Virtual World RE
Gotcha Force

Files patching process v1.0.6

Table of Contents

1. Summary:

Welcome! This guide is intended to explain how to help reverse engineering the GameCube game **Gotcha Force**. The main goal is to better understand some of the strategies to do reverse engineering on the Gocha Force files. Examples are done using the **USA GCM/iso** version. Here is presented the global patching process and tools used to achieve this. There is screens to show how to use Hexadecimal editor with bookmarks to investigate files easily with community sharing and a demo to automate the whole patching process using a python script that can be adapted.

Some techniques described in this guide could require some strong basis in informatics like python programming and file formats.

2. What tools do I need?

Hexadecimal editor

First of all you have to find your hexadecimal editor. Values are stored in game files or memory in raw “bytes” or if you prefer “8 bits” by “8 bits”.

The important thing to know about it is that when we store data we use an endianness. This mean that this same “bytes” can be stored in different order:

- Big-endian

- Little-endian

- (… others are possible)

So bits keep the same order but we store bytes (in files / memories) in one direction or in another. **All the individual bits are not reversed** though. Here is an example. Each 8 bits of the first byte has an index from 0 to 7: "0123 4567". The second byte is "XXXX XXXX". The value is "0123 4567 XXXX XXXX": 2 bytes long.

1. If we store it in **big endian** we will have:
   "0123 4567 XXXX XXXX"
2. And now in **little endian** it will be:
   "XXXX XXXX 0123 4567"

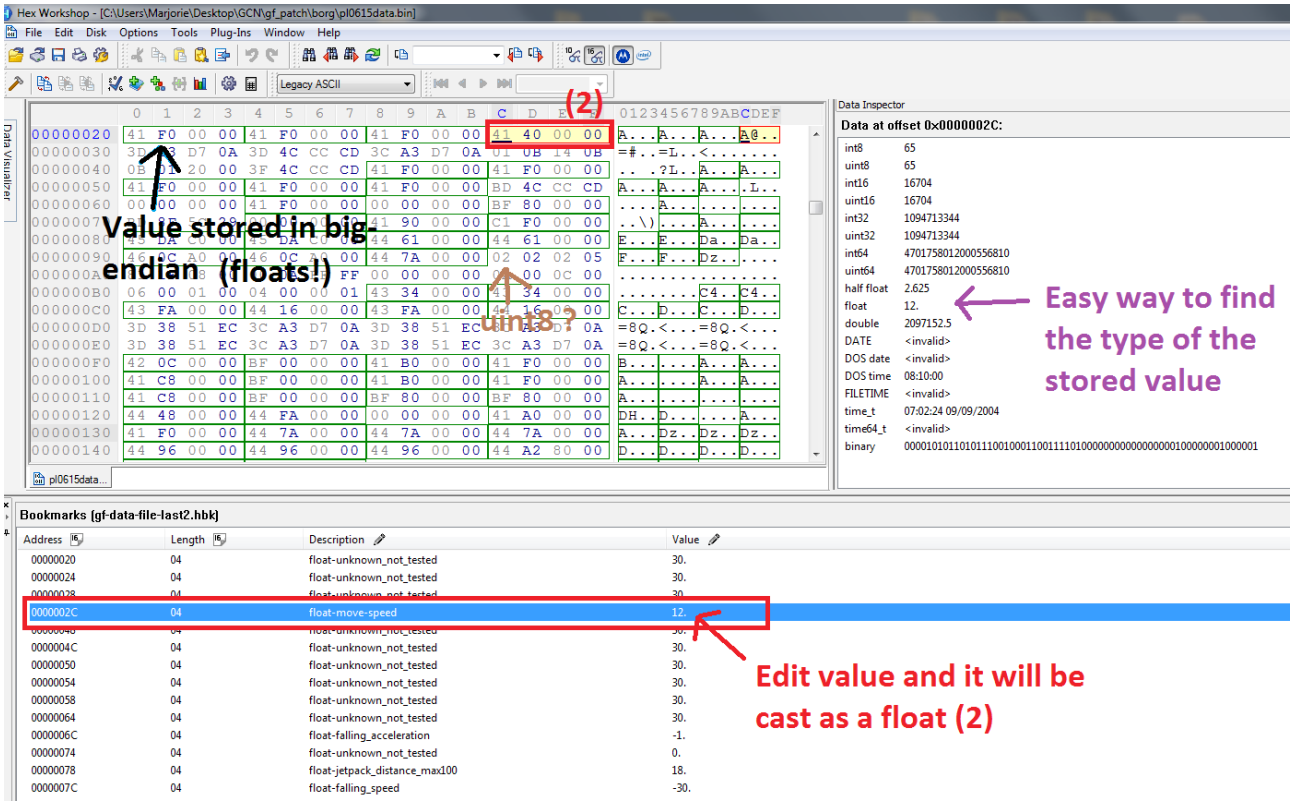
To simplify reading we use hexadecimal representation rather than binary. It's easy to read and convert one byte by splitting the 8 bits in two 4 bits block:

Any hex editor is good if you can:

- patch any byte of the file with a new value or remove / add data
- compare 2 files between them
- See values in different encoding (int8, int16, int32, uint8, float...)
- Use bookmarks or a system to create a memory mapping of the values present in the file.

I use hex workshop and it's bookmarks to easily determine every type of variable and automatically encode in the right format when I change a value. Find a good one!

Here is an example of the work on the standard "plxxxxdata.bin" files of borgs:



Numbers encoding is a way to store values in files or memory. It's hardware dependent and so embedded sys like GameCube could use specifics encodings. Here is a script I made that try to approach numbers:

https://github.com/tmpz23/scripts/blob/main/numbers.py

Python 3

Get python 3 to script patching process, use Virtual World RE scripts, and gain lot of time to do easily tests on large amounts of files.

GameCube Emulator

A good GameCube Emulator will help you when dynamically reversing the game:

- Patch the game
- Run and see what's changed
- Patch again
- Run again

- Use dynamic debugging of the game! (change registers or memory values, put breakpoints...)

Dolphin Emulator is what I use.

It is interesting to note that community has developed python bindings for IA/CPU Games creators. The API allow to interact with memory.
https://github.com/Felk/dolphin/releases/tag/scripting-preview1 (release)
https://github.com/Felk/dolphin (project)
https://github.com/Felk/dolphin/blob/master/python-stubs/dolphin/memory.pyi (api doc)

GameCube DVD patching tool

You need a tool to export and import back files from and to the Gotcha Force **GCM/iso** file. Virtual Word RE provide a full command line tool, gcmtool.py:

https://github.com/Virtual-World-RE/NeoGF/blob/main/gcmtool

This tool can patch every files from the initial GCM iso like sys files (apploader.img boot.dol) and files inside the DVD changing their length if needed.

AFS patching tool

AFS is a special GameCube file format often used in games that pack the game files. You need a special tool to unpack or pack files to this AFS.

Virtual World RE provide afstool.py:

https://github.com/Virtual-World-RE/NeoGF/blob/main/afstool

With the AFS format the main problem is "how the game (boot.dol) retrieve files inside the afs?" afstool.py allow you to use different strategies. You have to configure it to a rebuild by "index" in the Table Of Content (TOC). This will allow you to change length of every file you want to patch.

PZZ patching tool

PZZ is a file format storing multiple files with a compression algorithm. Virtual World RE provide pzztool.py:

https://github.com/Virtual-World-RE/NeoGF/blob/main/pzztool

There is multiple command to optimize patching process by handling compression algorithm during unpack or not. If you target a specific file then it's better to use unpack and the decompress /compress only the targeted file.

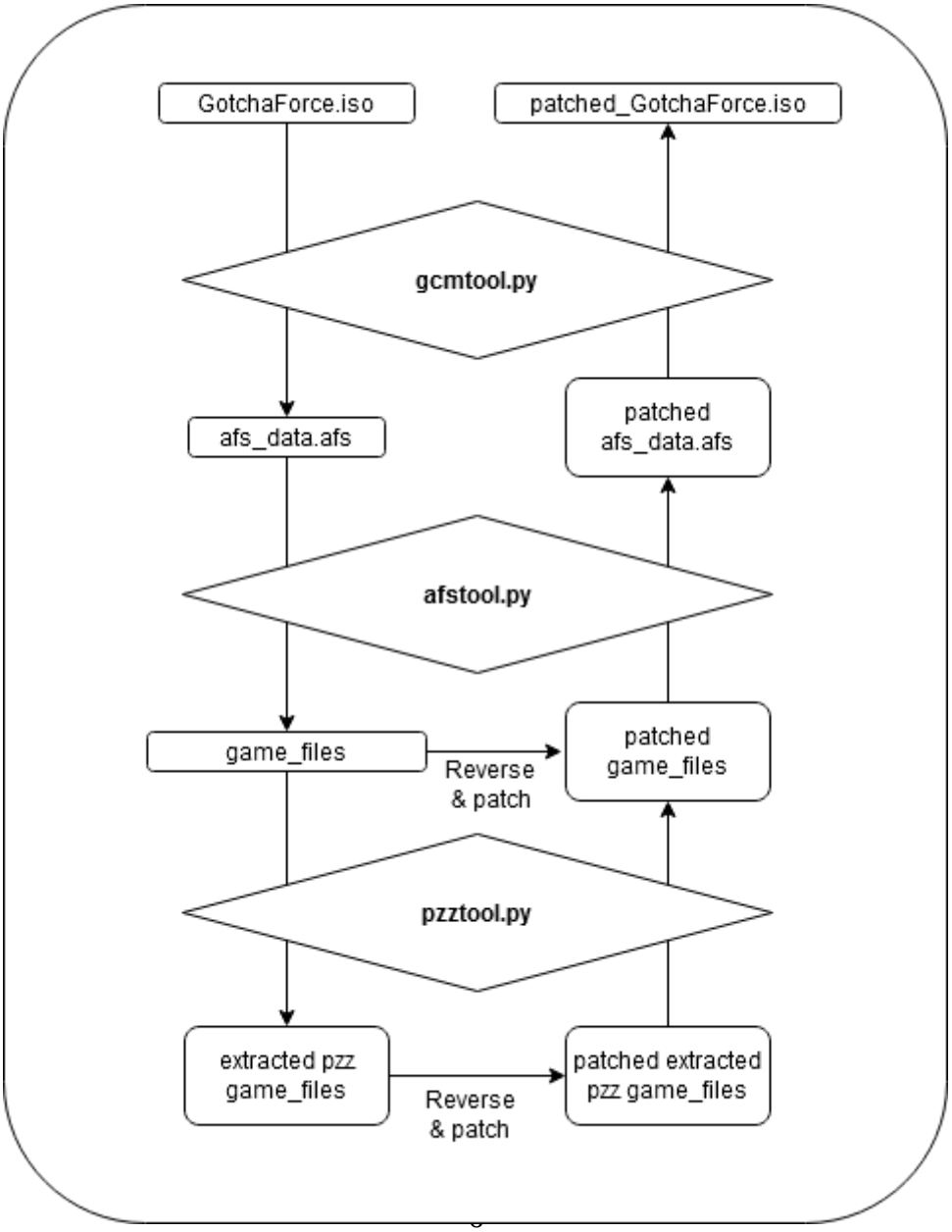
Dolphin Memory Engine (optional)

https://github.com/aldelaro5/Dolphin-memory-engine

This tool can help you to manage the memory during dolphin emulator research sessions.

3. How to patch the Game?

Here is a diagram explaining the different steps to patch the **GCM/iso** game:



In a first time it the whole process will be explained in details. Then we will see an automated script example that do all this command without having to waste time to do it manually.

First of all you have to be organized and use correct naming of files during the patching process. It will help you to find which file are patched and so on.

1. Make a folder named "gf_patch"
2. Put your Gotcha Force iso file "gotcha_force.iso" in it
3. Enter inside the gf_patch folder (with your prompt)
4. Create "tool" folder and download in it the Virtual World RE tools: **gcmtool.py**, **afstool.py**, **pzztool.py**. (Next use tools/gcmtool.py for instance in each command)
5. You will now unpack files from the gf iso into a folder:
   - gcmtool.py -u gotcha_force.iso iso_unpack # unpack of the iso
6. Then we extract afs_data.afs:
   - afstool.py -u iso_unpack/root/afs_data.afs afsdata_unpack # unpack of the afs
7. Now we will configure the afs to enable changing length of files and remove empty padding to optimize it. Open the configuration file "**afsdata_unpack/sys/afs_rebuild.conf**" and change the following values:
   - In [Default] change **files_rebuild_strategy = index**
   - In [FilenameDirectory] change **toc_offset_of_fd_offset = auto**
   - In [FilenameDirectory] change **fd_offset = auto**
8. Now we rebuild the afs system files into the unpacked afs folder:
   - afstool.py -r afsdata_unpack # rebuild of TOC & FD
9. Now we pack the rebuilded afs into the unpacked GCM/iso folder:
   - afstool.py -p afsdata_unpack iso_unpack/root/afs_data.afs # pack of the afs
10. Now we rebuild the GCM/iso FST to optimize space and accelerate repack but also to change length of files inside it:
    - gcmtool.py -r iso_unpack # rebuild of the FST
11. And finally we pack the files in a new GCM/iso:
    - gcmtool.py -p iso_unpack gotcha_force_patched.iso # new patched iso

What to remember?

- Every time you patch one file in the unpacked AFS folder and change it's length with a too big value overflowing on the next file you have to rebuild the afs using the -r command on the folder.
- Every time you change the length of a file in the GCM/iso folder you have to rebuild the gcm using the -r command.

Using this process you can change every files as you want.

4. How to automate the patching process?

Like said before using a python script to automate the patching process is really useful since you don't have to repeat commands. Also to accelerate the dolphin emulator runs you can change options on the dolphin emulator configurations to run with max CPU speed, and so on! It could be really quick to patch a file with automated casts of values (like shown in Hex WorkShop in the "**What tools do I need**" section) and run to see the impact on the game.

Here is a script that I've made to patch the borg data file (plxxxxdata.bin) and see in-game impact that run automatically dolphin emulator:
https://github.com/tmpz23/scripts/blob/main/gf_research_tool.py

You can edit this to patch any other file you want. Or ask algoflash on Discord to do this.

What's interesting is that it have multiple ways to patch:

- **Patch adding X** to every selected value: this is useful because the game will have very lows changes and some values could make a black screen or eveen make no change since there overflow their max allowed value so they are ignored
- **Patch setting value to X**: this is useful to test at the limits! For instance when you want to set "-1" or "0" or any other values.

To optimize the process there are commands that use dichotomic search algorithms to patch a range of value.

The idea of this algorithm is to patch half of the bytes of both files:

- pl0615data.bin
- pl0615/000C_pl0615data.bin

1. Then it run dolphin. You will see changes ingame or not. Target one change that you don't know yet like a camera at the end of the battle in a new place, or an other characteristic.

2. Quit dolphin that's the initialization.

3. Dolphin will start again and you have to check if the change is here again. Quit dolphin.

4. The prompt will ask you if the change is still here. Answer yes or not.

5. The game will run again and again. Repeat the change identification (Is it here again or not?) And quit dolphin and respond to the command prompt.

6. After all questions the prompt will display to you the offset (byte) that have been changed and that is responsible of the change.

7. You know have to search with your hex editor and target this offset trying to find it's type (float? byte? signed? not signed?) and what it does.

8. Share the offset with community! =D

What's going on inside the script? It will patch the whole file without patching already found bytes to allow you identify a change.

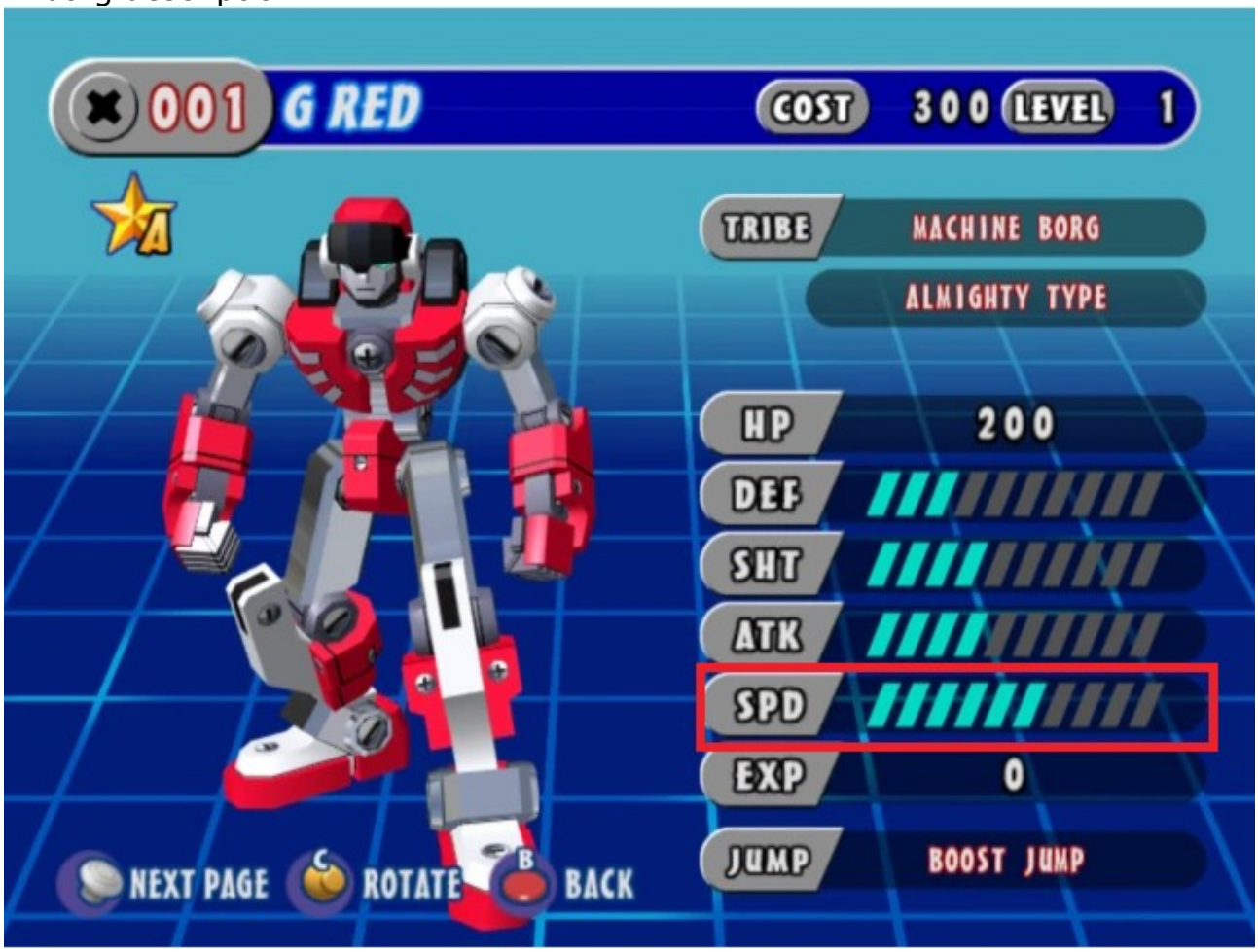
Then it will patch half of the file. And if there is a change ingame this half will be selected else other half should contain the bytes responsible of the change so the other half will be selected.

Then patch half of the half that have been selected and see if the change is here or not. Selected again the interval containing the change.

Repeat until you find the real offset and values that are stored in it.

5. Demo: change G Red speed in collections menu

In this demo we will patch the speed of G Red displayed in the menu collection → borg description:



This csv contains all borgs that are in the USA game. First column describe the filename and second column describe the borg name for instance "pl0615" is the filenames base for the first story borg used in the game "G Red".

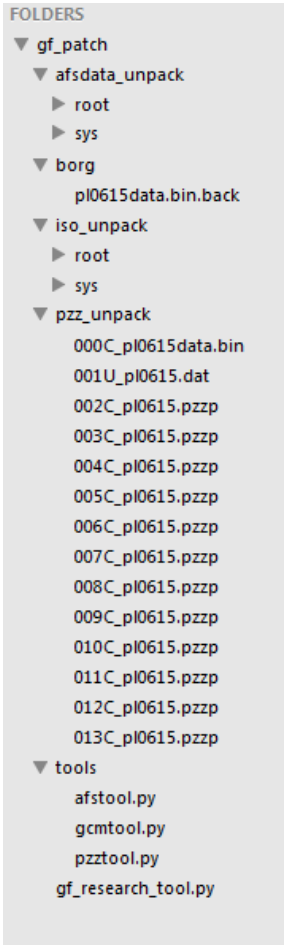
We will now add a cheat code in Gocha Force to have full borg "collection" available. Right click on the game then go to properties and AR Code and add the value "003BFF78 0000CD01" in a new AR Code.

If not already done download this tool:
https://github.com/tmpz23/scripts/blob/main/gf_research_tool.py

Inside this script edit this two variables with the path of your Gotcha Force ROM and with the full path of dolphin emulator exe:

```
# Original Gotcha Force GCM iso PATH
GF_ISO_PATH = Path("ROM/Gotcha Force (USA).iso")
# https://fr.dolphin-emu.org/download/
dolphin_path = Path("C:/Program Files/Dolphin/Dolphin.exe")
```

Then install using this command:
`gf_reasearch_tool.py -i` # it will create "**gf_patch**" folder and download Virtual World RE python tools.



At the left this is what your folder should looks like now.

Keep in mind that you must never add or remove files in the folder "gf_patch/afsdata_unpack" or in "gf_patch/iso_unpack".

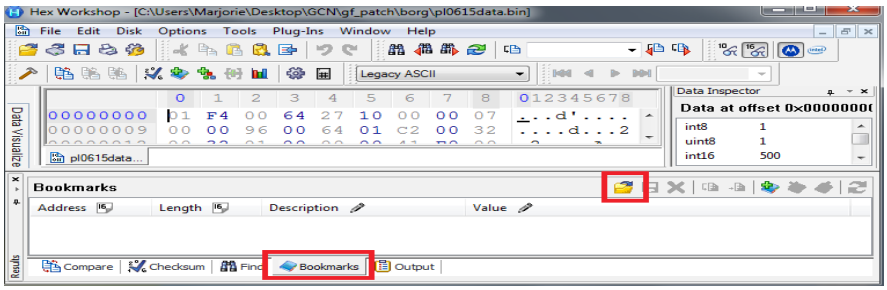
The working directory is the "**gf_patch/borg**".

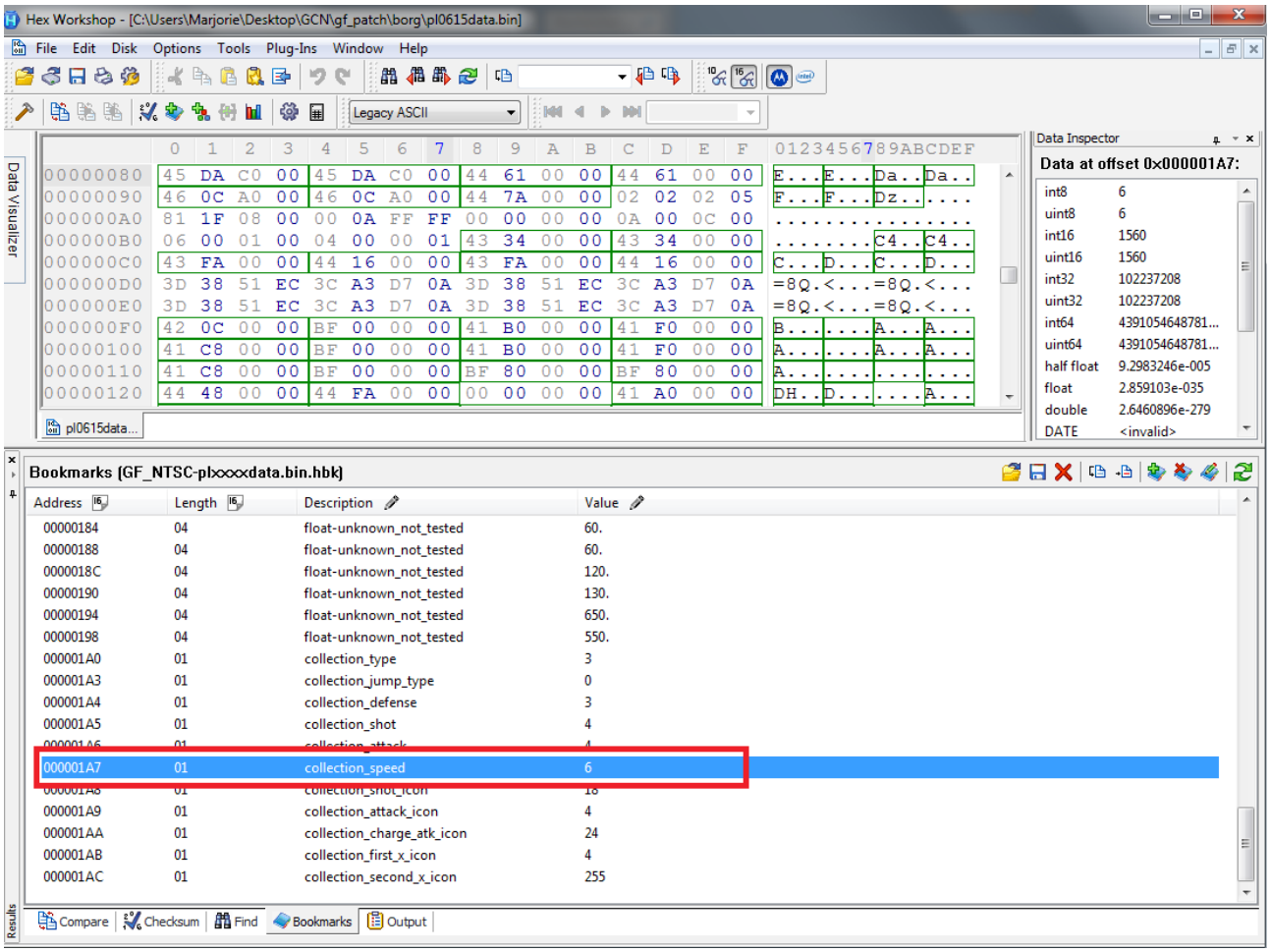
Copy pl0615data.bin.back and rename it with "**pl0615data.bin**". You will now edit this file to patch the game.

It will automatically patch the afsdata_unpack with:
(1) pl0615data.bin
(2) pl0615.pzz
Both have to contains the sames data to avoid crashes.

Now open **gf_patch/borg/pl0615data.bin** with Hex Workshop and then go to the "bookmarks" tab and open this file (put it in your working borg dir):

https://github.com/Virtual-World-RE/NeoGF/blob/main/data/GF_NTSC-plxxxxdata.bin.hbk





Change the value of collection_speed with "10" and save. Then enter the command:
gf_reasearch_tool.py -pr

This command will patch the borg pzz after patching the data file inside at position 000. Then it will patch the unpacked afsdata with both (data.bin and .pzz) and pack the afs into the unpacked gcm and pack the gcm into gf_patched.iso and run it into dolphin emulator.

Then ingame go to the menu "collection" and view the borg "G Red". You should now see "10" in the speed value.

With the -pr argument you can test all values in this file. But it will work if you edit the script to achieve this with another file. The main advantage is that it is really fast and allow to see changes in multiple prompt and dolphin emulator windows.

6. How to help?

You now get the patching process and you can automate it using a python script. The main goal is to find what values in the borg data files are used for.

Here is a trick:
For a known offset you can list ALL the values that are taken in all plxxxxdata.bin files with a simple python script:
https://github.com/tmpz23/scripts/blob/main/binreverse_list_gf_values.py
Adapt it with the files you want to list.

Conclusion

You are now a confirmed GameCube files reverse engineer! Come on discord and share the new offset you have found and their impact on the game! Pl0615 (G Red) is the first borg of the story so I like to use it for tests. But you can also adapt script to use another borg.

Also you can switch borgs by renaming all their files (plxxxx.pzz and plxxxx*) and erasing the targeted borg you want to change. Here is a screen of changing files to test ingame the different G Red _mdl:

Hope this tutorial gave you new ideas or knowledge. See you soon!