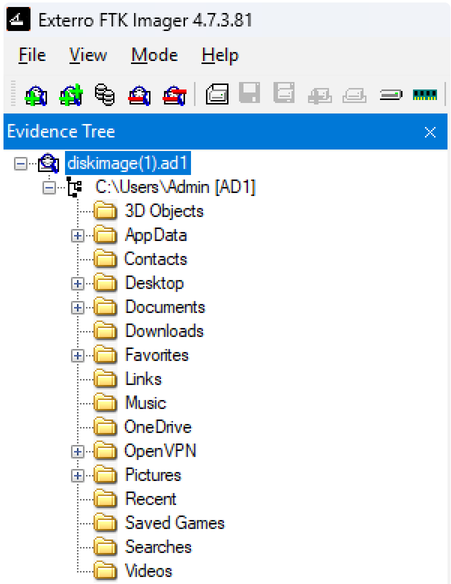
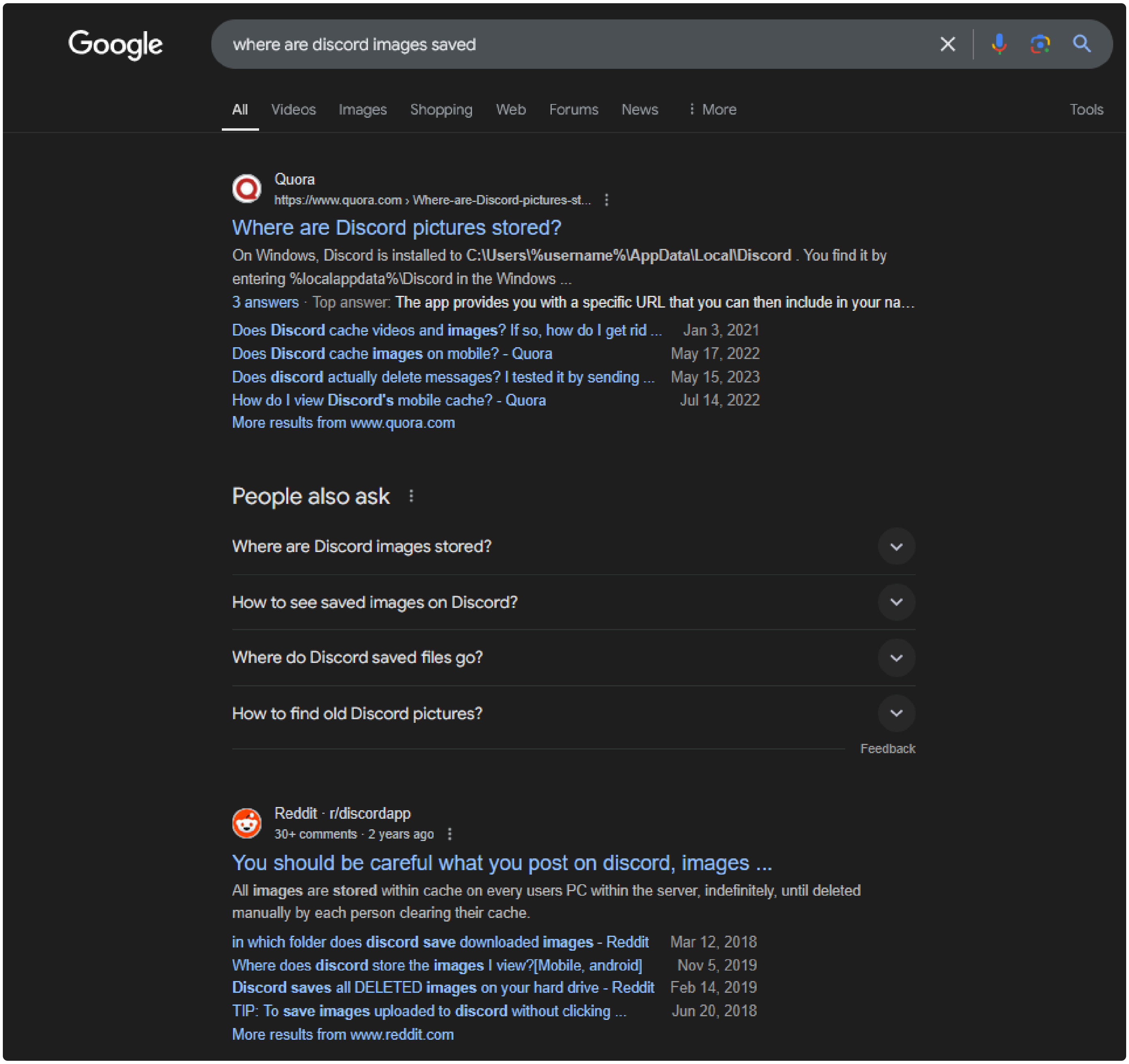
We're given a disk image (an `.ad1`, a logical image file) and the following facts:

- that our flag was sent as an image over Discord;
- that the user ran a downloaded executable, presumably a game.

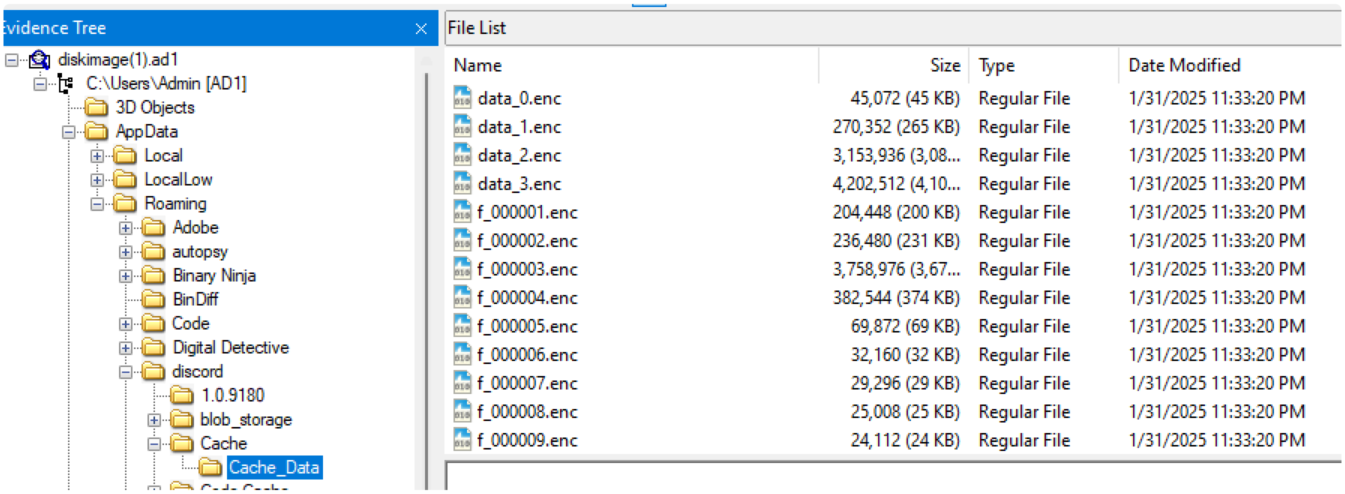
We can import the disk image into FTK Imager and observe the following file structure:



A Google search for "where are Discord images saved" should allow you to discover that Discord caches downloaded images, even if their containing message is deleted after the fact:



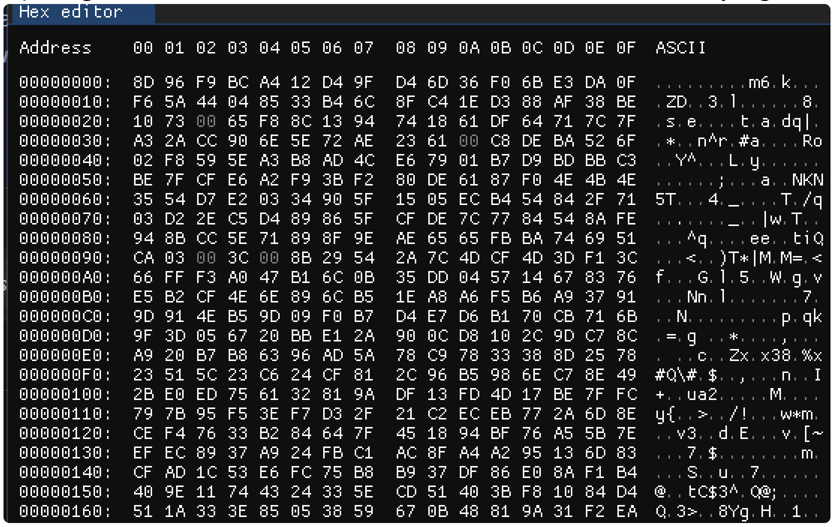
The location of this cache is typically `%appdata%/Discord/Cache/Cache_Data` on Windows machines. Indeed, we have a Windows disk image, and can navigate to the corresponding folder, `C:\Users\Admin\AppData\Roaming\discord\Cache\Cache_data` as shown:



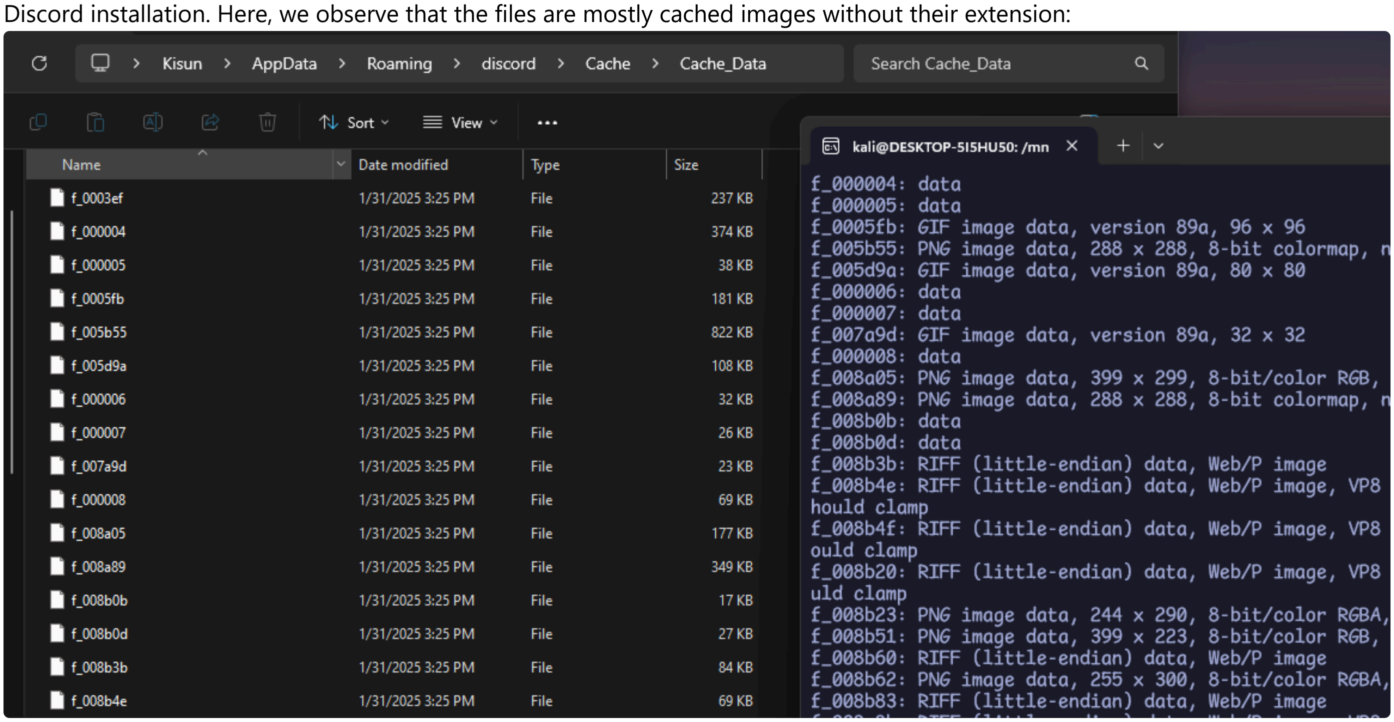
We can go ahead and export this entire folder to our local machine for further analysis. Running `file *` on the entire folder suggests that the data is mostly garbage:

```
❯ file *
data_0.enc:   data
data_1.enc:   data
data_2.enc:   data
data_3.enc:   data
f_00000a.enc: data
f_00000b.enc: data
f_00000c.enc: data
f_00000d.enc: data
f_00000e.enc: data
f_00000f.enc: data
f_000001.enc: data
f_00001a.enc: data
f_00001b.enc: PGP Secret Sub-key -
f_00001c.enc: data
f_00001d.enc: data
f_00001e.enc: data
f_00001f.enc: data
f_000002.enc: data
f_00002a.enc: data
f_00002b.enc: data
f_00002c.enc: data
f_00002d.enc: data
f_00002e.enc: data
f_00002f.enc: data
f_000003.enc: OpenPGP Public Key
f_00003a.enc: data
f_00003b.enc: data
```

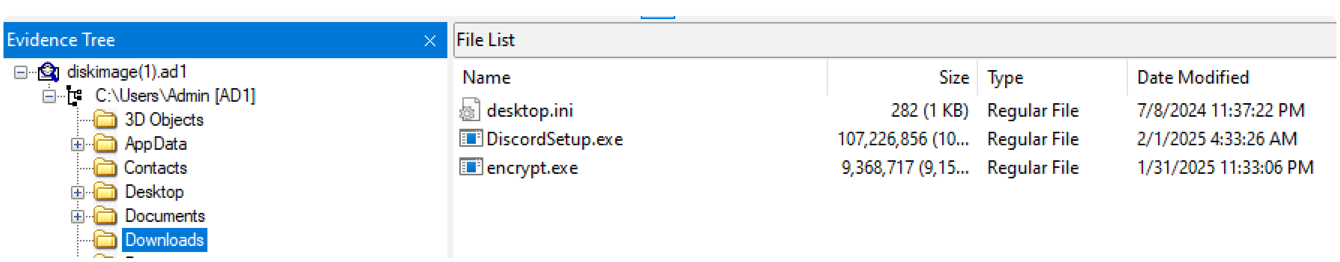
Opening these files in a hex editor reveals that these files exhibit very high entropy, a sign of compression or encryption:



We can confirm that these files are not normally compressed (or encrypted) by navigating to the `Cache_Data` folder in a "real" Discord installation. Here, we observe that the files are mostly cached images without their extension:



We can reasonably conclude that each of the files ending in `.enc` are encrypted, and that one of these files likely contains our flag after decryption. The challenge mentioned that an executable had been downloaded and run prior to the disk image being taken, so let's look at the Downloads folder:



Indeed, there is a not-so-suspicious file called `encrypt.exe`. We can export it and analyze it in Binja, though the icon for the file might give us a hint as to what it is:

If we jump to `main`, we see that it calls `sub_140002b80` ...

```
140001000    int64_t main(int32_t arg1, int64_t arg2)

140001000    {
140001000        **(uint32_t**)&data_140040000 = arg1;
140001010        *(uint64_t*)(data_140040000 + 8) = arg2;
14000101b        /* tailcall */
14000101b        return sub_140002b80(data_140040000);
140001000    }
```

Which reveals a function characteristic of PyInstaller setup:

```
140002b80    uint64_t sub_140002b80(int32_t* arg1)

140002b80    {
140002b80        __chkstk(0x2050);
140002b9a        void var_2068;
140002b9a        int64_t rax_1 = __security_cookie ^ &var_2068;
140002bb7        int32_t r8;
140002bb7        int64_t r9;
140002bb7        setbuf(sub_140016260(2), 0, r8, r9);
140002bc7        uint64_t result;
140002bc7
140002bc7        if (sub_140002a70(&arg1[4]) >= 0)
140002bc7        {
140002be1            char* rax_4 = sub_140001930(&arg1[4]);
140002be6            *(uint64_t*)((char*)arg1 + 0x2010) = rax_4;
140002bed            int32_t result_2 = -1;
140002bf7            void* rbx_1;
140002bf7
140002bf7            if (!rax_4)
140002bf7            {
140002ce6                int64_t* rax_8 = sub_1400039d0(&arg1[4], &data_14002d824);
140002ce6
140002cf1                if (!rax_8)
140002cf1                {
140002d38                    sub_140001e50("Could not load PyInstaller's emb…", &arg1[4]);
140002d3d                    result = 0xffffffff;
140002cf1                }
140002cf1                else
140002cf1                {
140002cff                    int64_t var_2048 = 0xe0b0a0b0049454d;
140002d11                    *(uint8_t*)((char*)var_2048)[3] = 0xd;
140002d11
140002d20                    if (sub_1400073d0(rax_8, &var_2048, 8))
```

The Python bytecode from PyInstaller executables can be extracted with a tool like pyinstxtractor. Let's go ahead and run that on the executable:

```
python pyinstxtractor.py encrypt.exe

[+] Processing encrypt.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.11
[+] Length of package: 9038480 bytes
[+] Found 107 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
```

```
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: encrypt.pyc
[+] Found 159 files in PYZ archive
[+] Successfully extracted pyinstaller archive: encrypt.exe

You can now use a python decompiler on the pyc files within the extracted directory
```

This gives us a folder with a bunch of DLLs and `.pyc` files. Our goal is to find the `.pyc` file that contains the program logic. Of the `.pyc` files extracted from the `.exe`, only `encrypt.pyc` looks interesting, so let's go ahead and analyze it.

encrypt.pyc	1/31/2025 5:54 PM	Compiled Python File	4 KB
pyi_rth_inspect.pyc	1/31/2025 5:54 PM	Compiled Python ...	2 KB
pyiboot01_bootstrap.pyc	1/31/2025 5:54 PM	Compiled Python ...	2 KB
pyimod01_archive.pyc	1/31/2025 5:54 PM	Compiled Python ...	6 KB
pyimod02_importers.pyc	1/31/2025 5:54 PM	Compiled Python ...	32 KB
pyimod03_ctypes.pyc	1/31/2025 5:54 PM	Compiled Python ...	7 KB
pyimod04_pywin32.pyc	1/31/2025 5:54 PM	Compiled Python ...	2 KB
struct.pyc	1/31/2025 5:54 PM	Compiled Python ...	1 KB

There are a variety of `.pyc` decompilers, such as [pycdc/Decompyle++]() and the online service [PyLingual](). For simplicity, we'll upload `encrypt.pyc` to PyLingual. This gives us the following result, which is [permalinked on PyLingual]():

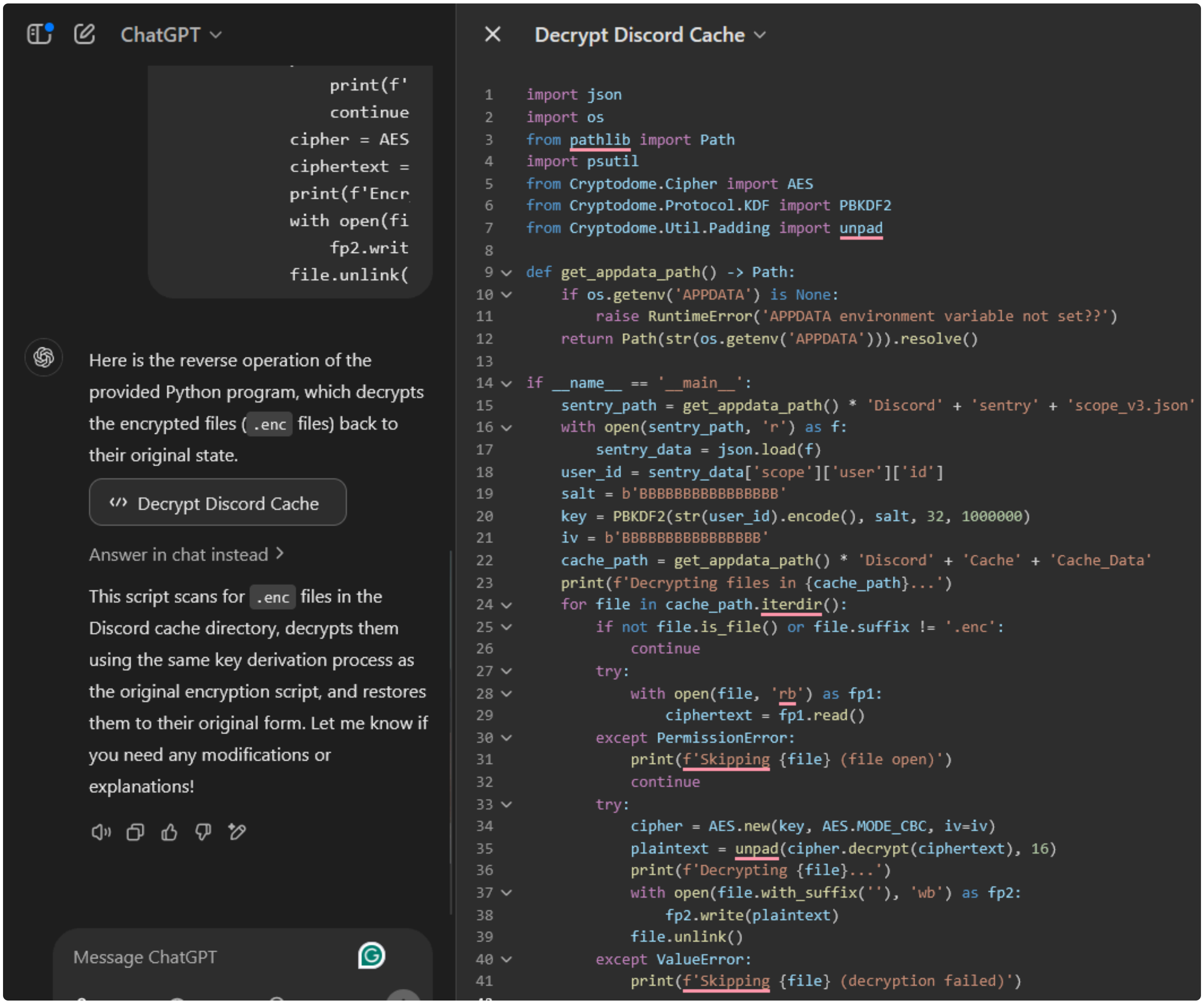
Original Bytecode | Patch Python | Empty Editor | Submit Patch Edit

```python
# Decompiled with PyLingual (https://pylingual.io)
# Internal filename: encrypt.py
# Bytecode version: 3.11a7e (3495)
# Source timestamp: 1970-01-01 00:00:00 UTC (0)

import json
import os
from pathlib import Path
import psutil
from Cryptodome.Cipher import AES
from Cryptodome.Protocol.KDF import PBKDF2
from Cryptodome.Util.Padding import pad

def get_appdata_path() -> Path:
    if os.getenv('APPDATA') is None:
        raise RuntimeError('APPDATA environment variable not set??')
    return Path(str(os.getenv('APPDATA'))).resolve()
if __name__ == '__main__':
    for proc in psutil.process_iter():
        if proc.name() == 'Discord.exe':
```

This seems to be what we're interested in. If we take a deeper look at the decompiled code, we can observe that it basically does three things:

- It tries to kill any processes called `Discord.exe`
- It recovers the Discord user ID from a file called `scope_v3.json`
- It uses the user ID to derive an AES-256-CBC key, which is then used to encrypt every file under the Discord `Cache_Data` directory

While we could write the decrypt tool manually, we can also just feed it to ChatGPT or GitHub Copilot. With a prompt of `Contained in triple backticks below is a Python program. Please write the program that performs the reverse operation.`, we get the following result from the free version of ChatGPT:



At this point, all we need is to export `scope_v3.json` from the disk image and update the path contained in the decryption script. After exporting `scope_v3.json` to the same folder as this script and changing some paths, we get this decryption script:

```
import json
from pathlib import Path
from Cryptodome.Cipher import AES
from Cryptodome.Protocol.KDF import PBKDF2
from Cryptodome.Util.Padding import unpad

if __name__ == '__main__':
    with open("scope_v3.json", 'r') as f:
        sentry_data = json.load(f)
    user_id = sentry_data['scope']['user']['id']
    salt = b'BBBBBBBBBBBBBBBB'
    key = PBKDF2(str(user_id).encode(), salt, 32, 1000000)
    iv = b'BBBBBBBBBBBBBBBB'

    cache_path = Path(".").resolve()
    print(f'Decrypting files in {cache_path}...')
    for file in cache_path.iterdir():
        if not file.is_file() or file.suffix != '.enc':
            continue
        try:
```

```
            with open(file, 'rb') as fp1:
                ciphertext = fp1.read()
        except PermissionError:
            print(f'Skipping {file} (file open)')
            continue
        try:
            cipher = AES.new(key, AES.MODE_CBC, iv=iv)
            plaintext = unpad(cipher.decrypt(ciphertext), 16)
            print(f'Decrypting {file}...')
            with open(file.with_suffix(''), 'wb') as fp2:
                fp2.write(plaintext)
            file.unlink()
        except ValueError:
            print(f'Skipping {file} (decryption failed)')
```

Running this script correctly decrypts the files, which we can now view normally (although without extensions) :

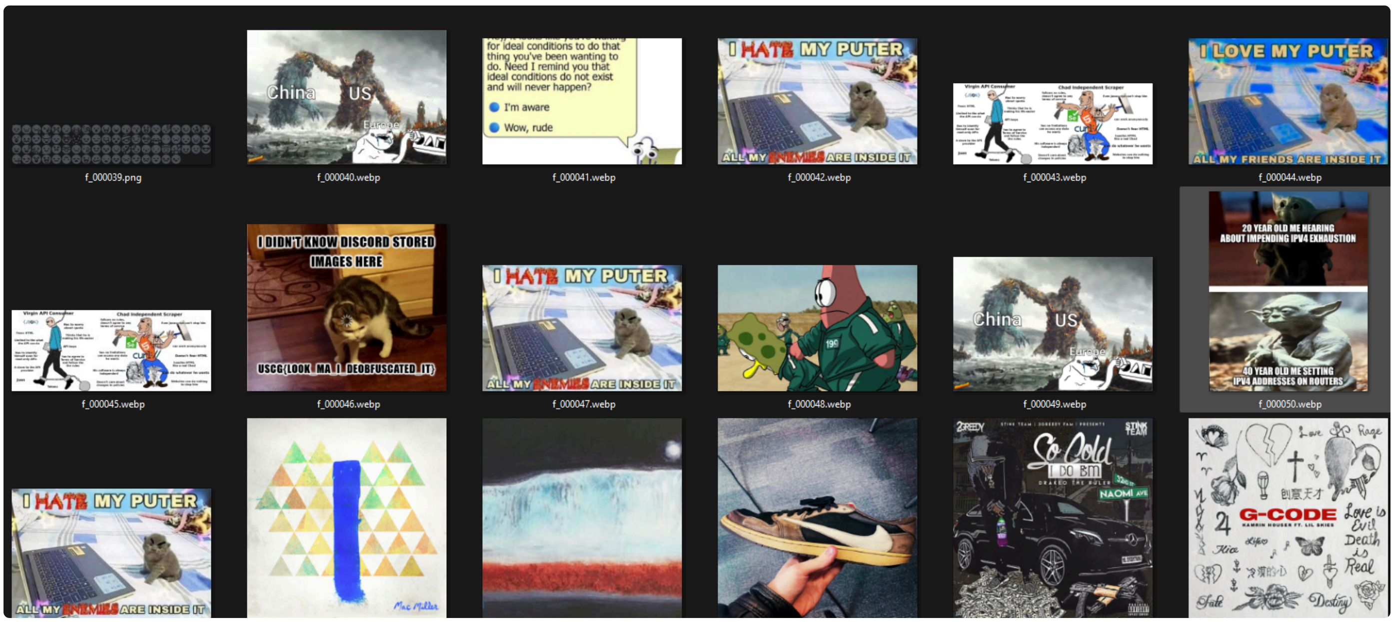
```
f_00002d:      data
f_00002e:      data
f_00002f:      gzip compressed data, original size modulo 2^32 889140
f_000003:      data
f_00003a:      PNG image data, 960 x 192, 8-bit/color RGBA, non-interlaced
f_00003b:      Web Open Font Format (Version 2), TrueType, length 25760, version 1.66
f_00003c:      gzip compressed data, from Unix, original size modulo 2^32 149580
f_00003d:      RIFF (little-endian) data, Web/P image
f_00003e:      RIFF (little-endian) data, Web/P image
f_00003f:      RIFF (little-endian) data, Web/P image
f_000004:      data
f_00004a:      RIFF (little-endian) data, Web/P image
f_00004b:      data
f_00004c:      RIFF (little-endian) data, Web/P image
f_00004d:      RIFF (little-endian) data, Web/P image
f_00004e:      data
f_00004f:      RIFF (little-endian) data, Web/P image
f_000005:      data
f_00005a:      RIFF (little-endian) data, Web/P image
f_00005b:      JPEG image data, JFIF standard 1.01, resolution (DPCM), density 118x118, segment length 16,
```

Again, while we could manually add extensions, we can write/ask for a script that does this for us:

```
import os
import magic

def add_extension(file_path):
    # Detect the file type
    file_type = magic.from_file(file_path, mime=True)

    # Map MIME types to file extensions
    mime_to_extension = {
        'text/plain': '.txt',
        'image/jpeg': '.jpg',
        'image/png': '.png',
        'application/pdf': '.pdf',
        'application/zip': '.zip',
        'application/x-tar': '.tar',
        'application/x-gzip': '.gz',
        'application/json': '.json',
        'text/html': '.html',
        'text/csv': '.csv',
        'image/webp': '.webp'
        # Add more mappings as needed
    }

    # Get the corresponding file extension
    file_extension = mime_to_extension.get(file_type)
```

```
    if file_extension:
        new_file_path = file_path + file_extension
        os.rename(file_path, new_file_path)
        print(f'Renamed {file_path} to {new_file_path}')
    else:
        print(f'Could not determine the file extension for {file_path} ({file_type})')

if __name__ == '__main__':
    directory = '.'  # Change this to the directory containing your files
    for file_name in os.listdir(directory):
        file_path = os.path.join(directory, file_name)
        if os.path.isfile(file_path) and '.' not in file_name:
            add_extension(file_path)
```

If we look through the files with extensions, we'll see a variety of cached images, one of which is the flag:



The flag is `uscg{look_ma_i_deobfuscated_it}`.