Sunrisedotdev: Settlement Sale

Security Review

Cantina Managed review by:
Sujith Somraaj, Lead Security Researcher
Jay, Security Researcher

April 25, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3

3 Findings **4**
- 3.1 Medium Risk . . . 4
  - 3.1.1 Forced lockup from permit payload not enforced on-chain . . . 4
- 3.2 Informational . . . 4
  - 3.2.1 Missing validation on Ed25519 precompile instruction format . . . 4
  - 3.2.2 `bidder` unnecessarily marked mutable in `CancelOrReduce` and `ClaimRefund` . . . 5
  - 3.2.3 Unused error variant `InvalidEd25519Instruction` . . . 5
  - 3.2.4 Missing CPI guard on Ed25519 signature verification . . . 5
  - 3.2.5 Single-step authority transfer allows no recovery from erroneous transfers . . . 6
  - 3.2.6 Purchase permit signer cannot be rotated without contract upgrade . . . 6
  - 3.2.7 Function `validate_structure()` accepts zero-duration permit windows . . . 6
  - 3.2.8 Misleading error variant name `Ed25519MultipleSignatures` . . . 7
  - 3.2.9 Missing documentation for `minAmount` non-enforcement during commitment reduction . . . 7
  - 3.2.10 Function `set_allocation` emits `AllocationSet` events for no-op allocations . . . 7
  - 3.2.11 Function `cancel_bid` lacks a zero-amount guard . . . 8
  - 3.2.12 Role grants accept `Pubkey::default()` as an operator . . . 9

4 Appendix **10**
- 4.1 Test Coverage Improvements: `settlement_sale::place_bid()`, `cancel_bid()`, `process_refund()` . . . 10

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Base is a secure, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain.

From Apr 9th to Apr 17th the Cantina team conducted a review of svm-spearbit on commit hash e8dd91aa. The team identified a total of **13** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	0	0	0
Gas Optimizations	0	0	0
Informational	12	12	0
Total	**13**	**13**	**0**

2.1 Scope

The security review had the following components in scope for svm-spearbit on commit hash e8dd91aa:

```
programs/settlement-sale/src
├── ed25519.rs
├── errors.rs
├── lib.rs
└── state.rs
```

3 Findings

3.1 Medium Risk

3.1.1 Forced lockup from permit payload not enforced on-chain

Severity: Medium Risk

Context: lib.rs#L555

Description: In `place_bid()`, the `lockup` parameter is a free argument passed by the bidder. The only on-chain enforcement is the monotonic ratchet in `update_bid()` — once `lockup` is set to `true`, it cannot revert to `false`. However, nothing forces it to be `true` in the first place.

The `PurchasePermitV3` contains a `payload` field that is signed by the permit signer and, in the EVM implementation, is decoded to extract a `forcedLockup` flag that is enforced on-chain. In the Solana program, this `payload` is explicitly marked as opaque and never decoded:

```
/// Opaque payload bytes for off-chain use. Not decoded by the program.
pub payload: Vec<u8>,
```

The error variant `BidMustHaveLockup` is defined but never referenced in any instruction logic, suggesting enforcement was intended but not implemented. A bidder can submit `lockup: false` regardless of the permit signer's intent, silently bypassing the forced lockup requirement.

Recommendation: Decode the permit payload on-chain and enforce the `forced_lockup` flag, matching the EVM implementation's behavior:

```
if permit.forced_lockup && !lockup {
   return Err(SettlementSaleError::BidMustHaveLockup.into());
}
```

Alternatively, if the payload is intentionally kept opaque for the Solana program, consider adding `forced_lockup` as an explicit field on `PurchasePermitV3` so it is covered by the permit signature and enforceable on-chain.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix. A new Borsh-serialized `PurchasePermitPayload` struct was introduced with a `forced_lockup: bool` field (mirroring the EVM version), and a `decode_payload()` method was added to `PurchasePermitV3` that returns None for an empty payload or the decoded struct otherwise.

In `place_bid`, right after permit validation, the decoded payload is checked with `require!(!payload.forced_lockup || lockup, SettlementSaleError::BidMustHaveLockup)`, finally wiring the previously-orphaned error variant.

3.2 Informational

3.2.1 Missing validation on Ed25519 precompile instruction format

Severity: Informational

Context: ed25519.rs#L111

Description: `parse_ed25519_instruction` verifies `ix.program_id == ED25519_PROGRAM_ID` but does not verify that the instruction has zero accounts. The `Ed25519` precompile is stateless and should never have accounts attached. Additionally, `parse_ed25519_instruction_data` accepts `message_data_size == 0`, allowing an empty message to pass parsing. The precompile ignores extraneous accounts, and an empty parsed message would fail the downstream comparison against the serialized permit in `verify_ed25519_signature`, but adding these checks hardens the instruction validation and is consistent with best practices in other audited Solana programs.

Recommendation: Add an accounts check in `parse_ed25519_instruction` and an optional message size check in `parse_ed25519_instruction_data`:

```
require!(ix.accounts.is_empty(), SettlementSaleError::Ed25519MalformedData);

require!(message_data_size > 0, SettlementSaleError::Ed25519MalformedData);
```

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.2 `bidder` unnecessarily marked mutable in `CancelOrReduce` and `ClaimRefund`

Severity: Informational

Context: lib.rs#L1230, lib.rs#L1340

Description: The `bidder` account is marked `#[account(mut)]` in both `CancelOrReduce` and `ClaimRefund`, but it is never mutated. It is only used for PDA seed derivation, signer verification, and event emission via `.key()`. The token transfer goes from vault to `bidder_token_account` signed by the sale PDA, so no lamports touch the `bidder`'s system account. The unnecessary write lock slightly increases compute cost and reduces transaction parallelism. For comparison, `SetAllocation` and `ProcessRefund` correctly leave their signer non-mutable.

Recommendation: Remove mut from bidder in both structs.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.3 Unused error variant `InvalidEd25519Instruction`

Severity: Informational

Context: errors.rs#L7

Description: The error variant `InvalidEd25519Instruction` is defined in `errors.rs` but never referenced anywhere in the codebase. The Ed25519 verification logic uses more specific variants such as `Ed25519InstructionMissing`, `Ed25519WrongProgram`, and `Ed25519MalformedData` instead. This is dead code that adds unnecessary surface area to the error enum.

Recommendation: Remove the unused variant from `SettlementSaleError`.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.4 Missing CPI guard on Ed25519 signature verification

Severity: Informational

Context: ed25519.rs#L64-L65

Description: `load_current_index_checked` returns the top-level instruction index, not the index of the CPI callee. If `place_bid` is invoked via CPI from a malicious wrapping program, the Ed25519 signature check still passes because it resolves the instruction at `current_index - 1` to the top-level Ed25519 precompile instruction placed by the attacker. This allows the wrapping program to control the amount, price, and `lockup` parameters passed to `place_bid` within the valid permit bounds, on behalf of a victim who signs the malicious transaction. The victim must sign the transaction for the attack to succeed, and the attacker cannot exceed backend-authorized permit limits, bounding the financial impact.

Recommendation: Add a CPI guard to `verify_ed25519_signature` that confirms the current instruction belongs to this program. Something like:

```
let current_ix = load_instruction_at_checked(current_index as usize, instructions_sysvar)
    .map_err(|_| SettlementSaleError::Ed25519InstructionMissing)?;
require!(current_ix.program_id == crate::ID, SettlementSaleError::UnauthorizedCpi);
```

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.5 Single-step authority transfer allows no recovery from erroneous transfers

Severity: Informational

Context: lib.rs#L530

Description: The `transfer_authority()` function performs an immediate, single-step ownership transfer by directly overwriting the `sale.admin` field with the provided `new_authority` pubkey. There is no intermediate "pending" state and no acceptance step required from the incoming authority. Once the transaction confirms, the old `admin` irrevocably loses all administrative privileges.

If an admin mistakenly passes an incorrect pubkey: a typo, a wrong copy-paste, or the address of a keypair whose private key is lost, administrative control of the sale is permanently and irrecoverably transferred.

This is a well-known footgun pattern in both EVM and Solana programs. While operational security (e.g., using multisigs, careful key management) is ultimately the strongest mitigation, a defensive two-step pattern at the protocol level provides an additional safety net at low complexity cost.

Recommendation: Implement a two-step (propose/accept) authority transfer pattern:

1. `propose_authority(new_authority)` called by the current `admin`; stores the proposed new authority in a `pending_admin` field on the `SettlementSale` account without modifying the active `admin`.
2. `accept_authority()` called by the proposed new authority; promotes `pending_admin` to `admin` and clears the pending field.

This guarantees the receiving address is controlled by an entity capable of signing, eliminating the class of bugs where authority is transferred to an unreachable key.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.6 Purchase permit signer cannot be rotated without contract upgrade

Severity: Informational

Context: lib.rs#L280

Description: The `permit_signer` is set once during `initialize_sale` and stored on the `SettlementSale` account. There is no instruction to update it. If the signing key is compromised or needs routine rotation, a program upgrade is required.

Recommendation: Consider documenting this behavior as a deliberate design choice and prepare necessary playbooks beforehand to deal with an emergency situation where the `permit_signer` might be required to be changed.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix. A new "Permit Signer Key Management" section to `ARCHITECTURE.md` that documents the non-rotatability as a deliberate design choice and provides a four-step compromise-response playbook. No code changes were made.

3.2.7 Function `validate_structure()` accepts zero-duration permit windows

Severity: Informational

Context: lib.rs#L180

Description: Function `validate_structure()` uses a `<=` check for the time window, allowing `opens_at == closes_at`. Such a permit passes structural validation but can never be used. `validate_against_sale()` requires `current_time >= opens_at` AND `current_time < closes_at`, which is unsatisfiable when the two values are equal.

A permit signer mistake issuing such a permit would silently block the affected wallet from bidding, with no on-chain error until the bid is actually attempted.

Recommendation: Consider using a strict `<` check (`opens_at < closes_at`) in `validate_structure()` to reject zero-duration windows early, surfacing the permit signer error at validation time rather than at bid time.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.8 Misleading error variant name `Ed25519MultipleSignatures`

Severity: Informational

Context: ed25519.rs#L120

Description: The error variant `Ed25519MultipleSignatures` fires for any `num_signatures != 1`, including `num_signatures == 0`. The error message ("must contain exactly one signature") correctly describes the constraint, but the variant name implies only the `>1` case. This can be confusing when debugging a zero-signature scenario, as the error name suggests multiple signatures were found.

Recommendation: Consider renaming the variant to `Ed25519InvalidSignatureCount` to accurately reflect that it covers both zero and multiple signature cases.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.9 Missing documentation for `minAmount` non-enforcement during commitment reduction

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the Solana program, `reduce_commitment` and `cancel_bid` allow entities to reduce their committed amount below the `min_amount` threshold originally enforced by the purchase permit at bid submission time.

This is intentional: `min_amount` is a bid-time constraint, not an ongoing floor. However, unlike the EVM implementation which explicitly documents this behavior ("No minimum floor is enforced on the resulting commitment. The minAmount constraint from the purchase permit applies only at bid submission and entities may reduce below that threshold here."), the Solana program has no equivalent documentation.

Recommendation: Add inline documentation clarifying that no minimum floor is enforced on the resulting commitment and that the `min_amount` constraint from the purchase permit applies only at bid submission.

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 1.

Cantina Managed: Verified fix.

3.2.10 Function `set_allocation` emits `AllocationSet` events for no-op allocations

Severity: Informational

Context: lib.rs#L701

Description: The `set_allocation` function emits an `AllocationSet` event even when there is no effective state change. This occurs in two scenarios:

1. When both `prev_accepted` and `accepted_amount` are zero. In this case, the `if prev_accepted > 0` branch is skipped, and calling `add_accepted(0)` has no effect on `wallet_binding.accepted_amount`, `entity_state.accepted_amount`, or `sale.total_accepted`.
2. When `prev_accepted` equals `accepted_amount` (both non-zero, with `allow_overwrite` = `true`). Here, `sub_accepted(X)` followed by `add_accepted(X)` results in no net change across the same state variables.

In both situations, an `AllocationSet { prev_accepted, accepted_amount, … }` event is still emitted despite no observable on-chain change. While this does not impact funds or break accounting invariants, it

can pollute the event stream and potentially mislead off-chain systems that rely on event diffs or treat these events as indicators of meaningful allocation changes.

Recommendation: Add an early return before the emit when the call is a no-op:

```
if prev_accepted == accepted_amount {
   return Ok(());
}
```

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 2.

Cantina Managed: Verified fix.

3.2.11 Function `cancel_bid` lacks a zero-amount guard

Severity: Informational

Context: lib.rs#L641

Description: When a wallet calls `cancel_bid`, the handler reads their refundable balance and passes it straight into the shared reduction helper without first checking that the amount is actually greater than zero.

```
pub fn cancel_bid(ctx: Context<CancelOrReduce>) -> Result<()> {
  let amount = ctx.accounts.wallet_binding.refund_amount()?;
  apply_reduction(..., amount)?;
//...
}
```

Inside `apply_reduction`, a zero amount takes a special shortcut: it flips the wallet's `refunded` flag to `true` and returns early, skipping the token transfer, the entity-state update, and the usual accounting adjustments.

```
if amount == 0 {
   binding.refunded = true;
   return Ok(());
}
```

In practice this branch is unreachable today. For `refund_amount()` to be zero during Cancellation, the binding's `contributed_amount` would have to be zero and the only way that happens is via a previous full cancellation, which would have already set `refunded = true` and caused the upstream `require!(!binding.refunded)` check to reject the call before it ever reaches the shortcut.

Still, it's a behavioral outlier worth cleaning up:

- The EVM contract never allows this path. `cancelBid` silently skips (wallet, token) pairs whose refundable amount is zero, and `_reduceCommitment` explicitly reverts with `ZeroAmount` if called with zero.
- The sibling Solana instruction `reduce_commitment` already does the right thing and rejects zero with `require!(amount > 0, ZeroAmount)` at the top of the handler.

So `cancel_bid` is the only place, across either implementation, where the `refunded` flag can in theory be flipped without any corresponding transfer or state change.

Recommendation: Add the same zero-amount guard that `reduce_commitment` already uses, so `cancel_bid` becomes consistent with both its sibling and the EVM reference:

```
  pub fn cancel_bid(ctx: Context<CancelOrReduce>) -> Result<()> {
    let amount = ctx.accounts.wallet_binding.refund_amount()?;
+   require!(amount > 0, SettlementSaleError::ZeroAmount);
    apply_reduction(..., amount)?;
  //...
  }
```

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 2.

Cantina Managed: Verified fix.

3.2.12 Role grants accept `Pubkey::default()` as an operator

Severity: Informational

Context: lib.rs#L323, lib.rs#L337

Description: Function `grant_role` doesn't check that the `operator` argument is non-zero. An admin passing `Pubkey::default()` creates a `RoleBinding` PDA that no one can ever exercise (no signer can satisfy `authority.key() == Pubkey::default()`), leaving an orphaned account consuming rent until the admin manually revokes it.

This is inconsistent with the rest of the program, which rejects zero pubkeys at every other input boundary: `initialize_sale` (`ZeroPermitSigner`, `ZeroProceedsReceiver`), `set_proceeds_receiver` (`ZeroProceedsReceiver`), and `transfer_authority` (`ZeroAuthority`).

Recommendation: Add the same zero-pubkey guard used elsewhere to `grant_role` instruction:

```
require!(operator != Pubkey::default(), SettlementSaleError::ZeroAuthority);
```

Coinbase: Fixed in sunrisedotdev/svm-spearbit PR 2.

Cantina Managed: Verified fix. PR 2 adds a zero-pubkey guard in `grant_role` via `require!(operator != Pubkey::default(), SettlementSaleError::ZeroAuthority)`. As Coinbase noted, since a `RoleBinding` PDA seeded with `Pubkey::default()` cannot be created in the first place due to seed constraints, the guard in `revoke_role` can never fire — but the check in `grant_role` makes the intent explicit and consistent with every other input boundary in the program.

4 Appendix

4.1 Test Coverage Improvements: `settlement_sale::place_bid()`, `cancel_bid()`, `process_refund()`

Overview

The existing test suites for `place_bid`, `cancel_bid`, and `process_refund` in `programs/settlement-sale` covered the main happy paths and most stage/pause/refunded error cases, but had several gaps:

1. No test pinning `max_wallets_per_entity` as an inclusive upper bound
2. No test for the zero-delta (price-only) re-bid code path.
3. No regression guards on `wallet_count` / `sale.total_committed` across multi-wallet bid sequences.
4. No test that `entity.refunded` flips to `true` after every wallet on the entity cancels.
5. No test that cancelling after a same-wallet re-bid refunds the full accumulated `contributed_amount`.
6. No test pinning the intentional asymmetry where `process_refund` succeeds while the sale is paused (unlike `claim_refund`).

This appendix documents the tests added to close those gaps.

`test_place_bid.rs : place_bid()`

Test	Details	Status
`accepts_bids_up_to_max_wallets_per_entity`	Sale initialized with `max_wallets_per_entity = 2`; two distinct wallets bid on the same entity; `entity.wallet_count` reaches the configured maximum of 2 (inclusive upper bound, no off-by-one)	✓
`places_bid_with_same_amount_higher_price_transfers_nothing`	Re-bid with same `amount` and higher `price` updates `entity.current_price` only; `delta = 0` path skips `transfer_to_vault`, `sale.add_committed`, and `wallet_binding.add_contribution`; vault balance, bidder balance, `contributed_amount`, and `total_committed` all unchanged	✓
`test_place_bid` (strengthened)	Inline `entity.wallet_count` and `sale.total_committed` assertions added after each of the three sequential bids; pins that `wallet_count` does not re-increment on re-bid from an already-bound wallet and that `sale.add_committed(delta)` runs on every non-zero-delta bid	✓

test_cancel_bid.rs : cancel_bid()

Test	Details	Status
cancel_bid_marks_entity_refunded_when_all_wallets_cancel	Two wallets bid on the same entity; after the first cancels, entity.refunded stays false and entity.current_amount > 0; after the second also cancels, entity.refunded = true, entity.current_amount = 0, sale.total_committed = 0, sale.total_cancelled = 2 * BID_AMOUNT	✓
cancel_bid_after_rebid_refunds_total_contributed_amount	Same wallet places BID_AMOUNT, then re-bids 2 * BID_AMOUNT; cancel must refund the full accumulated contributed_amount = 2 * BID_AMOUNT, zero out contributed_amount, and set cancelled_amount = sale.total_cancelled = 2 * BID_AMOUNT	✓

test_process_refund.rs : process_refund()

Test	Details	Status
process_refund_succeeds_while_paused	Sale is paused in Done stage; authority-initiated process_refund still succeeds, transfers the correct delta (BID_AMOUNT - ACCEPTED_AMOUNT) from vault to the wallet's token account, marks binding.refunded = true, and updates sale.total_refunded; pins the intentional asymmetry with claim_refund (which is paused-gated)	✓

Test Code

test_place_bid.rs : accepts_bids_up_to_max_wallets_per_entity

```
/// max_wallets_per_entity is an inclusive upper bound: wallet_count may reach
/// the configured maximum.
#[tokio::test]
async fn accepts_bids_up_to_max_wallets_per_entity() {
    let sale_uuid = [36u8; 16];
    let entity_id = [42u8; 16];
    let mut ctx = context::setup(&sale_uuid).await;
    ctx.initialize_sale(2, true, false, true, ctx.authority.insecure_clone().pubkey())
        .await;

    let (entity_pda, _) = pda::derive_entity_state_pda(&ctx.sale_pda, &entity_id,
    ↪  &ctx.program_id);

    let bidder = ctx.bidder.insecure_clone();
    let bidder_ta = ctx.bidder_token_account;
    let (wb_pda, _) = pda::derive_wallet_binding_pda(&ctx.sale_pda, &bidder.pubkey(),
    ↪  &ctx.program_id);
    ctx.place_bid(&bidder, &bidder_ta, &entity_id, &entity_pda, &wb_pda, BID_AMOUNT,
    ↪  BID_PRICE, false)
        .await
        .unwrap();

    let entity = ctx.fetch_entity(entity_pda).await;
```

```
    assert_eq!(entity.wallet_count, 1);

    let bidder2 = ctx.bidder2.insecure_clone();
    let bidder2_ta = ctx.bidder2_token_account;
    let (wb2_pda, _) = pda::derive_wallet_binding_pda(&ctx.sale_pda, &bidder2.pubkey(),
    ↪  &ctx.program_id);
    ctx.place_bid(
        &bidder2,
        &bidder2_ta,
        &entity_id,
        &entity_pda,
        &wb2_pda,
        BID_AMOUNT * 2,
        BID_PRICE,
        false,
    )
    .await
    .unwrap();

    let entity = ctx.fetch_entity(entity_pda).await;
    assert_eq!(entity.wallet_count, 2);
    assert_eq!(entity.current_amount, BID_AMOUNT * 2);
}
```

test_place_bid.rs : places_bid_with_same_amount_higher_price_transfers_nothing

```
/// Re-bidding with the same amount but a higher price must update the price
/// without transferring tokens (delta = 0 path skips add_committed /
/// add_contribution / transfer_to_vault).
#[tokio::test]
async fn places_bid_with_same_amount_higher_price_transfers_nothing() {
    let sale_uuid = [37u8; 16];
    let entity_id = [42u8; 16];
    let mut ctx = context::setup(&sale_uuid).await;
    ctx.initialize_default_sale(5).await;

    let (entity_pda, _) = pda::derive_entity_state_pda(&ctx.sale_pda, &entity_id,
    ↪  &ctx.program_id);
    let bidder = ctx.bidder.insecure_clone();
    let bidder_ta = ctx.bidder_token_account;
    let (wb_pda, _) = pda::derive_wallet_binding_pda(&ctx.sale_pda, &bidder.pubkey(),
    ↪  &ctx.program_id);

    ctx.place_bid(&bidder, &bidder_ta, &entity_id, &entity_pda, &wb_pda, BID_AMOUNT,
    ↪  BID_PRICE, false)
        .await
        .unwrap();

    let bidder_ta_before = ctx.token_balance(bidder_ta).await;
    let vault_before = ctx.token_balance(ctx.vault_pda).await;
    let sale_before = ctx.fetch_sale().await;
    let wb_before = ctx.fetch_wallet_binding(wb_pda).await;

    ctx.place_bid(&bidder, &bidder_ta, &entity_id, &entity_pda, &wb_pda, BID_AMOUNT,
    ↪  BID_PRICE + 50, false)
        .await
        .unwrap();

    let entity = ctx.fetch_entity(entity_pda).await;
    assert_eq!(entity.current_amount, BID_AMOUNT);
    assert_eq!(entity.current_price, BID_PRICE + 50);

    let wb_after = ctx.fetch_wallet_binding(wb_pda).await;
    assert_eq!(wb_after.contributed_amount, wb_before.contributed_amount);

    let sale_after = ctx.fetch_sale().await;
```

```
    assert_eq!(sale_after.total_committed, sale_before.total_committed);

    assert_eq!(ctx.token_balance(bidder_ta).await, bidder_ta_before);
    assert_eq!(ctx.token_balance(ctx.vault_pda).await, vault_before);
}
```

`test_place_bid.rs : test_place_bid` (strengthened assertions).

```
// Section 1 (after first bid by `bidder`):
assert_eq!(entity.wallet_count, 1);
assert_eq!(sale.total_committed, BID_AMOUNT);

// Section 2 (after second distinct wallet `bidder2` bids):
assert_eq!(entity.wallet_count, 2);
assert_eq!(sale.total_committed, 20_000_000);

// Section 4 (after `bidder` re-bids with a higher amount):
// Re-bid by an already-bound wallet must NOT re-increment wallet_count.
assert_eq!(entity.wallet_count, 2);
assert_eq!(sale.total_committed, 25_000_000);
```

test_cancel_bid.rs : cancel_bid_marks_entity_refunded_when_all_wallets_cancel

```
/// After every wallet on an entity cancels, `entity.refunded` must flip to true
/// (second half of the two-wallet case; existing test only covers the first
/// cancel where one binding still remains).
#[tokio::test]
async fn cancel_bid_marks_entity_refunded_when_all_wallets_cancel() {
    let (mut ctx, _, entity_pda, wb_pda, wb2_pda) = setup_with_two_bids().await;
    let bidder = ctx.bidder.insecure_clone();
    let bidder_ta = ctx.bidder_token_account;
    let bidder2 = ctx.bidder2.insecure_clone();
    let bidder2_ta = ctx.bidder2_token_account;

    ctx.cancel_bid(&bidder, &bidder_ta, &entity_pda, &wb_pda).await.unwrap();

    let entity = ctx.fetch_entity(entity_pda).await;
    assert!(!entity.refunded);
    assert_eq!(entity.current_amount, BID_AMOUNT);

    ctx.cancel_bid(&bidder2, &bidder2_ta, &entity_pda, &wb2_pda)
        .await
        .unwrap();

    let entity = ctx.fetch_entity(entity_pda).await;
    assert!(entity.refunded);
    assert_eq!(entity.current_amount, 0);

    let sale = ctx.fetch_sale().await;
    assert_eq!(sale.total_committed, 0);
    assert_eq!(sale.total_cancelled, BID_AMOUNT * 2);
}
```

test_cancel_bid.rs : cancel_bid_after_rebid_refunds_total_contributed_amount

```
/// Cancelling after a same-wallet re-bid must refund the full latest
/// `contributed_amount` (pins monotonic accumulation through re-bids).
#[tokio::test]
async fn cancel_bid_after_rebid_refunds_total_contributed_amount() {
    let sale_uuid = [4u8; 16];
    let mut ctx = context::setup(&sale_uuid).await;
    ctx.initialize_default_sale(3).await;

    let entity_id = [42u8; 16];
    let (entity_pda, _) = pda::derive_entity_state_pda(&ctx.sale_pda, &entity_id,
    ↪  &ctx.program_id);
```

```
    let bidder = ctx.bidder.insecure_clone();
    let bidder_ta = ctx.bidder_token_account;
    let (wb_pda, _) = pda::derive_wallet_binding_pda(&ctx.sale_pda, &bidder.pubkey(),
    ↪  &ctx.program_id);

    ctx.place_bid(
        &bidder,
        &bidder_ta,
        &entity_id,
        &entity_pda,
        &wb_pda,
        BID_AMOUNT,
        BID_PRICE,
        false,
    )
    .await
    .unwrap();
    ctx.place_bid(
        &bidder,
        &bidder_ta,
        &entity_id,
        &entity_pda,
        &wb_pda,
        BID_AMOUNT * 2,
        BID_PRICE,
        false,
    )
    .await
    .unwrap();

    let binding_before = ctx.fetch_wallet_binding(wb_pda).await;
    assert_eq!(binding_before.contributed_amount, BID_AMOUNT * 2);

    ctx.open_cancellation().await;

    let balance_before = ctx.token_balance(bidder_ta).await;
    let vault_before = ctx.token_balance(ctx.vault_pda).await;

    ctx.cancel_bid(&bidder, &bidder_ta, &entity_pda, &wb_pda)
        .await
        .unwrap();

    let balance_after = ctx.token_balance(bidder_ta).await;
    let vault_after = ctx.token_balance(ctx.vault_pda).await;
    assert_eq!(balance_after - balance_before, BID_AMOUNT * 2);
    assert_eq!(vault_before - vault_after, BID_AMOUNT * 2);

    let binding = ctx.fetch_wallet_binding(wb_pda).await;
    assert!(binding.refunded);
    assert_eq!(binding.contributed_amount, 0);
    assert_eq!(binding.cancelled_amount, BID_AMOUNT * 2);

    let sale = ctx.fetch_sale().await;
    assert_eq!(sale.total_committed, 0);
    assert_eq!(sale.total_cancelled, BID_AMOUNT * 2);
}
```

test_process_refund.rs : process_refund_succeeds_while_paused

```
/// `process_refund` must still succeed when the sale is paused — it is an
/// authority-driven refund path and intentionally has no `require_not_paused`
/// gate (asymmetric with `claim_refund`, which IS paused-gated).
#[tokio::test]
async fn process_refund_succeeds_while_paused() {
    let (mut ctx, _, entity_pda, wb_pda) = setup_in_done().await;
    let bidder = ctx.bidder.insecure_clone();
```

```
    let bidder_ta = ctx.bidder_token_account;

    let authority = ctx.authority.insecure_clone();
    let pause_ix = instructions::set_paused(&ctx.program_id, &authority.pubkey(),
    ↪  &ctx.sale_pda, true);
    ctx.send_tx_with_signer(&[pause_ix], &authority).await.unwrap();
    assert!(ctx.fetch_sale().await.is_paused);

    let balance_before = ctx.token_balance(bidder_ta).await;
    let vault_before = ctx.token_balance(ctx.vault_pda).await;

    ctx.process_refund(&entity_pda, &wb_pda, &bidder.pubkey(), &bidder_ta, false)
        .await
        .unwrap();

    let balance_after = ctx.token_balance(bidder_ta).await;
    let vault_after = ctx.token_balance(ctx.vault_pda).await;
    assert_eq!(balance_after - balance_before, BID_AMOUNT - ACCEPTED_AMOUNT);
    assert_eq!(vault_before - vault_after, BID_AMOUNT - ACCEPTED_AMOUNT);

    let binding = ctx.fetch_wallet_binding(wb_pda).await;
    assert!(binding.refunded);

    let sale = ctx.fetch_sale().await;
    assert_eq!(sale.total_refunded, BID_AMOUNT - ACCEPTED_AMOUNT);
    assert!(sale.is_paused);
}
```