

Tinyhack.com

Just another WordPress site

Reverse Engineering Pokémon GO Plus

TL;DR; You can clone a Pokemon GO Plus device that you own. I have managed to get the certification algorithm. However, there is a per device blob used (specific to a Bluetooth Mac Address) for key generation. I have not figured out how you can generate your own blob and key. Using other's people blob may be blacklisted in the future (or Niantic may ban your account).

[Pokemon GO Plus](#), (which I will refer from now on as PGP) is a wearable Bluetooth Low Energy (BLE) device to be used with the Pokemon GO game for Android or iOS. There have been many attempts to clone this device, but only Datel seems to figure out the algorithm, while the other clones are cloning the exact hardware and firmware.

I will explain the complete certification algorithm that I obtain from reverse engineering a PGP clone, and then I will explain how I did the reverse engineering and how you can extract your own blob and key if you want to clone your own device. I am providing a reference implementation for ESP32 so you can test this yourself, the source code DOES NOT INCLUDE the BLOB and DEVICE KEY.

Before I begin, let me start with the current state of Pokemon GO Reverse and PGP reverse engineering.

A short history of past reverse engineering attempts

I am writing this to clear up some confusion that people have on the current state of Pokemon GO Game/App and PGP reverse engineering. The first few versions of Pokemon GO were not protected at all. In a short amount of time people were making bots and maps. Then things changed when Niantic implemented a complex hashing algorithm for the requests to their servers, but this too was quickly defeated with collaboration from many hackers.

Starting from version 0.37 on around second week of September 2016 (which is the first version that supports PGP), Niantic added a very complex obfuscation to the native code, and since then they have changed the hashing method several times, and lately also added encryption. Since the obfuscated version came out, only a few people worked on cracking the new algorithms. For a while, there was a group that runs a hashing-as-a-service with a paid subscription but since a few months ago they haven't reopened their service. Either they have not figured out the latest protection or may be catching up with Niantic is getting boring or no that profitable.

On the PGP side, ever since this device was announced in 2016, many have tried to reimplement it in some form of hardware (for example [this](#)). Before the device was announced, the Pokemon GO app was still not obfuscated and the certification algorithm was not included yet. When the PGP device was finally released, the corresponding Pokemon GO app that supports it was already obfuscated.

On January 2017, a Reddit user BobThePigeon__ wrote [a quite detailed article about reverse engineering PGP device](#). He figured out part of the certification process, but it turns out that it was not the complete process (this is the reconnection protocol). The certification is done at first connection which generates a key to be used at subsequent connections. His write-up only covers the reconnection part. Unfortunately, he didn't continue this effort, and he never posted anything related to this. So currently (until now) there is no open source PGP device available since the device release data in 2016.

Even though no one has published the certification algorithm, [Datel](#)/Codejunkies has managed to reverse engineer this and released their clone: Gotcha and Gotcha Ranger. Just for your information, this company has been in this reverse engineering business for a few decades.

PGP BLE Peripheral

This background information is needed to understand the certification algorithm. PGP is a BLE peripheral that provides three services:

- Battery Level (a standard service)
- LED and button (a custom service)
- Certification (a custom service)

To be recognized by Pokemon Go app/game, it needs to announce its name as: “Pokemon GO Plus”, “Pokemon PBP”, or “EbisuEbisu test”.

Before the LED and Button can be used, we need to pass the certification process. There are three characteristics (“characteristics” is a BLE term) provided by the certification service:

- SFIDA_COMMANDS (for notifying the game to continue to the next step)
- CENTRAL_TO_SFIDA (for sending data to PGP)
- SFIDA_TO_CENTRAL (for reading data from PGP)

The flow of data is:

- The app can write anytime to CENTRAL_TO_SFIDA
- When PGP needs to send something, it sets the SFIDA_TO_CENTRAL characteristic value and notifies the app using SFIDA_COMMANDS notification
- The app can read anytime from SFIDA_TO_CENTRAL

I will not go into detail about the other two services:

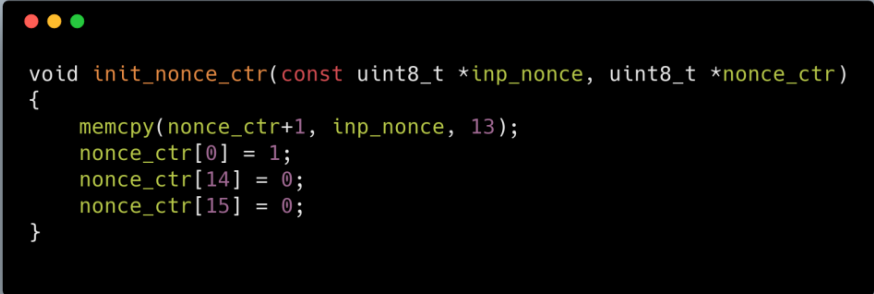
- The battery service must exist because the app reads it
- Someone already reversed engineered the LED pattern, so I won't go into detail about this

Certification Algorithm

Several people have tried reverse engineering based on the Bluetooth Low Energy (LE) traffic but were unable to get the detail of the certification algorithm. This is because the protocol uses AES encryption with a key that is not in the transferred data (so protocol analysis based on traffic alone is not possible).

First I will describe three special functions needed by the protocol. The first one is AES CTR. This is the same as normal AES CTR, except for the counter initialization and the increment function.

Please note that when exchanging nonce we exchange 16 bytes, only 13 bytes are used, and the other 3 bytes are overwritten. The nonce for AES CTR is prepared as pictured: first byte, and last two bytes are set to 0, and we copy the 13 bytes of the nonce (starting from offset 0) to offset 1 in the nonce.



```
void init_nonce_ctr(const uint8_t *inp_nonce, uint8_t *nonce_ctr)
{
    memcpy(nonce_ctr+1, inp_nonce, 13);
    nonce_ctr[0] = 1;
    nonce_ctr[14] = 0;
    nonce_ctr[15] = 0;
}
```

To increment the nonce, we increment the last byte (offset 15), and when it becomes 0, we increment the previous byte (offset 14)

```
void inc_ctr(uint8_t * ctr)
{
    ctr[15]++;
    if (ctr[15] == 0) {
        ctr[14]++;
    }
}
```

And this is how the AES-CTR is implemented

```
void aes_ctr(AES_Context *ctx, const uint8_t *nonce,
             const uint8_t *data, int count,
             uint8_t *output)
{
    uint8_t ctr[16];
    uint8_t ectr[16];

    init_nonce_ctr(nonce, ctr);

    int blocks = count/16;
    const uint8_t *tmpdata = data;
    uint8_t *outptr = output;

    for (int i = 0; i < blocks; i++) {
        inc_ctr(ctr);
        aes_encrypt(ctx, ctr, ectr);

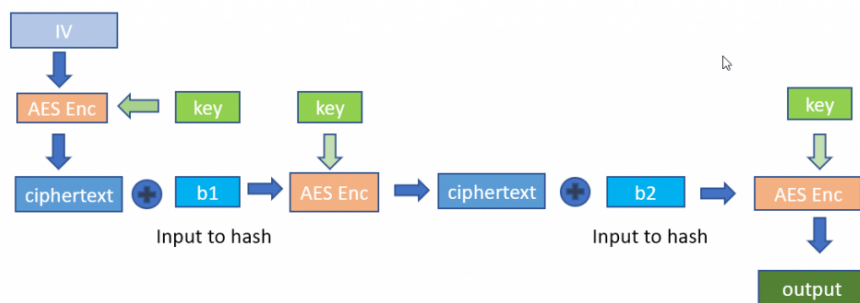
        for (int j = 0; j < 16; j++) {
            *outptr = ectr[j] ^ *tmpdata;
            ++outptr;
            ++tmpdata;
        }
    }
}
```

The second one I named it AES Hash, which uses AES to create a 128 bit hash from data. This one requires another nonce which is derived from nonce. This time we set the first byte to 57, and the last 2 bytes to the size of the data to hash.

```
void init_nonce_hash(const uint8_t *inp_nonce,
                    const int datalen,
                    uint8_t *nonce_hash)
{
    memcpy(nonce_hash+1, inp_nonce, 13);

    nonce_hash[0] = 57;
    nonce_hash[14] = (datalen>>8) & 0xff;
    nonce_hash[15] = datalen & 0xff;
}
```

And here is the hash algorithm. This just encrypts the nonce, then xor it with each block of input data, then encrypt again.



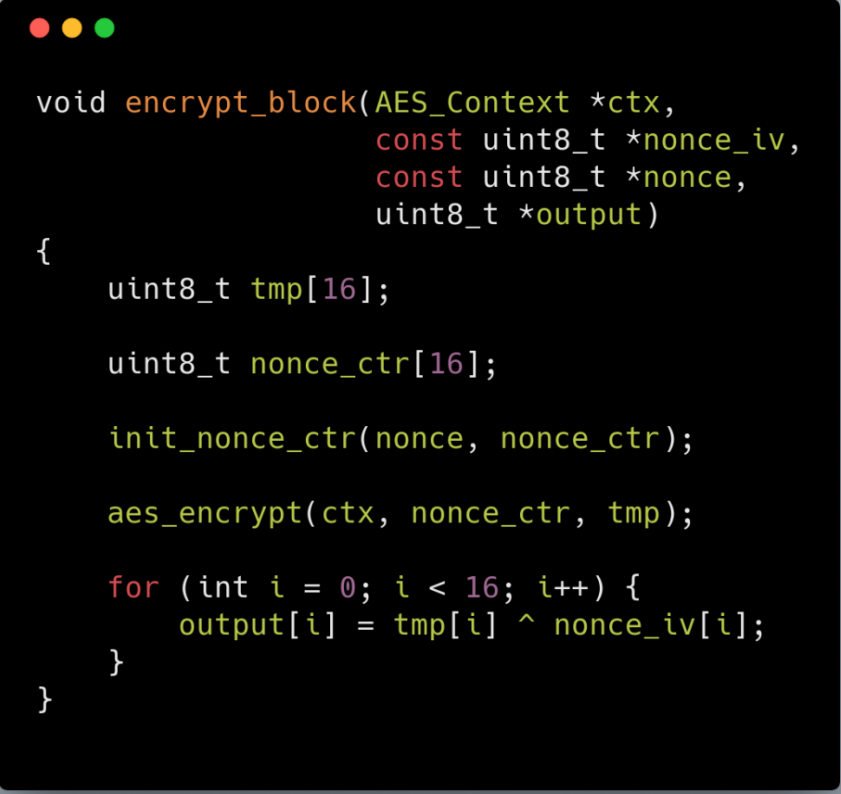
And the code is as follows

```
void aes_hash(AES_Context *ctx,
              const uint8_t *nonce,
              const uint8_t *data,
              const int count,
              uint8_t *output)
{
    uint8_t tmp[16];
    uint8_t tmp2[16];
    uint8_t nonce_hash[16];

    init_nonce_hash(nonce, count, nonce_hash);

    aes_encrypt(ctx, nonce_hash, tmp); //encrypt nonce
    int blocks = count/16;
    const uint8_t *tmpdata = data;
    for (int i = 0; i < blocks; i++) {
        for (int j = 0; j < 16; j++) { //xor with input
            tmp[j] ^= tmpdata[j];
        }
        tmpdata += 16;
        memcpy(tmp2, tmp, 16); //copy to temp
        aes_encrypt(ctx, tmp2, tmp);
    }
    memcpy(output, tmp, 16);
}
```

The third one is Encrypt Block which encrypts a nonce that is initialized as if it is going to be used in AES-CTR, then xors it with a data.



```
void encrypt_block(AES_Context *ctx,
                  const uint8_t *nonce_iv,
                  const uint8_t *nonce,
                  uint8_t *output)
{
    uint8_t tmp[16];

    uint8_t nonce_ctr[16];

    init_nonce_ctr(nonce, nonce_ctr);

    aes_encrypt(ctx, nonce_ctr, tmp);

    for (int i = 0; i < 16; i++) {
        output[i] = tmp[i] ^ nonce_iv[i];
    }
}
```

Now we can discuss the protocol.

PGP will generate a random 16 bytes challenge (A), a random 16 bytes session key (Sk), and a random 16 bytes nonce (N1). Encrypt A using Sk. This key, encrypted data, encrypted hash and nonce along with Bluetooth address (in reversed order) and some data obtained from the SPI flash (in my case it is always all 0) is then packed in the structure pictured below.


```
struct main_challenge_data {  
    uint8_t bt_addr[6];  
    uint8_t key[16];  
    uint8_t nonce[16];  
    uint8_t encrypted_challenge[16];  
    uint8_t encrypted_hash[16];  
    uint8_t flash_data[10];  
} __attribute__((packed)) ;
```

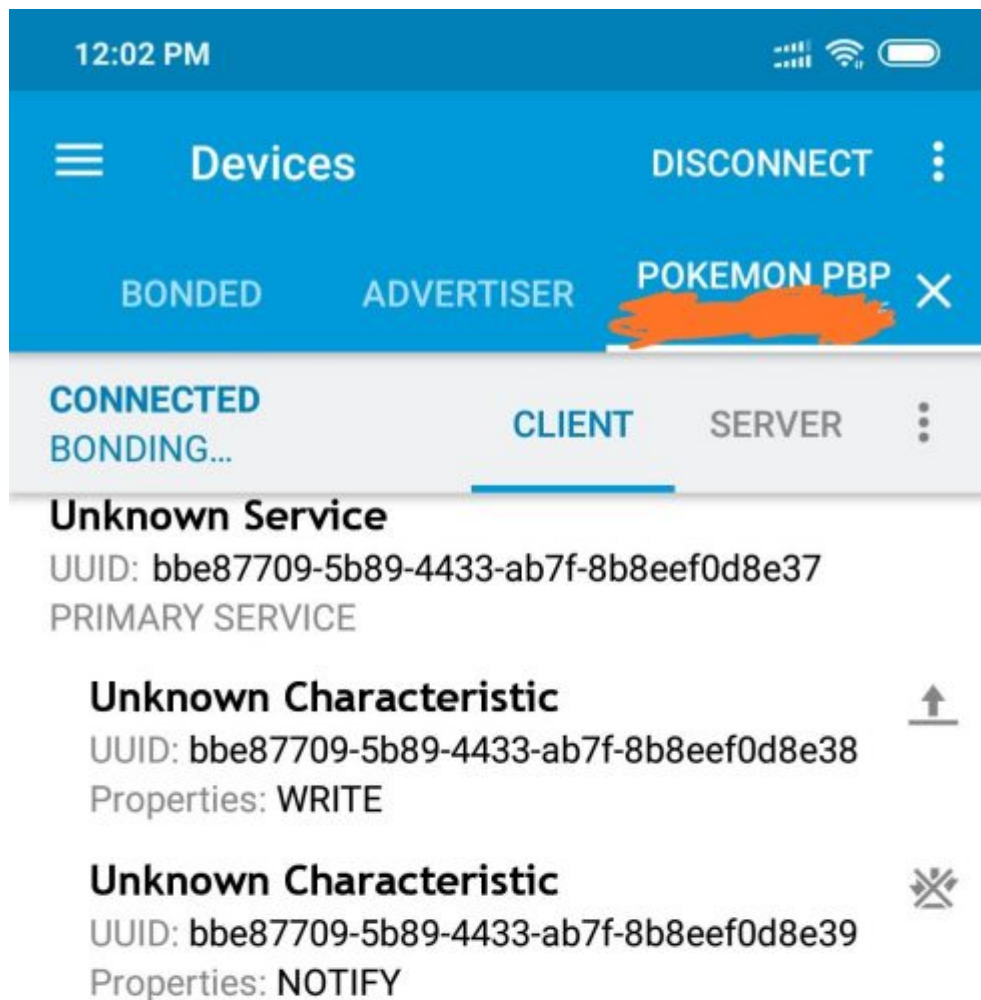
The content of `encrypted_challenge` is the output of `aes_ctr`. The content of `encrypted_hash` is result of `aes_hash` encrypted with `encrypt_block`. This explanation also applies to next parts wherever we have “`encrypted_challenge`” and “`encrypted_hash`”.

The 80 bytes is then encrypted using a “device key” which is dependent on the device. PGP prepares 378 bytes of data, consisting of:

- State (always 00 00 00 00)
- The encrypted `main_challenge_data` (80 bytes)
- A nonce (this nonce can be different from the nonce inside `main_challenge_data`)
- The encrypted hash
- Bluetooth Mac address (this is also on the encrypted `main_challenge_data`)
- 256 bytes of data blob from OTP (one-time programmable memory)

```
struct challenge_data {  
    uint8_t state[4];  
    uint8_t nonce[16];  
    uint8_t encrypted_main_challenge[80];  
    uint8_t encrypted_hash[16];  
    uint8_t bt_addr[6];  
    uint8_t blob[256];  
} __attribute__((packed));
```

When Pokemon GO App connects to PGP, PGP will prepare the challenge data, then signals the Pokemon Go app to read the data. The Pokemon Go app knowing the “device key” will be able to extract the challenge A and sends back A , with prefix 00 00 00 00 to PGP.



Value: (0x) 00-00-00-00

Descriptors:

Client Characteristic Configuration



UUID: 0x2902


Value: Notifications enabled

Unknown Characteristic



UUID: bbe87709-5b89-4433-ab7f-8b8eef0d8e3a

Properties: READ

Value: (0x) 00-00-00-00-44-44-44-44-44-44-44-44-44-44-44-44-44-44-44-44-59-93-99-E3-2A-7D-35-8A-C4-07-47-D2-2C-03-C9-59-AF-7D-AD-2C-1C-F0-4F-02-5A-CD-40-6F-89-24-65-FD-29-36-AE-11-6F-D5-48-78-B7-04-88-80-A6-D8-44-54-0A-F3-B7-C8-96-E5-FD-88-77-9B-64-3F-40-A6-79-C5-6B-37-A7-74-1D-D6-EC-DA-33-BD-31-3A-87-9A-64-57-7D-D3-83-17-E2-34-B1-D0-71-D5-81-75-2B-12-B7-81--3B-8B-CE-AE-C3-B0-48-16-4C-05-1C-CD-26-97-7B-77-A5-0C-78-26-30-B1--7-3E-A4-51-9E-C1-E3-06-71-EE-F0-AB-51-8C-5A-9-6E-F1-DF-D0-28-DD-58-A6-B4-6B-B7-BB-D8-E5-09-8A-23-7A-3C-17-19-FB-A9-C6-56-56-C2-80-4F-3A-5B-86



Viewing challenge data using nRF Connect app

PGP will check that the 16 bytes (A) are indeed the same as the one that was sent. If not then it will terminate the connection. In general, in any step, if something is not right, the connection will be terminated.

For the next few steps, the challenge will have the following format (size of this is 52 bytes).

```
struct next_challenge {  
    uint8_t state[4];  
    uint8_t nonce[16];  
    uint8_t encrypted_challenge[16];  
    uint8_t encrypted_hash[16];  
} __attribute__((packed)) ;
```

PGP will encrypt this static data: 0xaa followed by 15 bytes of NULs (0x00), using the session key (Sk), sets the state to 01 00 00 00 and notify the app to read the data.

The app will decrypt the data, and check if the decrypted data is 0xaa followed by 15 zeroes. If it is as expected, the app will generate random 16 bytes data, encrypts it and pack it in the same format. Note that in BLE world, app payload packet is limited to 20 bytes, so this will come in several packets.

PGP now needs to decrypt this data, and prepare a buffer prefixed by 02 00 00 00 and notify the app to read it. This proves to the app that the PGP device can decrypt the data from the app.

The app will send 52 bytes of data (again according to the next_challenge structure) then PGP will just notify with the value: 04 00 23 00, signaling that everything is OK. When decrypted this final challenge contains the string "PokemonGooooooooo". The app will then subscribe to button notifications and will start writing to LED characteristics when it finds a Pokemon or a Poke gym.



At this point, the green color should light up on the PGP icon.

We can tap on the green icon, and the app will disconnect from the PGP. We can tap it again to reconnect. At this point, we can just forget about everything and starts the protocol from the beginning again, or we can perform a reconnection protocol which is faster. This reconnection protocol is the one that is explained by BobThePigeon__.

For this reconnection, we will use the `session_key` (S_k) that we use in the previous exchange. PGP will generate two 16 bytes random value (let's call them A and B), and expects the App to respond with:

```
AES_ENCRYPT(session_key, A) xor B
```

After PGP verifies that it is correct, the app will then send another two 16 bytes random (let's call them C and D), and expects PGP to respond with:

```
AES_ENCRYPT(session_key, C) xor D
```

When the app verifies that everything is fine, it will send 03 00 00 00 01, and PGP will then acknowledges by notifying the value: 04 00 02 00.

LED, Vibration and Button

When we encounter a Pokemon or Pokestop, the app will send a pattern of lights to be played by PGP. The app will then read the button status to decide what to do with the information. So we can't reprogram it to select a particular ball or to give berries.

Because other people have explained this better than me, I will not repeat it again. Here is a good explanation from a Reddit user [on this thread](#).

Share

LED_VIBRATE_CTRL (21c50462-67cb-63a3-5c4c-82b5b9939aec)

Direction: Central to Peripheral (Write)

Method: Request

Prerequisites: Authentication (via CERTIFICATE_SERVICE)

This characteristic controls the LED and vibration motor. Sequences containing up to 31 patterns are transmitted to the GO Plus in a single write, where they are played back immediately as long as they are valid (if a sequence is already in progress the Priority must also be the same as or greater than the current one). Button status is available via the BUTTON_NOTIF characteristic only during sequence playback.

Packet sequence

| Byte | 0-3 | 4-6 | 7-9 | ... | 8+3(n-1) | 4+3n |
|---------|--------|------------|------------|-----|--------------|------------|
| Content | Header | Pattern(0) | Pattern(1) | ... | Pattern(n-1) | Pattern(n) |

Header packet

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|---|---|---|---|---|---|---|---|---|---|
| 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| 1 | | | | | | | | | Reserved - no apparent meaning - set to 0 | |
| 2 | | | | | | | | | Reserved - no apparent meaning - set to 0 | |
| 3 | | | | | | | | | Reserved - no apparent meaning - set to 0 | |
| 4 | | | | | | | | | Priority (0 to 7) | |
| 5 | | | | | | | | | Number of patterns (0 to 31) | |

Priority - if sequence playback is currently in progress, the new sequence will only be displayed if the Priority of the new sequence is greater than or equal to the Priority of the old sequence. All sequences sent by the game have a Priority of 0.

Number of patterns - this must match the number of Pattern packets, or the pattern will be ignored. To cancel an in-progress sequence, set Number of patterns to 0, send no Pattern packets, and set the priority to be equal to or greater than the in-progress sequence.

Pattern packet

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|---|---|---|---|---|---|---|---|---------------------|---|
| 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| 1 | | | | | | | | | Duration (0 to 255) | |
| 2 | | | | | | | | | Red (0 to 15) | |
| 3 | | | | | | | | | Blue (0 to 15) | |
| 4 | | | | | | | | | Interpolate | |
| 5 | | | | | | | | | Vibration (0 to 7) | |

Duration - period of the pattern in units of 50 milliseconds

Green - intensity of the green LED where 0: off and 15: maximum

Red - intensity of the red LED where 0: off and 15: maximum

Interpolate - applies to LED colour only and not to the vibration

0: Do not use interpolation for this pattern

1: Interpolate the LED colour from the previous pattern using 9-bit PWM

Vibration - any non-zero value enables vibration for this pattern. Vibration is either on or off and the intensity is not adjustable.

Blue - intensity of the blue LED where 0: off and 15: maximum

How-to (using nRF Connect)

1. Connect to the GO Plus in the game (required for authentication)
 2. Connect to the GO Plus in Nordic Semi nRF Connect (without disconnecting it first)
 3. Locate the LED_VIBRATE_CTRL characteristic according to its UUID
 4. Tap the Write button, enter the sequence (in hexadecimal), ensure you are sending a Request not a Command (under Advanced), and tap Send
- The GO Plus should now respond if the sequence was entered correctly. Try the sequence below to check it is working.

Try it yourself

What will the following byte array generate?

00:00:00:01:14:0F:17

Enter it in nRF Connect (or scan the QR code and copy and paste) to find out!

**BUTTON_NOTIF (21c50462-67cb-63a3-5c4c-82b5b9939aed)**

Direction: Peripheral to Central (Notify)

Prerequisites: Only returned when sequence is in progress

When a sequence is in progress and notifications are enabled for this characteristic, the value of the button is sampled every 50 ms and two bytes are sent to the central every 500 ms containing the ten samples right-aligned and zero-padded. Pressing the button does not cancel the sequence in progress; the central must do this if this is required.

Packet format

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|---|---|---|---|---|---|---|---|----------|-----------|
| 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| 1 | | | | | | | | | Sample 1 | Sample 2 |
| 2 | | | | | | | | | Sample 3 | Sample 4 |
| 3 | | | | | | | | | Sample 5 | Sample 6 |
| 4 | | | | | | | | | Sample 7 | Sample 8 |
| 5 | | | | | | | | | Sample 9 | Sample 10 |

Reserved - all zeros, but don't assume this will be always be the case if a GO+2 gets released

Samples - Sample 1 first, sample 10 last

0: Button was not pressed

1: Button was pressed

by @mavensGalore

GO Plus LED_VIBRATE_CTRL and BUTTON_NOTIF**Reimplementing the hardware**

The next logical step after understanding the process is to reimplement this algorithm in a new hardware to test that it is indeed correct. I started with Android (turns out to be almost impossible), then Raspberry Pi Zero W (got stuck on some bluez stuff), and finally resorts to ESP32.

Android BLE peripheral emulation

At first, I thought that this is the best method: anyone that has a spare Android can test this. It turns out to be not easy: Android can act as a BLE peripheral but will **randomize** its MAC address on every announcement. This is done for privacy reason but I thought that it will make it impossible to implement PGP emulation because the PGP protocol uses Mac Address in the encryption process.

It turns out that on iOS, the app can't get BLE address of the peripheral, so I (or someone) should try again reimplementing this in Android. For the Android game version: Niantic should be able to detect/block this easily.

Pi Zero W peripheral emulation

The next arsenal that I have is a Raspberry Pi Zero W. I have checked that it is possible to do peripheral emulation using bluez, and it is also possible to change the mac address for BLE. It is also possible to program this using Python, so it seems to be a good choice.

However, I was stuck with the bluez/dbus API. The documentation is quite sparse. So I gave up with Pi Zero W. I think it should be possible to do this in Pi Zero W. I don't want to spend a lot of time debugging the bluez stack so I switched to something that is easier to debug.

ESP32

I chose this device because this device is very cheap (the cheapest is around 5 USD delivered), is easy to program, and I happen to have a few of them. I didn't have any experience before in programming BLE for ESP32, but programming BLE in this platform is very straightforward. Please note that I just copied and modified the examples provided in the [esp-idf](#), so what I did may not be the most correct or efficient way to do it.

```
I (3481791) BT_GATT: GATT_GetConnectionInfo conn_id=3
I (3481801) PGPEMU: ESP_GATTS_EXEC_WRITE_EVT
I (3481801) PGPEMU: 00 00 00 0f 10 f0 f0 08 00 00 10 f0 f0 08 00 00
I (3481811) PGPEMU: 10 f0 f0 08 00 00 10 f0 f0 08 00 00 10 f0 f0 08
I (3481821) PGPEMU: 00 00 10 f0 f0 08 00 00 40 80 80 ff 80 80 89 80
I (3481821) PGPEMU: 80
I (3481831) PGPEMU: LED: Pattern Count=15 priority: 0
I (3481831) PGPEMU: *(16) #000f00
I (3481841) PGPEMU: *(8) #000000
I (3481841) PGPEMU: *(16) #000f00
I (3481851) PGPEMU: *(8) #000000
I (3481851) PGPEMU: *(16) #000f00
I (3481851) PGPEMU: *(8) #000000
I (3481861) PGPEMU: *(16) #000f00
I (3481861) PGPEMU: *(8) #000000
I (3481871) PGPEMU: *(16) #000f00
I (3481871) PGPEMU: *(8) #000000
I (3481871) PGPEMU: *(16) #000f00
I (3481881) PGPEMU: *(8) #000000
I (3481881) PGPEMU: *(64) #000800
I (3481881) PGPEMU: *(255) #000800
I (3481891) PGPEMU: *(137) #000800
```

I also provided Makefile.test which can be used on the desktop to test the encryption algorithms, just run `make -f Makefile.test` and run `cert-test`.

You can download the code from GitHub:

<https://github.com/yohanes/pgpemu>

This app doesn't have a visual indicator, after flashing with "make flash", run "make monitor" to see pairing progress. Press "q" to simulate button press and "w" to clear button press notification (although this doesn't seem to be necessary).

I didn't test the implementation for an extended amount of time. I only tested the following:

- It can be paired, disconnected, reconnected
- It can receive notifications when there are Pokemon around me

- It can send button press (using 'q' key in the serial monitor) to catch the Pokemon

You will need the following data from the device that you clone:

- Bluetooth MAC address (easily extracted using Bluetooth connection)
- Fixed data (easily extracted using Bluetooth connection)
- device key (currently requires soldering)

You can read the method to extract the device key on the next part.

Reversing PGP

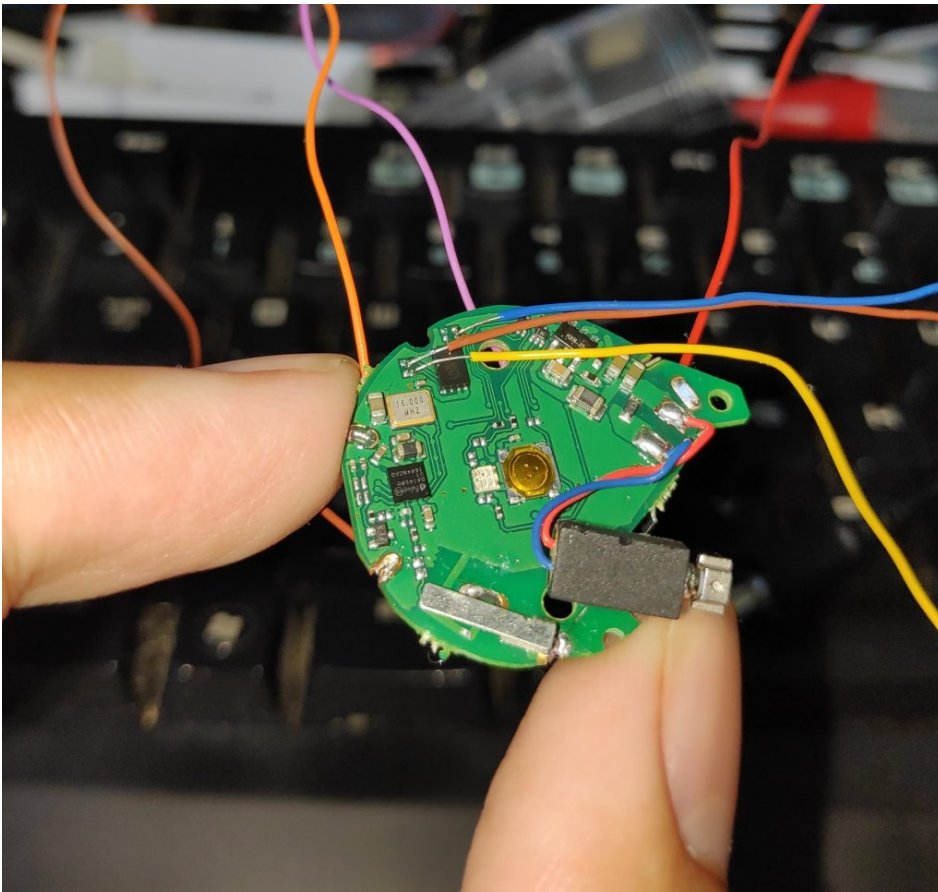
This is the details of the reverse engineering part. This part is divided into two main parts: the hardware reversing and firmware reversing.

The hardware

I bought a Chinese clone of the PGP for about \$20 including shipping (the original one would cost me \$88 including shipping to Thailand)., and when I opened the PGP, it turns out the be an exact clone of the original. It uses the same DA14580 chip with the same PCB layout.

The first difficult part in reversing a hardware is to extract the firmware (since no one has shared this on the internet). To be precise: the difficult part is soldering the wires to the SPI flash chip. Information from [BobThePigeon_ post helped a lot because I don't need to figure out the pinout.](#)

Fortunately, this one is a bit easier to solder due to the solder pads that exists in the cloned version.



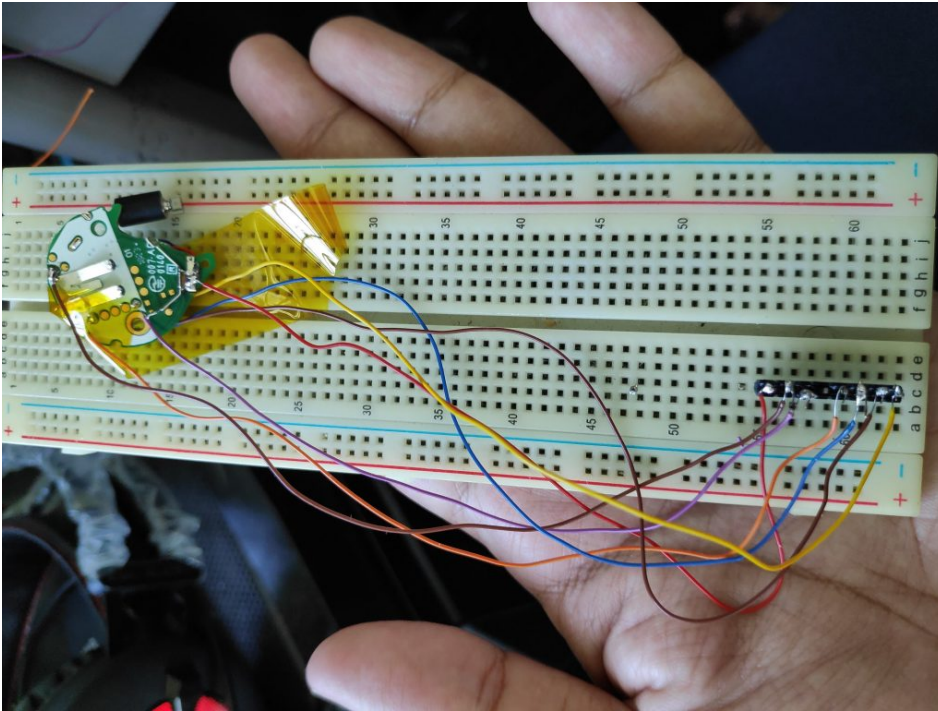
This is the clone device that I have, same MCU, same PCB layout, same firmware as the original

I used a \$5 USB Soldering iron with a wrapping wire and it works quite OK. I was a bit amazed that the everything works on the first try.



\$5 Soldering iron

I held the board in place with Kapton tape on a breadboard.



Since the hardware is the same, I tried following what BobThePigeon_ already did: holding the RST and read the flash. I use the flashrom package on Raspberry Pi software to read the SPI flash (you can also use Arduino board, Bus Pirate, or anything that can read SPI Flash).

To detect if the SPI connection works:

```
flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000
```

To read the flash:

```
flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000 -r pogoplus-31-10-2018.bin
```

Later on I also made a code to patch and reencrypt the firmware which can be uploaded using the same SPI connection.

After extracting it and comparing the description with his write-up, it turns out to be using the exact same firmware as described by BobThePigeon_. So all the AES keys that encrypt the firmware is also the same. It also means that when there is a new update for PGP, this device should also be updateable.

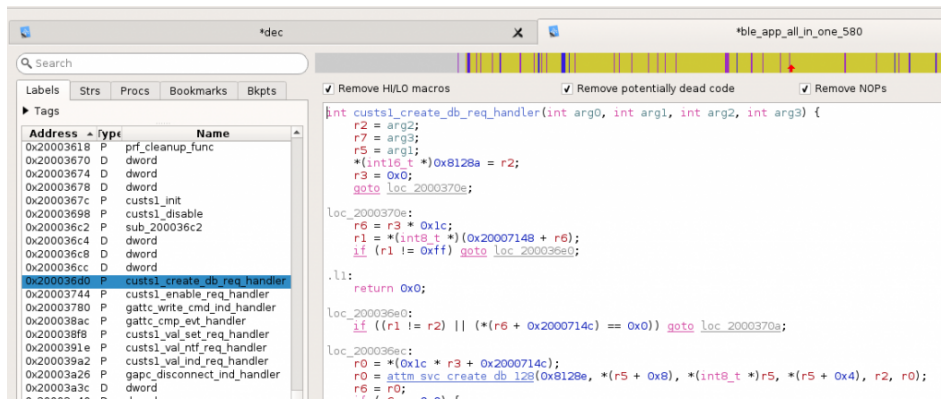
For the hardware debugging, I did not use JTAG to debug the firmware. I did not perform a dynamic analysis using a debugger. So basically I only used a few cables to read and write the SPI flash.

Deeper into the firmware code

BobThePideon_ wrote that *“All this information is in Dialog Semiconductor’s DA14580 SDK, however, you have to jump through some hoops to get the SDK.”*. Well, it turns out getting the SDK is quite an easy process, and having the SDK helps a lot in understanding the firmware. Even though I don’t have a DA14580 devkit board, I can try to compile and see what the resulting code will look like.

The main firmware is 31984 bytes long and since this is Cortex-M0, it uses Thumb instruction set. This firmware is loaded at starting memory location: 0x20000000. It is not easy to understand the code just by looking the code in a disassembler, so my first approach is to try to see what a real firmware would look like if it has a full debugging information.

After installing the DA1458x_SDK and Keil uVision5 we can compile the examples (for instance ble_app_all_in_one). My first thought was to generate assembly code from C file (like the -S option in gcc), but this is not allowed in the free version of Keil uVision. But we have the next best thing: an ELF with debugging symbol. On the output folder, I saw an AXF file, which is actually an ELF file with debug information. This helps a lot in understanding a firmware designed for DA14580.



Opening AXF file, this is much easier to understand

Now we can see clearly how the code calls ROM functions which is located from 0x20000-0x35000, and I can understand the mapping in rom_symdef.txt file:

sdk/common_project_files/misc/rom_symdef.txt

For example, in the PGP firmware, the function at the address 20006e24 just calls 0x33b21 which according to the rom_symdef.txt is __aeabi_memcpy. Renaming these procedure is like finding the edges of a puzzle.

I spent quite a lot of time looking at SDK to understand more about its structure, convention, and constants that might help.

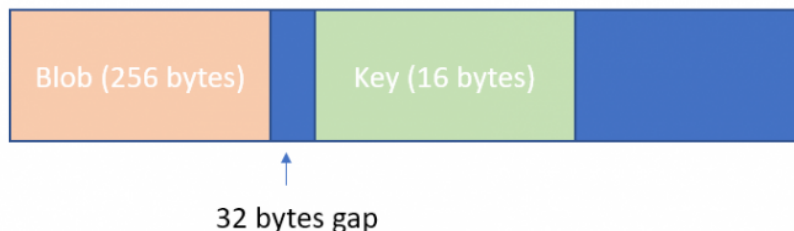
Inside platform\core_modules\rwip\api\rwip_config.h we can see the standard TASK number (for example for battery service, firmware update service, etc). This task numbers also helps identify different subroutines that call ke_msg_alloc, by convention, the ID of the message is: TASK_ID << 10 + message

Some interesting subroutine/function locations:

- At address 0x20005758 is the function that copies data from OTP (blob and device key)
- At address 0x200065DC is the main AES encryption (also the read AES implementation in sdk/platform/core_modules/crypto/), by finding cross-reference to this, we can get the subroutine that does AES-CTR, AES-HASH, etc
- At address 0x2000644E is the handler that will handle different states of certification

The rest is just patience to trace every input and output of the subroutine to see how each value is generated. After reading things very carefully for a couple of days, I figured out all of the algorithms. It took me another few days to track the key being used, it turns out that the device key is not stored in the SPI flash but in the OTP (one time programmable) area (starting from 0x47000). The blob is broadcasted on the challenge, but the key is not.

0x47000



The secret recipe (or how to extract your key)

So the code uses some data from the OTP area of the chip for the device encryption key. How can we read this? there are a lot of ways to do it:

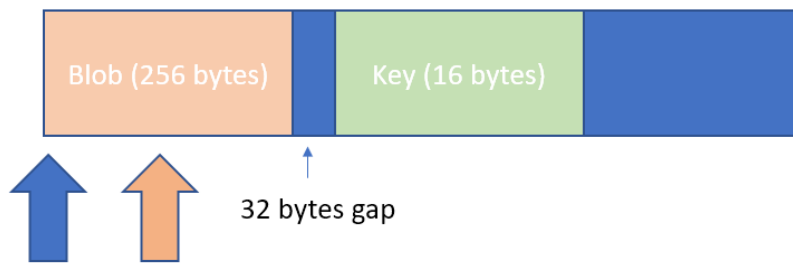
- Write a new code to read the OTP area
- Use a debugger
- Patch the existing code

Writing a new code or using JTAG debugger will require me to solder more wires and to get the debugging tools to work. I am trying not to disturb the current soldering that I already made, and I don't want to setup environment to write a new code, flash it and read the result.

I found a simple way to extract the key by patching a single byte on the firmware. I just shifted the constant that was used to send the blob. Instead of sending only the blob area, it will send also the embedded key that is located 32 bytes after the blob.

Compared to JTAG method, this patch is easier to reproduce by anyone that has a soldering iron and Raspberry Pi or Arduino.

0x47000



This is just a one-byte patch, by changing the value at 0x6425 (file offset in the decrypted main firmware) from 0x4c to 0x7c, we can extract the device key.

I provided a script to decrypt firmware from SPI flash image, and also another script re-encrypt the decrypted firmware to make a new flashable image. So the steps to get the keys are:

- Get the SPI flash content
- Extract/decrypt the main firmware
- Patch the main firmware (remember to make a backup of the original)
- Repack/re-encrypt the patched firmware
- Flash the firmware
- Start the PGP
- Read the challenge data using any software, the last 16 bytes is the device key
- Flash the original firmware

Is there another way to extract the device key? It may be possible to extract it from the memory of the Pokemon Go app while it is running. But even if it is possible now, Niantic may change it so that extraction will not be possible in the future (e.g: by clearing keys after use or even do the decryption on the Niantic server).

Reversing the Pokemon GO game?

I tried reverse engineering the Pokemon GO game, but it is heavily obfuscated. I spent a few hours a day for several days but didn't get very far. I can explain some of the things that I observed, but I won't go into much detail since Niantic will probably change them anyway.

Before going deep into the code, I checked on the version history of both the APK and IPA files, hoping that maybe in the past they have included the certification process in an unobfuscated form (or less obfuscated form compared to what we have now).

All versions before the release of PGP hardware contains incomplete PGP certification code which is still not obfuscated. Unfortunately, both the iOS and Android version contain obfuscation at the time of the PGP release. The old files are also not easier to read compared to the newer ones.

iOS version of the game

The iOS version consists of a single huge binary named pokemongo. This is a mix of Objective-C, Unity and native C++ code. The binary uses ARM64 code. Due to the size, navigating and extracting something useful from this monolithic binary is quite hard.

There is a group that actively maintains a patched version of Pokemon GO called PokeGO++ (they even have a subscription for this service). They patched the security checks that exist in the binary and added a new library with method swizzling to add new features to the game (such as Teleport, IV Checker, etc). If anyone is interested in reversing the Pokemon GO game, then this would be a good starting point.

I did not investigate the iOS version further, apart from the big binary size, the other reason is that I only have an iPhone 5S which is already too slow to run the Pokemon GO game.

Android version of the game

In the Android version, Niantic employs [SafetyNet](#) so that changing the APK will stop it from working. It is also very sensitive to any leftover trace of rooting tools, and the existence of some files will make it refuse to connect.

The Android version consists of Java/Smali Code, Unity code, and native library (accessed through JNI). Using existing tools we can decompile the Java part, but nothing interesting is there. We can also look at names of the Unity classes using existing tools, but the implementation is in native code (not in .NET IL). The native library uses THUMB instruction set instead of ARM/ARM64 (even on 64 bit Android).

Almost every subroutine in the native code is split into tens to hundreds of blocks. My guess is they are using a custom obfuscating compiler, probably a fork of [llvm-obfuscator](#). One subroutine is split into multiple blocks using MOV Rx, PC. This can be fixed using some pattern matching, but after you merged the routines, it turns out that it is still split into multiple small subroutines located far away. These small subroutines only do one thing, for example $a+7$ or $a + b$.

Strings are encrypted (obviously), but the decryption is not done in a single place. It is done when needed, and it also uses a different encryption method in each subroutine. So string extraction is not easy.

Although I didn't implement it, in my opinion, it is possible to unobfuscate a lot of the code. This will require quite a lot of coding. And when you succeed, they will probably already release a new version with a different obfuscation method that breaks your tool.

The native code is accessed using JNI, but it only exports several symbols named `java_XXX`, for the rest, it uses "[registerNativeMethods](#)". Of course, the address and the name of the methods are obfuscated.

The process for PGP is separate from the game process. The game communicates using SSL. Bypassing the SSL is not too hard, but apart from the initial handshake, subsequent packets are encrypted with custom encryption (they use another layer of encryption on top of SSL).

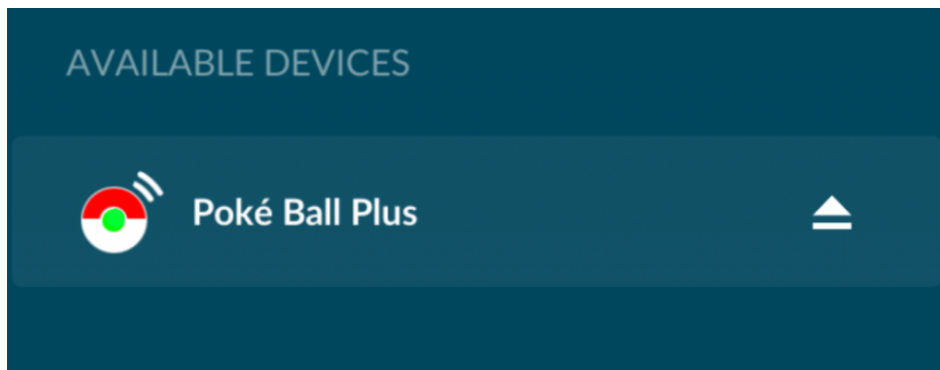
For the main game process, we can intercept the Unity code to see the request/response in Protobuf form, but unfortunately, this is not possible in the PGP process which is completely native code (it doesn't load the unity library at all).

During the pairing process, the Pokemon Go game must be connected to the internet. This seems to indicate that whatever process required to generate the key was done on the server (Niantic) side.

Reversing other devices?

Currently I do not own any other Pokemon GO related devices such as Poke Ball Plus, Nintendo Switch or other implementation of PGP (such as Gotcha, Gotcha Ranger, and Pocket Egg) so I leave it to others to do it, or I might do it when I have the device(s).

One interesting thing is that we can rename the device to Pokemon PBP and it will be recognized and paired as Poke Ball plus.



How can datel/codejunkies and the Chinese do it?

The short answer is I don't know. I don't know how they can generate the combination of a new blob, mac address, and device encryption key.

Changing a byte in the blob (with same Mac) causes the challenge to be rejected.

Changing a byte in the mac address also causes the challenge to be rejected. I only have one device to test and although I have many guesses, I am not sure which one is the answer.

Some of my speculations are:

- There are only a few combinations of Mac/Blob/Keys in the cloned devices being sold (as noted by many people, [it seems that most/all Gotchas have the same MAC address](#)), or
- There is a simple formula relating these three, and it doesn't use any secret key, or
- There is an implementation of this algorithm in one of the old version of the game, or
- The algorithm was leaked from the PGP factory

Or the explanation could be very different from the one listed above. I am also considering to release my Blob and Key, but I am afraid of these:

- Niantic might block this Mac Address from connecting (a bit unlikely, since they seem to allow Gotcha devices with same Mac address, but of course they can always change their mind)
- If this mac address is used by many people, everyone will be blocked
- I might get sued for publishing the secret key

But for now, I decided to play safe.

If someone wants to sacrifice their PGP and spread the blob/key combination, I suggest to use an original PGP, so that the Mac address is unique. And if that gets banned, you won't upset a lot of people buying cloned devices.

Please also note Niantic's stance on this.



Brandon @Brandon70351580 · Oct 10, 2018



So i just got the go-tcha for #PokemonGo. I orginally wanted the plus. Whats the big difference? @PokemonGoApp @NianticHelp



Niantic Support ✓

@NianticHelp

Hey Brandon! Pokémon GO Plus is currently the only authorized accessory for Pokémon GO. Using other accessories may negatively affect your gameplay and may result in account blocking or termination. Hope that helps! ^AM

♡ 64 12:03 PM - Oct 11, 2018



💬 43 people are talking about this



What can you do now with this information?

There are several **legal** things that you can do with the information presented:

- You can clone your own device, and make it better (e.g: in a better form factor, with a better display, battery, etc). Cloning your own device *for your own use* should be undetectable by Niantic.
- You can modify the firmware of your PGP (e.g: auto catch or auto spin only)
- You can write an app that can communicate with your PGP

Future work

I only played Pokemon GO casually with my family, I am still at level 33 after two years playing this game on and off. Reversing this Pokemon GO Plus is only for fun and to satisfy my curiosity. But I am not that curious to spend a lot of money to acquire other kinds of Pokemon Go related hardware (Gotcha, original Pokemon Go Plus, Poke Ball Plus, etc).

If you want to help me buy other Pokemon GO related hardware or just tip me for this article, you can send it via:

- [Paypal](#)
- Bitcoin (19mkof1of9yC5TNWbPw5gjGrcL2NHiHim9)
- Ethereum or other tokens (0x618b59AF01DC11b7fBb00f700E9b78A5cc2e234e)



admin / November 21, 2018 / hardware, writeup

2 thoughts on “Reverse Engineering Pokémon GO Plus”

Pingback: [Reverse Engineering Pokemon GO Plus #ReverseEngineering #Pokemon #PokemonLetsGo #ESP32 « Adafruit Industries – Makers, hackers, artists, designers and engineers!](#)



Sora

December 2, 2018 at 10:02 am

You left your cursor in the screenshot of your flow chart.

Just kidding. Thanks for the amazing write up and all your hard work. Gold star. ★

Tinyhack.com / Proudly powered by WordPress

