WPCTF 2025 - Shadow Garden - Writeup

Category: MISC (Forensics / Steganography)
Difficulty: Easy-Medium
Note: The flag used is fictitious and meant to preserve the challenge-solving experience.

Challenge Description

An encrypted transmission from The Eminence himself has been intercepted, an ancient video relic known only as intro.mkv.

Once a symbol of power, it has now been tainted.

The Cult of Diablos has corrupted the signal, twisting its form and shrouding its true purpose in static and chaos.

Our analysts believe fragments of Shadow's will still remain within.

What you see is merely the surface, the truth lies buried beneath layers of distortion.

Your task is simple in words, but heavy in consequence:
restore what has been lost.

Recover the hidden essence of the transmission and prove your worth to the Shadow Garden.

"Those who see only light will never uncover the truth that dwells in the dark."

Files: `ιηтrσ.mkv (522 MB)`

Initial Reconnaissance

First Look at the File

Opening the video in VLC shows something interesting:

- **0:00 - 0:22**: Normal anime intro (The Eminence in Shadow)
- **0:22 - 1:29**: Video becomes completely pixelated/glitched
- **0:22 onwards for a few seconds**: High-pitched annoying noise starts

Metadata Analysis

Always check metadata first in forensics/steganography/misc challenges:

Note: The data shown are only a subset of the available ones. Not all are relevant to the challenge, so only the most notable fields are included.

```
$ file ιηтrσ.mkv
ιηтrσ.mkv: Matroska data

$ mediainfo ιηтrσ.mkv
General
Format                                   : Matroska
File size                                : 522 MiB
Duration                                 : 1 min 29 s
Comment                                  : S33d 0.O p3 ratio Nx0r

Video
Format                                   : AVC
Width                                    : 1 920 pixels
Height                                   : 1 080 pixels
Frame rate                               : 60.000 FPS

Audio
Format                                   : PCM
Channels                                 : 1 channel
Sampling rate                            : 44.1 kHz
```

Critical finding! The comment field contains: S33d 0.O p3 ratio Nx0r
Let's decode this hint:
S33d 0 → Seed 0
.O p3 ratio Nx0r → Operation XOR

This hint will be crucial for the video section. Let's bookmark it and start with the audio.

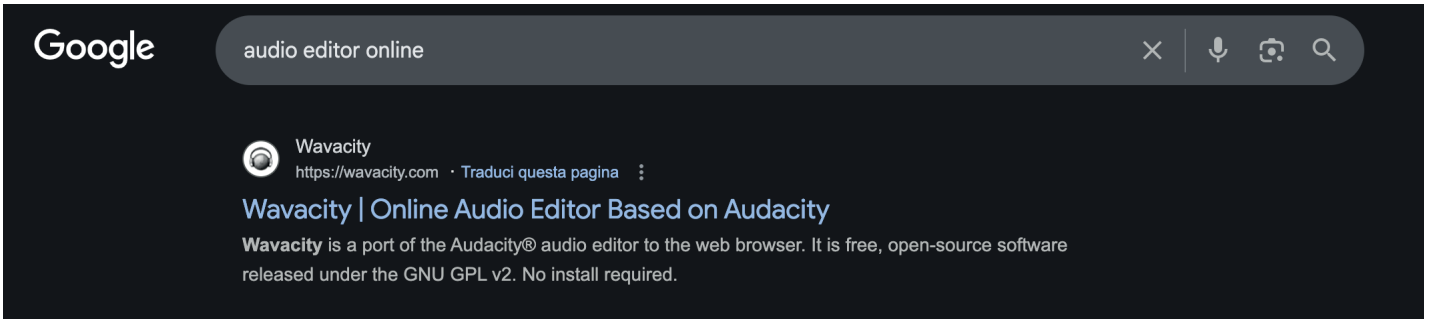
Part 1: Audio Analysis

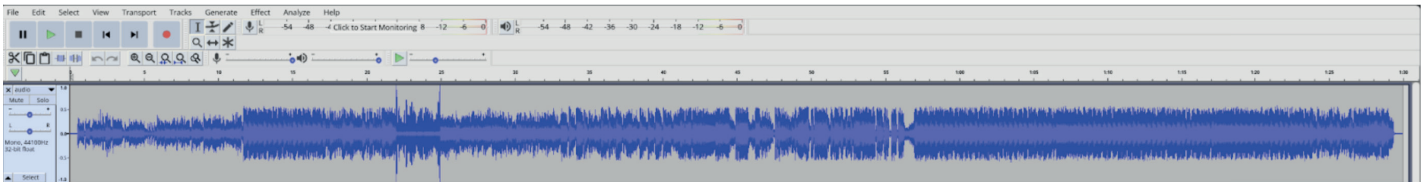
Extracting the Audio Track

```
$ ffmpeg -i ιηтrσ.mkv -vn audio.wav
```

Waveform Analysis

Let's visualize the waveform in Audacity or something similar.
Without installing any additional software it has been used an online video editor:

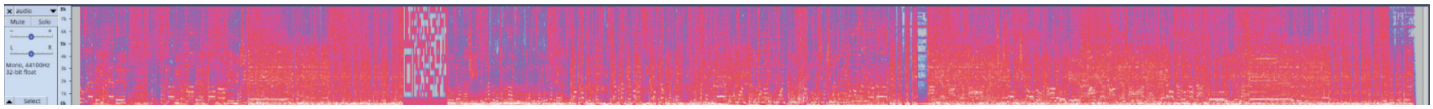




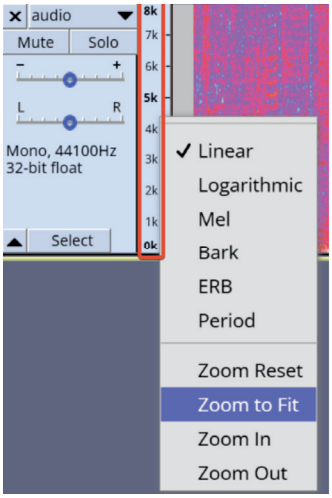
The high-pitched noise starts exactly at 22 seconds and continues for a few seconds. Looking at the spectrogram view it reveals something very interesting...

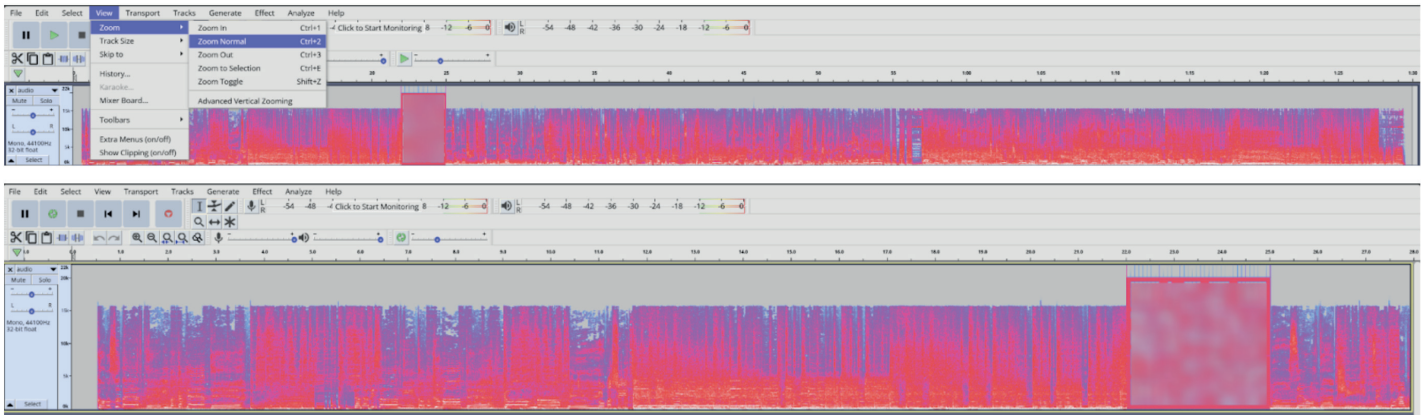
Spectrogram Analysis

In CTF, audio steganography often hides data in the frequency domain.



Whoa! That's clearly a **QR code pattern** in the spectrogram!
From now on we just have to adjust the **zoom**





Success! We got the flag prefix: **WPCTF{Sh4d0w5_**
This is only the beginning of the flag. The rest must be hidden in the video track.

Note: This is not the real one.

Part 2: Video Analysis

Now let's tackle the glitched video section. We already have a critical hint from the metadata: **S33d 0.O p3 ratio Nx0r** (Seed 0, Noise XOR).

Understanding the Visual Glitch

Let's extract some frames around the glitch start to analyze:

```
$ ffmpeg -i ιηrrσ.mkv -vf "select='between(t,21,23)'" -vsync 0
frame_%04d.png
```

Comparing frames:

- **Frame 1260 (21s)**: Normal anime content
- **Frame 1320 (22s)**: Completely pixelated
- **Frame 1320+**: Pixelated with visible horizontal movement

The transition is **instant** and the pattern shows **left-scrolling motion**.

Frame-by-Frame Difference Analysis

```
import cv2
import numpy as np

def calculate_frame_difference(frame1, frame2):
    """Calculate mean absolute difference between frames"""
    gray1 = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)
    return np.mean(cv2.absdiff(gray1, gray2))

def detect_glitch_start(video_path, threshold=30.0, min_-
stable=3):
    """
    Detect when the glitch starts by finding sustained high frame
differences
    """
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)

    frame_idx = 0
    prev_frame = None
    stable_count = 0
    first_anomaly = None

    print("[*] Scanning for glitch start...")
```

```
  while True:
        ret, frame = cap.read()
        if not ret:
            break

        if prev_frame is not None:
            diff = calculate_frame_difference(prev_frame, frame)

            if diff > threshold:
                if stable_count == 0:
                    first_anomaly = frame_idx
                stable_count += 1

                if stable_count >= min_stable:
                    cap.release()
                    print(f"[+] Glitch detected at frame {fir-
st_anomaly} (~{first_anomaly/fps:.2f}s)")
                    return first_anomaly
            else:
                stable_count = 0
                first_anomaly = None

        prev_frame = frame.copy()
        frame_idx += 1

    cap.release()
    return None

# Detect glitch
glitch_start = detect_glitch_start('ιηтrσ.mkv')
```

Output:

```
[*] Scanning for glitch start...
[+] Glitch detected at frame 1320 (~22.00s)
```

Perfect! The glitch starts exactly at **frame 1320.**
Generating Noise Pattern from Metadata Hint
Now let's use the metadata hint we found earlier: **S33d 0.O p3 ratio Nx0r**

```
import cv2
import numpy as np
```

```
# Get video dimensions
cap = cv2.VideoCapture('ιηтrσ.mkv')
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
cap.release()

print(f"[*] Video dimensions: {width}x{height}")

# Generate noise using seed from hint
SEED = 0
np.random.seed(SEED)
noise = np.random.randint(0, 256, (height, width, 3), dtype=n-
p.uint8)

print(f"[+] Generated noise matrix: {noise.shape}")
print(f"[*] Noise sample (top-left corner):")
print(noise[0:3, 0:3, 0])
```

Output

```
[*] Video dimensions: 1920x1080
[+] Generated noise matrix: (1080, 1920, 3)
[*] Noise sample (top-left corner):
[[172 140 196]
[106 192 104]
[136 125 154]]
```

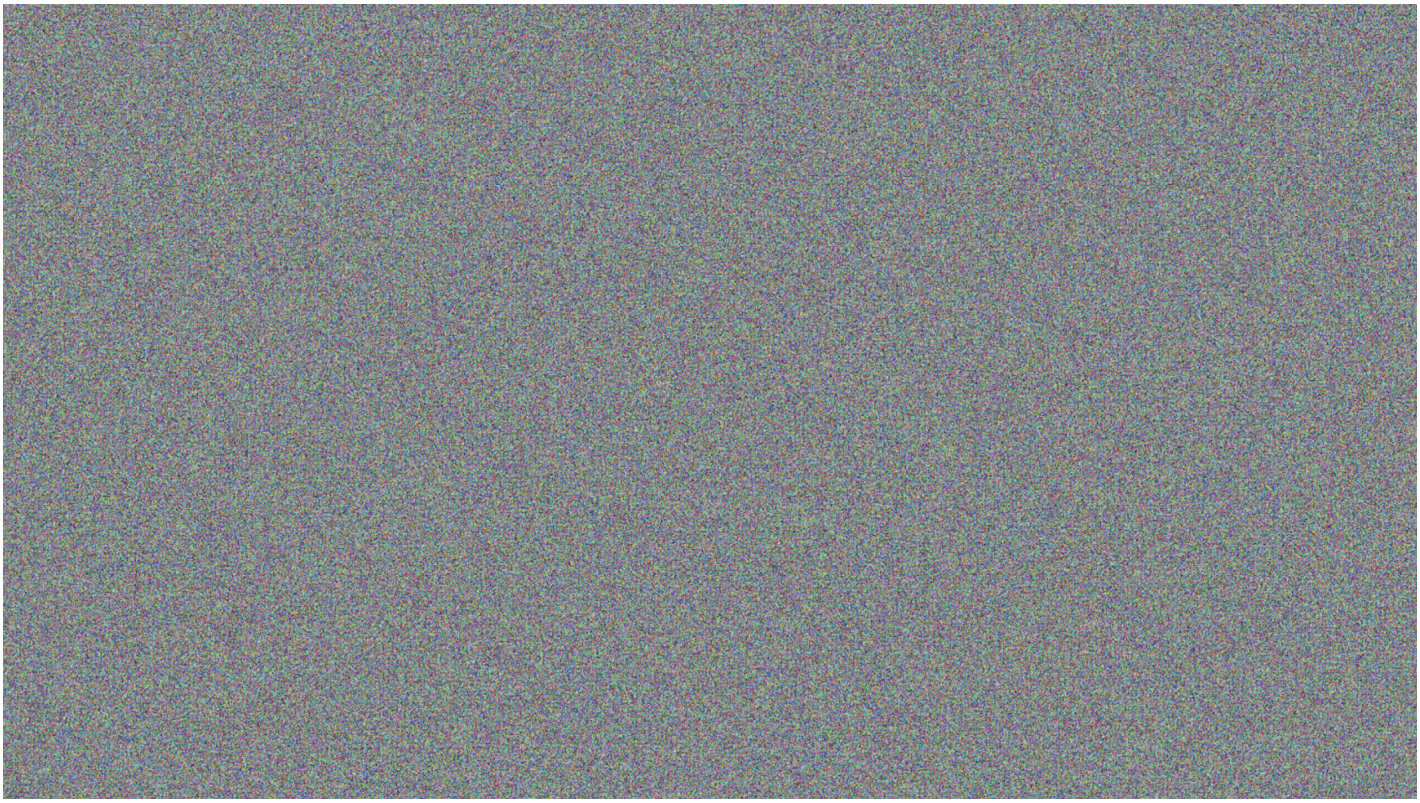
Now we have our noise matrix. According to the hint, we need to XOR frames with this noise somehow.

Direct XOR

The most straightforward approach. Let's try XOR-ing a glitched frame directly with our noise:

```
import cv2
import numpy as np


# Generate noise using seed from hint
SEED = 0
np.random.seed(SEED)
noise = np.random.randint(0, 256, (1080, 1920, 3), dtype=n-
p.uint8)
```

```
# Extract a frame from the middle of the glitch
cap = cv2.VideoCapture('ιηтгσ.mkv')
test_frame = 2000  # Middle of glitched section
cap.set(cv2.CAP_PROP_POS_FRAMES, test_frame)
ret, frame = cap.read()
cap.release()

# Direct XOR with noise
decrypted_direct = cv2.bitwise_xor(frame, noise)

# Save for visual inspection
cv2.imwrite('test_direct_xor.png', decrypted_direct)
print(f"[+] Saved test_direct_xor.png")
print("[*] Checking result...")

# Try to detect QR code
from pyzbar.pyzbar import decode
gray = cv2.cvtColor(decrypted_direct, cv2.COLOR_BGR2GRAY)
qr_codes = decode(gray)

if qr_codes:
    print(f"[+] QR detected: {qr_codes[0].data.decode()}")
else:
    print("[-] No QR code detected")
```



The frame is still completely pixelated. Direct XOR isn't the solution.

Thinking: The visual observation showed horizontal scrolling movement. If the encryption were static, we wouldn't see any movement at all. The noise must be **changing per frame somehow**!

Hypothesis: Rolling Noise Pattern

The left-scrolling motion suggests the noise is being **shifted horizontally** for each frame. Let's test this theory:

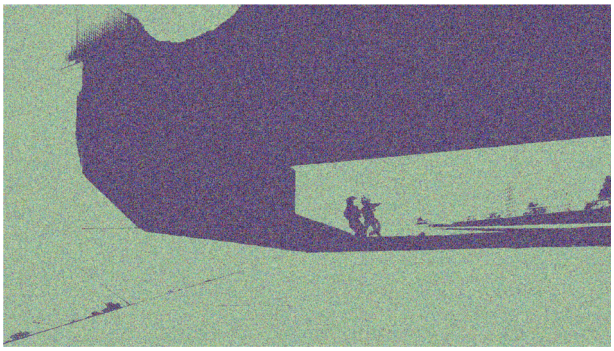
```
import cv2
import numpy as np

# Generate noise using seed from hint
SEED = 0
np.random.seed(SEED)
noise = np.random.randint(0, 256, (1080, 1920, 3), dtype=n-
p.uint8)

glitch_start = 1320

print("\n[*] Testing rolling noise hypothesis...")

# Mathematical model:
# If noise scrolls left, that means each frame uses the noise
# rolled/shifted by some amount based on the frame number

def test_rolling_pattern(video_path, noise, glitch_start, te-
st_frames):
    """Test XOR with horizontally rolled noise"""
    cap = cv2.VideoCapture(video_path)
    width = noise.shape[1]

    print(f"[*] Testing on frames: {test_frames}")

    for frame_num in test_frames:
        cap.set(cv2.CAP_PROP_POS_FRAMES, frame_num)
        ret, frame = cap.read()

        # Calculate offset from glitch start
        offset = frame_num - glitch_start

        # Hypothesis: shift amount = offset (modulo width for
cycling)
        shift = offset % width

        # Roll noise LEFT (negative) - matches visual left-scrol-
ling
        rolled_noise = np.roll(noise, -shift, axis=1)

        # XOR with rolled noise
```

```
 decrypted = cv2.bitwise_xor(frame, rolled_noise)

        # Save result
        filename = f'test_rolling_frame{frame_num}.png'
        cv2.imwrite(filename, decrypted)

    cap.release()

# Test on several frames
test_rolling_pattern('ιηтгσ.mkv', noise, glitch_start,
                     test_frames=[2000, 2100, 2200])
```

Excellent! The rolling noise approach works, QR codes are now visible.

test_rolling_frame2000.png

test_rolling_frame2100.png



test_rolling_frame2200.png

Understanding the mechanism:

Frame	offset	shift	effect
1320 (glitch_start)	0	0	original noise
1321	1	1	roll left 1 pixel
1322	2	2	roll left 2 pixels
3240	1920	0	back to original (cycle)
...	...	...	...

This creates the visual left-scrolling effect!

Analyzing Batch Structure

We're getting different characters from different frames. Let's figure out how many frames contain each QR code:

```
import cv2
import numpy as np
from pyzbar.pyzbar import decode

# Generate noise matrix using seed from metadata hint
SEED = 0
np.random.seed(SEED)
noise = np.random.randint(0, 256, (1080, 1920, 3), dtype=n-
p.uint8)

# Frame where glitch starts
glitch_start = 1320

def extract_complete_flag(video_path, noise, glitch_start, au-
dio_prefix, batch_size):
    cap = cv2.VideoCapture(video_path)
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))

    # Start from glitch frame
    cap.set(cv2.CAP_PROP_POS_FRAMES, glitch_start)

    frame_idx = glitch_start
    frames_in_batch = 0
    flag = audio_prefix
    last_qr = None

    print(f"Progress: {flag}", end='', flush=True)

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Decrypt: XOR with horizontally rolled noise
        offset = frame_idx - glitch_start
        shift = offset % width
        rolled_noise = np.roll(noise, -shift, axis=1)
        decrypted = cv2.bitwise_xor(frame, rolled_noise)

        # Preprocess for better QR detection
        gray = cv2.cvtColor(decrypted, cv2.COLOR_BGR2GRAY)
        blurred = cv2.GaussianBlur(gray, (5, 5), 0)
        _, binary = cv2.threshold(blurred, 0, 255, cv2.THRESH_BI
NARY + cv2.THRESH_OTSU)

        # Detect QR codes
        qr_codes = decode(binary)
```

```
        # Get first detection
        if qr_codes:
            try:
                last_qr = qr_codes[0].data.decode('utf-8')
            except:
                pass

        frames_in_batch += 1

        # Process batch every N frames
        if frames_in_batch % batch_size == 0:
            if last_qr:
                flag += last_qr
                print(last_qr, end='', flush=True)

                # Stop at closing brace
                if last_qr == '}':
                    break

            last_qr = None

        frame_idx += 1

    cap.release()
    print(f"\n\nFinal flag: {flag}")
    return flag

# Run extraction
final_flag = extract_complete_flag('ιηтrσ.mkv', noise, glitch_-
start, audio_prefix="WPCTF{Sh4d0w5_", batch_size=30)
```

Output:

```
Progress: WPCTF{Sh4d0w5_D44nncc33__11nn...
```

Close! We're getting multiple times the same value, we need to find the right batch size.

After different attempts, **batch_size=50** got the flag.

FLAG

WPCTF{Sh4d0w5_D4nc3_1n_7h3_D4rkn355}

Note: This is not the real one.