Tetsuji: Remote Code Execution on a 22 Years Later

2022-08-27

Introduction

It's that time of year again - the Binary Golf Grand Prix is back for a third year running! You can also check out my entries to the first and second times this amazing competition ran.

The theme this year was to produce a binary that crashes a given program. Bonus points for hijacking execution, and submitting a patch to the project that fixes the vulnerability. Coinciding with the announcement of this year's competition, @netspooky told me about a little-known accessory for the GameBoy/GameBoy Colour/GameBoy Advance called the Mobile Adapter GB, which let players connect their console to the internet via their mobile phone. This accessory was only released in Japan and lasted only 2 years before being killed off.

In keeping with the BGGP theme of "crash", I looked up the games which had support for this adapter, and found that Pokemon Crystal was one of them. Sadly, I can't use my exploit (dubbed "Tetsuji") as a BGGP entry as it isn't a binary. However, I believe this is possibly the first case of a remote code execution exploit on a GameBoy Colour - discovered 22 years after the accessory was released!

You can find all the code I wrote to exploit this bug on the repo.

> Thanks to mid-kid on GitHub, this article has been updated with some corrections/additional info!

The Mobile Adapter GB

The Mobile Adapter GB was Nintendo's first attempt at online handheld gaming. It was only available in Japan and lasted just under 2 years before it was killed. There's tonnes of incredibly valuable detail (focused on emulation) on Shonumi's site here. Imagine in 2001 plugging your GameBoy Colour into your mobile phone, and trading Pokemon over HTTP and POP3?!

Fortunately, there was some previous work by Háčky back in 2016 that gave me a great headstart (see here), but sadly it seems like the interest has died out since. Apparently the Mobile Adapter's protocol is very similar to the GameBoy Printer.

The real guts of the protocol is in the commands that can be sent. Shonumi's site (here) contains excellent detail on all of the commands available. As mentioned, Háčky started work on a script which was my main starting point to actually see this protocol in action.

When you bought one of these adapters back in 2001, it came with a "Mobile Trainer GB" cartridge, which I managed to pick up for a tenner on eBay. This cartridge was used to configure the memory in the Mobile Adapter GB with things like your dial-up username and password, as well as containing a *very* simple web browser and email client. Running this in the BGB emulator, we can use the link cable emulator to simply connect over a local socket to read and write bytes as we would any other connection.

Partly as an exercise in eduation, but also for my own legibility, I rewrote most of Háčky's work to make it easier to expand as I started to prod and poke. A great illustration of this protocol in action is reading an email over POP3 using the Mobile Trainer GB ROM:

```
[+] 0x10: Opening Session (NINTENDO)
[+] 0x11: Closing Session
[+] 0x10: Opening Session (NINTENDO)
[+] 0x19: Read Config: 96 bytes @ 0
```

```
00000000: 4D 41 81 00 D2 C4 03 B7  D2 8D 70 A3 67 31 32 33
MA........p.g123
00000010: 34 35 36 37 38 39 00 00  00 00 00 00 00 00 00 00
456789..........
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 6E 69 6E 74
............nint
00000030: 65 6E 38 38 40 67 62 61  61 2E 64 69 6F 6E 2E 6E
en88@gbaa.dion.n
00000040: 65 2E 6A 70 00 00 00 00  00 00 6D 61 69 6C 2E 67
e.jp......mail.g
00000050: 62 61 61 2E 64 69 6F 6E  2E 6E 65 2E 6A 70 70 6F
baa.dion.ne.jppo
[+] 0x19: Read Config: 96 bytes @ 96
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
p.gbaa.dion.ne.j
00000010: 70 00 00 00 00 00 A9 67  7F 00 00 F0 00 00 44 49
p......g......DI
00000020: 4F 4E 20 50 44 43 2F 43  44 4D 41 4F 4E 45 FF FF  ON
PDC/CDMAONE..
00000030: FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 FF FF  FF FF FF FF FF FF 00 00
................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 35 99
..............5.
[+] 0x11: Closing Session
[+] 0x10: Opening Session (NINTENDO)
[+] 0x19: Read Config: 96 bytes @ 0
00000000: 4D 41 81 00 D2 C4 03 B7  D2 8D 70 A3 67 31 32 33
MA........p.g123
00000010: 34 35 36 37 38 39 00 00  00 00 00 00 00 00 00 00
456789..........
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 6E 69 6E 74
............nint
00000030: 65 6E 38 38 40 67 62 61  61 2E 64 69 6F 6E 2E 6E
en88@gbaa.dion.n
00000040: 65 2E 6A 70 00 00 00 00  00 00 6D 61 69 6C 2E 67
e.jp......mail.g
00000050: 62 61 61 2E 64 69 6F 6E  2E 6E 65 2E 6A 70 70 6F
baa.dion.ne.jppo
```

```
[+] 0x19: Read Config: 96 bytes @ 96
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
p.gbaa.dion.ne.j
00000010: 70 00 00 00 00 00 A9 67  7F 00 00 F0 00 00 44 49
p......g......DI
00000020: 4F 4E 20 50 44 43 2F 43  44 4D 41 4F 4E 45 FF FF  ON
PDC/CDMAONE..
00000030: FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 FF FF  FF FF FF FF FF FF 00 00
................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 35 99
..............5.
[+] 0x17: Check Telephone Line
[+] 0x17: Line Free
[+] 0x12: Dialling #9677
[+] 0x21: Log in to DION
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x28: DNS Query for pop.gbaa.dion.ne.jp
[+] 0x23: Open TCP Connection to 19.55.19.55:110
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: USER ninten88
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: PASS bcdeZ01
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: STAT
[+] 0x95: POP Send: +OK 1 292
[+] 0x15: POP Recv: TOP 1 0
[+] 0x95: POP Send: +OK
From: MISSINGNO.
Date: Sat, 27 Jan 2001 12:34:56 +0900
X-Game-code: CGB-BXTJ-00
X-GBmail-type: exclusive
X-Game-result: 1 01378ffa 0185 0317 1
```

```
.
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x15: POP Recv: QUIT
[+] 0x95: POP Send: +OK
[+] 0x24: Close TCP Connection
[+] 0x22: Log out of DION
[+] 0x13: Hanging Up
[+] 0x11: Closing Session
```

This is what's going on:

1. The session is opened and immediately closed. This happens almost every time the adapter is used. Seems to be generic way to check if the adapter is connected.
2. The session is opened for real and the configuration data is read from the adapter in 96 byte chunks. Háčky documents the configuration structure pretty well here and there is also the `parse_config()` class in `mobile_adapter.py` .
3. The line is checked to make sure it's free before dialing the DION ISP dial-up number `#9677` (this varied depending on which adapter you had).
4. A DNS lookup is performed against `pop.gbaa.dion.ne.jp` (not the actual DNS protocol, which was handled by the mobile phone and not the adapter).
5. A TCP connection is opened to the POP3 server on port 110.
6. The standard POP3 protocol is spoken (this both originates and is parsed by the ROM itself!)

▶ Don't worry about the content of the email, that'll be explained further down.

7. The "game" closes the TCP connection, logs out from the DION ISP,

hangs up the line and finally closes the session with the adapter.

A couple of observations that hit me when I first saw this in action:

- The ROM doesn't handle DNS, it just has a command to ask the mobile phone to lookup a domain for it and send an IP address back.
- The ROM *does* handle POP3 itself (and HTTP too!). There is actually a partial HTTP and POP3 parser *in the ROM*! Sounds ripe for bugs?
- By far the most interesting command is `0x15` (or `0x15 | 0x80 = 0x95` if it's a response rather than a request), which is the command for transferring arbitrary data.

Fortunately, it seems that Nintendo must've distributed an SDK for working with the Mobile Adapter GB as the above behaviour is roughly the same for all the games I tried. This means that observations we make with the "reference design" of the Mobile Trainer will be largely reflected in other ROMs too.

Pokemon Crystal

The Japanese version of Pokemon Crystal was the only Pokemon game to feature support for the Mobile Adapter GB. In order to unlock most of these features in the game, you have to connect the Mobile Adapter to the GameBoy as the game loads. You'll see a "Mobile Adapter GB" splash screen as the open session/close session "handshake" happens before the regular Game Freak screen appears.

In the JP version of Crystal, the Pokemon Center in Goldenrod City is replaced by a larger building called the "Pokemon Communication Center". As well as functioning like a regular Pokemon Center, it also has some people in it you can talk to that let you send and receive Pokemon via the Mobile Adapter. Until the features have been "unlocked" as described above, there's a girl standing in front of the lady you need to speak to, preventing you from interacting with her.

Although, my eventual exploit doesn't use this feature, I spent *a lot* of time trying to find a vulnerability in this mechanism, so I'll document it here for the sake of completeness.

Sending a Pokemon with an HTTP Request

Sending a Pokemon with an HTTP Request

In order to send a Pokemon, we visit the PCC (Pokemon Communication Center) in Goldenrod City and talk to the lady at the desk to the right. After saving the game, she asks us to pick a Pokemon that we want to send away, and then to choose one that we want to receive. This is very important: both Pokemon are recorded in the trade request and, as we'll see when we go to collect our Pokemon email, we can't respond with a different Pokemon to the one that was requested.

Here is a demo of sending a Pokemon away from the game's perspective, as well as a log of what the Mobile Adapter is doing.


```
[+] 0x10: Opening Session (NINTENDO)
[+] 0x11: Closing Session
[+] 0x10: Opening Session (NINTENDO)
[+] 0x19: Read Config: 96 bytes @ 0
00000000: 4D 41 81 00 D2 C4 03 B7  D2 8D 70 A3 67 31 32 33
MA........p.g123
00000010: 34 35 36 37 38 39 00 00  00 00 00 00 00 00 00 00
456789..........
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 6E 69 6E 74
............nint
00000030: 65 6E 38 38 40 67 62 61  61 2E 64 69 6F 6E 2E 6E
en88@gbaa.dion.n
```

```
00000040: 65 2E 6A 70 00 00 00 00  00 00 6D 61 69 6C 2E 67

e.jp......mail.g
00000050: 62 61 61 2E 64 69 6F 6E  2E 6E 65 2E 6A 70 70 6F
baa.dion.ne.jppo
[+] 0x19: Read Config: 96 bytes @ 96
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
p.gbaa.dion.ne.j
00000010: 70 00 00 00 00 00 A9 67  7F 00 00 F0 00 00 44 49
p......g......DI
00000020: 4F 4E 20 50 44 43 2F 43  44 4D 41 4F 4E 45 FF FF  ON
PDC/CDMAONE..
00000030: FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 FF FF  FF FF FF FF FF FF 00 00
................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 35 99
..............5.
[+] 0x11: Closing Session
[+] 0x10: Opening Session (NINTENDO)
[+] 0x19: Read Config: 96 bytes @ 0
00000000: 4D 41 81 00 D2 C4 03 B7  D2 8D 70 A3 67 31 32 33
MA........p.g123
00000010: 34 35 36 37 38 39 00 00  00 00 00 00 00 00 00 00
456789..........
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 6E 69 6E 74
............nint
00000030: 65 6E 38 38 40 67 62 61  61 2E 64 69 6F 6E 2E 6E
en88@gbaa.dion.n
00000040: 65 2E 6A 70 00 00 00 00  00 00 6D 61 69 6C 2E 67
e.jp......mail.g
00000050: 62 61 61 2E 64 69 6F 6E  2E 6E 65 2E 6A 70 70 6F
baa.dion.ne.jppo
[+] 0x19: Read Config: 96 bytes @ 96
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
p.gbaa.dion.ne.j
00000010: 70 00 00 00 00 00 A9 67  7F 00 00 F0 00 00 44 49
p......g......DI
00000020: 4F 4E 20 50 44 43 2F 43  44 4D 41 4F 4E 45 FF FF  ON
PDC/CDMAONE..
```

```
00000030: FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 FF FF  FF FF FF FF FF FF 00 00
................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 35 99
..............5.
[+] 0x17: Check Telephone Line
[+] 0x17: Line Free
[+] 0x12: Dialling #9677
[+] 0x21: Log in to DION
[+] 0x28: DNS Query for gameboy.datacenter.ne.jp
[+] 0x23: Open TCP Connection to 19.55.19.55:80
[+] 0x15: HTTP Recv: GET /cgb/download?name=/01/CGB-
BXTJ/exchange/index.txt
[+] 0x95: HTTP Send: 186 bytes
[+] 0x95: HTTP Server Closed Connection
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x17: Check Telephone Line
[+] 0x17: Line Busy
[+] 0x23: Open TCP Connection to 19.55.19.55:80
[+] 0x15: HTTP Recv: GET /cgb/upload?name=/01/CGB-
BXTJ/exchange/10upload.cgi
[+] 0x95: HTTP Send: 57 bytes
[+] 0x95: HTTP Server Closed Connection
[+] 0x23: Open TCP Connection to 19.55.19.55:80
[+] 0x15: HTTP Recv: GET /cgb/upload?name=/01/CGB-
BXTJ/exchange/10upload.cgi
[+] 0x95: HTTP Send: 49 bytes
[+] 0x95: HTTP Server Closed Connection
[+] 0x23: Open TCP Connection to 19.55.19.55:80
[+] 0x15: HTTP Recv: POST /cgb/upload?name=/01/CGB-
BXTJ/exchange/10upload.cgi
00000000: 6E 69 6E 74 65 6E 38 38  40 67 62 61 61 2E 64 69
```

```
ninten88@gbaa.di
00000010: 6F 6E 2E 6E 65 2E 6A 70  00 00 00 00 00 00 01 37
on.ne.jp.......7
00000020: 8F FA 01 9A 03 4A 87 D8  8C 50 50 9A 00 0F 4B 22
.....J...PP...K"
00000030: 94 01 37 09 B8 D5 C7 05  EA 9C E1 4A E5 59 D6 96
..7........J.Y..
00000040: 4D CF 1E 19 0F 14 FF 00  85 81 55 00 00 01 1F 01
M.........U.....
00000050: 1F 00 CB 00 F8 00 D4 00  DC 00 F9 87 D8 8C 50 50
..............PP
00000060: A0 05 95 82 9F 00 00 00  00 00 00 00 00 00 00 00
................
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00
...............
OrderedDict([('email', 'ninten88@gbaa.dion.ne.jp'),
             ('trainer_id', bytearray(b'\x017')),
             ('secret_id', bytearray(b'\x8f\xfa')),
             ('offer_gender', 'MALE'),
             ('offer_species', 154),
             ('request_gender', 'EITHER'),
             ('request_species', 74),
             ('trainer_name', bytearray(b'\x87\xd8\x8cPP')),
             ('pokemon_struct',
              OrderedDict([('pkm', 154),
                           ('item', 0),
                           ('move1', 15),
                           ('move2', 75),
                           ('move3', 34),
                           ('move4', 148),
                           ('ot_id', bytearray(b'\x017')),
                           ('exp', bytearray(b'\t\xb8\xd5')),
                           ('hp_ev', bytearray(b'\xc7\x05')),
                           ('att_ev', bytearray(b'\xea\x9c')),
                           ('def_ev', bytearray(b'\xe1J')),
                           ('spd_ev', bytearray(b'\xe5Y')),
                           ('spc_ev', bytearray(b'\xd6\x96')),
                           ('iv', bytearray(b'M\xcf')),
```

```
                              ('pp1', 30),
                              ('pp2', 25),
                              ('pp3', 15),
                              ('pp4', 20),
                              ('frndshp', 255),
                              ('pokerus', 0),
                              ('caught', bytearray(b'\x85\x81')),
                              ('level', 85),
                              ('status', 0),
                              ('curr_hp', bytearray(b'\x01\x1f')),
                              ('max_hp', bytearray(b'\x01\x1f')),
                              ('att', bytearray(b'\x00\xcb')),
                              ('def', bytearray(b'\x00\xf8')),
                              ('spd', bytearray(b'\x00\xd4')),
                              ('sp_att', bytearray(b'\x00\xdc')),
                              ('sp_def', bytearray(b'\x00\xf9'))])),
             ('pokemon_ot_name', bytearray(b'\x87\xd8\x8cPP')),
             ('pokemon_nickname', bytearray(b'\xa0\x05\x95\x82\x9f')),
             ('mail_data',
              OrderedDict([('message',

bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00'

b'\x00\x00\x00\x00\x00\x00\x00\x00'

b'\x00\x00\x00\x00\x00\x00\x00\x00'

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00')),
                           ('sender_name',
bytearray(b'\x00\x00\x00\x00\x00')),
                           ('sender_trainer_id',
bytearray(b'\x00\x00')),
                           ('pokemon_species', 0),
                           ('item_index', 0)]))])
[+] 0x95: HTTP Send: 20 bytes
[+] 0x95: HTTP Server Closed Connection
[+] 0x22: Log out of DION
[+] 0x13: Hanging Up
[+] 0x11: Closing Session
```

There's some interesting things to point out here:

- The game opens a TCP connection to `gameboy.datacenter.ne.jp` on port 80
- First, it sends a `GET` request to `/cgb/download?name=/01/CGB-BXTJ/exchange/index.txt`
  - In the response, we provided two links (see `mobile_trainer.py`)
    - `http://gameboy.datacenter.ne.jp/cgb/upload?name=/01/CGB-BXTJ/exchange/10upload.cgi`
    - `http://gameboy.datacenter.ne.jp/cgb/upload?name=/01/CGB-BXTJ/exchange/cancel.cgi`
- Next the game sends another `GET` but this time to the `10upload.cgi` URL that we gave it
  - We have to return a `401`, and set the `Gb-Auth-ID` HTTP header to something. I followed Háčky's example and set it to `HAIL GIOVANNI`.
- Then another `GET` to the same path, but this time the game sends the `Gb-Auth-ID` header in the request.
  - We catch this request here and return a `200` this time
- Lastly, the game sends a `POST` to the `10upload.cgi` endpoint with a big binary blob of data in the body

Importantly, there's the email of the requester, the offered species (`154` is Meganium, as shown in the GIF above) and the requested species (`74` is Geodude, which is shown in the GIF further down) - what a terrible trade! Following that is a 143 byte struct which contains a lot of important information. In particular, it contains the raw PKM data structure which is what gets written to SRAM and used by the game to store information about Pokemon - this is what kept me digging this hole to try and inject corrupted Pokemon data into the game!

At this point, the game actually enforces a 1 hour wait before it'll let you check your inbox to see if you have a match on your trade.

Receiving Pokemon in an Email

Once the 1 hour wait has elapsed, we can go back and talk to the lady in the PCC and she'll helpfully check our inbox for us to see if we have a new Pokemon waiting. Here's what it looks like as this goes down.


```
[+] 0x10: Opening Session (NINTENDO)
[+] 0x11: Closing Session
[+] 0x10: Opening Session (NINTENDO)
[+] 0x19: Read Config: 96 bytes @ 0
00000000: 4D 41 81 00 D2 C4 03 B7  D2 8D 70 A3 67 31 32 33
MA........p.g123
00000010: 34 35 36 37 38 39 00 00  00 00 00 00 00 00 00 00
456789..........
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 6E 69 6E 74
............nint
00000030: 65 6E 38 38 40 67 62 61  61 2E 64 69 6F 6E 2E 6E
en88@gbaa.dion.n
00000040: 65 2E 6A 70 00 00 00 00  00 00 6D 61 69 6C 2E 67
e.jp......mail.g
00000050: 62 61 61 2E 64 69 6F 6E  2E 6E 65 2E 6A 70 70 6F
baa.dion.ne.jppo
[+] 0x19: Read Config: 96 bytes @ 96
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
```

```
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
p.gbaa.dion.ne.j

00000010: 70 00 00 00 00 00 A9 67  7F 00 00 F0 00 00 44 49
p......g......DI
00000020: 4F 4E 20 50 44 43 2F 43  44 4D 41 4F 4E 45 FF FF  ON
PDC/CDMAONE..
00000030: FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 FF FF  FF FF FF FF FF FF 00 00
................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 35 99
..............5.
[+] 0x11: Closing Session
[+] 0x10: Opening Session (NINTENDO)
[+] 0x19: Read Config: 96 bytes @ 0
00000000: 4D 41 81 00 D2 C4 03 B7  D2 8D 70 A3 67 31 32 33
MA........p.g123
00000010: 34 35 36 37 38 39 00 00  00 00 00 00 00 00 00 00
456789..........
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 6E 69 6E 74
............nint
00000030: 65 6E 38 38 40 67 62 61  61 2E 64 69 6F 6E 2E 6E
en88@gbaa.dion.n
00000040: 65 2E 6A 70 00 00 00 00  00 00 6D 61 69 6C 2E 67
e.jp......mail.g
00000050: 62 61 61 2E 64 69 6F 6E  2E 6E 65 2E 6A 70 70 6F
baa.dion.ne.jppo
[+] 0x19: Read Config: 96 bytes @ 96
00000000: 70 2E 67 62 61 61 2E 64  69 6F 6E 2E 6E 65 2E 6A
p.gbaa.dion.ne.j
00000010: 70 00 00 00 00 00 A9 67  7F 00 00 F0 00 00 44 49
p......g......DI
00000020: 4F 4E 20 50 44 43 2F 43  44 4D 41 4F 4E 45 FF FF  ON
PDC/CDMAONE..
00000030: FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 FF FF  FF FF FF FF FF FF 00 00
................
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 35 99
..............5.
```

```
[+] 0x17: Check Telephone Line
[+] 0x17: Line Free
[+] 0x12: Dialling #9677
[+] 0x21: Log in to DION
[+] 0x28: DNS Query for pop.gbaa.dion.ne.jp
[+] 0x23: Open TCP Connection to 19.55.19.55:110
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: USER ninten88
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: PASS ABCDE
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: STAT
[+] 0x95: POP Send: +OK 1 292
[+] 0x15: POP Recv: LIST 1
[+] 0x95: POP Send: +OK 1 292
[+] 0x15: POP Recv: TOP 1 0
[+] 0x95: POP Send: +OK
From: MISSINGNO.
Date: Sat, 27 Jan 2001 12:34:56 +0900
X-Game-code: CGB-BXTJ-00
X-GBmail-type: exclusive
X-Game-result: 1 01378ffa 0185 0317 1

.
[+] 0x15: POP Recv: RETR 1
[+] 0x95: POP Send: +OK
From: MISSINGNO.
Date: Sat, 27 Jan 2001 12:34:56 +0900
X-Game-code: CGB-BXTJ-00
X-GBmail-type: exclusive
X-Game-result: 1 01378ffa 0185 0317 1

4+Pj4+OFAP////8BNwAfQAAAAAAAAAAAAABM9SMeDyhyAJQQFAAAADUANQAcAB0AIQAZACGH
2IxQUIHjG4FQAAAAAAAAAAAAA
[+] 0x95: POP Send: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

.
[+] 0x15: POP Recv: DELE 1
[+] 0x95: POP Send: +OK
[+] 0x15: POP Recv: QUIT
```

```
[+] 0x95: POP Send: +OK
[+] 0x24: Close TCP Connection
[+] 0x22: Log out of DION
[+] 0x13: Hanging Up
[+] 0x11: Closing Session
```

Okay, let's pick this apart too:

- After the usual fluff of reading the configuration data, the game opens a TCP connection to `pop.gbaa.ne.jp` on port 110
- The POP messages you see above actually originate from the game itself (notice the password `ABCDE` that I had to type in in the above GIF)
- According to Háčky, the `From:` field is only checked to be present, the actual content (`MISSINGNO.`) isn't checked at all.
- We have a couple of custom email headers:
  - `X-Game-code` : Indicates which game the email is meant to be processed by. In this case `CGB-BXTJ-00` is the JP version of Pokemon Crystal.
  - `X-GBmail-type` : Here, `exclusive` , indicates that only the intended Game (Crystal) should process this email. Other games will ignore the email if it's not intended for it if this header is set.
  - `X-Game-result` : Game-dependent.

The `X-Game-result` header is broken down as follows:

Always present	Trainer ID	Secret ID	Offered Gender	Offered Species	Requested Gender	Requested Species	Search Result
1	0137	8ffa	01	85	03	17	1

In our case,

- Offered Gender is `01` , which corresponds to Male
- Offered Species is `85` , i.e. `133` in decimal which corresponds to Eevee

- Requested Gender is `03` , which means Any

- Requested Species is `17` , i.e. `23` in decimal whcih corresponds to Ekans, as we see in the GIF
- Search Result is either `1` or `2` to indicate that a Pokemon was found to complete to the trade or not. If the trade failed, then the original Pokemon is returned to the player.

Why is any of this important? Well, it all comes down the Base64 blob in the body of the email. This is the same format as the request that was `POST` ed to the `10upload.cgi` endpoint earlier, but this time, it only contains the 5 byte player name, followed by the PKM Struct, with some extra data like any mail the Pokemon is holding.

Where does the start of this struct come from with the Trainer ID and Secret ID, etc? It comes from the `X-Game-result` header! This makes it a little more difficult to scew around with as we *have* to give the game what it asked for Pokemon-wise. The observant among you dear readers may have noticed that the two example trades above don't match - the first is a Meganium/Geodude trade, while the second is a Eevee/Ekans trade. This is down to the 1 hour enforced wait - I just have a couple of save files backed up so I can restore a save that's already waited for an hour. I didn't want to wait an hour to record that GIF so just used the one I already had handy :).

So what can we do? First, let's look at this Base64 encoded struct in the parser I wrote:

```
OrderedDict([('email', 'ninten88@gbaa.dion.ne.jp'),
             ('trainer_id', bytearray(b'\x017')),
             ('secret_id', bytearray(b'\x8f\xfa')),
             ('offer_gender', 'MALE'),
             ('offer_species', 133),
             ('request_gender', 'EITHER'),
             ('request_species', 23),
             ('trainer_name', bytearray(b'\xe3\xe3\xe3\xe3\xe3')),
             ('pokemon_struct',
              OrderedDict([('pkm', 133),
                           ('item', 0),
```

```
                                ('move1', 255),
                                ('move2', 255),
                                ('move3', 255),
                                ('move4', 255),
                                ('ot_id', bytearray(b'\x017')),
                                ('exp', bytearray(b'\x00\x1f@')),
                                ('hp_ev', bytearray(b'\x00\x00')),
                                ('att_ev', bytearray(b'\x00\x00')),
                                ('def_ev', bytearray(b'\x00\x00')),
                                ('spd_ev', bytearray(b'\x00\x00')),
                                ('spc_ev', bytearray(b'\x00\x00')),
                                ('iv', bytearray(b'L\xf5')),
                                ('pp1', 35),
                                ('pp2', 30),
                                ('pp3', 15),
                                ('pp4', 40),
                                ('frndshp', 114),
                                ('pokerus', 0),
                                ('caught', bytearray(b'\x94\x10')),
                                ('level', 20),
                                ('status', 0),
                                ('curr_hp', bytearray(b'\x005')),
                                ('max_hp', bytearray(b'\x005')),
                                ('att', bytearray(b'\x00\x1c')),
                                ('def', bytearray(b'\x00\x1d')),
                                ('spd', bytearray(b'\x00!')),
                                ('sp_att', bytearray(b'\x00\x19')),
                                ('sp_def', bytearray(b'\x00!'))])),
              ('pokemon_ot_name', bytearray(b'\x87\xd8\x8cPP')),
              ('pokemon_nickname', bytearray(b'\x81\xe3\x1b\x81P')),
              ('mail_data',
               OrderedDict([('message',

bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00'

b'\x00\x00\x00\x00\x00\x00\x00\x00'

b'\x00\x00\x00\x00\x00\x00\x00\x00'

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00')),
```

```
                            ('sender_name',
 bytearray(b'\x00\x00\x00\x00\x00')),
                            ('sender_trainer_id',
 bytearray(b'\x00\x00')),
                            ('pokemon_species', 0),
                            ('item_index', 0)]))])
```

You might have already noticed some weird things in here. In particular, the moves are all set to `0xff = 255`. You may be wondering if there are that many moves in Gen 2, and the answer is no. According to Bulbapedia, the moves only go up to `251` for Gen 2, so `255` is certainly not valid. In fact, in the GIF further up, when I bring up the summary for Ekans, the game locks up and freezes when I hit A to go to the next screen where the moveset would be shown. I spent a lot of time trying to figure out how to screw with this mechanic and time and time again all I managed to do was lock the game up and occasionally reboot.

The 0x1500 Trick

After doing a lot of research on existing Pokemon Crystal exploits (often used by the speedrunning community) I stumbled across the “0x1500 Trick”. The core idea behind this trick is down to the character map used by Pokemon Crystal. ~~Nintendo~~ Game Freak certainly don’t bother using ASCII - you can see the full layout here (spoiler: it will be very important later on!). On that page, scrolling down the Japanese layout, we see that a lot of characters are just left as `*`. In general, these are control characters than ~~Nintendo~~ Game Freak used for various things. Unfortunately, this trick doesn’t work on the JP version (or at least I couldn’t get it to work), but I spent so long on it, this might prove useful to someone else (and it does eventually set me on the right path to successful exploitation).

The `0x15` control character is used for “Mobile Scripts”. I’m not 100% sure exactly what each of these do, but the excellent PokeCrystal project on GitHub gives some hint. Unfortunately for us, this is the US version and the JP release is different enough that I had to spend a lot of time disassembling the JP ROM (especially around the Mobile

Adapter functionality!).

In a nutshell, when the game’s print routine parses a string, if it hits a `0x15` byte, it jumps to the `RunMobileScript` routine at location `$70aa` in bank `5F` (all these offsets are for the JP version of the game, sha1sum: `95127b901bbce2407daf43cce9f45d4c27ef635d`). Here, we have:

```
RunMobileScript:        ; 5f:$70aa
    ld a, $06
    call OpenSRAM
    inc de

.loop:
    call _RunMobileScript
    jr c, .finished
    jr .loop

.finished:
    call CloseSRAM
    ret

_RunMobileScript:       ; 5f:$70bb
    ld a, [de]
    inc de
    cp $50

    jr z, .finished

    cp $10
    jr nc, .finished

    dec a
    push de
    ld e, a
    ld d, $00
    ld hl, .Jumptable
    add hl, de
    add hl, de
    ld a, [hl+]
    ld h, [hl]
```

```
    ld l, a

    jp hl

.finished:
    scf
    ret
```

What on earth is going on here? The interesting part is down at the `cp $10` instruction. What happens is the game reads the next byte following `0x15` and makes sure its less than `0x10`, then *decrements it* before using it as an offset into the `.Jumptable`. This is so that the string `1501` will call the *first entry in the table* - someone at ~~Nintendo~~ Game Freak in 2001 liked their arrays starting at 1!

The vulnerability here is a simple one by today's standards: if we get the print routine to parse a string that contains `1500`, the `00` byte will pass the `cp $10` test, then be decremented and underflow to `ff` before being used as the jumptable offset at the `jp hl` instruction. This will cause execution to jump way past ROM and into WRAM at location `$cd52`. I haven't been able to work out who first found this trick, but it's a nice one!

In speedruns, it's common to now use several different tricks involving the Pokemon boxes in the PC to control the memory in WRAM around `$cd52` and then use a glitch to get `1500` into a string that the print routine will parse.

Well, we can control a bunch of strings in our email! However, it seems that ~~Nintendo~~ Game Freak were one step ahead of me here - if you overwrite any of the strings in the big base64'd email struct (like trainer name), with any of the control characters (like `0x15`), either the lady in the PCC tells you that something went wrong and you get your original Pokemon back, or the troublesome bytes get replaced with `E6`, which is just `?` in-game. :(

At this point, I felt like I'd hit a dead end. However, I kept thinking about this problem for a while and eventually decided to look at some of the features that hadn't already been reversed engineered by Háčky.

Enter, the battle protocol.

The Battle Protocol

Something that Háčky briefly mentions in their writeup is the Battle Colliseum mode. This is accessed by talking to one of the ladies upstairs in any Pokemon Center. But, as you might expect, it comes with a few quirks.

There is a 10 minute daily timer on the use of this feature. In various places online, there is reference to an "unlimited battle adapter" that removes this limit. This turned into an enormous rabbit hole ~~which once again turned out to be fruitless. There *is* a mechanism in the game for unlimited play, but it requires manually modifying the save file. After spending many weeks knee-deep in the disassembly, I am 99% certain that it is not possible to remove this limit from the game with the Mobile Adapter alone, despite what is often claimed online. The fact that it's possible to remove by editing your save leads me to believe that it was intended to only be used by Nintendo/Game Freak internally during testing.~~

> Update: As mid-kid pointed out to me, on Dan Docs, it clearly explains that in the reply to the "Telephone Status" command (`0x17`), "The third byte is unknown, and usually hardcoded to 0. However, Pokemon Crystal reacts to the third byte being 0xF0 by allowing the player to bypass the 10 min/day battle time limit.". By modifying the `mobile_trainer.py` script, we can set this third byte to `0xF0` and the game does indeed believe that we are using an unlimited adapter. I can't seem to find much about this adapter online though, so I do wonder whether it was actually ever sold or only used internally to Game Freak for testing? Does anyone have more details on it?

Okay, that's not too big a deal, I just have to keep restoring a save to before the timer expired. Another quirk of this mode is that (similar to the Battle Frontier in Pokemon Emerald), you have to choose three Pokemon from your team to battle with.

three Pokemon from your team to battle with.

From my research online, this protocol hasn't been documented well yet, so I'm going to go into a little more detail than I did in the other parts of the protcol.

First thing I noticed is that the game asks you for a phone number to dial - I assume this must've been the mobile number of your friend you wanted to battle against (still blows my mind that all this was possible in 2001!). After dialing that number, the game goes straight to the `TRANSFER DATA` command of the Mobile Adapter GB, without setting a port number via the open TCP/UDP commands. This makes it easy to catch because the port is set to `0` by default in the `mobile_adapter.py` script.

The simplest thing to do to begin with was to just echo back whatever the game sends. By doing this, I'm able to launch a battle against myself! If you're able to defeat yourself, the game goes into a loop:

Here's the high level packet structure I was able to figure out:

- The first byte is always `0xff`
- ~~The second byte is some kind of identifier, possibly a command ID. Packets that share the same first byte are very similar in structure. I'll refer to this as the `packet type`.~~
  - In retrospect, this is clearly the length of the packet. Thanks to

mid-kid for pointing it out!

- The byte third before the end is a packet counter, it starts at `0x01` and increments with each subsequent packet.
  - This counter only increments *after* the response packet, so I guess it's more of a sequence ID.
- The penultimate two bytes are a checksum, which is just a little endian sum of all the preceeding bytes modulo `2^16`.

There are *many* messages sent back and forth that consist of only `0xff` and seems to be a keepalive (seeing as this is essentially real-time communication between two GameBoys).

> As pointed out by mid-kid, and explained on Dan docs, this byte is actually a socket id. When transfer data packets are sent without first opening a TCP/UDP connection, this byte is ignored and always set to `0xff`.

Let's take a look at an example set of messages sent during a battle and go through them one-by-one. Note that the final byte of each message is the packet counter described above. Something important to keep in mind when looking at these is that `0x50` or `P` in ASCII is used as a string terminator by Pokemon Crystal. This makes it easier to identify strings that are being sent around without having to lookup *every* byte to see if it corresponds to a printable character in the character map. Another helpful piece of information is that `87 D8 8C` is the player name in the save that I found online (thanks to クリス, or "Kurisu" whoever you are!).

> Clearly, it's been too long since I last played Gen II casually - as mid-kid pointed out, "Kurisu" is the Japanese form for "Kris" - the default female character name.

```
[-] Battle: Initial Packet Received
00000000: 19 67 10 01 6C 69 6D 69  74 5F 63 72 79 73 74 61
.g..limit_crysta
00000010: 6C 00 01                                          l..
```

This initial packet is always of packet type `0x15`. I'm not entirely sure what the first 4 bytes are (`19 67 10 01`), but the `limit_crystal` string indicates that the player has the 10 minute timer enabled. If you enabled the mythical "unlimited battle adapter", the string that gets sent in this message is `free__crystal`.

```
[-] Battle: 0x0d Packet Received
00000000: 9C E9 34 7E C8 11 59 A0  E7 2D 02                 ..4~..Y..-.
```

This message appears to always be random and always 10 bytes long.

> A good theory for this packet is that it's the RNG seed. Thanks again mid-kid.

```
[-] Battle: 0x4d Packet Received
00000000: 01 87 D8 8C 50 50 50 87  D8 8C 50 50 50 01 37 8F
....PPP...PPP.7.
00000010: FA FF BF FF EF FF FF 00  04 00 FF 00 00 00 00 00
................
00000020: 10 00 00 FF FF EE FF FF  FF FF EF 00 00 00 00 00
................
00000030: 00 00 00 FF FF FF FF FF  FF FF FF 00 00 00 00 00
................
```

```
00000040: 00 00 01 FF FF F7 FF FF  FF FF 03                 ...........
```

This packet has been bugging me for a while. The `87 D8 8C 50 50` is the 5 byte Player Name (and it appears twice), but this blob of bytes doesn’t seem to appear verbatim in the save file, indicating it’s assembled on the fly.

```
[-] Battle: 0x53 Packet Received
00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000050: 04                                                .

[-] Battle: 0x53 Packet Received
00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000050: 05                                                .

[-] Battle: 0x53 Packet Received
00000000: 00 00 00 00 00 00 00 00  BA DE C6 C1 CA 50 50 50
.............PPP
00000010: 50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50
PPPPPPPPPPPPPPPP
00000020: 50 50 50 50 50 50 50 50  50 00 00 00 00 00 00 00
```

```
00000020: 50 50 50 50 50 50 50 50  50 00 00 00 00 00 00 00
PPPPPPPPP.......

00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000050: 06                                               .
```

Ignoring the final counter byte (`04` , `05` and `06` here), these three messages are the contiguous chunk of save data from `0x600` - `0x6f0` (3 lots of `0x50` bytes). I haven't been able to figure out what these bytes are. The many `0x50` bytes indicates that this is a string of 32 bytes (with an extra `0x50` to always "null" terminate). Looking up `BA DE C6 C1 CA` in the character encoding table gives us `こんにちは` or "Kon'nichiwa"! So this is clearly a greeting towards the end of this blob.

I wondered if maybe this string is printed during the battle, but didn't see those characters at any point. Not being able to read Japanese, I modified my script to alter this packet before replying by inserting the `15 00` string discussed earlier. My breakpoint at `$cd52` never triggered like before, so I'm pretty confident that this is just unused. Perhaps it's a leftover from a feature that was never finished?

```
[-] Battle: 0x0f Packet Received
00000000: 00 00 00 00 00 00 00 00  00 00 00 00 07
.............
```

This packet is always 12 nulls. Not really much more to say.

```
[-] Battle: 0x53 Packet Received
00000000: 87 D8 8C 50 50 50 05 82  8F F9 9A 17 FF FF 01 37
...PPP.........7
00000010: 82 00 46 7F 39 FA 01 37  03 51 9C 31 1B 3F 5A 52
```

```
..F.9..7.Q.1.?ZR
00000020: D2 29 D9 25 6B EA AA 0F  0F 0F 0F ED 00 9E A6 37
.).%k.........7
00000030: 00 00 00 B8 00 B8 00 AE  00 7A 00 76 00 5F 00 8B
.........z.v._..
00000040: 8F 92 F9 22 AD 9C 01 37  03 4A 08 0B AC 0E 7A 0D
..."...7.J....z.
00000050: 08                                              .

[-] Battle: 0x53 Packet Received
00000000: DD 0C 04 0C 37 DF 97 0F  0F 0F 0A 93 00 B2 3D 37
....7.........=7
00000010: 00 00 01 08 01 08 00 94  00 64 00 37 00 5A 00 8C
.........d.7.Z..
00000020: F9 00 10 38 69 13 01 37  04 CD 9A 06 55 07 17 07
...8i..7....U...
00000030: 36 06 BE 07 16 FE 4E 23  05 14 0F 7D 00 BC 1F 3F
6.....N#...}...?
00000040: 00 00 00 DE 00 DE 00 8F  00 C0 00 9A 00 8E 00 DE
................
00000050: 09                                              .

[-] Battle: 0x53 Packet Received
00000000: 9A 00 0F 4B 22 94 01 37  09 B8 D5 C7 05 EA 9C E1
...K"..7........
00000010: 4A E5 59 D6 96 4D CF 1E  19 0F 14 FF 00 85 81 55
J.Y..M.........U
00000020: 00 00 01 1F 01 1F 00 CB  00 F8 00 D4 00 DC 00 F9
................
00000030: 17 00 FF FF FF FF 01 37  00 1F 40 00 00 00 00 00
.......7..@.....
00000040: 00 00 00 00 00 4C F5 2B  2B 2B 2B 46 00 94 10 14
.....L.++++F....
00000050: 0A                                              .

[-] Battle: 0x53 Packet Received
00000000: 00 00 00 2D 00 2D 00 1E  00 1B 00 21 00 17 00 1C
...-.-.....!....
00000010: AF 00 2D CC 94 00 01 37  00 00 64 00 00 00 00 00
..-....7..d.....
```

```
00000020: 00 00 00 00 00 56 CD 28  14 14 00 DB 00 81 90 05  .....V.
(........
00000030: 00 00 00 13 00 13 00 07  00 0C 00 08 00 0A 00 0C
................
00000040: 87 D8 8C 50 50 50 87 D8  8C 50 50 50 87 D8 8C 50
...PPP...PPP...P
00000050: 0B                                                .

[-] Battle: 0x3b Packet Received
00000000: 50 50 87 D8 8C 50 50 50  87 D8 8C 50 50 50 87 D8
PP...PPP...PPP..
00000010: 8C 50 50 50 06 AD A5 13  8C 50 85 1A 09 AB 50 50
.PPP.....P....PP
00000020: A6 06 80 50 50 50 A0 05  95 82 9F 50 81 E3 1B 81
...PPP.....P....
00000030: 50 50 93 08 41 E3 50 50  0C                       PP..A.PP.
```

Following the same trick as the previous block of `0x53` packets, if we remove the final counter bytes, we end up with a large contiguous blob of bytes (clearly `0x50` must be the maximum packet size for this battle protocol). I had a feeling this this blob must contain the details of the Pokemon in my team (especially as they haven't been transferred yet and we're almost out of packets!).

Having noticed that we earlier got a whole sled of bytes straight from the save game, I tried my luck and picked an entropic looking sequence and searched for it in the `.sav` file from the emulator. I struck gold! The byte `05` at offset `0x06` in the first packet lies at offset `0x281a` of the save file followed by rest of the bytes, all the way to the end of the `0x3b` type packet.

I tried my luck and looked up the save data structure to see if `0x281a` corresponded to anything and it turns out to be the team pokemon list in the Japanese version of Crystal! I wrote a *very* basic parser for this structure to make sure that it was making sense with the bytes that were being sent.

The structure is as follows:

Offset	Bytes	Meaning
0x0:0x6	87 D8 8C 50 50 50	Player Name
0x6:0x7	05	Size of Party
0x7:0xc	82 8F F9 9A 17	Species of Pokemon in Party
0xc:0x10	FF FF 01 37	Unknown
0x10:0x40	82 00 46 ... 5F 00 8B	Pokemon 1 (Gyarados)
0x40:0x70	8F 92 F9 ... 5A 00 8C	Pokemon 2 (Snorlax)
0x70:0xa0	F9 00 10 ... 8E 00 DE	Pokemon 3 (Lugia)
0xa0:0xd0	9A 00 0F ... DC 00 F9	Pokemon 4 (Feraligatr)
0xd0:0x100	17 00 FF ... 17 00 1C	Pokemon 5 (Ekans)
0x100:0x130	AF 00 2D ... 0A 00 0C	Pokemon 6 (Togepi)
0x130:0x154	87 D8 8C ... 50 50 50	(5 byte OT name + 1 byte terminator) * 6
0x154:0x178	06 AD A5 ... E3 50 50	(5 byte Pokemon name + 1 byte terminator) * 6

All this is well and good, but what on Earth is Togepi doing in there? It turns out that this is just a quirk of the Pokemon Crystal save file. Togepi is actually in the PC as the Party Count is set to `5` . I

suppose it prevents having to constantly keep resizing list structures and offsets in SRAM.

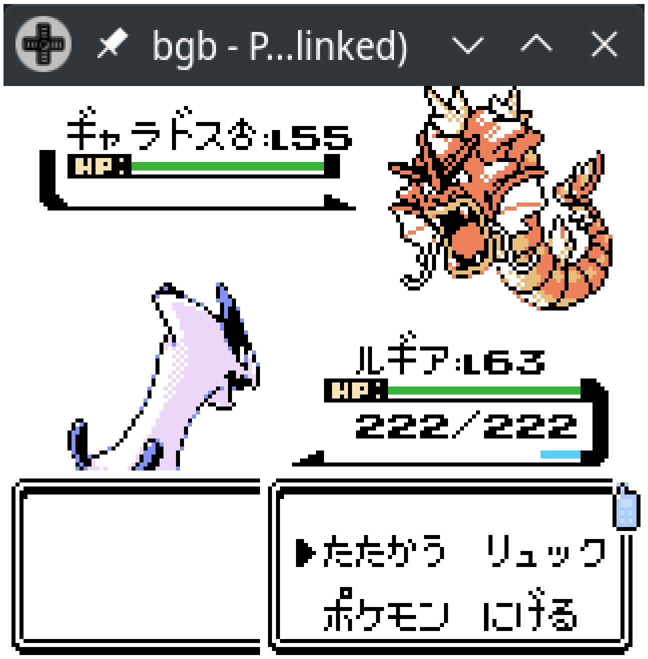
With that out the way, things are starting make a lot more sense!

```
[-] Battle: 0x09 Packet Received
00000000: 02 03 04 00 00 0C 0D                              .......
```

As mentioned earlier, when talking to the lady in the Pokemon Center to enter this mode, you're asked to select three Pokemon from your party. We saw above that the data for your entire party is sent to the opponent, so what's going on? For whatever reason, in my testing I chose the final three Pokemon in my party each time rather than the first three in the hopes that this choice would be apparent in this reverse engineering stage. Taking a look at the packet above, hopefully something will stand out to you! The first three bytes `02 03 04` is the Pokemon selection from the party (beginning at `00` for the first entry, naturally).

In order to test this theory, I again modified the packet so that in the response, the first byte was `00` instead of `02` , and sure enough, I was faced with Gyarados instead of Lugia as my opponent.

As for the remaining three bytes before the counter `00 00 0C` ? I have no idea. Corrupting them seems to have no effect.




```
[-] Battle: 0x0c Packet Received
00000000: 00 74 65 74 73 75 6A 69  00 0E                  .tetsuji..

[-] Battle: 0x0c Packet Received
00000000: 01 74 65 74 73 75 6A 69  00 0F                  .tetsuji..

[-] Battle: 0x0c Packet Received
00000000: 05 74 65 74 73 75 6A 69  00 10                  .tetsuji..

[-] Battle: 0x0c Packet Received
00000000: 0F 74 65 74 73 75 6A 69  00 11                  .tetsuji..
```

At this point, the battle has begun and the `0x0c` packets indicate the moves that are being used. The first byte indicates the action:

- `00` - `03` are the Pokemon's moves in the order they appear in it's struct
- `04`, `05` and `06` are swapping out to the first, second or third Pokemon in the party respectively (remember, we only get three Pokemon in these battles!).
- `0F` is running from the battle - yes we can actually run from a

- `0F` is running from the battle - yes we can actually run from a trainer battle!

However, you’ve probably noticed something in common with these `0x0c` packets and the title of this post! I was struck by the presence of `tetsuji` in these packets (a null-terminated ASCII string!). I felt like “Tetsuji” sounded like someone’s name, but not knowing Japanese, I turned to my friend Kyo to help me check. As far as I can tell, this is an easter egg *possibly* left by Tetsuji Oota (list of staff who worked on Pokemon Crystal here). If so, it’s possible that this has gone undiscovered for over 20 years - pretty cool! In honour of this, it seemed fitting to name the eventual exploit “Tetsuji”, hence the name of this write up and repo.

Finding the Vulnerability

Oof, that’s a lot of background before we even get to a vuln! Timeline wise, I was still figuring out the `0x1500` trick while playing with the battle protocol so I was actually trying to find away of injecting the bytes `15 00` into one of the strings. I started by searching for the 5 byte player name in my save `87 D8 8C 50 50` (the `50` s are just padding) and simply replacing it with `15 00 50 50 50` , while setting a breakpoint at `$cd52` . Sadly, as previously explained, the `0x1500` method seems to be a dead end on the Japanese version of Crystal and I couldn’t get it to work.

Re-reading once again the GlitchCity page on the 1500 trick, there are some “Self-contained setup and bootstrap” strings. Although none of these worked for me (it even says that they might not on the JP version), I did spend a lot of time single-stepping through the ROM to try and understand how these “mobile scripts” are handled.

While following the “self-contained” examples step-by-step in the BGB debugger, I observed that the `15` control character often “consumed” a number of bytes that followed it, depending on which byte followed `15` (remember, this byte corresponds to an index in the “mobile script” jump table). The other thing I noticed on the GlitchCity page was the example `4F 15 08 05 C9 00 [code] 37 C9` . What’s `4F` do? Checking the character table once more, and it appears that `4F` is another control character!

> As explained to me by mid-kid, `4F` is the <LINE> character. It sets the text pointer to a known location, and is later written to the address that is jumped to (and therefore the bytes that make up the pointer are actually executed first prior to reaching the payload). As it turns out, mid-kid was also able to get this self-contained example to work on the Japanese version - unfortunately the reason why it didn't work for me is that the name fields (only 5 bytes in the JP version) seem to be too short to anything very interesting.

At this point, I decided to try some of the other control characters from the table (indicated by a `*`). For whatever reason, I decided to try `3F` rather than `4F` and execution jumped to an address just a few bytes before the mobile adapter buffer!

As described above, the `0x53`-type battle protocol packet is used to transfer the save data containing the party data. However, the party data only began at offset `0x6`, while the first 5 bytes of the blob contains the Player Name. This is the name that is printed to the screen when it says "so-and-so wants to battle!".

By replacing this with `3F 00 00`, execution reliably jumps to just before `$ca42`, which is where the Mobile Adapter GB SDK writes packets to. By the time the print routine is called, we've also sent 3 more `0x53`-type packets, an `0x3b`-type packet and finally an `0x09`-type packet (which indicates which Pokemon from our party that we're battling with).

This buffer isn't cleared inbetween writes! If you break at the point immediately after the final packet (`0x09`-type) is copied into memory, you can see the remnants of the packet that was sent prior to it - the `0x3b`-type packet! This packet is the final part of the party save data which contains the OT and Pokemon names. The final packet sent after this one is only 7 bytes long so most of the `0x3b` packet is still intact - in fact it's consistently intact from offset `0xd` onwards. This corresponds to address `$ca4f` in memory.

onwards. This corresponds to address `$ca4f` in memory.

Fortunately for us, there are few enough instructions between where the `3F 00 00` player name string lands us and `$ca4f` that they don't lock up the GameBoy! Actually, they're are quite a few bytes on the way, but almost all of them are `0x00`, which is a `nop` ;)

The Vulnerability Details

What follows here is what was wonderfully explained to me by mid-kid on GitHub.

It turns out that `$3f` is the `<ENEMY>` byte, which is used to print the name of the opponent player during link battles (handy as you don't have to copy the opponent name to a string buffer before printing, you can just print `\x3f wants to battle`, etc). By setting the first byte of our player name to `$3f` itself, the game enters a recursive infinite loop, which in turn triggers a stack overflow from all the return addresses that get piled onto the stack. The stack is usually somewhere in WRAM: `$c000` - `$cfff`. As the stack grows down, eventually the stack pointer will point into SRAM (`$b000` - `$bfff`), which should be an issue as SRAM has to be opened before it can be read or written to.

Before long, the LCD interrupt will trigger, and execution will break from this infinite loop and jump to `LCD` at address `$0552`. Importantly, the return address will be pushed to the stack. When this happens, the write into SRAM (where the stack pointer points) will silently fail. At the end of this interrupt is a `reti` instruction at `$0567` and the address to return to would normally be popped off the stack. What's *supposed* to happen is that we would read `$ffff` as SRAM is still closed. Instead what we get is something a bit weirder.

When a CPU reads a value from memory (e.g. an instruction to execute, or a read/store instruction, etc) it sets the address lines on it's bus accordingly and then reads the data lines to see what it's got. At the point where the return address is meant to be popped off the stack, the very last thing read off the data lines by the GameBoy's CPU is the `reti` instruction, with opcode `$d9`. When SRAM is closed, the data lines can't be pulled when the address lines are set to an address in

SRAM. As mentioned, we're *supposed* to "read" `$ffff` in this scenario,

which is because the data lines in this case are *high* when not in use (i.e. set to `1` rather than `0`). This is called an "open bus". The issue here is that, much like how data can still be recovered from your PC's RAM for a very short time after powering off, the intrinsic electrical capacitance of the wires in the PCB connecting the data lines to the CPU make it possible for the last value that was read to be *re-read*! In practice this is what happens:

1. The DMG CPU reads the `reti` instruction from the address pointed to by the instruction pointer.
2. The value `$d9` is read over the data lines.
3. The DMG CPU attempts to read two bytes from the address pointed to by the stack pointer to return to.
4. SRAM is closed, so the data lines do not change (or update). They are "on their way" to returning to `$ffff` as nothing is being read.
5. The "ghost" of the value `$d9` is read by the DMG CPU *twice* to form the form the memory address `$d9d9` .
6. The GameBoy "returns" to address `$d9d9` from the interrupt.

Weird, right? This is referred to as "open bus behaviour". So, what's there to return to at `$d9d9` ? Fortunately, not a whole lot - in fact it's mostly all `00` s, so in effect is one big nop sled. In all the tests I did, I found that there was always an `$ff` byte at address `$da66` . As an opcode, `$ff` decodes to `rst $38` . Jumping over to `$0038` , we again see `rst $38` . This loop continues a few times and pushes the stack pointer a little further into SRAM until a timer interrupt triggers.

This is where things start to pick up again. The timer interrupt is at `$0050` and immediately jumps to `$3e20` here - note however that this function is *very* different in the Japanese version of the game. Since the game thinks we're using a mobile adapter, the appropriate mobile timer needs to be called at `$58de` in bank `$44` . In order to do that, the game has to bankswitch, which is done via interrupt `$10` and is pretty short:

```
ld (ff00+9d), a
```

```
ld (2000), a
ret
```

The details of those loads aren't important, but what is *vital* is the `ret` at `$0015`. Remember that we're still in the middle of a timer interrupt, so *timers are disabled until we hit a `reti`* (unless manually triggered with `rst`). When the `ret` instruction is executed, the exact same process that we had with the `reti` instuction. The opcode for `ret` is *re-read* over the data bus and interpreted as a memory address. This is because the stack pointer *still points into SRAM and SRAM is still closed*.

The opcode for `ret` is `$c9`, so we end up "returning" to address `$c9c9`. When an interrupt triggers, further interrupts are disabled until a `reti` instruction is reached. Because we never reached a `reti` after the timer interrupt triggered, this means that no further interrupts will get in our way, and execution will continue through the convenient nop sled of `$00` s until we reach the player name string at address `$ca4f`!

It's important to make clear just how lucky this setup is. If the aforementioned timer interrupt triggered at some point during the infinite loop caused by `$3F` in the enemy name, the stack pointer wouldn't be pointing to somewhere in SRAM and we wouldn't be able to benefit from the open bus behaviour in order to gain control of execution. There's also the wonderful coincidence that the opcode for `ret` is `$c9` and `$c9c9` is so close to the controllable buffer at `$ca4f`. It's also apparently very unlikely that this will work with a GameBoy flashcart or with other emulators that don't perfectly emulate this open bus behaviour. It's really a testament to how accurate BGB is that this works at all!

Huge thanks once again to mid-kid for pointing out that the open bus behaviour was the culprit behind `$3f` allowing us to hijack execution! You can read his explanation here.

Exploiting the Vulnerability

So how many bytes do we have to play with? We're injecting into the

So how many bytes do we have to play with? We're injecting into the `0x3b` -type packet which is only 56 bytes (plus an `0xc` counter byte on the end), but we have to start at offset `0xd` , leaving us 43 bytes to play with. That's not much.

Although I'd long since dropped the idea of this being a valid entry for BGGP, I still thought it would be cool to print a single `3` to the screen, and I figured 43 bytes should be enough to do that.

The next question is how do we write to the screen on the GameBoy Colour? Well, the PPU memory lives at `$9800` , but most of the time we can't read or write to it. Why's that? We can only address the PPU's memory-mapped region during vblank - this is the time in-between the PPU successively drawing the image in it's memory to screen. Fortunately, there's a memory-mapped register called `LY` at `$ff44` which contains the current line being drawn. The GameBoy Colour has 144 lines, so that's when vblank starts. Easy enough to wait for `$ff44` to be bigger than `144` before continuing.

Next we have to turn the LCD off. We do this using another memory-mapped register, this one called `LCDC` or "LCD Control" and it lives at `$ff40` . It's a bit-mapped register, but we only care about bit 7 which dictates whether the screen is on or off. The easiest thing to do is just zero out the whole register.

Now, finally we can write to PPU memory at `$9800` . What does `$9800` actually represent? It represents the top-left most `8x8` block of screen, which is the size of a sprite. Looking up the character map again, we see that the character `3` corresponds to byte `$f9` , so that's what we need to write to address `$9800` .

Now, there's a slight snag. Somewhere on the way to this code, the GameBoy's palette has been wiped. This means that if we turn the LCD back on, our `3` will be there, but it'll be white text on a white background. Not much fun.

In order to change the palette, we have to use two more memory-mapped registers. These are `BGPI` and `BGPD` and they work together (see here for more detail). Essentially, `BGPI` ("BackGround Palette Index"), which lives at `$ff68` is the index into the background palette data and `BGPD` ("BackGround Palette Data") at `$ff69` is the actual data at the index we requested.

By examining the *very helpful* BGB palette data screen, I knew that the index that was active when I re-enabled the LCD was `0x38`, which meant I needed to write `$38` to `$ff68`. However, the palette data is actually a 16-bit value, and helpfully, if we set the MSB in our write to `$ff68`, it'll automatically increment after we write the value to `$ff69`. Nice. So, first we write `$b8` to `$ff68`, then we just do 2 successive writes to `$ff69` with our palette data. Great. What's our palette data? It's simply broken up into R/G/B intensity, so to set the background to black, we just write `0` twice.

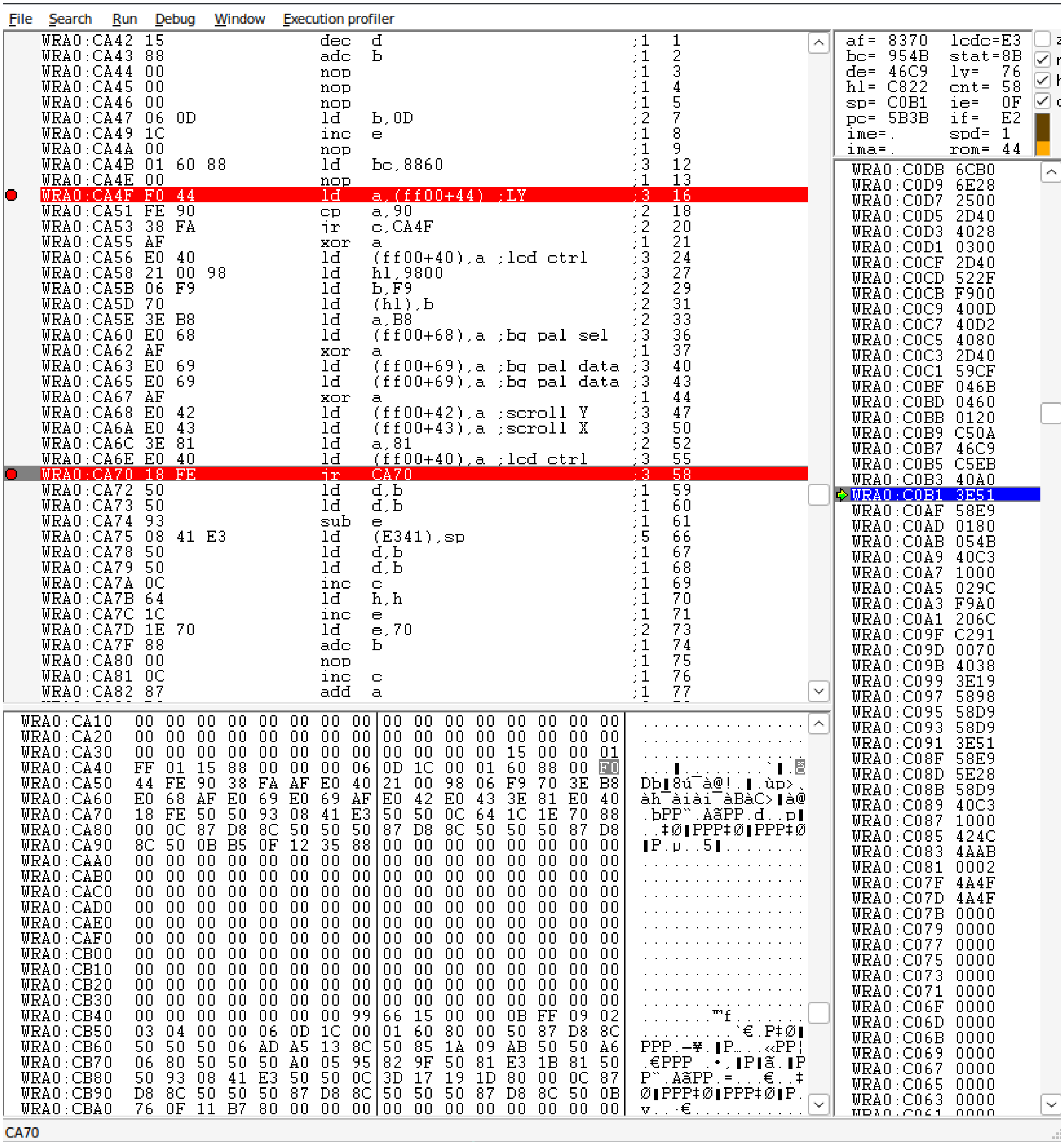
Lastly, we need to set which part of the PPU's memory is the part we want on the screen. This might sound weird. The PPU's memory is actually quite a bit larger than the screen size, which allows for things to be loaded into video memory before it's due to be on the screen (think about the scrolling world of Super Mario Land). Because we wrote to `$9800`, which is the start of video memory, we set the `LY` and `LX` registers (memory-mapped to `$ff42` and `$ff43` respectively) to `0`. The very last thing we need to do is turn the LCD back on by setting the MSB and LSB of `LCDC` again. As mentioned earlier the MSB enables/disables the LCD, but the LSB controls the Background Display Priority, which we want on.

What does all this look like? It looks like the below. Note that although the disassembly shows absolute addresses, the opcodes are actually all relative jumps. Saying that, this code is always executed from `$ca4f`, so the disassembly below is all relative to that, despite being a "position independent exploit".

```
ld a, (FF00+44)        ; Load LY register into A register
cp a, $90              ; Are we past vblank?
jr c, $CA4F            ; Loop until we are

xor a                  ; Clear A
ld (FF00+40), a        ; Reset LCDC register

ld hl, $9800           ; Load $9800 into HL register
ld b, $F9              ; Load $F9 into B register
```

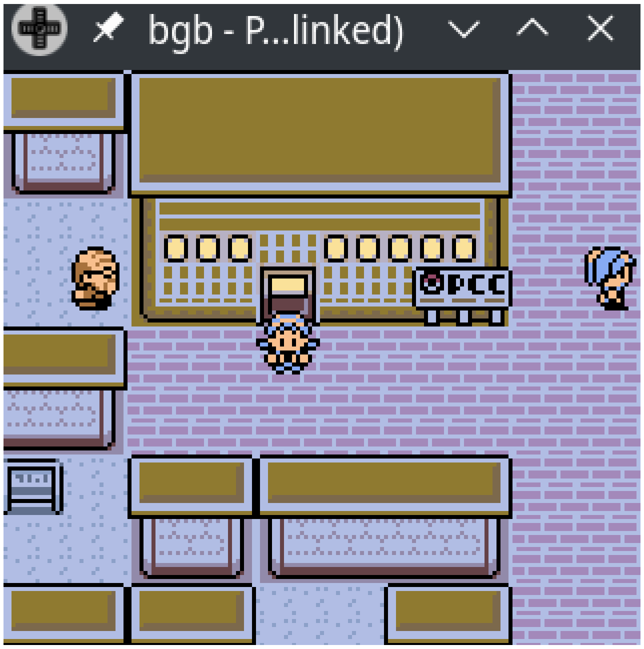
```
ld b, $F9               ; Load $F9 into B register
ld (hl), b              ; Load $F9 into the address pointed to by HL

($9800)

ld a, $B8               ; Index $38 into BG Palette Data, with Auto-
Increment On
ld (FF00+68), a         ; Load $B8 into BGPI
xor a                   ; Clear A
ld (FF00+69), a         ; Write 0 into BGPD
ld (FF00+69), a         ; Write 0 into BGPD

xor a                   ; Clear A
ld (FF00+42), a         ; Load 0 into LY
ld (FF00+43), a         ; Load 0 into LX
ld a, %10000001         ; Set MSB and LSB Only
ld (FF00+40), a         ; Write $81 into LCDC - the LCD is now on!
jr $CA70                ; Infinite Loop
```

How did I assemble this? I used the opcode table and wrote it out manually because I couldn’t find a simple assembler that didn’t give me an entire ROM, and it only turned out to be 35 bytes in the end, well short of the 43 bytes we had available.



Above, you can see the payload between the two breakpoints at `$ca4f` and `$ca70` .

I guess it's time to see it in action! Well, here we are:



The Finale

So there we go - remote code execution on a GameBoy Colour 22 years after the product that makes it possible was released! Massive thanks to @netspooky for mentioning the existence of the Mobile Adapter GB to me and setting me off on this journey.

Thanks as well to the Binary Golf Association. Initially I was planning to make this an entry to this year's competition, but unfortunately the exploit is not a binary (and not that small either). Be sure to check out all the amazing write ups people have put together for their entries this year!

Hopefully you've read this article and feel like hacking around with GameBoy ROMs isn't as scary as it might sound. :)

Shoutouts to dnz, yuu, hermit, gren, bane, remy, kyo, rqu, gilda, harmony and all the ghosts. Support your local Binary Golf Association!

Until next time…

READ OTHER POSTS

Janus: A Polyglot Binary for BGGP 2021 →