Pump Security Review

Pashov Audit Group

Conducted by: FrankCastle, mahdiRostami, shaflow, ayeslick, Giannis443, Tigerfrake

March 18th 2025 - March 22th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **pump-fun/fixed-price-token-sale** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Pump

Pump Token Fixed Price Sale is a smart contract-managed event where users deposit USDC, USDT, or SOL at a fixed price during a predefined period, with PUMP tokens distributed after the sale ends, while integrated CEXs track contributions via PDAs and API calls to prevent overselling. The settlement involves manual reconciliation between the project and exchanges, with CEXs handling final token distribution to their users.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 52715ef1e5639b8192d17764ef0cc746b782c879

fixes review commit hash - 328cc64ea2dc9293f0e72771a7a97c43c15be71b

Scope

The following smart contracts were in scope of the audit:

- `blacklist_wallet`
- `initialize_global_state`
- `initialize_global_vault`
- `mod`
- `post_cex_sale`
- `update_global_state`
- `update_oracle`
- `withdraw`
- `buy`
- `distribute`
- `mod`
- `constants`
- `errors`
- `lib`
- `states`
- `utils`

7. Executive Summary

Over the course of the security review, FrankCastle, mahdiRostami, shaflow, ayeslick, Giannis443, Tigerfrake engaged with Pump to review Pump. In this period of time a total of **13** issues were uncovered.

Protocol Summary

Protocol Name	Pump
Repository	https://github.com/pump-fun/fixed-price-token-sale
Date	March 18th 2025 - March 22th 2025
Protocol Type	Tokensale

Findings Count

Severity	**Amount**
Medium	4
Low	9
Total Findings	**13**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Initializing SOL price makes trading unfair	Medium	Resolved
[M-02]	The blacklist_user_ix instruction skips is_blacklisted update	Medium	Resolved
[M-03]	Buying post-distribution locks funds	Medium	Acknowledged
[M-04]	Unauthorized Global and Oracle State Initialization	Medium	Acknowledged
[L-01]	Add min received token amount to buy instruction	Low	Resolved
[L-02]	User may cause DoS on CEX update when sales near expected amount	Low	Acknowledged
[L-03]	Insufficient token handling in buy_ix	Low	Resolved
[L-04]	Rounding down of fees for blacklist_fee	Low	Resolved
[L-05]	is_active condition blocks sale completion	Low	Resolved
[L-06]	Unnecessary is_active check may block event emission on delays	Low	Resolved
[L-07]	Missed initializeCexEvent emission	Low	Resolved
[L-08]	Missing validation for max_staleness_allowed	Low	Resolved
[L-09]	Hardcoded stablecoin price	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Initializing SOL price makes trading unfair

Severity

Impact: Medium

Likelihood: Medium

Description

In the `initialize_global_state` instruction, the SOL price is initialized at 200 USD/SOL.

```
pub fn initialize_global_state_ix(
    ctx: Context<InitializeGlobalState>,
    start_time: i64,
    end_time: i64,
    tokens_for_sale: u64,
) -> Result<()> {
    ...
    oracle_account.mint = constants::DEPOSIT_MINT3;
    oracle_account.price = 200_00000;
    Ok(())
}
```

However, the actual SOL price may differ significantly. For example, at the time of writing the report, the SOL price is 128.7 USD/SOL.
This allows the first batch of traders to buy tokens with SOL at a higher price before the `oracle_account` is updated, enabling them to profit.

Recommendations

It is recommended not to use hardcoded values to initialize the SOL price. Instead, the `super_admin` should provide the real price for initialization.

[M-02] The `blacklist_user_ix` instruction skips `is_blacklisted` update

Severity

Impact: Medium

Likelihood: Medium

Description

The `super_admin` can call the `blacklist_user_ix` instruction to refund the tokens purchased by `user_state`, but the contract does not update the `is_blacklisted` flag in `user_state` to `true`, which fails to prevent this user from continuing to purchase tokens.

Recommendations

Set the `is_blacklisted` flag of `user_state` to `true` in the `blacklist_user_ix` instruction.

```
pub fn blacklist_user_ix<'a, 'b, 'c, 'info>(
    ctx: Context<'a, 'b, 'c, 'info, BlacklistUser<'info>>,
) -> Result<()> {
    //...
+   user_state.is_blacklisted = true;
    Ok(())
}
```

[M-03] Buying post-distribution locks funds

Severity

Impact: Medium

Likelihood: Medium

Description

Allowing users to purchase tokens after distribution can result in a permanent loss of funds. The `distribute` function can only be called once per user, meaning any additional purchases made afterward will leave the funds locked indefinitely, as distribution cannot be re-executed.

The relevant code snippet:

```
if user_state.tokens_received > 0 {
        // Only one distribute per user
        return Err(SaleError::AlreadyDistributed.into());
    }
```

Since users can buy multiple times and across multiple mints, this creates a risk where they may unknowingly lock their funds without the ability to claim tokens.

Recommendations

Introduce an `is_distributed` flag in `user_state` and enforce a check in the `buy` function to prevent purchases once distribution has occurred. Example:

```
if user_state.is_distributed {
    return Err(SaleError::AlreadyDistributed.into());
}
```

[M-04] Unauthorized Global and Oracle State Initialization

Severity

Impact: Medium

Likelihood: Medium

Description

The `InitializeGlobalState` instruction allows any user to initialize both a `global_state` and `oracle` account using arbitrary values for `super_admin` and `admin`. The instruction lacks necessary access controls:

```
#[derive(Accounts)]
pub struct InitializeGlobalState<'info> {
    #[account(mut)]
    pub super_admin: Signer<'info>,

    /// CHECK: no need to check
    #[account(mut)]
    pub admin: AccountInfo<'info>,

    #[account(
        init,
        seeds = [
            GLOBAL_STATE_SEED.as_bytes(), super_admin.key().as_ref()
        ],
        bump,
        payer = super_admin,
        space = GlobalState::SIZE
    )]
    pub global_state: Account<'info, GlobalState>,

    #[account(
        init,
        seeds = [
            ORACLE_STATE_SEED.as_bytes(), admin.key().as_ref()
        ],
        bump,
        payer = super_admin,
        space = OracleAccount::SIZE
    )]
    pub oracle: Account<'info, OracleAccount>,

    pub clock: Sysvar<'info, Clock>,
    pub system_program: Program<'info, System>,
}
```

There is no constraint enforcing that `super_admin` is a trusted authority, nor is `admin` required to be a signer. This allows any user to create a fake `global_state` and `oracle` using arbitrary inputs — including real `admin` public keys.

Impact 1: Token Extraction via Fake Global State

A malicious actor can initialize a fake `global_state` under their control. Since there is no validation that `super_admin` is legitimate, a user calling the `buy` function might unknowingly interact with this fake state:

```
pub struct Buy<'info> {
    #[account(mut)]
    pub wallet_address: Signer<'info>,

    /// CHECK: No need to check
    pub super_admin: AccountInfo<'info>,

    #[account(
        mut,
        has_one = super_admin,
        seeds = [GLOBAL_STATE_SEED.as_bytes(), super_admin.key().as_ref()],
        bump
    )]
    pub global_state: Account<'info, GlobalState>,
}
```

As a result:

1. Normal users may send tokens to a malicious vault controlled by the attacker.
2. The attacker can later call a `withdraw` function to extract those tokens for themselves.
3. Users have no way of verifying whether the `global_state` they are interacting with is trusted.

This leads to **unrecoverable loss of tokens** by unsuspecting users.

Impact 2: Denial-of-Service for Legitimate Admin Oracle Account Creation

Because the `oracle` account is derived using the `admin`'s public key — but the `admin` is **not required to sign** during initialization — a malicious actor can **preemptively create an oracle account for any admin**:

```
#[account(
    init,
    seeds = [
        ORACLE_STATE_SEED.as_bytes(), admin.key().as_ref()
    ],
    bump,
    payer = super_admin,
    space = OracleAccount::SIZE
)]
pub oracle: Account<'info, OracleAccount>,
```

Attack Flow:

1. Attacker initializes a `global_state` using a real `admin`'s public key, and arbitrary (invalid) oracle parameters such as incorrect start/end times or tokens for sale.
2. Since PDAs are deterministic, the oracle account becomes locked in.
3. When the legitimate admin later attempts to initialize their oracle, the PDA already exists — preventing them from setting up a valid oracle.
4. This effectively blocks any admin from operating in the system if their PDA has been hijacked.

This results in a **denial-of-service condition for legitimate admins**, preventing them from establishing their intended configuration.

Recommendations

- **Restrict `super_admin`**: Set `super_admin` to a constant trusted address (hardcoded or passed as a build-time config). This ensures only the deployer or a governance authority can initialize `global_state` and `oracle` accounts.

- **Require `admin` to be a signer** in `InitializeGlobalState` to prevent oracle PDAs from being hijacked using the admin's public key without their consent.

- **Reconsider PDA derivation**: If multiple legitimate `global_state` accounts are needed, use `admin.key()` instead of `super_admin.key()` in the seeds for `global_state`. This allows legitimate admins to control their own namespaces.

These changes will prevent both user token theft and oracle initialization DoS attacks.

8.2. Low Findings

[L-01] Add min received token amount to buy instruction

When buying tokens using the `buy` instruction with SOL, the execution of the transaction may result in the user receiving fewer tokens due to oracle price updates.

```
pub fn buy_ix(ctx: Context<Buy>, amount: u64) -> Result<()> {
    ...
    let buy_value =
        oracle_state.get_token_value_usd
          (mint_key, amount as u128, ctx.accounts.mint.decimals)?;

    let number_of_tokens = global_state.usd_to_tokens(buy_value)?;

    user_state.deposit_value += buy_value;
    user_state.tokens_bought += number_of_tokens;
    global_state.tokens_sold += number_of_tokens;
    ...
}
```

It is recommended to add a minimum received token amount to prevent users from unexpectedly receiving fewer tokens.

[L-02] User may cause DoS on CEX update when sales near expected amount

When the selling quantity is about to reach the expected amount, if the CEX completes the final sale and calls an instruction to update on-chain, a malicious user can buy a small amount of tokens, causing the total token supply to exceed expectations after the CEX update, resulting in the transaction reverting.

```
pub fn is_active(self, current_time: i64) -> Result<()> {
        ...
        if self.tokens_sold + self.cex_tokens_sold >= self.tokens_for_sale {
            return err!(SaleError::SaleEnded);
        }
        Ok(())
    }
```

Even if the administrator blacklists the malicious address, the attacker can switch wallets to continue the attack.

CEX transaction updates to the blockchain have latency. It is recommended to distinguish between the expected token sale quantity on the CEX and the expected token purchase quantity on-chain.

[L-03] Insufficient token handling in `buy_ix`

In the `buy_ix` function, the user specifies an amount of one of three mints and the function calculates the amount of tokens to sell to users, however, if the requested amount exceeds the available tokens `(>tokens_for_sale)`, the function reverts. As a result, valid swaps that could partially fulfil the user's request are prevented. This issue prevents partial purchases when the requested token amount exceeds the remaining tokens.

Recommendations:

Modify the buy function to handle the case where the requested amount exceeds the available tokens. If the requested amount is greater, the function should consider the amount needed to complete `tokens_for_sale`.

[L-04] Rounding down of fees for `blacklist_fee`

In `blacklist_user_ix` function, the `blacklist_fee` is calculated as follows:

```
let blacklist_fee = amount
            .checked_mul(BLACKLIST_FEE_BPS as u64)
            .unwrap()
            .checked_div(10_000 as u64)
            .unwrap();
```

Which can result in scenarios where the fee is reduced to zero, especially for small transactions. This can cause the protocol to lose potential fees.

Recommendation:

To ensure that the protocol receives the correct fee, the function should round the fee up instead of down. This guarantees that even in cases where the fee calculation results in a fractional value, it is rounded up to the nearest integer to prevent fee loss.

```
let blacklist_fee = amount
            .checked_mul(BLACKLIST_FEE_BPS as u64)
            .unwrap()
            .checked_add(9_999)
            .unwrap()
            .checked_div(10_000 as u64)
            .unwrap();
```

By adding 9_999 before division, the fee calculation will always round up.

[L-05] `is_active` condition blocks sale completion

The `is_active` function currently prevents a full sale due to this condition:

```
if self.tokens_sold + self.cex_tokens_sold >= self.tokens_for_sale
```

This condition does not allow the last token to be sold, effectively blocking a complete sale.

Recommendation:

Modify the condition to use `>` instead of `>=`:

```diff
- if self.tokens_sold + self.cex_tokens_sold >= self.tokens_for_sale {
+ if self.tokens_sold + self.cex_tokens_sold > self.tokens_for_sale {
```

[L-06] Unnecessary `is_active` check may block event emission on delays

For delayed transactions, there is no state change required, meaning that checking the `is_active` state is unnecessary. However, the current implementation still verifies `is_active`, and if the state becomes inactive between a regular transaction and a delayed one (previously scheduled), the delayed transaction will not emit an event. This can lead to missing logs and inconsistencies in tracking transaction execution.

[L-07] Missed `initializeCexEvent` emission

The `initialize_sale_state_ix` function is supposed to emit the `InitializeCexEvent` as it is implemented. However, the function does not emit this event.

[L-08] Missing validation for `max_staleness_allowed`

The `max_staleness_allowed` variable is assigned a value but is not considered in the price validation logic from the oracle account. This oversight allows the system to use outdated price data without any constraints, potentially leading to inaccurate token valuations.

The issue originates in the `initialize_global_state_ix` function:

```
pub fn initialize_global_state_ix(
    ctx: Context<InitializeGlobalState>,
    start_time: i64,
    end_time: i64,
    tokens_for_sale: u64,
) -> Result<()> {
    ...

    oracle_account.max_staleness_allowed = u64::MAX;
    ...
}
```

Furthermore, the function `get_token_value_usd` retrieves token prices using `get_token_price`, but there is no check to ensure the retrieved price is within the allowed staleness threshold:

```
pub fn get_token_value_usd
  (&self, mint: Pubkey, amount: u128, decimals: u8) -> Result<u64> {
    let price = self.get_token_price(mint)? as u128;
    let value_in_u128 = price
        .checked_mul(amount)
        .ok_or_else(|| SaleError::MathOverflow)?
        .checked_div(10u128.pow(decimals as u32))
        .ok_or_else(|| SaleError::MathOverflow)?;
    let value_in_u64: u64 = value_in_u128
        .try_into()
        .map_err(|_| SaleError::MathOverflow)?;
    Ok(value_in_u64)
}
```

Without validating `max_staleness_allowed`, the contract risks using outdated or manipulated prices, which could lead to incorrect token valuations and unfair trading conditions.

Recommendations:

Implement a `max_staleness_allowed` check in the price validation logic to ensure that token prices remain up to date. This check should be enforced when retrieving the token price in `get_token_price`. Example:

```
pub fn get_token_price(&self, mint: Pubkey) -> Result<u64> {
    if self.last_update_timestamp + self.max_staleness_allowed < Clock::get
      ()?.unix_timestamp {
        return Err(SaleError::StalePriceData.into());
    }

    if mint == constants::DEPOSIT_MINT1 || mint == constants::DEPOSIT_MINT2 {
        return Ok(100_000);
    } else if mint == constants::DEPOSIT_MINT3 && mint == self.mint {
        return Ok(self.price);
    } else {
        return Err(SaleError::UnsupportedMint);
    }
}
```

[L-09] Hardcoded stablecoin price

In `states.rs` the function `get_token_price` assumes the prices of stablecoins to be always 1 USD.

```
pub fn get_token_price(&self, mint: Pubkey) -> Result<u64> {
    if mint == constants::DEPOSIT_MINT1 || mint == constants::DEPOSIT_MINT2 {
        return Ok(100000);
    } else if mint == constants::DEPOSIT_MINT3 && mint == self.mint {
        return Ok(self.price);
    } else {
        return err!(SaleError::UnsupportedMint);
    }
}
```

This hardcoded price will lead to issues at depegging events. For example, if the stable depegs to 0.95, the users can still contribute the stable coin and get assigned tokens of value equal to 1 USD instead of 0.95 USD.

Recommendations:

Consider adding an oracle price for stablecoins as well.