

ChargePoint Home security research

Dmitry Sklyar, @d_sklyar

Kaspersky Lab Security Services, @kl_secservices

Contents

1. Introduction.....	3
2. Research target.....	3
3. Mobile application analysis.....	5
4. Hardware revision.....	8
5. NAND image downloading	12
5.1. NAND image structure.....	14
6. Root access.....	15
7. Software analysis	17
7.1. HTTPS server.....	18
7.1.1. The uploadsm CGI binary	21
7.1.1.1. OS command injection in uploadsm	21
7.1.1.2. Arbitrary file write in uploadsm	21
7.1.2. The getsrvr CGI binary.....	22
7.1.2.1. Stack buffer overflow in getsrvr.....	23
7.1.3. The dwnldlogsm CGI binary.....	23
7.2. cpsrelay analysis	23
7.3. sshrevtunnel.sh analysis	24
7.4. Bluetooth communications	27
7.4.1. Stack buffer overflow in btclassic.....	27
8. Communications with ChargePoint Inc.....	28
9. Conclusion.....	29
10. References.....	30

1.Introduction

Home Electric Vehicle (EV) charging stations forms a relatively new class of electronic devices, which are designed for installations at private places, such as homes, private parking, etc.

Typically, hardware and software design of these devices is based on public charging stations with certain simplifications, as they don't have to deal with charge payment and advanced power grid management.

With the development of the industry, manufacturers add to home EV charging stations support of new features, such as remote charging process control, which can make these devices vulnerable to different types of attacks.

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

The "OCPP Protocol: Security Threats and Challenges" paper is devoted to basic properties of one of industry protocols, but not any particular device research. Mathias Dalheimer's talk is mostly devoted to public station's security aspects, such as billing transactions security and RFID cards data storage format weaknesses.

Here are links to the mentioned works:

https://www.researchgate.net/publication/313781416_OCPS_Protocol_Security_Threats_and_Challenges

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

2.Research target

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



Figure 1.ChargePoint Home appearance

Here are main technical characteristics of this charging station:

- 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 is developed for charging management in the ChargePoint ecosystem. This application is announced to have two main functions:

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

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

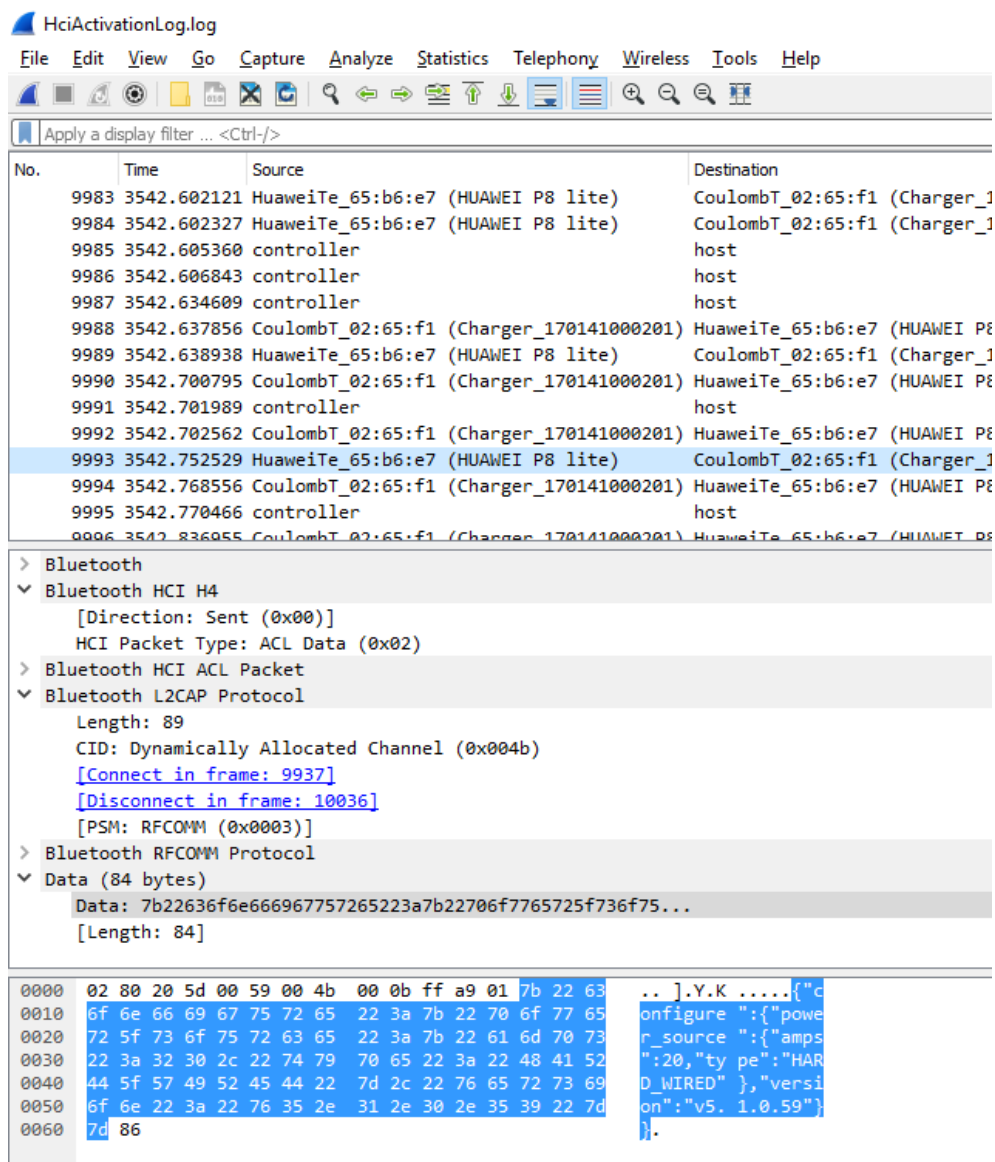


Figure 2. Wireshark view of the HCI dump

Turns out that the device supports following Bluetooth commands:

- Get_version – returns the software version
- Configure – sets maximum charging current and device's 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 smartphone's coordinates and the mobile application account id to the remote backend server
- Shutdown_Bluetooth – disables the Bluetooth service

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

Application is written in Java, and can be easily decompiled.

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

Listing 1. `ResetToFactoryDefaultsFlashSequenceActivity` pseudocode

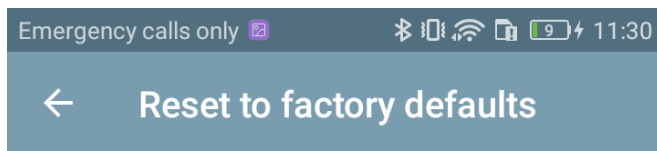
```
private void a() {  
    this.a = new FlashSequence();  
    if (PermissionUtil.requestCameraPermission(this, true)) {  
        return;  
    }  
    try {  
        var1_1 = this.a.a();  
    }
```

As one can see, this code requests camera permissions for some purpose. We were able to run that 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.ResetToF  
actoryDefaultsActivity
```

Running this activity leads to the control screen shown at Figure 3



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, a smartphone camera’s flash starts to play a special blinking pattern. There is a small photodiode window located on the bottom side of the device. If the flash is pointed to that window while playing the pattern, the device will perform a factory reset after the next reboot. This results in losing Wi-Fi and user account settings. Our further investigations show that there is only one pattern that can be recognized by the device, so no additional commands could be received by means of the photodiode.

4. Hardware revision

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



Figure 4. ChargePoint Home without a front panel

PCBs are connected to each other with a proprietary connector. For the sake of clarity, we will refer to the PCB denoted by “1” at 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.

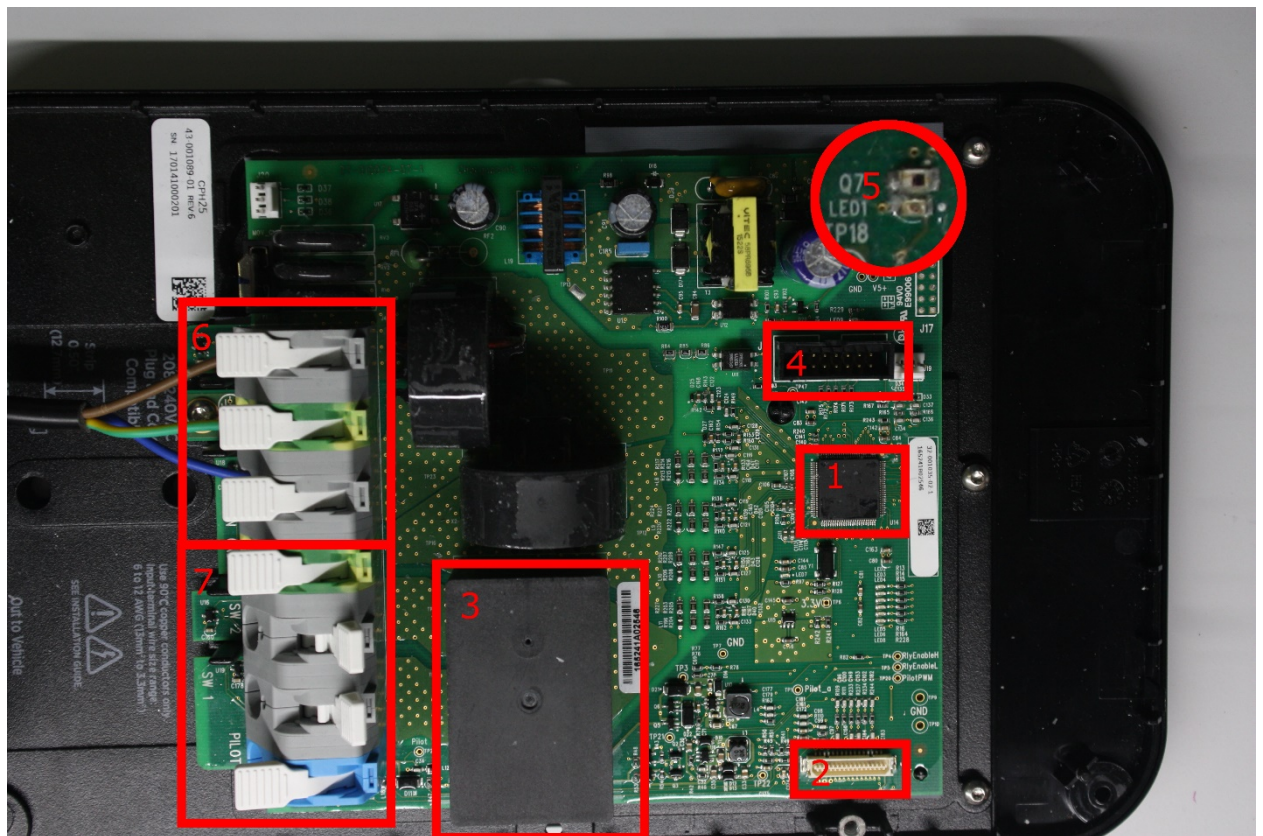


Figure 5. The Power board and its components

Following components are denoted with 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 done by this board. If the pattern is recognized, one dedicated GPIO pin in the connection socket will be turned on. This is the way how a factory reset is triggered. We didn't spend much time on this board during the research, so it may be a target for further investigations.

The Power board is based on a 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 on it are available at Texas Instruments website (<http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html>).

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

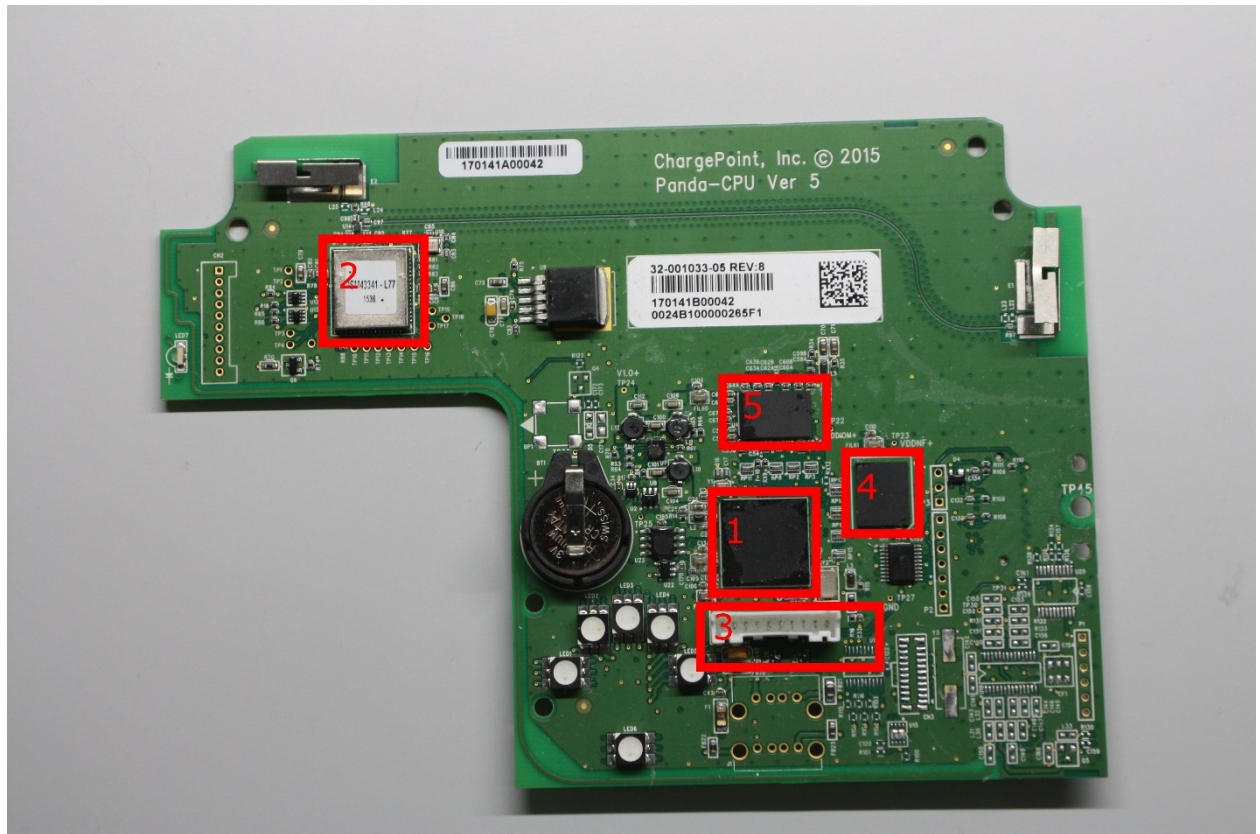


Figure 6. The Panda board and its components

Following components are denoted with 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 a MPU with an 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 base 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 software components designed for that in the ChargePoint Home station, so this protocol could be used in other ChargePoint Inc. products, that utilizes the same hardware design.

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

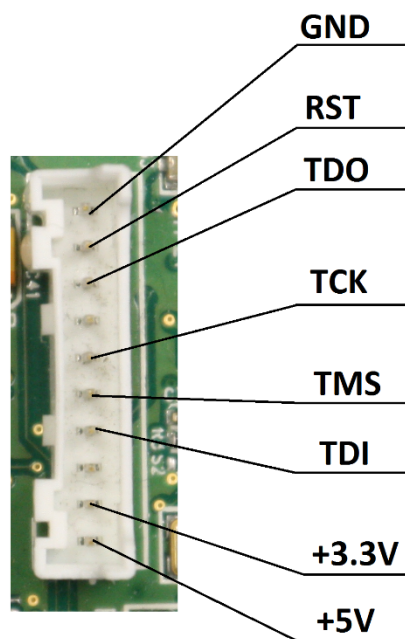


Figure 7. Debug socket pinout

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 distributive. 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 that 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 in an on-chip mask ROM. It can work only with an internal SRAM.
2. AT91Bootstrap. This is a 2nd stage open-source bootloader that can already initialize an external RAM and thus load bigger images.
3. U-Boot. This is a one of well-known Linux bootloaders
4. Linux kernel

We decided to use the AT91 BootROM code to read the NAND memory content, as this method is simple and universal, so it can be later applied to 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, address of the NAND read procedure was found. 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 a 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 appropriate arguments' values set and dumps 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 others, the image contains an UBI file system. For 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 a part of the emulated NAND chip to analyze their content, and found mounting scripts that contain the information about the image partitioning (Table 1).

Table 1. NAND image partitioning

	- 2 nd 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. 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, which are mounted to the root file system during boot and initialization process. These volumes are not duplicated. In addition, there are AT91-Bootstrap image and SSH key recovery partition, which will be discussed in [section 7.3](#).

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

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

All parameters contained in the current parameters 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 telnet port open 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 the authentication with an incorrect password.

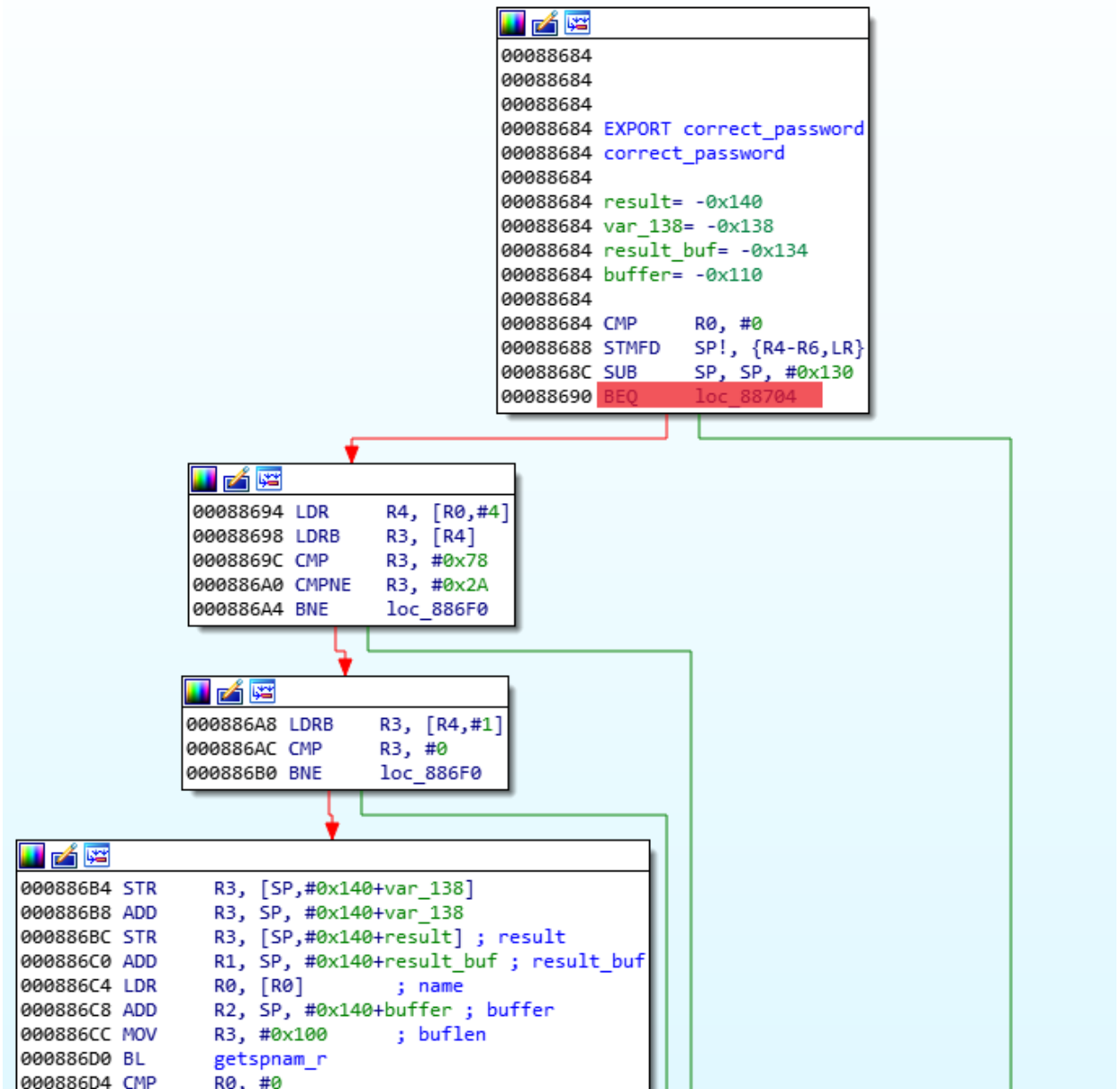


Figure 9. correct_password function

To patch the code in the RAM, we need to set a breakpoint somewhere before the target instruction execution and wait until this breakpoint will be hit. As the target MPU works in the protected mode with the 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 constant memory content, for example, a signature that is located somewhere in the .text section

After bypassing the 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
onboarder	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 uses certificates from the parameters section: the system.crt certificate as a server's certificate and the ca.crt certificate as a certificate authority for client's certificate validation.

Originally, 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 the 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 parameters 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. During SSL connection establishment, the stunnel binary verifies only 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 some other, unknown certificate was used for the system.crt certificate signing.

Listing 4 shows the thttpd configuration file

Listing 4. tthttpd configuration file

```
# BEWARE : No empty lines are allowed!
# This section overrides defaults
dir=/
chroot
user=nobody
logfile=/dev/null
pidfile=/var/run/tthttpd.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 tthttpd 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, which are used in public places. In the latest firmware version, https is enabled only on the localhost interface and unreachable from a Wi-Fi network.

The tthttpd daemon is launched with “nobody” user rights, but all CGI binaries have the SUID bit set. It leads to the fact that these binaries will be 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 the device 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 for 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 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 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's 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 at 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 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.

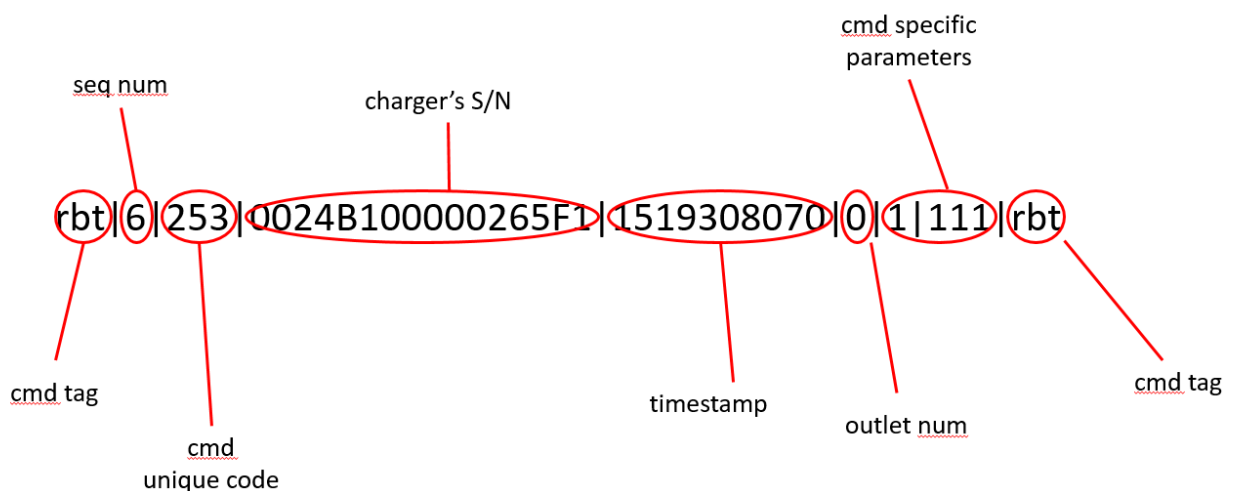


Figure 10. getsrvr command format

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 in common case unsafe against a buffer overflow.

For instance, when a packet, where the "cmd tag" token is equal to the "touconf" string is received, the process parses its parameters with the "sscanf" function, which 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 "[^|]" specificator, which doesn't specify 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 otherwise obtain correct memory addresses to obtain reliable remote code execution.

7.1.3. The dwnldlogsm CGI binary

The dwnldlogsm binary is used for downloading different logs from the device. Type of a 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 "sscanf" function and have the same exploitation characteristics, so we will not discuss them in details here

7.2. cpsrelay analysis

The cpsrelay process is used for the remote backend communications. All remote controlling functions, that are available to a smartphone application's 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.

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

As the modified stunnel binary, the cpsrelay binary uses the same compiled-in library for certificate decryption, and it also uses the same system.crt certificate. This certificate is used to establish connection with the remote backend server, so it can give an adversary a possibility to communicate with the server. Analysis of server-side security problems was out of scope during this research.

To communicate with the remote backend server, the binary uses the libcxx.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 messages – “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 a command set is different.

Also, we found a series of stack buffer overflows in server’s response processing. They are caused by the similar usage of “sscanf” function, as in the getsrvr and downldlogsm CGI binaries, and have the same exploitation characteristics, so we will not discuss them in details. The only difference here is that due to the fact that device establishes communication with the server using its DNS name, to successfully exploit this vulnerability an attacker must have 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. The 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_PARTITION=/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" $REVSYSYSTEMPORT -N -T -R $REVPOR:localhost:23 $REVHOST_2016 &

        else

            ...

            ssh -o "StrictHostKeyChecking no" -o "ExitOnForwardFailure
yes" $REVSYSYSTEMPORT -N -T -R $REVPOR: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

```

Generally, the only thing that this script is doing, is trying 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). SSH key from the SSH key recovery partition is used for device authentication on the server, so, with this key, an adversary can possibly perform malicious actions on the server. Analysis of server-side security problems was out of scope during this research.

7.4. Bluetooth communications

Bluetooth communications scheme is shown in Figure 11.

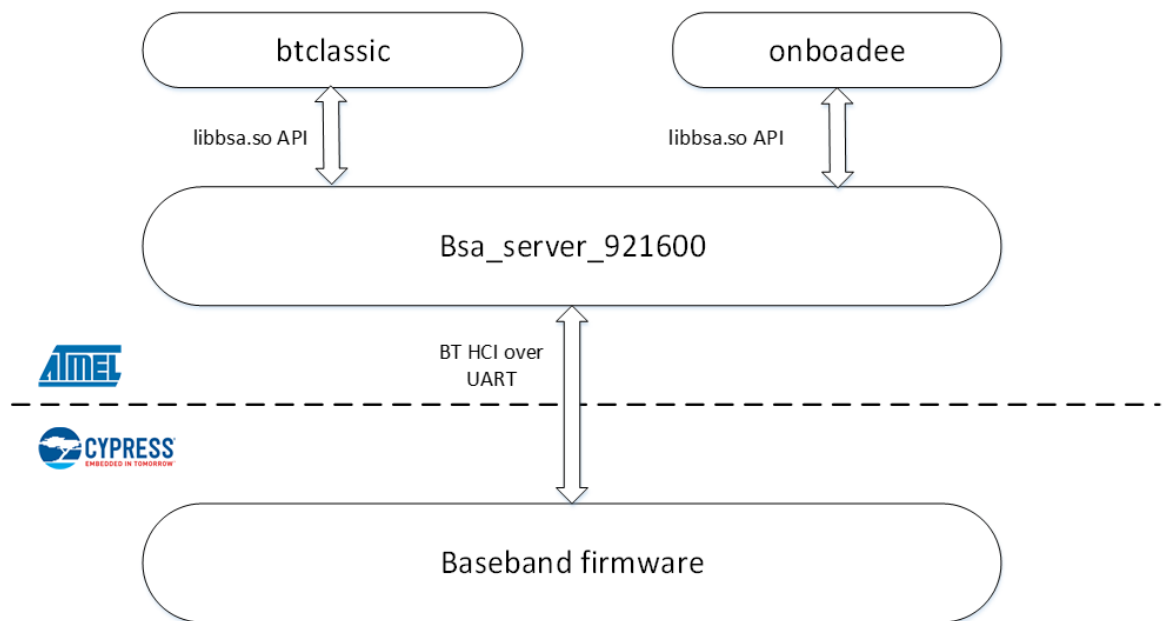


Figure 11. Bluetooth stack scheme

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`. There is the `libbsa.so` dynamic library, which 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 a RFCOMM protocol. We found a buffer overflow vulnerability in this process.

`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 made and it may be a target 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. That 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 put 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 communications with the vendor.

Table 3. Vendor response timeline

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

Mitigation measures implemented by vendor for all 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 the research of the ChargePoint home charging station we analyzed general system software and hardware components focusing on those responsible for wireless communications, especially Wi-Fi and Bluetooth. As a result, several security problems in the device’s firmware were identified, which could lead to gaining full control over the device. Generally, these security issues were caused by unsafe string functions and lack of input length validation.

It’s worth noticing that device vendor, ChargePoint Inc., turned out to be really concerned about the product security. They use modern stack of technologies, which includes improved backend communication protocol as well as reliable firmware updating system. This makes it possible to address vulnerabilities quickly and introduce additional security mechanisms without significant architectural modifications and end-user interaction.

We appreciate ChargePoint Inc. commitment to securing their devices and coordinating the efforts with information security community. All our findings were taken into account quickly, and by the time of the publication of this research all identified vulnerabilities had been patched.

The EV industry in general, and charging stations in particular open a wide area for further information security research projects. At this point we can outline such research topics, as EV communication protocols, potential of payment system fraud, and security of backend communications.

10. References

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

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

Microchip. SAM9N12/SAM9CN11/SAM9CN12 Microprocessors Specification. 2017.

Open Charge Alliance. OCPP2.0 specification. 2018.