Challenge 5 – Hidden Data Extraction from PNG

Objective
The objective of this challenge was to analyze the given PNG file (`final.png`), identify any hidden embedded data inside it, and recover the secret flag.

Initial Observation
At first glance, the file appeared to be a normal PNG image.
However, in CTF challenges, image files often contain hidden payloads inside their binary data or metadata.
So instead of treating it as a simple image, the file was analyzed at the binary-string level.

`final.png`

Step 1 – Extracting Raw Printable Data
Tool Used: `strings` Command executed:

```
strings final.png > out.txt
```

This command extracted all printable character sequences from the binary PNG file.
At this stage, most of the output appeared as unreadable or random characters.
This was expected because PNG is a compressed binary format.
(Pehle pura rubbish data nikala — real forensic workflow ka first step.)

Step 1 – Extracting Raw Printable Data

Tool Used: `strings`

Command executed:

```
strings final.png > out.txt
```

This command extracted all printable character sequences from the binary PNG file.

At this stage, most of the output appeared as unreadable or random characters.

This was expected because PNG is a compressed binary format.

(Pehle pura rubbish data nikala — real forensic workflow ka first step.)

Step 2 – Searching for Flag-like Patterns

CTF flags usually follow a structure with curly braces `{ }`.

So the next step was to search for any brace-enclosed sequences inside the extracted text.

Using PowerShell:

```
Select-String -Pattern "\{.*?\}" -Path out.txt
```

This returned many matches, but most were random brace sequences caused by compressed PNG data — false positives.

- Total corrupted entries detected: ~130+
- Common pattern: { } appearing with random ASCII symbols
- Sample extracted anomalies:

d{\} G{}*6 O={} nw{} F{}r:
W{}>: XF{} {}IM {}$> {}B$
VE{}K P{:} Q{}q#3 I{Z}j W{+}G

- Indicates:
  - Non-readable binary fragments
  - Possible compression / encoding artifacts
  - Required post-cleaning & filtration

Step 3 – Filtering Meaningful Data

Among all extracted brace patterns, one sequence stood out because it was short, clean, and readable:

`{93F_z}`

All other brace matches contained unreadable or random symbols, confirming they were just compression artifacts.

Thus `{93F_z}` was identified as the meaningful hidden payload.

Step 4 – Constructing the Final Flag

Based on the challenge naming convention, the expected flag format was:

`SAIC{...}`

Combining the prefix with the extracted payload:

`SAIC{93F_z}`

How the Hidden Data Was Decoded

The PNG file contained embedded readable data inside its binary structure.

By first dumping all printable strings and then filtering for structured brace patterns, the real hidden sequence was isolated.

Random brace outputs were discarded as compression noise, while the clean readable token {93F_z} revealed the embedded message.

Techniques and Tools Used

- **`strings`** → to extract printable sequences from binary PNG
- PowerShell **`Select-String`** → to filter brace patterns
- Manual inspection → to distinguish real payload from binary noise

Final Recovered Flag

SAIC{93F_z}