Pashov Audit Group

Thrust Security Review

September 1st 2025 - September 7th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Thrust

Thrust consists of two components: a Rust-based presale contract system and a token staking platform. It allows users to participate in early token sales where purchases, lockups, and gradual releases are managed automatically, and to stake and withdraw tokens under specific rules.

5. Executive Summary

A time-boxed security review of the **thrustcom/thrust-staking** and **thrustcom/thrust-presale** repositories was done by Pashov Audit Group, during which **ctrus**, **0xAlix2**, **Johny**, **0xbrivan**, **jesjupyter** engaged to review **Thrust**. A total of **17** issues were uncovered.

Protocol Summary

Project Name	Thrust
Protocol Type	Token Presale Platform
Timeline	September 1st 2025 - September 7th 2025

Review commit hashes:

- fdd4b0bfab5f8c078801c37462ac5f078d132b85 (thrustcom/thrust-staking)
- 1801151b42eac6e7103120ca76318dcc426c4f6e (thrustcom/thrust-presale)

Fixes review commit hashes:

- b325c03dd8034c8183bf72b55807c3dc6e0773d1 (thrustcom/thrust-staking)
- 2f7a04f0ccebf858852b71975eeb02647f4063ad (thrustcom/thrust-presale)

Scope

`mod.rs` `stake.rs` `unstake.rs` `update_global_admins.rs` `update_global_whitelist.rs` `withdraw.rs` `global_state.rs` `stake_state.rs` `constants.rs` `errors.rs` `events.rs` `lib.rs` `claim.rs` `create_presale.rs` `deposit.rs` `refund.rs` `update_global_state.rs` `update_presale_whitelist.rs` `withdraw.rs` `presale_state.rs` `user_deposit_state.rs`

6. Findings

Findings count

Severity	Amount
Critical	3
High	1
Medium	5
Low	8
Total findings	**17**

Summary of findings

ID	Title	Severity	Status
[C-01]	User prevents token account closure by donating tokens	Critical	**Resolved**
[C-02]	Token amount calculation ignores mint decimals, massively under-credits buyers	Critical	**Resolved**
[C-03]	External token transfers to user deposit ATA cause permanent claim DoS	Critical	**Resolved**
[H-01]	Immediate withdrawal possible without `unstaked_at` check	High	**Resolved**
[M-01]	Allowing withdraw before presale officially closes can lock all users' token claims	Medium	**Resolved**
[M-02]	DoS when creating a new presale	Medium	**Resolved**
[M-03]	Vesting schedule truncates final partial hour, enabling early full claims	Medium	**Resolved**
[M-04]	Incorrect presale parameter validation allows invalid configurations	Medium	**Resolved**
[M-05]	Whitelist leaf is not user-scoped	Medium	Acknowledged
[L-01]	`token_collected` reports unsold not sold tokens in Withdrawn event	Low	**Resolved**
[L-02]	`Deposited` event overstates totals due to double addition	Low	**Resolved**

ID	Title	Severity	Status
[L-03]	Event field mismatch in `UpdateGlobalState`	Low	**Resolved**
[L-04]	Minor error variant mismatch	Low	**Resolved**
[L-05]	Early withdraw causes DoS for later deposits	Low	**Resolved**
[L-06]	Equality check on ATA balance enables refund DoS	Low	**Resolved**
[L-07]	Boundary flaw enables deposit and refund at presale close	Low	**Resolved**
[L-08]	Users can inflate unstake event amounts to exploit Off-Chain rewards	Low	**Resolved**

Critical findings

[C-01] User prevents token account closure by donating tokens

Severity

Impact: High

Likelihood: High

Description

In the `claim` instruction, after a user has claimed all their tokens, the program attempts to close the user's deposit token account.

```
        if claimed_total >= self.user_deposit_state.tokens_bought {
            //TODO maybe not required
            require!(
                self.user_deposit_state_token_ata.amount == claimable_token_amount,
                PresaleError::InvalidDepositState
            );
            //@audit dos by transferring 1/2 tokens; preventing ta closure
            //TODO close user deposit token ata -- done
            token::close_account(CpiContext::new_with_signer(
                  //closure
            ))?;

            //TODO close user deposit state account -- done
            self.user_deposit_state.close(self.user.to_account_info())?;
        }
```

If any external user transfers tokens into this account(`user_deposit_state_token_ata`), the account cannot be closed as it still holds presale mints. This results in a permanent lockup of tokens, as only the program (via PDA) can move tokens out, but no mechanism exists to recover or sweep these tokens. The user loses access to their funds and cannot reclaim rent.

eg. user had bought 50 tokens & has 50 tokens into his `user_deposit_state_token_ata` , claimable_token_amount is equal to tokens bought when vesting has ended and user has not claimed before, we transfer one token into this account, the token balance becomes 51 units, the program only transfers `tokens_bought` , not the total token amount before proceeding for account closure

```
      token::transfer(
            CpiContext::new_with_signer(
         //
            -> claimable_token_amount,
        )?;
```

So program would just transfer 50 tokens, the token account would still have 1 token left in it, and hence it can't be closed because token accounts can only be closed when their token balance is 0. So the account can never be closed, and hence users could never claim their tokens.

Here is why it's different from issues that have the same attack vector(donation of tokens) 1. even if we fix the strict equality by losing it as:

```
        require!(
              self.user_deposit_state_token_ata.amount >= claimable_token_amount,
              PresaleError::InvalidDepositState
          );
```

It does not prevent from this issue, as 51>50 would pass, but account closure would still fail. The intent behind this issue is not to break strict equality by donating tokens, but to break account closure, and fixing strict equality won't prevent from this issue unless we implement the suggestion that I made.

Recommendations

1. Before entering the if block, reload user's `user_deposit_state_token_ata`.
2. If there are any tokens, sweep those to admin's presale mint token account.
3. Then move forward to close the token account.

[C-02] Token amount calculation ignores mint decimals, massively under-credits buyers

Severity

Impact: High

Likelihood: High

Description

When users deposit base tokens, they get presale tokens according to a certain price, `price_per_token_in_base` is an integer price **in base mint smallest units** (e.g., USDC has 6 decimals, so 0.1 USDC = `100_000`). When transferring tokens, they should be transferred using their raw decimals. If the presale token has 6 decimals, then **1 token = 1_000_000 raw units**.

During deposit, the program computes how many sale tokens to give the user:

```
let token_bought = tokens_for_base(
    deposit_amount_in_base,
    self.presale_state.price_per_token_in_base,
);
```

But the helper is:

```
fn tokens_for_base(base_lamports: u64, price_per_token_in_base: u64) -> u64 {
    if price_per_token_in_base == 0 {
        return 0;
    }
    base_lamports / price_per_token_in_base
}
```

This uses **integer division only** and **does not scale** by the **presale token's decimals**. The result is a *human count of tokens* (e.g., `31`) rather than **raw mint units** (e.g., `31_000_000` for a 6-decimals mint). The code then transfers `token_bought` directly via `token::transfer(..., token_bought)`, sending only a tiny fraction of the intended amount.

Example

- Base deposit: `3_100_000` (3.1 USDC if base has 6 decimals).
- Price: `1_000` (0.001 USDC per token).
- Presale token decimals: `6`.

Expected raw units:

```
(3_100_000 * 10^6) / 1_000 = 3_100_000_000  // 3100.000000 tokens
```

Current code returns:

```
3_100_000 / 1_000 = 3_100  // raw units (!!) = 0.003100 tokens
```

This forces buyers to receive **far fewer tokens** than promised by the price.

Proof of Concept

```
it("Wrong token decimals", async () => {
    await setBlockchainTime(
        banksContext,
        BigInt(currentPresaleState4.presaleOpensAt.toNumber())
    );

    const data = {
        depositAmountInBase: new BN(3.1 * LAMPORTS_PER_TOKEN),
        merkleLeafIndex,
        merkleMaxTotalDeposit: merkleLeaf.max_total_deposit,
        merkleProof: proofArray,
    };

    await program.methods
        .deposit(
            data.depositAmountInBase,
            data.merkleLeafIndex,
            data.merkleMaxTotalDeposit,
            data.merkleProof
        )
        .accounts({
            user: user1Kp.publicKey,
            tokenMint: tokenMint4Kp.publicKey,
            baseMint: baseTokenMintKp.publicKey,
        })
        .signers([user1Kp])
        .rpc();

    const userDepositAccount = await program.account.userDepositState.fetch(
        PublicKey.findProgramAddressSync(
            [
                Buffer.from("user_deposit_state"),
                PublicKey.findProgramAddressSync(
                    [
                        Buffer.from("presale_state"),
                        currentPresaleState4.baseMint.toBuffer(),
                        currentPresaleState4.tokenMint.toBuffer(),
                    ],
                    program.programId
                )[0].toBuffer(),
                user1Kp.publicKey.toBuffer(),
            ],
            program.programId
        )[0]
    );

    expect(userDepositAccount.baseSpent.toString()).to.equal("3100000");
    expect(userDepositAccount.tokensBought.toString()).to.equal("3100");
});
```

Recommendations

Consider scaling by token mint decimals when computing `token_bought` so the function returns **raw mint units** suitable for `token::transfer`.

[C-03] External token transfers to user deposit ATA cause permanent claim DoS

Severity

Impact: High

Likelihood: High

Description

The `claim` function strictly validates that the ATA balance matches expected tokens bought:

```
require!(
    self.user_deposit_state_token_ata
        .amount
        .saturating_add(self.user_deposit_state.tokens_claimed)
        == self.user_deposit_state.tokens_bought,
    PresaleError::InvalidDepositState
);
```

However, this can be Dosed by sending minimum amount to break the check.

Attack scenario: 1. User deposits and receives tokens in their deposit ATA. 2. Attacker sends 1 lamport (or any amount) to the user's deposit ATA. 3. The balance validation fails: `ata_amount + tokens_claimed != tokens_bought`. 4. User can never claim their tokens, even after vesting completes. 5. The account cannot be closed due to the balance mismatch.

This is a permanent DoS attack that costs minimal funds (`1 lamport`) but completely blocks legitimate users from accessing their tokens.

This also works for `refund` (but less likely, since attacker may not get the `token` to dos):

```
        require!(
            self.user_deposit_state_token_ata.amount == self.user_deposit_state.tokens_bought,
            PresaleError::InvalidDepositState
        );
```

The close operation can be dosed.

```
        //TODO close user deposit token ata -- done
        token::close_account(CpiContext::new_with_signer(
            self.token_program.to_account_info(),
            token::CloseAccount {
                account: self.user_deposit_state_token_ata.to_account_info(),
                authority: self.user_deposit_state.to_account_info(),
                destination: self.user.to_account_info(),
            },
            &[&UserDepositState::get_signer(
                &self.presale_state.key(),
                &self.user.key(),
                &user_deposit_bump,
            )],
        ))?;
```

Recommendations

Avoid using strict balance comparison.

High findings

[H-01] Immediate withdrawal possible without `unstaked_at` check

Severity

Impact: High

Likelihood: Medium

Description

When users stake their tokens, one of the parameters encoded in the Merkle proof is `merkle_lockup_in_minutes`, which specifies the minimum time a user must wait after initiating an unstake before withdrawing. In `unstake`, the program records the initiation time and computes the unlock time:

```
pub fn unstake(
    &mut self,
    account_seed: u8,
    merkle_leaf_index: u32,
    merkle_lockup_in_minutes: u32,
    merkle_minimum_stake_amount: u64,
    merkle_proof: Vec<u8>,
) -> Result<()> {
    // --SNIP

    let now = Clock::get()?.unix_timestamp;
    self.stake_state.unstaked_at = now;
    self.stake_state.unlocks_at = now.saturating_add((merkle_lockup_in_minutes as
i64).mul(MINUTE));
}
```

However, `withdraw` only checks that the current timestamp is greater than or equal to `unlocks_at` and does not verify that an unstake was ever initiated (i.e., `unstaked_at` set). Because the default value of `unlocks_at` is 0, a user can call `withdraw` immediately after staking-without calling `unstake`-and pass the time check, enabling an immediate withdrawal that bypasses the lockup period.

```
pub fn withdraw(&mut self, stake_state_bump: u8, account_seed: u8) -> Result<()> {
    // --SNIP
@==>    require!(now >= self.stake_state.unlocks_at,StakingError::LockupNotEnded);
    token::transfer(
        CpiContext::new_with_signer(
            self.token_program.to_account_info(),
            token::Transfer {
                from: self.stake_state_token_ata.to_account_info(),
                authority: self.stake_state.to_account_info(),
                to: self.user_token_ata.to_account_info(), // needs to be
self.user_deposit_token_ata
```

```
            },
            &[&StakeState::get_signer(
                &self.token_mint.key(),
                &self.user.key(),
                &account_seed,
                &stake_state_bump,
            )],
        ),
        self.stake_state_token_ata.amount,
    )?;
}
```

Recommendations

Consider checking in `withdraw` that an `unstake` was initiated by checking `unstaked_at`.

Medium findings

[M-01] Allowing withdraw before presale officially closes can lock all users' token claims

Severity

Impact: Medium

Likelihood: Medium

Description

In the `withdraw` process, the following check is commented out:

```
// require!(
//     now >= self.presale_state.presale_closes_at,
//     PresaleError::NotEnded
// );
```

Instead, only the minimum base raised check is enforced:

```
require!(
    self.presale_state.base_raised_balance >= self.presale_state.base_min_target_balance,
    PresaleError::NotSuccessful
);
```

This allows the admin to call `withdraw` and start vesting(potentially) **before the presale has officially ended.** The withdraw logic closes the presale's token/base ATAs and sets:

```
self.presale_state.base_raised_balance = 0;
self.presale_state.token_balance = 0;
```

Now consider a scenario when: - The target amount was raised so quickly that admin called withdraw(presale is still on) - vesting started(presale is still not closed officially) - one user claims some of his vested tokens(presale is still not closed officially) - This user recreates the presale's token/base ATAs & he transfers these few tokens to presale's token ata

Now he deposits a small amount(such that he gets just the amount of presale mints he deposited in presale mint's token ata), Since the presale is still open, the following happens: - `self.presale_state.base_raised_balance` is set to a very low value (from the new deposit). - `self.presale_state.end_base_balance` is set to this low value, which is obviously lower than `self.presale_state.base_min_target_balance`

Now, when any genuine users try to claim their tokens via `claim`, the following check fails:

```
require!(
    self.presale_state.end_base_balance >= self.presale_state.base_min_target_balance,
    PresaleError::NotSuccessful
);
```

This blocks all claims, **permanently locking all user tokens** in the vesting process.

Recommendations

- **Enforce presale end before allowing withdraw:**
  Uncomment and require the presale to be closed before allowing withdraw: `rust require!( now >= self.presale_state.presale_closes_at, PresaleError::NotEnded );`

[M-02] DoS when creating a new presale

Severity

Impact: Medium

Likelihood: Medium

Description

The `CreatePresale` instruction initializes the presale state PDA and its associated token accounts:

- `presale_token_ata` (holding the presale's tokens for sale).
- `presale_base_ata` (holding the raised base tokens, e.g., USDC).

These ATAs are currently created using Anchor's strict `init` constraint:

```
#[account(
    init,
    payer = admin_creator,
    associated_token::mint = token_mint,
    associated_token::authority = presale_state
)]
presale_token_ata: Box<Account<'info, TokenAccount>>;

#[account(
    init,
    payer = admin_creator,
    associated_token::mint = base_mint,
    associated_token::authority = presale_state
)]
presale_base_ata: Box<Account<'info, TokenAccount>>;
```

This introduces a denial-of-service vector where an attacker can preemptively create these token accounts before the legitimate `CreatePresale` transaction is executed.

As a result, the admin will be unable to create the presale, since the instruction will always fail when it attempts to `init` the already-existing accounts.

Recommendations

Consider replacing `init` with `init_if_needed` for the presale ATAs.

[M-03] Vesting schedule truncates final partial hour, enabling early full claims

Severity

Impact: Medium

Likelihood: Medium

Description

When calculating the claimable amount, both vesting duration and elapsed time are converted to hour-based segments:

```
pub fn claim(&mut self, user_deposit_bump: u8) -> Result<()> {
    // --SNIP
    let total_duration = (self.presale_state.vesting_ends_at -
self.presale_state.vesting_starts_at).max(0);
    let elapsed = (now - self.presale_state.vesting_starts_at).max(0);

@==>    let segments_total = (total_duration / HOUR_IN_SECONDS).max(1) as u128;
@==>    let segments_elapsed = (elapsed / HOUR_IN_SECONDS).clamp(0, segments_total as i64) as
u128;

    let vested = ((self.user_deposit_state.tokens_bought as u128) * segments_elapsed) /
segments_total;
}
```

Because both `segments_total` and `segments_elapsed` are derived via integer division (floor), the final partial hour of the vesting window is discarded. Consider this example: - `total_duration = 27900` (7 hours and 45 minutes). - At claim time, `elapsed = 25200` (7 hours). Then: - `segments_elapsed = 25_200 / 3_600 = 7`. - `segments_total = 27900 / 3_600 = 7` (the extra 45 minutes are dropped).

Since `segments_elapsed == segments_total`, the computation vests 100% of tokens even though 45 minutes remain. Thus, the intended vesting period never fully completes.

Recommendations

Consider relying on seconds-based segments as it is less sensitive to division truncation.

[M-04] Incorrect presale parameter validation allows invalid configurations

Severity

Impact: Medium

Likelihood: Medium

Description

In `create_presale.rs`, the following check is used to validate the relationship between `token_supply`, `price_per_token_in_base`, and `base_target_balance`:

```
require!(
    (token_supply / price_per_token_in_base) > base_target_balance,
    PresaleError::InvalidAmount
);
```

This logic is incorrect and can allow invalid presale configurations. For example:

- `token_supply = 500`.
- `price_per_token_in_base = 5`.
- `base_target_balance = 2500`.

With the current check: - `token_supply / price_per_token_in_base = 500 / 5 = 100`. - `100 > 2500` is **false**, so the presale would be rejected even though the numbers are valid.

The correct relationship should be: - `token_supply * price_per_token_in_base >= base_target_balance`. - `500 * 5 = 2500 >= 2500` is **true**, so the presale should be allowed.

Recommendations

Replace the check with:

```
require!(
    token_supply
        .checked_mul(price_per_token_in_base)
        .ok_or(PresaleError::MathOverflow)?
        >= base_target_balance,
    PresaleError::InvalidAmount
);
```

This ensures the presale parameters are validated correctly and prevents configuration errors.

[M-05] Whitelist leaf is not user-scoped

Severity

Impact: Medium

Likelihood: Medium

Description

The Merkle leaf verified in `stake` and `unstake` does not include the user's public key. The leaf is constructed from only:

- `token_mint`
- `merkle_lockup_in_minutes`
- `merkle_minimum_stake_amount`
- `is_disabled = 0`

```
// stake.rs and unstake.rs (both)
let mut original_leaf = Vec::new();
original_leaf.extend_from_slice(&self.token_mint.key().to_bytes());
original_leaf.extend_from_slice(&merkle_lockup_in_minutes.to_le_bytes());
original_leaf.extend_from_slice(&merkle_minimum_stake_amount.to_le_bytes());
original_leaf.push(0u8); // is_disabled = false
```

Because the staker's identity is not committed to the leaf, any wallet can reuse the same `(index, proof)` indefinitely as long as the global root remains unchanged. The program also does not keep any nullifier or "consumed" tracking on-chain.

As a result, per-wallet allowlisting is not enforced. Any revealed proof can be relayed to arbitrary wallets, which will all pass the whitelist check. Additionally, per-wallet limits are unenforceable. A single wallet can open multiple stakes by varying `account_seed` (there's no global cap), and multiple wallets can share the same proof.

Recommendations

Bind the leaf to the user. Include `user_pubkey` in the leaf alongside the existing fields (and keep the same hashing algorithm to remain compatible with your off-chain tree tooling). For example, serialize: `leaf = keccak256("staking" || token_mint || user_pubkey || lockup_minutes || min_amount || is_disabled)`. Update both `stake` and `unstake` to recompute the identical user-bound leaf.

Low findings

[L-01] `token_collected` reports unsold not sold tokens in Withdrawn event

In the `withdraw` instruction, the `Withdrawn` event emits `token_collected` as:

```
let token_collected = self.presale_state.token_balance;
emit!(Withdrawn {
    // ...
    token_collected: token_collected,
    // ...
});
```

However, `self.presale_state.token_balance` represents the **remaining unsold tokens** in the presale, not the total tokens sold/collected. The correct value for tokens sold should be:

```
let token_collected = self.presale_state.token_supply - self.presale_state.token_balance;
```

Recommendations

Update the event emission to use the correct calculation:

```
let token_collected = self.presale_state.token_supply - self.presale_state.token_balance;
emit!(Withdrawn {
    // ...
    token_collected: token_collected,
    // ...
});
```

This ensures the event accurately reflects the total tokens sold during the presale.

[L-02] `Deposited` event overstates totals due to double addition

In `deposit`, the on-chain state is updated first:

```
self.user_deposit_state.base_spent += deposit_amount_in_base;
self.user_deposit_state.tokens_bought += token_bought;
```

but the event then emits:

```
base_deposit_total: self.user_deposit_state.base_spent + deposit_amount_in_base,
token_bought_total: self.user_deposit_state.tokens_bought + token_bought,
```

which double-adds the just-applied increment. As a result, off-chain indexers/analytics will record inflated running totals, leading to misleading dashboards or accounting.

Emit the post-update totals directly:

```
emit!(Deposited {
    presale: self.presale_state.key(),
    user: self.user.key(),
    base_deposit: deposit_amount_in_base,
    base_deposit_total: self.user_deposit_state.base_spent,
    token_bought: token_bought,
    token_bought_total: self.user_deposit_state.tokens_bought,
});
```

[L-03] Event field mismatch in `UpdateGlobalState`

In `update_global_state.rs`, the `GlobalStateUpdated` event assigns `authorized_withdrawers: self.global_state.authorized_admins` (likely a copy-paste). If the new state's `authorized_withdrawers` differs from `authorized_admins`, the event will misreport it, leading off-chain indexers and dashboards to record incorrect permissions.

Emit the actual `authorized_withdrawers` value, e.g.:

```
emit!(GlobalStateUpdated {
    authorized_admins: self.global_state.authorized_admins,
    authorized_withdrawers: self.global_state.authorized_withdrawers,
});
```

[L-04] Minor error variant mismatch

In `claim.rs`, the check for `claimable_token_amount == 0` uses `PresaleError::NothingToRefund`, which is semantically incorrect for a claim path and can confuse users, dashboards, or monitoring.

```
require!(claimable_token_amount > 0, PresaleError::NothingToRefund);
```

Return `PresaleError::NothingToClaim` instead.

[L-05] Early withdraw causes DoS for later deposits

The withdraw function can be executed before the presale closes, transferring all tokens and setting balances to zero, which causes subsequent deposit attempts to fail permanently.

```
// require!(
//     now >= self.presale_state.presale_closes_at,
//     PresaleError::NotEnded
// );
```

The function transfers all remaining tokens and base tokens from the presale:

```
let token_on_account = self.presale_token_ata.amount;
let base_on_account = self.presale_base_ata.amount;

// ... transfers all tokens and base tokens ...

```

```
token::transfer(/* transfers base_on_account */)?;
token::transfer(/* transfers token_on_account */)?;
```

Then sets the presale balances to zero:

```
self.presale_state.base_raised_balance = 0;
self.presale_state.token_balance = 0;
```

Consider the following case: - Presale reaches minimum target but hasn't closed yet. - Authorized withdrawer calls withdraw, draining all tokens. - Subsequent users try to deposit but fail because:

To mitigate, maybe revise the design.

[L-06] Equality check on ATA balance enables refund DoS

In the refund flow, the contract enforces strict equality between the ATA balance and the user's recorded purchase amount:

```
require!(
    self.user_deposit_state_token_ata.amount == self.user_deposit_state.tokens_bought,
    PresaleError::InvalidDepositState
);
```

The variable `tokens_bought` is **only increased during** `Deposit`. Deposits are only allowed while `now < presale_closes_at`. Once the presale closes, no further deposits can happen, and `tokens_bought` becomes immutable.

Refunds are enabled after close if the minimum base target was not reached:

```
require!(
    now >= self.presale_state.presale_closes_at,
    PresaleError::NotEnded
);
require!(
    self.presale_state.base_min_target_balance > self.presale_state.base_raised_balance,
    PresaleError::NotFailed
);
```

At this stage, a malicious actor who already holds these tokens can **donate tokens** directly into the user's `user_deposit_state_token_ata`. Because **anyone can transfer SPL tokens into any ATA without authorization**, the balance (`.amount`) can increase while `tokens_bought` stays fixed.

When the user later attempts to claim a refund, the strict equality check fails (`amount != tokens_bought`) and the transaction reverts with `PresaleError::InvalidDepositState`. This permanently blocks the user from recovering their base tokens, effectively locking their funds.

Recommendations

Consider removing the strict equality check since it is redundant, and when transferring tokens back from the user's ATA, **drain the full ATA balance.** This guarantees that the account will be emptied and `close_account` will not revert due to a non-zero balance.

```
- require!(
-     self.user_deposit_state_token_ata.amount == self.user_deposit_state.tokens_bought,
-     PresaleError::InvalidDepositState
- );

// ... snip ...

  token::transfer(
      CpiContext::new_with_signer(
          self.token_program.to_account_info(),
          token::Transfer {
              from: self.user_deposit_state_token_ata.to_account_info(),
              authority: self.user_deposit_state.to_account_info(),
              to: self.presale_token_ata.to_account_info(), // needs to be
self.user_deposit_token_ata
          },
          &[&UserDepositState::get_signer(
              &self.presale_state.key(),
              &self.user.key(),
              &user_deposit_bump,
          )],
      ),
-     self.user_deposit_state.tokens_bought,
+     self.user_deposit_state_token_ata.amount,
  )?;
```

[L-07] Boundary flaw enables deposit and refund at presale close

There's a boundary condition error between deposit and refund functions: - Deposit time check:

```
require!(
    now >= self.presale_state.presale_opens_at
        && now <= self.presale_state.presale_closes_at,
    PresaleError::NotInPresaleWindow
);
```

- Refund time check:

```
require!(
    now >= self.presale_state.presale_closes_at,
    PresaleError::NotEnded
);
```

The problem: At the exact moment `now == presale_closes_at` : - Deposit allows: now <= presale_closes_at. - Refund allows: now >= presale_closes_at.

Both conditions are satisfied simultaneously, creating an overlapping boundary where users can deposit and refund at the same time. This could create inconsistent operations.

Recommendations

Fix the boundary conditions to be mutually exclusive.

[L-08] Users can inflate unstake event amounts to exploit Off-Chain rewards

The `Unstaked` event currently emits the amount from the stake token account at the time of unstaking:

```
emit!(Unstaked {
    token_mint: self.token_mint.key(),
    user: self.user.key(),
    unstake_amount: self.stake_state_token_ata.amount,
    unlocks_at: self.stake_state.unlocks_at,
});
```

This value can be manipulated by users transferring additional tokens into the stake token account before calling `unstake`. If off-chain mechanism uses the `unstake_amount` from this event to calculate rewards for user, users can exploit this by inflating the amount, earning rewards far beyond what they actually staked.

Recommendations

- **Track Actual Staked Amount**: Add a `staked_amount` field to the `StakeState` struct and set it only during the initial stake operation.
- **Emit True Staked Amount**: Use the `staked_amount` field in the `Unstaked` event instead of the current token account balance.