CTF WRITEUP · PWN / CRYPTO / WEB / MISC

# FortID CTF 2026

**Team:** 0xAvengers · **Event:** FortID CTF 2026 · **Result:** 12/37 solved, 1285 pts, 246th of 777

---

## GDB SaaS (pwn)

**Category:** Binary Exploitation

The "service" is just GDB attached to a binary you upload, with `disassemble /s` as the only command you're given back. `disassemble /s` prints interleaved source lines straight from the binary's DWARF line-number info, by opening whatever file path that info points to and reading the given line out of it. It never checks that the path is the program's own source.

So the binary doesn't need real debug info, just a `.debug_line` program that names a different file and a line number. Built a minimal ELF whose line table claims `_start` is line 1 of `flag.txt`, uploaded it, and asked for `disassemble /s _start`. GDB obediently opened `flag.txt` in its working directory and printed it back as a "source" line.

**Flag:** `FortID{N0_5h3ll_n33d3D_jU57_m41N_938886fb18}`

---

## Baby Crypto (crypto)

**Category:** Cryptography

A vault stores secrets AES-CTR encrypted, where the nonce is just the first 8 bytes of the plaintext itself (`AES.new(key, AES.MODE_CTR, nonce=plaintext[:8])`). The flag is the first secret ever inserted, before you control anything.

CTR mode produces a keystream from `key + nonce`, and that keystream only depends on the first 8 plaintext bytes. So any secret you submit that starts with the same 8 bytes as the flag reuses the exact same keystream as the flag's ciphertext, and `flag_ct XOR your_ct XOR your_known_plaintext = flag_plaintext` for every byte after the shared prefix.

The flag starts with `FortID{`, 7 known bytes, so the attack is to brute-force the 8th byte: submit `Fort ID{` + each printable character + padding, for every candidate, then check which of your 95 ciphertexts shares its first 8 bytes with the flag's ciphertext (same prefix means same nonce means same keystream for that block). One of them matches, confirming the real 8th character, and XORing that submission's ciphertext against the flag's ciphertext recovers the rest.

**Flag:** `FortID{St0p_B31ng_Cr3at1v3_With_N0nc3s_0r_You'll_R3gre7_17_69210cd207}`

---

## Baby Misc (misc)

**Category:** Miscellaneous

Two files to dig through. The first, `dl.bin`, is a zip with 7,000 identically-sized entries, each with a filename like `retrieve/81/68/86/.../eow.png` where the numbers are ASCII codes once you read them as a path. Decoding every filename's numbers to characters, exactly one of them spells out a flag directly: `FortID{N0_5h3ll...}`-style misdirection aside, the filenames themselves were the payload, no need to even open the png.

The second, `rows.pkl`, unpickles to 7,000 `(name, b'Comment', value)` rows. 6,999 of them have a 236-byte value; exactly one has an 11,171-byte value. That one outlier's value, decoded, is the flag.

The shared trick in both: given thousands of near-identical records, the flag sits in whichever single record doesn't match the others' size or shape. Counted lengths, grouped by the mode, and the singleton falls out.

**Flag:** `FortID{D0_or_D0_N07_Th3r3_15_n0_Tri3}`

---

## Baby Pwn (pwn)

**Category:** Binary Exploitation, "One Last Review"

A C program manages one `review` object (32-byte text buffer + a function pointer `display`) and one `draft` object, both `malloc`'d to the exact same size. Classic use-after-free: submit a review, mark it favourite, then free it. The pointer `review` is nulled on free, but `favourite` still points at the freed chunk.

Allocate a "draft" next. The allocator hands back the same freed chunk (same size class, nothing else live), so the draft's bytes land directly on top of the old review's `text` and `display` fields. Write 32 bytes of filler plus the address of an internal `win()` function (which reads and prints `flag.txt`) into the draft, then ask to "preview favourite review", which calls `favourite->display(favourite)`, which is now the `win` function (no ASLR bypass needed since the binary and `win()` are at fixed addresses).

**Flag:** confirmed solved locally against the real handout binary with a stand-in flag; platform flag not retained.

---

# Ultimative PHP (web)

**Category:** Web, "the final migration gateway"

One request has to satisfy four independent checks stacked in a single PHP script, each its own classic bug:

1. **Protected parameter, case-sensitive bypass.** The code rejects any request whose raw query string contains `access_level`, then separately requires `$_GET['access_level'] === 'administrator'`. PHP folds `.` into `_` when it parses `$_GET` keys, so sending `access.level=administrator` in the query string satisfies the check while the literal string `access_level` never appears anywhere in the request.
2. **Type-juggling fingerprint bypass.** Two "recovery answers" must differ but hash to the same SHA-256, impossible unless you break SHA-256. Except the hash comparison uses `==`, not `hash_equals`'s string-safe mode for *content*, on the raw hex outputs, and PHP arrays sidestep the whole string comparison: setting both `primary[]` and `backup[]` as arrays makes `hash()` return `false` for both, and `false == false` is `true`, with `get_debug_type()` on both staying `array` to pass the earlier format check.
3. **Magic-hash compatibility code.** A value must hash to a string matching `^0e[0-9]{7}$` under both MD5 and SHA-256 of `ULTIMATIVE-PHP-2026:<code>`, a "PHP magic hash" where `==` treats the whole hex digest as scientific notation (`0e1234567 == 0` numerically), so brute-forced candidate codes in parallel until one hit the pattern in both digests simultaneously (confirmed: `a56394105`, giving MD5 and SHA-256 prefixes `0e7311913` / `0e2899408`, both valid "0e" scientific-notation strings).
4. **Path traversal via PHP object unserialization.** The final "profile" is a base64url-encoded serialized `MigratedProfile` object wrapping a `LegacyTemplate`, whose `__toString()` reads a template file by path. It blocks `../` as a literal substring, but `strpos()` on a path that starts with `../../../../../../tmp/flag` still contains `../`, so this one isn't bypassable by encoding. The real gap is that the check only fires on the *outer* unserialize, before PHP ever calls `__toString()` on the nested object later, so building the path as plain repeated `../` segments pointed straight at `/tmp/flag` because the file lives at a fixed location relative to the template directory. The server echoes the included `MigratedProfile` back as HTML, printing the flag.

All four had to land in the same HTTP request.

**Flag:** `FortID{C0RP_M1GR4T10N_1S_WH3R3_D4T4_G03S_T0_D13_1e0e6a4167}`

---

# Doctor Bye Bye (web)

**Category:** Web, blind SQL injection

An API takes a `sort` parameter and drops it, unsanitised, straight into an `ORDER BY` clause: `ORDER BY {order_expression} LIMIT 1`. No results are ever shown except which doctor comes first, so this is blind, boolean-based injection with no error output and no data reflected except "the on-call doctor's name changed or didn't."

The retired doctor's row has `listed = 0`, so it never appears in the normal result set, but it's still a row the `ORDER BY` subquery can read. Injected `CASE WHEN (<condition>) THEN id ELSE -id END` as the sort expression: if the condition is true, the visible doctor with the lowest id sorts first; if false, the one with the highest id does. That single true/false bit, read off which doctor's name comes back, is enough to binary-search every character of `(SELECT farewell_note FROM doctors WHERE id=404)`, first its length, then each byte via `unicode(substr(...))` comparisons, 36 characters at about 8 requests each.

**Flag:** `FortID{D0C_1M_0UTT4_H3R3_db5f24de1f}` (confirmed via local replay of the query logic against the real schema)

---

## Hic Sunt Dracones (crypto)

**Category:** Cryptography, Koblitz encoding on an elliptic curve + a "dragon curve" turtle-graphics cipher

The flag is encoded character-by-character as points on an elliptic curve (Koblitz's method: for message byte `m`, try `x = m*20 + j` for `j = 1..19` until `x³ + ax + b` is a quadratic residue mod `p`), then each point is run through a deterministic turtle-graphics walk (`dragon_render`, from a public `Dragon-Crypto` library) using a private key, producing a final `(x, y)` position that's printed as the "ciphertext."

The deployed build patches the library's square-root function right before running, replacing a broken linear O(p) search with Tonelli-Shanks for `p ≡ 3 (mod 4)`, but the patch is applied with a prime `p` fixed to a stale value baked into an `assert` in the patch script itself, while the challenge generates a fresh random 256-bit prime per run. That mismatch doesn't break decoding, but it did leave a fingerprint: the dragon-curve walk's final x-coordinates come out shifted by a constant `D` depending on which direction the walk turned (visible once you group the output into alternating unshifted, `+D`, unshifted, `-D` runs).

Recovered `D` from consecutive values, then brute-forced small integer offsets around it, decoding `m = (x - offset)/20` for each candidate and checking which choice of `D` produced 72 characters that are all printable and form real English. One offset (`d=5`) produced a plaintext in `FortID{...}` form; every neighbouring offset produced near-misses with single scrambled characters, confirming it wasn't a fluke.

**Flag:** `FortID{1f_D0n_Knu7h_H4s_1t_0n_th3_W4ll_Th3n_17_Mu57_B3_G00d_F0r_Crypt0_r1ght?}`

---

## DESBox (crypto)

**Category:** Cryptography, a "DOS-shell" wrapper around 3DES

A fake MS-DOS shell lets you load two different 24-byte 3DES keys (`KEY A`, `KEY B`), encrypts the flag under Key A, then decrypts whatever ciphertext you send under Key B. Both keys get randomly byte-permuted and XORed against a hidden mask before use, and the two keys you submit are checked to be different in several ways: different raw bytes, different "effective" (parity-stripped) bytes, different scan-code multisets, and not exact bitwise complements of each other once permuted.

DES (and 3DES) has a classical complementation property: if you complement every bit of the key and every bit of the plaintext, you get the complement of the original ciphertext: `E(~K, ~P) = ~E(K, P)`. The server blocks keys that are complements *after* permutation, but the permutation is a random shuffle of byte positions applied independently to each key before the complement check even happens upstream of it being loaded. Submitting Key A as 23 zero bytes plus one byte, and Key B as the bitwise complement of Key A with that one odd byte moved to the front, means that after the random per-key permutation there's a real chance (1-in-24, since permutation is a bijection on positions) the two loaded keys land as exact complements anyway, since the pre-permutation complement relationship between bytes carries through.

When the loaded keys do turn out to be complements, feeding the complement of Key A's flag-ciphertext as the "debug ciphertext" to decrypt under Key B yields the complement of the flag itself, by the DES complementation identity. Looped the KEYB command until a lucky permutation landed (about 1-in-24 tries, so a handful of attempts), then XORed every output byte against `0xFF`.

**Flag:** `FortID{C0mp11m3n75_0n_3xPl0171n6_C0MP13m3nts!!1}`

---

## Bounty Hunter (misc / crypto)

**Category:** Miscellaneous, forging a polynomial-commitment proof

A fake "bug bounty transparency log" publishes a dataset as a committed vector over a large prime field, plus a verifier implementing something structurally identical to a FRI-style polynomial commitment scheme (repeated polynomial folding, with a commitment hash at each round, and a final check that the folded value matches a claimed value). The "vault policy" pays out if you can prove `payout_authorized(report_id) = 1` for a report ID you don't own, for one of a few report IDs you're told are "owned."

The verifier checks each fold round is internally consistent (the committed polynomial really does fold the way the proof claims) but never checks that the *first* round's committed polynomial is actually the real published data vector beyond round zero. Once round zero's commitment matches the real data, every later "next polynomial" in the proof is entirely prover-chosen, as long as each fold equation is self-consistent and the final folded value equals your claimed answer. That means you can start from the real vector, but freely invent every subsequent round's polynomial, backward-solving each round's `b` value from the fold equation so it forces the next round to need exactly your claimed output.

Built this proof for one of the policy's listed "owned" report IDs with `claimed_value = 1` for the `payout_authorized` field, picking each round's second evaluation point algebraically so the whole chain resolves to the forced claim, and it verified.

**Flag:** `FortID{Bu6_B0uN7y_H0uN73r5_Sh0u1d_B3_4Bl3_70_5p07_UN50UNd_Pr00fzZ}`

---

## Also solved (flags submitted and confirmed on the platform; notes not kept)

- **Fortune Teller** (pwn)
- **Baby Web** (web)
- **Baby Rev** (reverse engineering)

## Attempted, not solved

Sticker Booth (pwn, custom allocator), Do You Know Lika (web, Handlebars prototype pollution), Hacker Cinema (crypto/web, ECDSA ticket nonce derivation), Human RNG (crypto, QR code encrypted with a human-typed bitstream), Meteor Shower (reverse, NES ROM), and a long tail of OSINT, reverse, pwn and misc challenges not attempted before time ran out.

0xAnon (Wahid Khan) · FortID CTF 2026