also three additional UBI partitions that are mounted to the root file system during boot and initialization process. These volumes are not duplicated. In addition, there is an AT91-Bootstrap image and an SSH key recovery partition, which will be discussed in section 7.3.

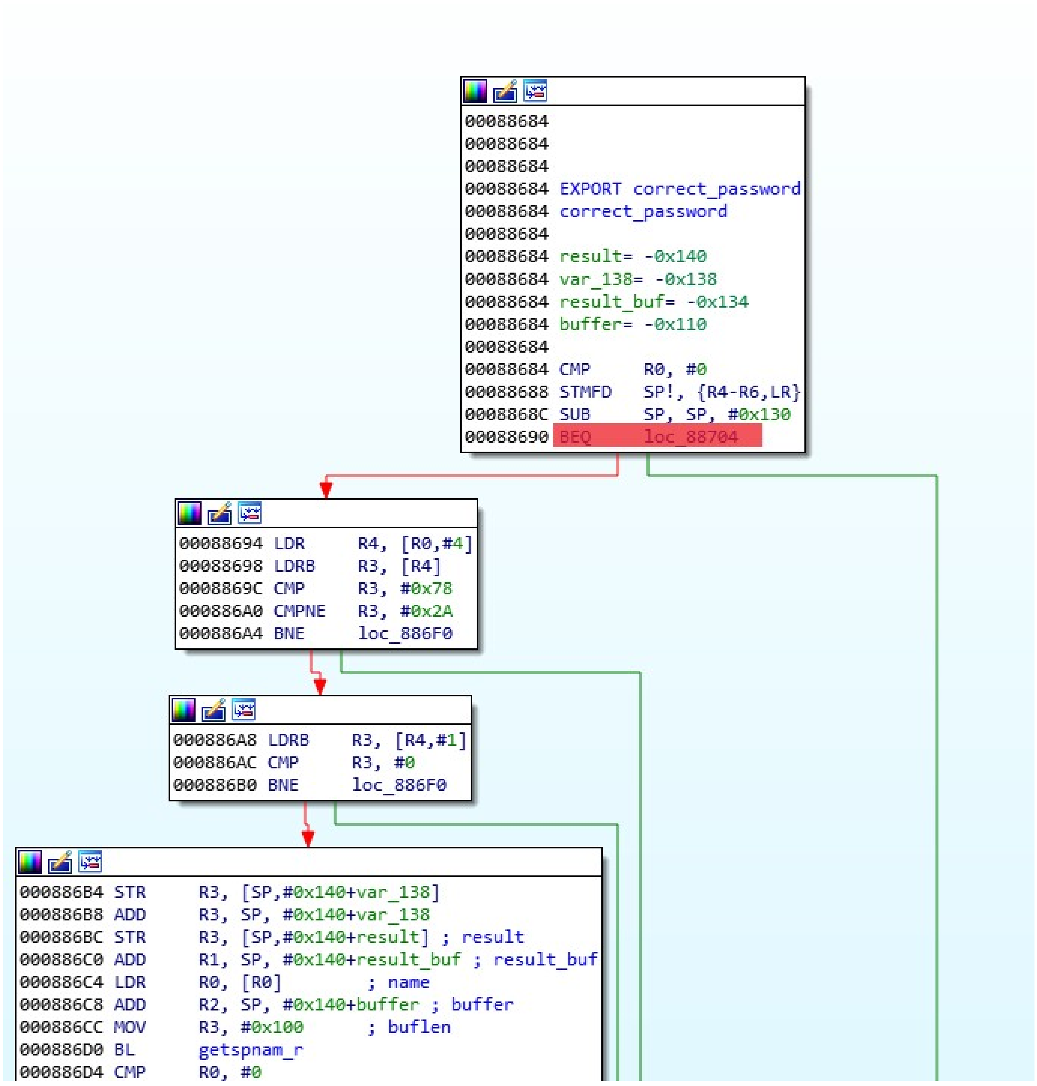
Parameter sections included in the Linux images are in proprietary format. They contain records with the following fields:

1. 4-byte parameter name
2. 2-byte parameter value length
3. Parameter value

All parameters contained in the current parameter section are parsed at boot time with the cfg_decoder binary and saved as separate files in the /var/config folder.

6. Root access

For further investigation, we connected the device to our Wi-Fi network. It had an open telnet port with password authentication. To bypass authentication, we used JTAG to inject our code into the password verification procedure. Our Linux image is based on the busybox binary, so this procedure is located in the login module of this binary. Figure 9 shows the procedure in the disassembled form with the highlighted BEQ instruction. If we change this instruction to the BNE instruction, we will bypass authentication with an incorrect password.



```
00088684
00088684
00088684
00088684 EXPORT correct_password
00088684 correct_password
00088684
00088684 result= -0x140
00088684 var_138= -0x138
00088684 result_buf= -0x134
00088684 buffer= -0x110
00088684
00088684 CMP     R0, #0
00088688 STMFD   SP!, {R4-R6,LR}
0008868C SUB     SP, SP, #0x130
00088690 BEQ     loc_88704
```

```
00088694 LDR      R4, [R0,#4]
00088698 LDRB     R3, [R4]
0008869C CMP      R3, #0x78
000886A0 CMPNE    R3, #0x2A
000886A4 BNE      loc_886F0
```

```
000886A8 LDRB     R3, [R4,#1]
000886AC CMP      R3, #0
000886B0 BNE      loc_886F0
```

```
000886B4 STR      R3, [SP,#0x140+var_138]
000886B8 ADD      R3, SP, #0x140+var_138
000886BC STR      R3, [SP,#0x140+result] ; result
000886C0 ADD      R1, SP, #0x140+result_buf ; result_buf
000886C4 LDR      R0, [R0]        ; name
000886C8 ADD      R2, SP, #0x140+buffer ; buffer
000886CC MOV      R3, #0x100      ; buflen
000886D0 BL       getspnam_r
000886D4 CMP      R0, #0
```

Figure 9. correct_password function

To patch the code in the RAM, we need to set a breakpoint somewhere before the target instruction execution and wait until this breakpoint is hit. As the target MPU works in protected mode with virtual addressing, a breakpoint is set on a virtual address and there might be several ‘false positive’ breakpoint hits caused by other processes. We need to recognize and skip these hits by checking the constant memory content, for example, a signature that is located somewhere in the .text section.

After bypassing authentication, we added a permanent user to the device.

7. Software analysis

Table 2 contains information about processes that are responsible for wireless communications.

Table 2. Processes that are responsible for wireless communications

Process name	Description
stunnel	Listens for incoming connections on TCP ports 443 and 55557; port 443 is used for HTTPS, port 55557 – for an encrypted telnet service
busybox	Listens for incoming connections on TCP port 23; provides a telnet service
cpsrelay	Connects to the remote server, implements main communication channel with the backend infrastructure that is used for remote controlling and monitoring
sshrevtunnel.sh	Connects to the remote server, implements additional communication channel with the backend infrastructure
btclassic	Implements main communication channel via Bluetooth that is used by the smartphone application
onboardee	Implements additional communication channel via Bluetooth

7.1. HTTPS server

There is an HTTPS server implemented with the thttpd daemon combined with the stunnel daemon for TLS support. Stunnel configuration is shown in Listing 3.

Listing 3. Stunnel configuration file

```
#
# Stunnel configuration file for handling incoming SSL/TLS
# connections to proxied services running locally on a Smartlet.
#
CAfile = /var/config/.keys/ca.crt
CApath = /etc/pki/certs
CRLFile = /etc/pki/crls/ca.crl
cert = /etc/pki/certs/system.crt
key = /etc/pki/keys/system.key
key_passphrase_type = 2
verify = 2
ciphers                                                                    =
ALL:!aNULL:!ADH:!eNULL:!LOW:!EXPORT:!SSLv2:RC4+RSA:+HIGH:+MEDIUM
#compression = zlib
# TODO: need to find out why background does not work.
# May be due to differing OpenSSL versions on server and Smartlet.
#foreground = no
foreground = yes
debug = 0
# Uncomment to enable debug.
#debug = daemon.info
client = no
pid = /var/tmp/stunnel-in.pid
session = 20
max_clients = 8


[thttpd]
```

```
accept = https

connect = 127.0.0.1:55555

TIMEOUTidle = 60


[telnet]

accept = 55557

connect = 127.0.0.1:telnet

TIMEOUTidle = 300
```

The stunnel service is configured for mutual authentication, so we can’t access the HTTPS server without an appropriate certificate, and it uses certificates from the parameters section: the system.crt certificate as a server certificate and the ca.crt certificate as a certificate authority for client certificate validation.

Initially, the stunnel binary supports only unencrypted certificates, and doesn’t support the key_passphrase_type configuration option, but the system.crt certificate’s private key is stored in encrypted form. We assumed that the stunnel binary was built from modified sources. After analyzing the stunnel binary file, we discovered that it was compiled with a static library that supports certificate decryption. Furthermore, every binary that supports incoming or outgoing TLS connection is statically linked with this library.

Among others, this library includes a function for generating the certificate decryption key. This function takes paths to master_key, system_mac and system.phr files as parameters. These files are derived from the parameter sections in the NAND chip. The system.phr file contains only one byte, which sets the certificate’s encryption mode. In our case, this byte is set to 2. In this research we didn’t analyze the key generation algorithm, and downloaded the key for the RAM via JTAG after that function has been executed. This is a 64-character string that consists of digits and small Latin characters.

Also, we discovered that the system.crt certificate is signed with the ca.crt certificate. When an SSL connection is established, the stunnel binary only verifies the certificate’s signature. Therefore, we are able to connect to the https server using the system.crt certificate. This issue could have been avoided if another unknown certificate was used for system.crt certificate signing.

Listing 4 shows the thttpd configuration file.

Listing 4. thttpd configuration file

```
# BEWARE : No empty lines are allowed!
# This section overrides defaults
dir=/
chroot
user=nobody
logfile=/dev/null
pidfile=/var/run/thttpd.pid
#charset=UTF-8
cgipat=/usr/bin/getsrvr|/usr/bin/uploadsm|/usr/bin/dwnldlogsm
port=55555
#Only localloopback addres
host=127.0.0.1
# This section _documents_ defaults in effect
# port=80
# nosymlink         # default = !chroot
# novhost
# nocgipat
# nothrottles
# host=0.0.0.0
# charset=iso-8859-1
```

The thttpd daemon is used only for the CGI (Common Gateway Interface) implementation and supports invocation of three ELF files: uploadsm, dwnldlogsm and getsrvr. This CGI interface seems to be redundant, and was left on the device as a part of the software subsystem that is utilized in more complex devices, such as charging stations that are intended for public use. In the latest firmware version, https is enabled only on the localhost interface and unreachable from a Wi-Fi network.

The thttpd daemon is launched with “nobody” user rights, but all CGI binaries have the SUID bit set. This results in these binaries being executed with the highest system privileges, but all their child processes will again have “nobody” user rights.

7.1.1. The uploadsm CGI binary

The uploadsm binary is used to upload files to different folders of thedevice depending on query string parameters. We found two vulnerabilities in this binary: OS command injection and arbitrary file write.

7.1.1.1. OS command injection in uploadsm

The uploadsm binary supports three parameters. One of these parameters is the "filename" parameter. When this parameter is passed to the "system" call, no verification of the command line delimiters is made, which leads to OS command execution.

For instance, if a "filename" parameter contains the ".bz2" substring, the process passes the parameter to the "system" function without proper validation, which is shown in Listing 5.

Listing 5. uploadsm OS command injection vulnerable code

```
sprintf((char *)queryString, "bunzip2 -f %s ", newFilePath);

res= system((const char *)queryString);
```

Thus, a lack of parameter validation leads to OS command execution on the device.

Due to the default thttpd configuration, the OS command is invoked with "nobody" user rights.

7.1.1.2. Arbitrary file write in uploadsm

While processing the “filename” parameter, the process passes it to the “fopen” function without proper validation against the "../" characters sequence, which is shown in Listing 6.

Listing 6. uploadsm OS path traversal vulnerable code

```
strcpy(newFilePath, "/otavdata/");

…

while ( 1 )

        {

          pointer2 = strchr(pointer1, '/');

          if ( !pointer2 )

            break;

          v7 = strlen(pointer1);

          v8 = strlen(pointer2);
          strncat(newFilePath, pointer1, v7 - v8);
          if ( strchr(pointer2, '/') )
          {
            mkdir(newFilePath, 0x1FDu);
            strcat(newFilePath, "/");
          }
          pointer1 = pointer2;
          if ( *pointer2 == '/' )
            ++pointer1;
        }
        strcat(newFilePath, pointer1);
        stream = fopen(newFilePath, "w");
```

That can give a remote attacker an opportunity to create/overwrite any file of the device's file system with the highest privileges.

7.1.2. The getsrvr CGI binary

The getsrvr binary is used to send different commands to the charger in the vendor-specific format, via the HTTP POST method. Each command is encoded with a string of tokens, separated with the '|' symbol. There are some common tokens that are included in all commands, and some command-specific tokens. An example of a command with this format is shown in Figure 10.

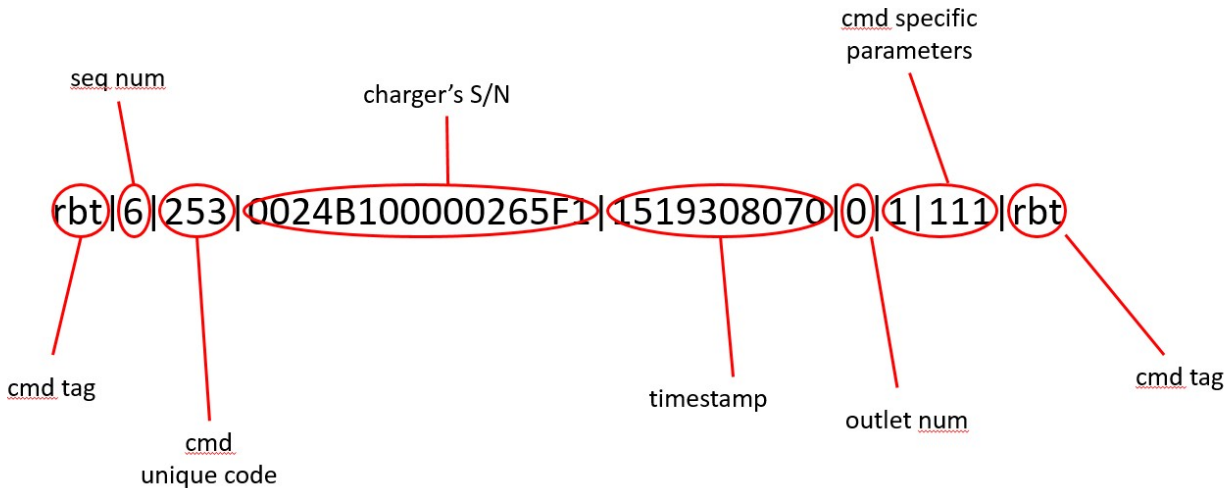


Figure 10. getsrvr command format

We found a series of stack buffer overflow vulnerabilities here.

7.1.2.1. Stack buffer overflow in getsrvr

Command parameters are parsed with the "sscanf" function that is generally unsafe against a buffer overflow.

For instance, when a packet is received where the “cmd tag” token is equal to the “touconf” string, the process parses its parameters with the “sscanf” function that is shown in Listing 7.

Listing 7. getsrvr vulnerable sscanf call

```
sscanf(fS,"%[^|]|%d|%d|%16s|%ld|%d|%d|%d|%[^|]|%[^|]|%s",&t1,&p1,&p2,&
p3,&p4,&p5,&p6,&p7,&p8,&p9,&t2)
```

p8 and p9 are buffers located on the stack, and they are initialized using the scanf “[^|]” specifier, which doesn’t specify a maximum buffer length. That can lead to a stack buffer overflow and remote code execution with the highest privileges.

Due to the fact that ASLR is turned on by default, a remote attacker may have to retry exploitation numerous times or obtain the correct memory addresses to achieve reliable remote code execution.

7.1.3. The dwnldlogsm CGI binary

The dwnldlogsm binary is used for downloading different logs from the device. The type of downloaded log is selected with commands whose format is the same as in the getsrvr binary.

We found a series of stack buffer overflow vulnerabilities in this module that are quite similar to those in the getsrvr binary. They are caused by similar usage of the “sscanf” function and have the same exploitation characteristics, so we won’t discuss them in detail here.

7.2. cpsrelay analysis

The cpsrelay process is used for remote backend communications. All remote controlling functions that are available to the smartphone application user are available via commands that are transmitted from this backend. The process has a configuration file, cps.conf, located at the /etc/coul path.

Besides the control server’s URL, which is set with the “WsUrl” option, there are three other URLs listed in this configuration file, which are set by the options Url, AuthUrl and KioskUrl. Further binary analysis shows that these three URLs are actually bogus: a function that is used for communications with them is stubbed with the “return 0” statement.

Like the modified stunnel binary, the cpsrelay binary uses the same compiled library for certificate decryption, and it also uses the same system.crt certificate. This certificate is used to establish a connection with the remote backend server, so it may present an attacker with the possibility of communicating with the server. Analysis of server-side security problems was beyond the scope of this research.

To communicate with the remote backend server, the binary uses the libocpp.so dynamic library. OCPP is an industry protocol that is based on the WebSocket protocol. It supports several types of messages, including connection establishment messages, heartbeat messages, data transfer messages, and so on. More detailed analysis of this library showed that it uses only one type of OCPP-defined message – "DataTransfer" messages. All commands and responses are passed between the device and the server in the same format as in the getsrvr and downldlogsm CGI binaries, but the command set is different.

Also, we found a series of stack buffer overflows in the server's response processing. They are caused by the same "sscanf" function that is used in the getsrvr and dwnldlogsm CGI binaries, and have the same exploitation characteristics, so we will not discuss them in detail. The only difference here is that due to the fact that the device establishes communication with the server using its DNS name, to successfully exploit this vulnerability an attacker must have the means to reroute traffic or impersonate the hostname of the server.

7.3. sshrevtunnel.sh analysis

The sshrevtunnel.sh executable is a bash script that cyclically tries to establish an SSH connection to the remote server. Partial sources of that script are shown in Listing 8.

Listing 8. sshrevtunnel.sh sources

```
#!/bin/sh

# Bring up pinned up reverse tunnel to mothership. Try forever, but back off

# connection attempts to keep from wasting resources.  Peg the retry time at

# some max and keep trying.



LOG="logger -p DEBUG $0 - "
```

```
KEY_RECOVERY_PARITION=/dev/mtd14
KEY_RECOVERY_OFFSET=1000          # allow for bad block table

#JB MINCONNECT=300
MINCONNECT=30

SERIAL_NUM=`cat /var/config/cs_sn`
SN_YEAR=`echo $SERIAL_NUM | head -c 2`
BASE_SERVER_PORT=20000
BASE_SERIAL=0
SERIAL_MODULO=10000
SERIAL_MINOR=`expr $SERIAL_NUM % $SERIAL_MODULO`
REVPORT=`expr $SERIAL_MINOR - $BASE_SERIAL`
REVPORT=`expr $REVPORT + $BASE_SERVER_PORT`
#FOR QA server please uncomment this line
#REVSYSTEM="pandagateway.ev-chargepoint.com"
REVSYSTEM="ba79k2rx5jru.chargepoint.com"
REVSYSTEMPORT="-p 343"
REVHOST="pandart@$REVSYSTEM"
REVHOST_2016="pandart@xiuq0o4yl57c.chargepoint.com"
#For 2017
REVHOST_2017=pandart@xiuq0o4yl57c2017.chargepoint.com
…
while true; do
        …
        if [ "$SN_YEAR" = "17" ]; then
            …
            ssh -o "StrictHostKeyChecking no" -o "ExitOnForwardFailure
yes" $REVSYSTEMPORT -N -T -R $REVPORT:localhost:23 $REVHOST_2017 &
        elif [ "$SN_YEAR" = "16" ]; then
            …
            ssh -o "StrictHostKeyChecking no" -o "ExitOnForwardFailure
yes" $REVSYSTEMPORT -N -T -R $REVPORT:localhost:23 $REVHOST_2016 &

        else

            …

            ssh -o "StrictHostKeyChecking no" -o "ExitOnForwardFailure
yes" $REVSYSTEMPORT -N -T -R $REVPORT:localhost:23 $REVHOST &

        fi


        SSHPID=$!

        wait $SSHPID

        ssh_rc=$?


    done


    # All attempts failed. Force delays to max until we see a connection.

    if [ $connectsuccess -eq 0 ]; then

        delays="900"

    else

        resetdelays

        connectsuccess=0

    fi

    …

done
```

As a rule, the only thing this script does is try to forward the local telnet port through an SSH tunnel to a port on the remote server (this server is called “mothership” in one of the script’s comments). The SSH key from the SSH key recovery partition is used for device authentication on the server, so, with this key, an intruder could possibly perform malicious actions on the server. Analysis of server-side security problems was beyond the scope of this research.

7.4. Bluetooth communications

The Bluetooth communications scheme is shown in Figure 11.

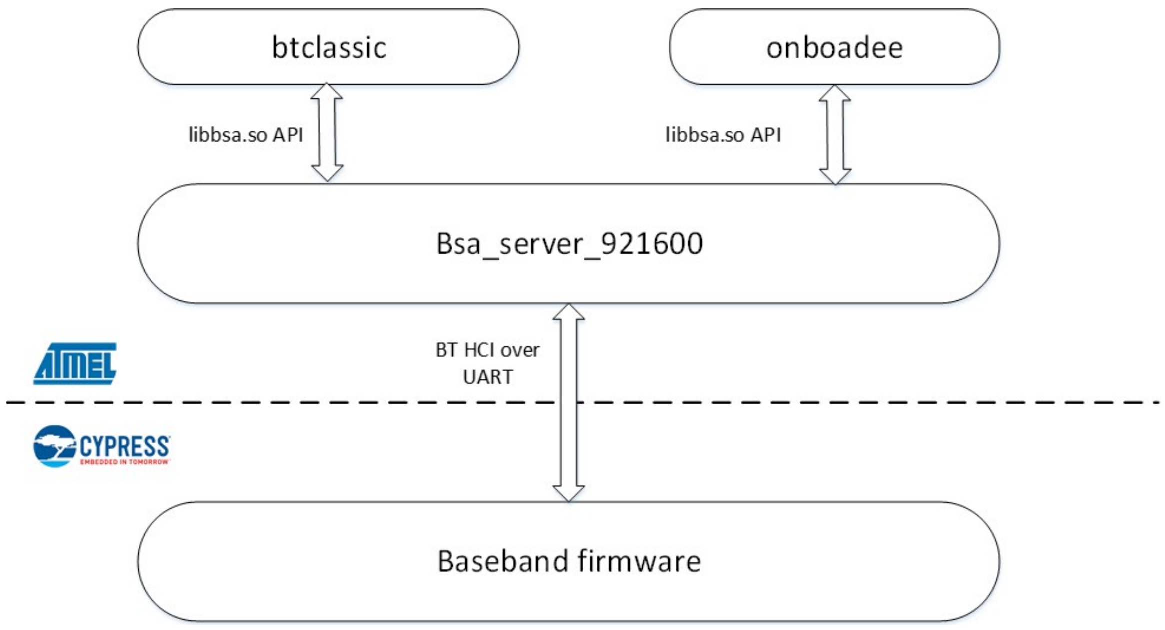


Figure 11. Bluetooth stack scheme

All the communications are based on the Broadcom/Cypress UART HCI stack. This stack is developed for working with a single UART line that is used for a Bluetooth modem connection, so it doesn't need any additional kernel components. Almost all its functionality is encapsulated in the user space process Bsa_server_921600. The libbsa.so dynamic library is used for calling this interface.

There are two processes that use the libbsa.so library's API: btclassic and onboardee.

The btclassic executable processes communications with the smartphone application that are carried over an RFCOMM protocol. We found a buffer overflow vulnerability in this process.

The Onboardee process implements an additional communication protocol that supports the same command set, but is carried over the dedicated L2CAP channel. Only basic analysis of this binary was conducted and it may be a subject for further analysis.

7.4.1. Stack buffer overflow in btclassic

When parsing the "password" parameter of the "connect_to_wifi" request, the service copies it to the stack buffer without proper length verification (see Listing 9).

Listing 9. Btclassic vulnerable code

```
pswd = (void *)json_dumps(joPassword, 512);

…

strcpy(.pswdHash, (const char *)pswd);
```

“pswdHash” here is a 0xD0-byte stack buffer. This can lead to a stack buffer overflow and a denial of service attack.

For successful vulnerability exploitation, the charging station needs to be in the unregistered state. To place the station into that state, an attacker may need to make a power-cycle prepended by the reset-to-factory-defaults procedure, which requires physical access to the charger.

8. Communications with ChargePoint Inc.

Table 3 contains the main stages of communication with the vendor.

Table 3. Vendor response timeline

08/07/18	Information about all the identified vulnerabilities sent to ChargePoint, Inc.
08/21/18	Vendor representatives provided a plan for a vulnerability mitigation process with detailed descriptions of measures to be undertaken.
09/14/18	We received the new firmware with all necessary patches.

The mitigation measures implemented by the vendor for all the discovered vulnerabilities are listed in Table 4.

Table 4. Mitigation measures

Vulnerability	**Mitigation measures**
Uploadsm vulnerability 1. OS command injection	additional input string validation
uploadsm vulnerability 2. Arbitrary file write	additional system() call parameters validation
getsrvr vulnerability 1. Stack buffer overflow	maximum-length sscanf specifier
dwnldlogsm vulnerability 1. Stack buffer overflow	maximum-length sscanf specifier
cpsrelay vulnerability 1. Stack buffer overflow	maximum-length sscanf specifier
btclassic vulnerability 1. Stack buffer overflow	safe string functions like strncpy()

We received the following official response from ChargePoint Inc.

“ChargePoint takes the security of our products and services seriously. We dedicate significant resources to this area including:

- *Following best practices for secure design and testing of our products*
- *Regular 3rd party penetration testing against our products and systems that store sensitive data*

Thank you, Kaspersky, for helping us enhance the security of our products!

- *Your patience and persistence were helpful as these were the first externally-detected vulnerabilities reported to us*
- *All the vulnerabilities identified have been patched*

If you feel you have discovered a possible privacy or security vulnerability, please contact us at security@chargepoint.com with a description of the issue.”

9. Conclusion

During our research of the ChargePoint Home charging station we analyzed the system software and hardware components focusing on those responsible for wireless communications, especially Wi-Fi and Bluetooth. As a result, several security problems were identified in the device's firmware that could lead to full control being gained over the device. These security issues were generally caused by unsafe string functions and a lack of input length validation.

It's worth noting that the device vendor, ChargePoint Inc., was very concerned about the product's security. They use modern stack of technologies, including an enhanced backend communication protocol as well as a reliable firmware updating system. This makes it possible to address vulnerabilities quickly and to introduce additional security mechanisms without significant architectural modifications or end-user interaction.

We appreciate ChargePoint Inc.'s commitment to securing their devices and coordinating their efforts with the information security community. All our findings were quickly addressed, and by the time this research was published all identified vulnerabilities had been patched.

The EV industry in general, and charging stations in particular, offers a wide field for further information security research projects. As of now, we can point to such research topics as EV communication protocols, the potential for payment system fraud, and the security of backend communications.

10. References

Alcaraz, C., Lopez, J., and Wolthusen, S. OCPP Protocol: Security Threats and Challenges. 2017.

(https://www.researchgate.net/publication/313781416_OCPP_Protocol_Security_Threats_and_Challenges)

Dalheimer, M. Ladeinfrastruktur für Elektroautos: Ausbau statt Sicherheit. In CCC Congress (2017).

(https://media.ccc.de/v/34c3-9092-ladeinfrastruktur_fur_elektroautos_ausbau_statt_sicherheit#t=2902)

Microchip SAM9N12/SAM9CN11/SAM9CN12 Microprocessors Specification. 2017.

Open Charge Aliance. OCPP2.0 specification. 2018.