Frank castle

CoolDex Audit

January 2025

Cool Final report

Content

About Frank Castle 🦀

Frank Castle is a profissional smart contract security researcher with a focused expertise in auditing Rust-based contracts and decentralized infrastructure across leading blockchain ecosystems, including Solana , Polkadot , and Cosmos (CosmWasm).🦀

Frank Castle has audited Lido ,Pump.fun, LayerZero, Synthetix , Hydration ,DUB Social and several multi-million protocols.

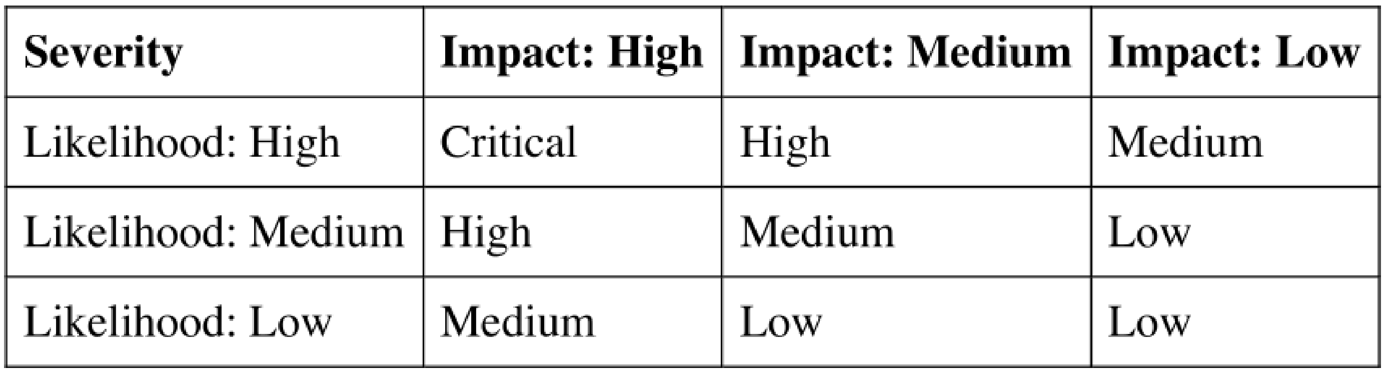
with more than ~25 Rust Audit , ~15 Solana Audits , and +100 criticals/highs found , All the reports can be found here

For private audit or consulting requests please reach out to me via Telegram @castle_chain , **Twitter** (@0xfrank_auditor) or **Discord** (castle_chain).

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.

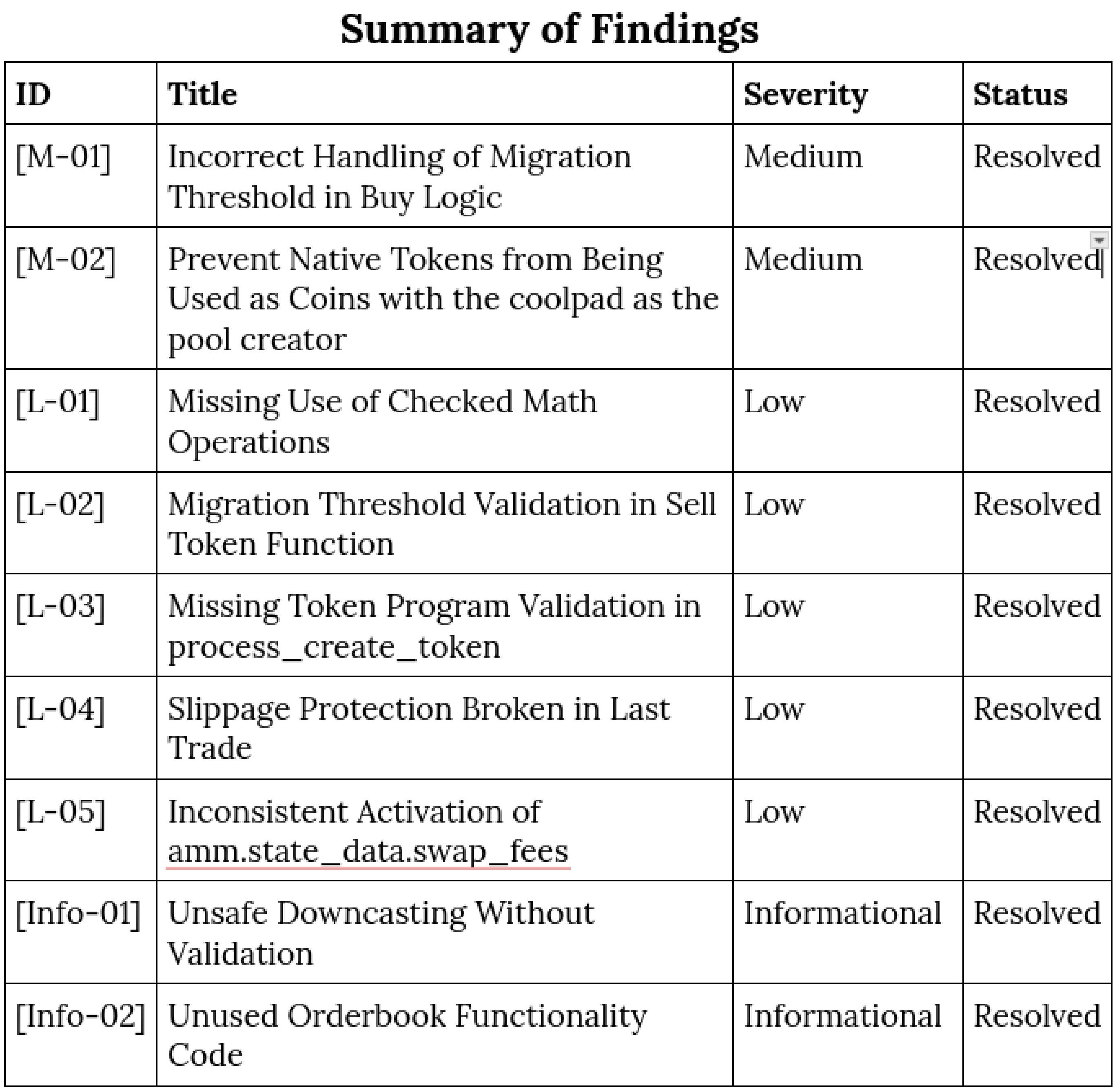
Risk Classification



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

Audit Scope

CoolDex : https://github.com/fablee1/cooldex/commit/93e64ea3fb788e46ccbfe95430f322d04b540056
CoolPad : https://github.com/fablee1/coolpad/commit/6ddf7d81c68c45e23319f60ce6dcb4832d8c1193



Summary of Findings

ID	Title	Severity	Status
[M-01]	Incorrect Handling of Migration Threshold in Buy Logic	Medium	Resolved
[M-02]	Prevent Native Tokens from Being Used as Coins with the coolpad as the pool creator	Medium	Resolved
[L-01]	Missing Use of Checked Math Operations	Low	Resolved
[L-02]	Migration Threshold Validation in Sell Token Function	Low	Resolved
[L-03]	Missing Token Program Validation in process_create_token	Low	Resolved
[L-04]	Slippage Protection Broken in Last Trade	Low	Resolved
[L-05]	Inconsistent Activation of amm.state_data.swap_fees	Low	Resolved
[Info-01]	Unsafe Downcasting Without Validation	Informational	Resolved
[Info-02]	Unused Orderbook Functionality Code	Informational	Resolved

Findings

1. Medium Findings

1.1. [M-01] Incorrect Handling of Migration Threshold in Buy Logic

Severity

Impact: Medium
Likelihood: Medium

Description

In the buy logic, after validating slippage, the program checks if there are enough tokens available to be sold to the user. Additionally, it verifies whether the expected amount to be sold (including the migration reserve) exceeds the token balance in the bonding curve. If the migration threshold is reached, the program is supposed to finalize the migration by setting the migration flag to `2`.

However, there is an issue when the condition `POOL_MIGRATION_RESERVES + expected_token = available_token` is met. In this case, the logic does not trigger the `else` branch to set the migration flag, leaving the migration unfinalized. This oversight creates a vulnerability where tokens can still be sold, bringing the balance back below the migration threshold after it was reached. This results in a situation where a subsequent user attempting to buy tokens will receive zero tokens, offering no incentive for users to trigger the migration flag.

Affected Code

```
let (sol_to_spend, token_to_receive, is_finalized) =
    if POOL_MIGRATION_RESERVES + expected_token <= available_token {
        msg!(
            "Pool migration reserves {} + expected_token {} >=
available_token {}",
            POOL_MIGRATION_RESERVES,
            expected_token,
            available_token
        );
        (amount_without_fee, expected_token, false)
    } else {
        msg!("Setting token sale as finalized");
        {
            // here we set the migration flag to 2
            let mut borrowed_data =
token_owner_pda.try_borrow_mut_data().unwrap();
            borrowed_data[0] = 2; // Mark as finalized
        }
        msg!(
            "Pool migration reserves {} + expected_token {} <
available_token {}",
            POOL_MIGRATION_RESERVES,
            expected_token,
            available_token
        );
        // the available to sell is the available token - the migration
reserves
        let token_to_sell = available_token - POOL_MIGRATION_RESERVES;
        // calc the amount of sol corresponding to the token to sell
        let allowed_sol_to_spend = ((pool_sol_balance as u128) *
(token_to_sell as u128)
            / ((pool_token_balance - token_to_sell) as u128))
            as u64;
        msg!(
            "So, amount of sol to spend POST FEE corrected to {}",
            allowed_sol_to_spend
        );
        (allowed_sol_to_spend, token_to_sell, true)
    };
```

Issue

When the condition `POOL_MIGRATION_RESERVES + expected_token = available_token` holds true, the `else` branch is not executed, leaving the migration flag unset. This opens a window for further token sales and prevents the migration from finalizing correctly.

Recommendations

Update the condition to ensure the migration flag is set when the sum of `POOL_MIGRATION_RESERVES` and `expected_token` equals `available_token`. Replace the current logic with the following code:

```
let (sol_to_spend, token_to_receive, is_finalized) =
    if POOL_MIGRATION_RESERVES + expected_token < available_token {
        msg!(
            "Pool migration reserves {} + expected_token {} <
available_token {}",
            POOL_MIGRATION_RESERVES,
            expected_token,
            available_token
        );
        (amount_without_fee, expected_token, false)
    } else {
        msg!("Setting token sale as finalized");
        {
            // here we set the migration flag to 2
            let mut borrowed_data =
token_owner_pda.try_borrow_mut_data().unwrap();
            borrowed_data[0] = 2; // Mark as finalized
        }
        msg!(
            "Pool migration reserves {} + expected_token {} >=
available_token {}",
            POOL_MIGRATION_RESERVES,
            expected_token,
            available_token
        );
        // the available to sell is the available token - the migration
reserves
        let token_to_sell = available_token - POOL_MIGRATION_RESERVES;
        // calc the amount of sol corresponding to the token to sell
        let allowed_sol_to_spend = ((pool_sol_balance as u128) *
(token_to_sell as u128)
            / ((pool_token_balance - token_to_sell) as u128))
            as u64;
        msg!(
            "So, amount of sol to spend POST FEE corrected to {}",
            allowed_sol_to_spend
        );
        (allowed_sol_to_spend, token_to_sell, true)
    };
```

1.2. [M-02] Prevent Native Tokens from Being Used as Coins whith the coolpad as the pool creator

Description

Native tokens should not be allowed as coins in the Coolpad program because they cannot be burned, and the burn functionality is not available for native tokens like SOL. This requires validation to ensure that if the Coolpad is created, the coin is not native.

According to the SPL token program implementation, attempting to burn a native token will result in an error in the `process_burn` function:

```
if source_account.is_native() {
    return Err(TokenError::NativeNotSupported.into());
}
```

Impact

Allowing native tokens as coins can lead to a denial-of-service (DoS) scenario where the trading functionality fails due to the inability to burn native tokens. This would disrupt the expected behavior of the program.

Recommendation

Prevent Native Tokens from Being Used as Coins whith the coolpad as the pool creator in the function `process_initialize2`

2. Low Findings

2.1. [L-01] Missing Use of Checked Math Operations

Description

The code contains multiple instances where standard arithmetic operations are used without checks for overflow or underflow. This could lead to unintended behavior or vulnerabilities if large values are involved. Using unchecked operations may result in overflow or underflow, potentially causing incorrect calculations.

Affected Code

An example of unchecked arithmetic is shown below:

```
let expected_token: u64 = ((pool_token_balance as u128) *
(amount_without_fee as u128)
      / ((pool_sol_balance + amount_without_fee) as u128))
      as u64;
```

In the above snippet, arithmetic operations such as multiplication, division, and addition are performed without safeguards against overflow or underflow. This lack of protection could cause the program to behave unpredictably if extreme values are encountered.

Recommendations

Use checked math operations such as `checked_mul`, `checked_div`, and `checked_add` to ensure that arithmetic operations do not result in overflow or underflow. If an overflow or underflow occurs, these functions return `None` instead of panicking or silently failing.

The corrected code should look like this:

```
let expected_token: u64 = (pool_token_balance as u128)
    .checked_mul(amount_without_fee as u128)
    .and_then(|result| result.checked_div((pool_sol_balance +
amount_without_fee) as u128))
    .unwrap_or_else(|| Err!("Math operation overflow/underflow occurred"))
as u64;
```

2.2. [L-02] Migration Threshold Validation in Sell Token Function

Description

In the `process_sell_token` function, there is no validation to check whether the migration threshold has been reached before executing a sell trade. If the `POOL_MIGRATION_RESERVES` equals or exceeds the `available_token`, the migration flag should be set to finalize the token sale. Failing to include this check can result in unintended behavior, allowing trades to proceed even when the vault is essentially depleted, and the migration should have been finalized.

Recommendation

Add the following validation before executing the sell trade:

```
if POOL_MIGRATION_RESERVES >= available_token {
    msg!("Setting token sale as finalized");
    {
        // Set the migration flag to 2
        let mut borrowed_data =
token_owner_pda.try_borrow_mut_data().unwrap();
        borrowed_data[0] = 2; // Mark as finalized
    }
}
```

This ensures that the migration flag is correctly set if the migration reserve is the only token value remaining in the vault, preventing further trades and maintaining consistency in the program's state.

2.3. [L-03] Missing Token Program Validation in `process_create_token`

Description

In the `process_create_token` function of the Coolpad program, the migration to Cooldex only accepts the SPL Token program. However, there is no validation to ensure the provided token program is the SPL Token program during the creation process.

Recommendation

Add the following check in the `process_create_token` function:

```
check_assert_eq!(
    *token_program_info.key,
    spl_token::id(),
    "spl_token_program",
    AmmError::InvalidSplTokenProgram
);
```

2.4. [L-04] Slippage Protection Broken in Last Trade

Description

During the trading process, the amount of tokens to be sold to the user in the last trade, before setting the completion flag, can differ from the user's expectations and the slippage parameter they set.

This issue is acknowledged by the developer, as it only affects the last trade. The current implementation prevents frontrunning of the last trade to ensure the migration flag is not set prematurely. While this approach works well to prevent frontrunning, it results in broken slippage protection for the last trade.

```
let token_to_sell = available_token - POOL_MIGRATION_RESERVES;
```

The amount of tokens the user receives becomes variable, depending on what is available in the vault. This prevents the user from accurately setting a slippage protection value.

2.5. [L-05] Inconsistent Activation of `amm.state_data.swap_fees`

Description

Consider either activating or deactivating `amm.state_data.swap_fees` consistently across the codebase. In the `base_amount_in` function, it is deactivated, whereas in the `base_amount_out` function, it is activated.

Here is the state where `swap_fees` is deactivated:
https://github.com/fablee1/cooldex/blob/4c51a67e473f4a510601b93747fc2d2d1003a01b/program/src/processor.rs#L2636-L2644

Here, `swap_fees` is activated:
https://github.com/fablee1/cooldex/blob/4c51a67e473f4a510601b93747fc2d2d1003a01b/program/src/processor.rs#L3262-L3268

Here, `swap_fees` is activated:
https://github.com/fablee1/cooldex/blob/4c51a67e473f4a510601b93747fc2d2d1003a01b/program/src/processor.rs#L3400-L3405

3. Informational Findings

3.1. [info-01] Unsafe Downcasting Without Validation

Description

The code uses the `as` keyword to downcast a value from a larger integer type (`u128`) to a smaller type (`u64`). This approach does not validate whether the value being cast is within the range of the target type, potentially leading to truncation or incorrect values if the value exceeds the bounds of `u64`.

Affected Code

An example of unsafe downcasting is shown below:

```
let sol_to_return: u64 = ((pool_sol_balance as u128) * (token_to_spend as
u128)
    / ((pool_token_balance + token_to_spend) as u128))
    as u64;
```

If the calculated value exceeds the maximum value of `u64`, the `as` cast will truncate it, resulting in incorrect behavior without any indication of the problem.

Recommendations

To safely downcast the value, use `u64::try_from()` to ensure that the value is within the valid range of `u64` before performing the downcast. This approach will return an error if the value cannot be represented as a `u64`, allowing the program to handle the situation gracefully.

3.2. [Info-02] Unused Orderbook Functionality Code

Description

The `enable_orderbook` variable is always set to `false`:

```
let enable_orderbook = false;
```

As a result, the code block for the orderbook functionality is never executed. This unused functionality can be removed for cleaner and more maintainable code.

Example of the unused code block:

```
if enable_orderbook {
    check_assert_eq!(
        *amm_open_orders_info.key,
        amm.open_orders,
        "open_orders",
        AmmError::InvalidOpenOrders
    );
    check_assert_eq!(
        *market_program_info.key,
        amm.market_program,
        "market_program",
        AmmError::InvalidMarketProgram
    );
    check_assert_eq!(
        *market_info.key,
        amm.market,
        "market",
        AmmError::InvalidMarket
    );
    let (market_state, open_orders) = Processor::load_serum_market_order(
        market_info,
        amm_open_orders_info,
        amm_authority_info,
        &amm,
        false,
    )?;
    let bids_orders = market_state.load_bids_mut(&market_bids_info)?;
    let asks_orders = market_state.load_asks_mut(&market_asks_info)?;
    (bids, asks) = Self::get_amm_orders(&open_orders, bids_orders,
asks_orders)?;
    (total_pc_without_take_pnl, total_coin_without_take_pnl) =
        Calculator::calc_total_without_take_pnl(
            amm_pc_vault.amount,
            amm_coin_vault.amount,
            &open_orders,
            &amm,
            &market_state,
            &market_event_queue_info,
            &amm_open_orders_info,
        )?;
}
```