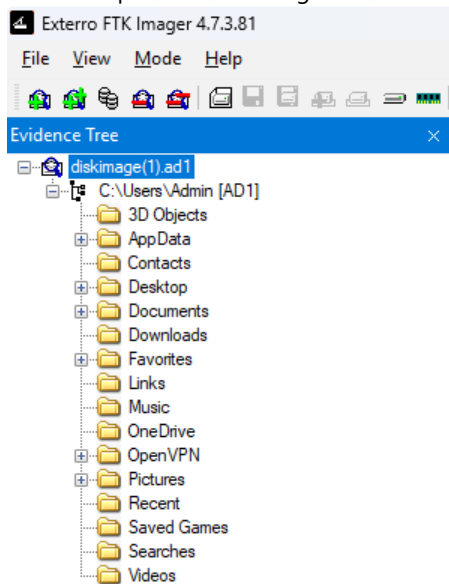


We're given a disk image (an `.ad1`, a logical image file) and the following facts:

- 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:



Quora

<https://www.quora.com/Where-are-Discord-pictures-stored> ⋮

### Where are Discord pictures stored?

On Windows, Discord is installed to `C:\Users\%username%\AppData\Local\Discord`. You find it by entering `%localappdata%\Discord` in the Windows ...

3 answers · Top answer: The app provides you with a specific URL that you can then include in your na...

Does Discord cache videos and images? If so, how do I get rid ... Jan 3, 2021

Does Discord cache images on mobile? - Quora May 17, 2022

Does discord actually delete messages? I tested it by sending ... May 15, 2023

How do I view Discord's mobile cache? - Quora Jul 14, 2022

More results from [www.quora.com](https://www.quora.com)

### People also ask ⋮

Where are Discord images stored?



How to see saved images on Discord?



Where do Discord saved files go?



How to find old Discord pictures?



[Feedback](#)

Reddit · [r/discordapp](https://www.reddit.com/r/discordapp)30+ comments · 2 years ago ⋮

### You should be careful what you post on discord, images ...

All images are stored within cache on every users PC within the server, indefinitely, until deleted manually by each person clearing their cache.

in which folder does discord save downloaded images - Reddit Mar 12, 2018

Where does discord store the images I view?[Mobile, android] Nov 5, 2019

Discord saves all DELETED images on your hard drive - Reddit Feb 14, 2019

TIP: To save images uploaded to discord without clicking ... Jun 20, 2018

More results from [www.reddit.com](https://www.reddit.com)

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:

Evidence Tree		File List			
diskimage(1).ad1		Name	Size	Type	Date Modified
C:\Users\Admin [AD1]		data_0.enc	45,072 (45 KB)	Regular File	1/31/2025 11:33:20 PM
3D Objects		data_1.enc	270,352 (265 KB)	Regular File	1/31/2025 11:33:20 PM
AppData		data_2.enc	3,153,936 (3,08...)	Regular File	1/31/2025 11:33:20 PM
Local		data_3.enc	4,202,512 (4,10...)	Regular File	1/31/2025 11:33:20 PM
LocalLow		f_000001.enc	204,448 (200 KB)	Regular File	1/31/2025 11:33:20 PM
Roaming		f_000002.enc	236,480 (231 KB)	Regular File	1/31/2025 11:33:20 PM
Adobe		f_000003.enc	3,758,976 (3,67...)	Regular File	1/31/2025 11:33:20 PM
autopsy		f_000004.enc	382,544 (374 KB)	Regular File	1/31/2025 11:33:20 PM
Binary Ninja		f_000005.enc	69,872 (69 KB)	Regular File	1/31/2025 11:33:20 PM
BinDiff		f_000006.enc	32,160 (32 KB)	Regular File	1/31/2025 11:33:20 PM
Code		f_000007.enc	29,296 (29 KB)	Regular File	1/31/2025 11:33:20 PM
Digital Detective		f_000008.enc	25,008 (25 KB)	Regular File	1/31/2025 11:33:20 PM
discord		f_000009.enc	24,112 (24 KB)	Regular File	1/31/2025 11:33:20 PM
1.0.9180					
blob_storage					
Cache					
Cache_Data					

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:

```
Hex editor
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII
00000000: 8D 96 F9 BC A4 12 D4 9F D4 6D 36 F0 6B E3 DA 0F .....m6.k...
00000010: F6 5A 44 04 85 33 B4 6C 8F C4 1E D3 88 AF 38 BE ...ZD..3.l....8.
00000020: 10 73 00 65 F8 8C 13 94 74 18 61 DF 64 71 7C 7F ...s.e...t.a.dq|.
00000030: A3 2A CC 90 6E 5E 72 AE 23 61 00 C8 DE BA 52 6F ...*.n^n.#a....Ro
00000040: 02 F8 59 5E A3 B8 AD 4C E6 79 01 B7 D9 BD BB C3 ...Y^...L.y....
00000050: BE 7F CF E6 A2 F9 3B F2 80 DE 61 87 F0 4E 4B 4E ...;...a...NKN
00000060: 35 54 D7 E2 03 34 90 5F 15 05 EC B4 54 84 2F 71 5T...4...T./q
00000070: 03 D2 2E C5 D4 89 86 5F CF DE 7C 77 84 54 8A FE ...|w.T...
00000080: 94 8B CC 5E 71 89 8F 9E AE 65 65 FB BA 74 69 51 ...^q...ee.tiQ
00000090: CA 03 00 3C 00 8B 29 54 2A 7C 4D CF 4D 3D F1 3C ...<...)M.M=<
000000A0: 66 FF F3 A0 47 B1 6C 0B 35 DD 04 57 14 67 83 76 f...G.l.5..W.g.v
000000B0: E5 B2 CF 4E 6E 89 6C B5 1E A8 A6 F5 B6 A9 37 91 ...Nn.l.....7.
000000C0: 9D 91 4E B5 9D 09 F0 B7 D4 E7 D6 B1 70 CB 71 6B ...N.....p.qk
000000D0: 9F 3D 05 67 20 BB E1 2A 90 0C D8 10 2C 9D C7 8C ...=g...*.
000000E0: A9 20 B7 B8 63 96 AD 5A 78 C9 78 33 38 8D 25 78 ...c..Zx.x38.%x
000000F0: 23 51 5C 23 C6 24 CF 81 2C 96 B5 98 6E C7 8E 49 #Q\#.$...n..I
00000100: 2B E0 ED 75 61 32 81 9A DF 13 FD 4D 17 BE 7F FC +...ua2....M....
00000110: 79 7B 95 F5 3E F7 D3 2F 21 C2 EC EB 77 2A 6D 8E y{...>./!...w*m.
00000120: CE F4 76 33 B2 84 64 7F 45 18 94 BF 76 A5 5B 7E ...v3..d.E...v.[~
00000130: EF EC 89 37 A9 24 FB C1 AC 8F A4 A2 95 13 6D 83 ...7.$.....m.
00000140: CF AD 1C 53 E6 FC 75 B8 B9 37 DF 86 E0 8A F1 B4 ...S...u...7.....
00000150: 40 9E 11 74 43 24 33 5E CD 51 40 3B F8 10 84 D4 @...tC$3^..Q@;...
00000160: 51 1A 33 3E 85 05 38 59 67 0B 48 81 9A 31 F2 EA Q.3>...8Yg.H..1..
```

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:

Name	Date modified	Type	Size
f_0003ef	1/31/2025 3:25 PM	File	237 KB
f_000004	1/31/2025 3:25 PM	File	374 KB
f_000005	1/31/2025 3:25 PM	File	38 KB
f_0005fb	1/31/2025 3:25 PM	File	181 KB
f_005b55	1/31/2025 3:25 PM	File	822 KB
f_005d9a	1/31/2025 3:25 PM	File	108 KB
f_000006	1/31/2025 3:25 PM	File	32 KB
f_000007	1/31/2025 3:25 PM	File	26 KB
f_007a9d	1/31/2025 3:25 PM	File	23 KB
f_000008	1/31/2025 3:25 PM	File	69 KB
f_008a05	1/31/2025 3:25 PM	File	177 KB
f_008a89	1/31/2025 3:25 PM	File	349 KB
f_008b0b	1/31/2025 3:25 PM	File	17 KB
f_008b0d	1/31/2025 3:25 PM	File	27 KB
f_008b3b	1/31/2025 3:25 PM	File	84 KB
f_008b4e	1/31/2025 3:25 PM	File	69 KB

```
kali@DESKTOP-S1SHU50: /mn x + v
f_000004: data
f_000005: data
f_0005fb: GIF image data, version 89a, 96 x 96
f_005b55: PNG image data, 288 x 288, 8-bit colormap, r
f_005d9a: GIF image data, version 89a, 80 x 80
f_000006: data
f_000007: data
f_007a9d: GIF image data, version 89a, 32 x 32
f_000008: data
f_008a05: PNG image data, 399 x 299, 8-bit/color RGB,
f_008a89: PNG image data, 288 x 288, 8-bit colormap, r
f_008b0b: data
f_008b0d: data
f_008b3b: RIFF (little-endian) data, Web/P image
f_008b4e: RIFF (little-endian) data, Web/P image, VP8
ould clamp
f_008b4f: RIFF (little-endian) data, Web/P image, VP8
ould clamp
f_008b20: RIFF (little-endian) data, Web/P image, VP8
ould clamp
f_008b23: PNG image data, 244 x 290, 8-bit/color RGBA,
f_008b51: PNG image data, 399 x 223, 8-bit/color RGB,
f_008b60: RIFF (little-endian) data, Web/P image
f_008b62: PNG image data, 255 x 300, 8-bit/color RGBA,
f_008b83: RIFF (little-endian) data, Web/P image
f_008b84: RIFF (little-endian) data, Web/P image, VP8
```

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:

Name	Size	Type	Date Modified
desktop.ini	282 (1 KB)	Regular File	7/8/2024 11:37:22 PM
DiscordSetup.exe	107,226,856 (10...	Regular File	2/1/2025 4:33:26 AM
encrypt.exe	9,368,717 (9,15...	Regular File	1/31/2025 11:33:06 PM

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

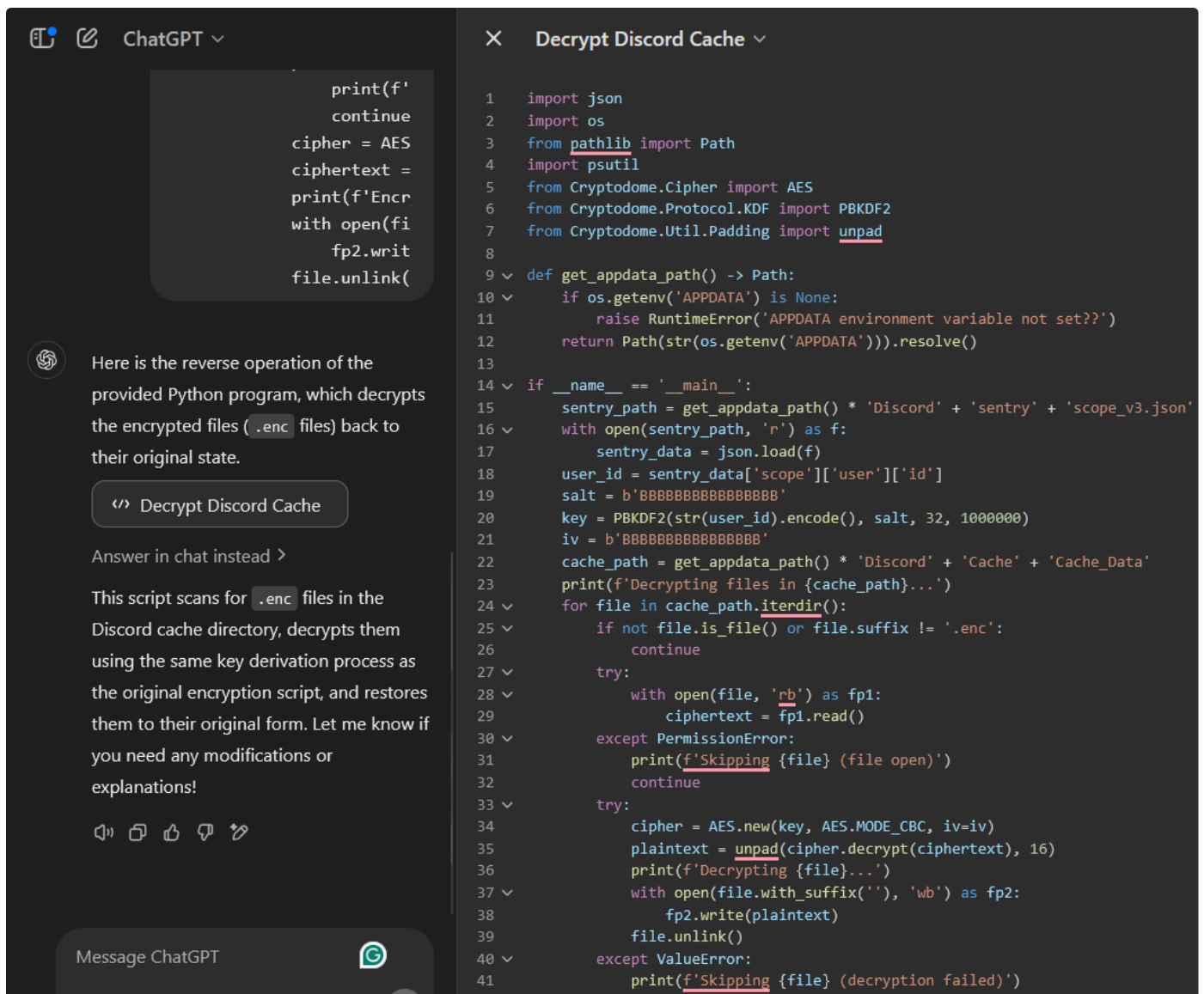
There are a variety of .pyc decompilers, such as [pycdc/Decompyle++](#) and the online service [PyLingual](#). For simplicity, we'll upload encrypt.pyc to PyLingual. This gives us the following result, which is [permalinked on PyLingual](#):

Original Bytecode	Patch Python	Empty Editor	Submit Patch Edit ↗
<pre>1  # Decompiled with PyLingual (<a href="https://pylingual.io">https://pylingual.io</a>) 2  # Internal filename: encrypt.py 3  # Bytecode version: 3.11a7e (3495) 4  # Source timestamp: 1970-01-01 00:00:00 UTC (0) 5 6  import json 7  import os 8  from pathlib import Path 9  import psutil 10 from Cryptodome.Cipher import AES 11 from Cryptodome.Protocol.KDF import PBKDF2 12 from Cryptodome.Util.Padding import pad 13 14 def get_appdata_path() -&gt; Path: 15     if os.getenv('APPDATA') is None: 16         raise RuntimeError('APPDATA environment variable not set??') 17     return Path(str(os.getenv('APPDATA'))).resolve() 18 if __name__ == '__main__': 19     for proc in psutil.process_iter(): 20         if proc.name() == 'Discord.exe':</pre>			

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_00003: 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_00004: 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_00005: 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)

```



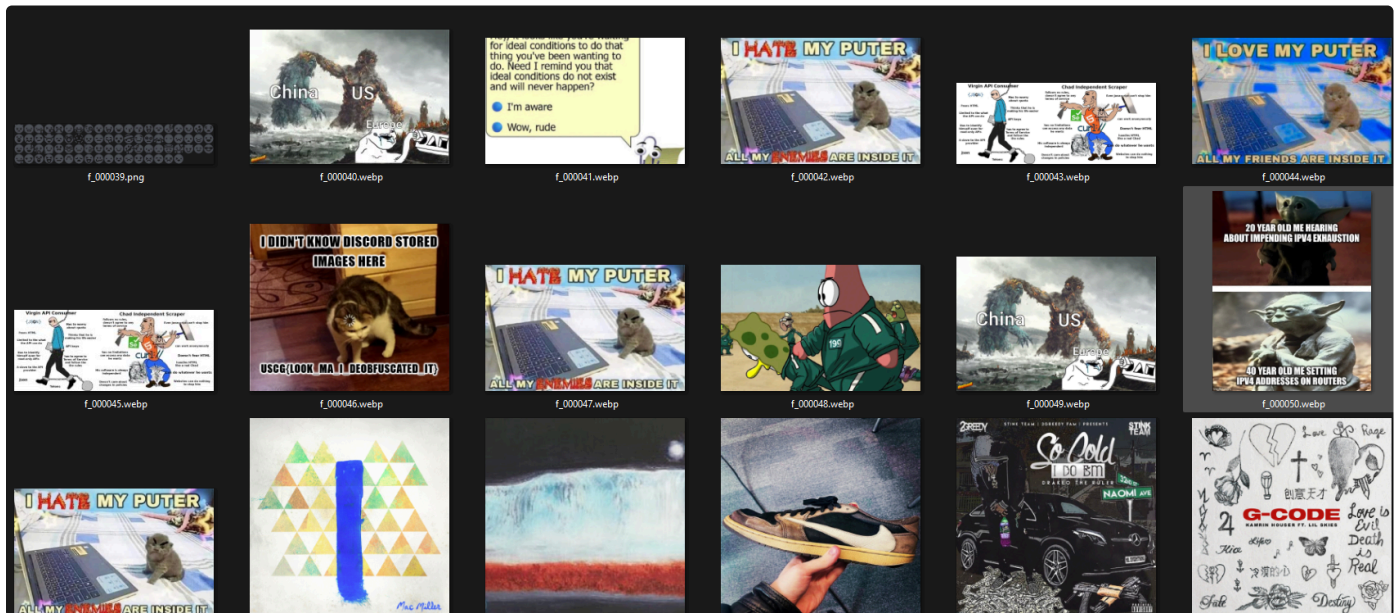
```

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}`.