GhostImage POC - Payload Structure Overview

Objective:

Demonstrate how a `.heic` image file can appear normal while containing a covert data layer (JSON payload) that is parsed when certain apps or environments mishandle metadata or preview routines.

1. File Structure Breakdown (Safe Emulated):

[HEIC Container]

??? ftyp (File Type Box)

??? meta (Metadata Box)

?   ??? iloc (Item Location)

?   ??? iinf (Item Info)

?   ??? iprp (Item Properties)

?   ?   ??? JSON payload inside a custom "Exif" or "XMP" segment

??? mdat (Media Data Box)

2. Injected JSON Payload Example:

Embedded inside a custom Exif-like segment:


```
{
  "ghostcore_tag": "ghostImage_v1",
  "timestamp": "2025-04-17T04:44:00Z",
  "origin": "scan.sigils.x.com",
  "reaction": {
    "alert": false,
    "redirect": "https://ghostcore.local/init",
    "note": "Silently processed in background view"
  }
}
```

This would sit encoded in UTF-8 within a valid block like:

45786966 0000... 7b226768 6f737463 6f72655f 74616722...

3. Simulated Use Case:

- A social media app auto-previews uploaded images.

- Its metadata parser is built in JS/Native bridge.

- A background service reads all Exif for indexing.

- If that bridge fails to sanitize the metadata fields, it could unintentionally invoke a call.

4. Ethical Safe Testing Note:

This is not a working exploit-just an illustrated proof-of-format.

No code executes. No malicious shell. No obfuscation beyond harmless JSON.

If you're testing detection systems or image sanitation layers, this is a viable vector to examine.