How to think in binary
a guide for cats

By NekoMimi

NekoMimi@NekoLabs (nekomimi@tilde.team)

Appendix

Herro everynyan :3

Chudey yuu wirru acquayaah e greit pawah!

Za pawah chuu concuer aru za deta yuu wishu to comando

Neko has created this wonderful guide for you to release your inner hacker and be able to decode any random sequence of data and deobfuscate it

This guide came out as a means to fight the proprietary data formats which can’t be documented, like for example game data that one wishes to translate, or a save file that one wishes to create a trainer tool for

We will go through all the things necessary to go from “what on earth is this?” to “ah yeah this goes here, that goes there”

If you found any of this useful please make sure to show Neko some support.

A Hacker’s Environment

You want to get into hacking this binary/hex data but you don’t know what tools your’e gonna need

For this guide we will assume you have access to the following tools:

a hexdump tool, a string dump tool.

We encourage you to use a Linux environment as working with windows terminal is hell

for Linux the tools you need are already pre-installed (`hexdump` and `strings`)

I recommend also getting the “Reverse Engineer’s Hex Editor” as it’ll be better than just looking at the raw output of hexdump

Make sure your hex output looks like this:

```
00 01 02 03  04 05 06 07         08 09 0A 0B  0C 0D 0E 0F
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00 00 00 00  00 00 00 00         00 00 00 00  00 00 00 00 |…….. ……..
```

as in you got 16 entries in each line, and next to them you got the strings output, this will make reading the data much easier

And one more important thing is knowning the byte order, for example 32 byte integers take 4 of these spaces (for example 00, 01, 02 and 03 for a single int) but that doesn’t mean that they are read in that order necessarily, it depends on the architecture’s endian type, big endian is read in order however little endian is read backwards (03 → 02 → 01 → 00) so when you’re figuring out pointers keep this in mind.

The Data

You look at a random string of binary data, be it a save file, some game files, anything to that matter, you have a goal in mind but you do not know the steps to get to the finish line.

To decipher this data you'll need to notice a couple of things:

- Header
- Strings
- Pointers
- Hashes
- Patterns

We will go with what it takes to notice these things and what other things you should pay attention to

Header

The header is usually the first couple of bytes, in most cases it's the first 32 or 16 bytes

It will likely contain some of these objects:

[The file header]

    3 or 4 (or even more) characters to identify the file type

[Counters]

    be it file counts, data counts, anything relevant

[Flags]

bytes that define the attributes of the file, examples like if it contains media files or scripts or if the files are compressed or not

There are no easy ways to notice these objects other than the file header but knowing what type of file you're working with should give you a hint, for example a proprietary archive file like the Sega XBB format contains a file count flag right after the file header and a couple of unknown flags

The best way to learn where these flags are, is by cross-examining multiple of the same file types you are working with, if it's a single save file then keep playing the game and save different copies as you progress, this will help in the other steps too, the best weapon you got is all the knowledge you can aquire, the more samples you get, the more easier it is to label each byte's job

Strings

Now comes the fun part, searching for strings, anything that makes sense in human language.
Before going on with using the strings command you can try and skim through the first couple of lines after the header since most useful strings would be there.

The reason for this is that we currently don't really care about the strings that represent the data, as we want to learn the structure of the file first, anything like filenames, extensions, data titles are useful here

for example in Sega's XBB file format after a couple of lines you can start to see some filenames along their extensions, plus they are separated with a nullptr (0x0), in this example we learn two

important things, the filenames needed to extract the files and the separator being the nullptr.

Your case will be different but try to search for any strings relevant to the file type you are working with (filenames for archives, player names for save files, etc)

And now we get a full scan of strings using the strings command which will show us all the strings and their locations, note that some strings might look like gibberish so discard those

Pointers

Pointers are a bit tricky, you'll have to know the addresses of the data you're searching for prehanded (or you can write a program that will check for potential pointers by comparing each iteration of 4 consecutive bytes after the header to the file size, you can also add a check to see if they point to locations that each start with the same or similar patterns)
sometimes they might point to a new file that starts with a file header so this might come in handy for checking whether it's a valid pointer or not.

Pointers can also point to the end of the data (or the size of the data) (which you can construct an end pointer by adding the size to the start pointer) most end pointers will lead to an address that has a bunch of nullptrs after it (usually a padding since there would be no need for an end pointer otherwise, you can use the next start pointer for that, but for old games that were read from a disc it is very common to segment data to chunks of equal data lengths for easier reading and better error handling thus they are padded and contain a size/end pointer)

but pointers are very likely to be part of the data you are examining so keep looking for them.

Hashes

There are two main type of integrity check hashes that are commonly used:

crc and adler

each of these two also has different versions based off their polynomial, but most used are 32 and 4.

However 32 is the most used in the industry, if that fails then try 4 next otherwise it’s most likely a different integrity hash

we will not go deep into these algorithms but you should know that each of these algorithms has a different polynomial, and if you don’t have it you’ll need to figure out somehow, most likely by bruteforcing, which a simple multi-threaded python program can complete both crc32 and adler32 polynomials simultaniously in about a day, take a look at the following example:

```
import zlib
import time
import signal
import multiprocessing

from pyfiglet import os
from NekoMimi import utils as nm

threads= []

def calculate_checksums(file_path, crc32, adler32):
   # Read the file in binary mode
   with open(file_path, 'rb') as f:
       while True:
           # Read a chunk of the file
           data = f.read(4096)  # Read in 4KB chunks
           if not data:
               break

           # Update CRC32 checksum
           crc32 = zlib.crc32(data, crc32)

           # Update Adler-32 checksum
           adler32 = zlib.adler32(data, adler32) #ignore this, this is useless

   return crc32, adler32
```

```
def worker(i, c, steps):
   r= c
   while True:
       crc32_checksum, adler32_checksum = calculate_checksums(file_path, r, r)
       if crc32_checksum == YOUR_KNOWN_HASH_HERE:
           print("crc32 found!  "+str(r))
           nm.write(str(r), f"thread{str(i)}-poly-c32.txt")
           break

       if adler32_checksum == YOUR_KNOWN_HASH_HERE: #ignore this, this is useless
           print("adler32 found!   "+str(r))
           nm.write(str(r), f"thread{str(i)}-poly-a32.txt")
           break

       r= r+1

       if r%10000 == 0:
           print(
       f"[W{str(i)}] P:{str(r*100/steps)[4]}% C:{hex(r)} " \
       +f"HC:{hex(crc32_checksum)} HA:{hex(adler32_checksum)}"
       )
           nm.write(str(r), f"thread{str(i)}-save.txt")

       if r == steps:
           break
```

```
def stopper(sig, frame):
    for w in threads:
        w.kill()
    exit()
def constr(i, s):
    if os.path.exists(f"thread{str(i)}-save.txt"):
        c= int(nm.read("thread{str(i)}-save.txt"))
        return multiprocessing.Process(target=worker, args=[i, c, s])
    else:
        return multiprocessing.Process(target=worker, args=[i, s*i, s*(i+1)])
if __name__ == "__main__":
    file_path = "YOURFILEHERE"
    signal.signal(signal.SIGINT, stopper)
    threads.append(constr(0, 0x20000000))
    threads.append(constr(1, 0x20000000))
    threads.append(constr(2, 0x20000000))
    threads.append(constr(3, 0x20000000))
    threads.append(constr(4, 0x20000000))
    threads.append(constr(5, 0x20000000))
    threads.append(constr(6, 0x20000000))
    threads.append(constr(7, 0x20000000))

    for w in threads:
        w.start()
    while True:
        time.sleep(1)
        pass
```

this example is a proof of concept script that will go through all possibilities for the polynomial, just give it some time to do it's thing, you use this app on a chunk of data that you know the hash for and it will keep comparing it till it finds the polynomial, this is essential for when you want to repack a modified file since you'll need to recalculate the hash for the game to read it.

Hashes look like 4 consecutive bytes of very random data eg: 0x7A824DC5 for a crc32 or an adler32.

Try searching for these assortment of bytes in a pattern at the beginning of the file (with the pointers) or directly in the end of each blob a pointer points to (before the start of a new blob).

You should search the architecture of your game since this will help tell you the maximum bit size the CPU can handle (like if a game is 16bits it will only be able to use up to crc16 for example)

Patterns

Data will often be written in a certain pattern, eg: (start ptr → size → filename ptr → hash) and continues with a second file/save/data type in the same way.

Comparing between the different samples you got will also show a pattern that helps aknowledge the use for each byte.

There is no way I can tell you what patterns to seek but any should be sufficient to give you more data to work with, plus patterns are the only thing besides pointers that help hackers reverse engineer a blob of raw data, if it's hard to do it manually then writing a script should make it easier

Where are we now?

Now that we've covered all the steps it's up to you to know where to go and to look for.

We hope that we managed to cram all this information into a short but informative guide that allows anyone to understand binary data files with no documentation, however we didn't include a lot of resources and we expect you to know a teeny tiny bit of general information otherwise you'll need to learn much more than just this small guide.

We wish you luck!
~Neko

[Resources]

https://en.wikipedia.org/wiki/Cyclic_redundancy_check

https://en.wikipedia.org/wiki/Adler-32

https://en.wikipedia.org/wiki/Endianness

https://github.com/WerWolv/ImHex