Frank castle

TailWind launchpad audit

Scope

- tailwind-zone/contracts/launchpad
- tailwind-zone/contracts/astroport-pool-migrate
- tailwind-zone/contracts/launchpad-vesting

Finding summary

ID	Title	Severity
Major-1	unsafe cast in curve math , will lead to permanent Dos of the protocol and loss of funds	Major
Major-2	Unhandled Scenario of token0_denom Equal to token1_denom Causes Loss of Funds in astroport-pool-migrate	Major
Minor-1	Storage can be bloated with zero value deposits stored in OWNERSHIP_DEPOSITED ,and VESTING_CLAIMED mappings , which will cause serious damage to the network.	Minor
info-1	Fund Locking Vulnerability in Launchpad and Launchpad-Vesting Contracts due to Lack of Mechanism to Withdraw Non-Relevant Locked Tokens .	informational
info-2	Enhancement of migrate_funds Function in Launchpad Contract to Handle Unredeemed Fees.	informational

Findings

1. Majors

1.1 unsafe cast in curve math , will lead to permanent Dos of the protocol and loss of funds .

Location

contracts/launchpad/src/curve.rs#L138-L140

Description:

in the function `slope_from()` , it uses `to_precision` to scale the number to a certain precision , in order to preform the curve calculation .
In the function `to_precision`

```
pub fn to_precision(num: Uint128, scale: u32) -> Decimal {
    Decimal::from_i128_with_scale(num.u128() as i128, scale)
}
```

there is unsafe cast from `Uint128` to `i128` , which can result in negative value in case of using `Uint128::MAX` , which will **NOT REVERT** but it will have a negative value .
as shown in this test

```

    fn to_prec1() {

        let per :u32 = 6 ;
        let n =Uint128::MAX ;
        let total = to_precision(n, per);
        println!("{:?}" , total);
        println!("{:?}" , Uint128::MAX.u128() as i128);
    }
```

the printed value is

```
-0.000001
-1
```

so the function `to_precision` will not revert but it will return negative value from the unsafe cast .
the function `to_precision` reverts if the value is greater than `2^96` , but `-1` or `-0.000001` is not greater than `2^96` so they are acceptable .

although that negative slope is not allowed

```
        if slope <= Decimal::ZERO {
            return Err(ContractError::InvalidCurveParams {
                msg: format!("Slope cannot be negative {}",
slope).to_string(),
            });
        }
```

, but the calculation of the slope is

```
K = (2(M - cS))/S^2
```

another unexpected behaviour

setting the `launch_goal` `S` to high value should result in low value `slope` , as per calculation :

```
K = (2(M - cS))/S^2
```

this is valid untill the `launch_goal` is set to `Uint128::MAX` this will result in low value `- 0.000001` , the square of this value is `0.00000000001` , putting this value in the denominator will result in very large value

```
    fn test_curve_from_goals() {
        let curve = CurveState::from_goals(
            Uint128::new(60_000_000_000),
            6,
            Uint128::MAX,
            6,
            Uint128::zero(),
        )
        .unwrap();
        println!("{:?}" , curve.slope);
    }
```

the value of the slope will be

```
"1200000000000000000"
```

Recommendation

the function `to_precision` should revert if any of the values = `Uint128::MAX`.

```
pub fn to_precision(num: Uint128, scale: u32) -> Decimal {
+ if (num == Uint128::MAX ) revert ;
    Decimal::from_i128_with_scale(num.u128() as i128, scale)
}
```

1.2 Unhandled Scenario of token0_denom Equal to token1_denom Causes Loss of Funds in astroport-pool-migrate .

Location

contracts/astroport-pool-migrate/src/contract.rs#L84-L136
contracts/astroport-pool-migrate/src/contract.rs#L250-L276

Description:

n the function `create_pool_and_migrate()`, which is designed to create a new liquidity pool and migrate liquidity, there is a crucial oversight. The function accepts two token denoms as input and validates them using the `parse_token_pair()` function. However, `parse_token_pair()` does not check if `token0_denom` and `token1_denom` are the same, allowing both denoms to be set to the same token. This can lead to significant loss of funds.

Code Snippet

```
fn parse_token_pair(
    actual_funds: &[Coin],
    token0_denom: String,
    token1_denom: String,
) -> Result<(Uint128, Uint128), ContractError> {
    if actual_funds.len() != 2 {
        return Err(ContractError::InvalidFunds {
            msg: format!("Expected 2 tokens, got {}", actual_funds.len()),
        });
    }

    let token0 = actual_funds
        .iter()
        .find(|c| c.denom == token0_denom)
        .ok_or(ContractError::InvalidFunds {
            msg: format!("token0_denom not included: {}",
token0_denom).to_string(),
        })?;

    let token1 = actual_funds
        .iter()
        .find(|c| c.denom == token1_denom)
        .ok_or(ContractError::InvalidFunds {
            msg: format!("token1_denom not included: {}",
token1_denom).to_string(),
        })?;

    Ok((token0.amount, token1.amount))
}
```

Issue Description

The `parse_token_pair` function does not validate that `token0_denom` is different from `token1_denom`. An attacker can exploit this by setting both `token0_denom` and `token1_denom` to the same value (e.g., `token0_denom`). When the function processes the input, it will assign the amount of `token0` to both tokens. For example, if 500 units of `token0` and 200 units of `token1` are provided, the function will incorrectly return 500 units for both tokens. This will cause the `astroport-pool-migrate` contract to deposit more funds than intended into the pool.

The `create_pool_and_migrate()` uses the resulting values and store them to be provided to the pool , which is considered a huge loss of funds .

```
    ADD_LIQUIDITY_CONTEXT.save(
        deps.storage,
        &AddLiquidityContext {
            token0_denom,
            token0_amount,
            token1_denom,
            token1_amount,
                  },
    )?;
```

Recommendations

Prevent setting `token0_denom` to `token1_denom` by modifying the `parse_token_pair()` function as follows:

```
fn parse_token_pair(
    actual_funds: &[Coin],
    token0_denom: String,
    token1_denom: String,
) -> Result<(Uint128, Uint128), ContractError> {
    if actual_funds.len() != 2 {
        return Err(ContractError::InvalidFunds {
            msg: format!("Expected 2 tokens, got {}", actual_funds.len()),
        });
    }
+   if token0_denom == token1_denom {
+       return Err(ContractError::InvalidFunds {
+           msg: "token0_denom and token1_denom must be
different".to_string(),
+       });
+   }
```

2 Minors

2.1 Storage can be bloated with zero value deposits stored in OWNERSHIP_DEPOSITED ,and VESTING_CLAIMED mappings , which will cause serious damage to the network.

Location

contracts/launchpad-vesting/src/contract.rs#L178-L207

Description :

The `launchpad-vesting` contract is susceptible to storage bloating due to the lack of checks for zero value deposits in the `OWNERSHIP_DEPOSITED` and `VESTING_CLAIMED` mappings. Specifically, the `transfer_ownership` function creates new storage slots without verifying that the amount to be stored is greater than zero. This allows any user to call `transfer_ownership` an unlimited number of times with a deposit equal to `zero` , leading to unnecessary storage usage.

Additionally, the `execute_deposit` function does not enforce a minimum deposit amount, further contributing to the risk of storage bloating by allowing trivial deposits to create storage entries.

```
pub fn execute_transfer_ownership(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    recipient: String,
) -> Result<Response, ContractError> {
    let recipient_addr = deps.api.addr_validate(&recipient)?;

    let ownership_deposited = OWNERSHIP_DEPOSITED.load(deps.storage,
&info.sender)?;
    let total_vested = VESTING_CLAIMED.load(deps.storage, &info.sender)?;

    OWNERSHIP_DEPOSITED.save(deps.storage, &info.sender, &Uint128::zero())?;
    VESTING_CLAIMED.save(deps.storage, &info.sender, &Uint128::zero())?;

   -> OWNERSHIP_DEPOSITED.update(
        deps.storage,
        &recipient_addr,
        |old| -> Result<_, ContractError> { Ok(old.unwrap_or_default() +
ownership_deposited) },
    )?;
   -> VESTING_CLAIMED.update(
        deps.storage,
        &recipient_addr,
        |old| -> Result<_, ContractError> { Ok(old.unwrap_or_default() +
total_vested) },
    )?;

    Ok(Response::default().add_attributes(vec![
        ("action", "execute_transfer_ownership"),
        ("recipient", &recipient_addr.to_string()),
    ]))
}
```

Recommendation

To prevent storage bloating, the contract should enforce checks to ensure that zero value deposits are not allowed. The following code snippet provides a fix:

```
pub fn execute_transfer_ownership(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    recipient: String,
) -> Result<Response, ContractError> {
    let recipient_addr = deps.api.addr_validate(&recipient)?;

    let ownership_deposited = OWNERSHIP_DEPOSITED.load(deps.storage,
&info.sender)?;
    let total_vested = VESTING_CLAIMED.load(deps.storage, &info.sender)?;
+   if ownership_deposited.is_zero() && total_vested.is_zero() {
+       return Err(ContractError::InvalidAmount {});
+   }

    OWNERSHIP_DEPOSITED.save(deps.storage, &info.sender, &Uint128::zero())?;
    VESTING_CLAIMED.save(deps.storage, &info.sender, &Uint128::zero())?;

    OWNERSHIP_DEPOSITED.update(
        deps.storage,
        &recipient_addr,
        |old| -> Result<_, ContractError> { Ok(old.unwrap_or_default() +
ownership_deposited) },
    )?;
    VESTING_CLAIMED.update(
        deps.storage,
        &recipient_addr,
        |old| -> Result<_, ContractError> { Ok(old.unwrap_or_default() +
total_vested) },
    )?;

    Ok(Response::default().add_attributes(vec![
        ("action", "execute_transfer_ownership"),
        ("recipient", &recipient_addr.to_string()),
    ]))
}
```

3 Informationals

3.1 Fund Locking Vulnerability in Launchpad and Launchpad-Vesting Contracts due to Lack of Mechanism to Withdraw Non-Relevant Locked Tokens .

Location

contracts/launchpad/src/contract.rs#L446-L558

Description :

In the Launchpad contract, the `buy_launch_tokens` and `sell_launch_tokens` functions validate the input funds sent with the call. However, other functions such as `redeem_fees` and `migrate_funds` , do not prevent sending funds to the contract, and there is no mechanism to withdraw these funds. As a result, any tokens sent to these functions will be locked within the contract indefinitely.

Example Function: `redeem_fees`

```
fn redeem_fees(
    deps: DepsMut<SeiQueryWrapper>,
    _env: Env,
    info: MessageInfo,
    recipient: Option<String>,
    amount: Option<Uint128>,
) -> Result<Response<SeiMsg>, ContractError> {
    let Config {
        funding_token_denom,
        fee_admin_addr,
        ..
    } = CONFIG.load(deps.storage)?;
    if info.sender != fee_admin_addr {
        return Err(ContractError::Unauthorized {});
    }

    let recipient = recipient.map_or_else(
        || Ok(fee_admin_addr),
        |addr_unvalidated| deps.api.addr_validate(&addr_unvalidated),
    )?;

    let fee_total = FEE_TOTAL.load(deps.storage)?;
    let fee_to_send = amount.unwrap_or(fee_total);
    if fee_to_send > fee_total.clone() {
        return Err(ContractError::InvalidFunds {
            msg: format!(
                "Fees specified {} exceed fees in contract: {}",
                fee_to_send,
                fee_total.clone()
            ),
        });
    }

    FEE_TOTAL.update(deps.storage, |fee_total| -> StdResult<Uint128> {
        Ok(fee_total - fee_to_send)
    })?;
    let msg = BankMsg::Send {
        to_address: recipient.into(),
        amount: vec![Coin {
            denom: funding_token_denom,
            amount: fee_to_send,
        }],
    };

    Ok(Response::new().add_message(msg))
}
```

Recommendations :

To address this issue, the following steps are recommended:

1. **Create a Withdrawal Mechanism**: Add a `withdraw_tokens` function to allow the withdrawal of locked tokens from the contract. Ensure that the function can handle and differentiate between relevant and non-relevant tokens.
2. **Prevent Unintended Fund Transfers**: Implement checks to prevent the sending of funds with calls to functions that do not handle them, ensuring only intended transfers are allowed.

Proposed `withdraw_tokens` Function

```
fn withdraw_tokens(
    deps: DepsMut<SeiQueryWrapper>,
    info: MessageInfo,
    amount: Uint128,
    denom: String,
    recipient: String,
) -> Result<Response<SeiMsg>, ContractError> {
    let Config {
        funding_token_denom,
        vesting_token_denom,
        launch_token_denom,
        ..
    } = CONFIG.load(deps.storage)?;

    // Check that the token to be withdrawn is not the funding, vesting, or
launch token
    if denom == funding_token_denom || denom == vesting_token_denom || denom
== launch_token_denom {
        return Err(ContractError::UnauthorizedTokenWithdrawal {
            msg: "Withdrawal of funding, vesting, or launch tokens is not
allowed".to_string(),
        });
    }

    let recipient_addr = deps.api.addr_validate(&recipient)?;
    let balance =
deps.querier.query_balance(deps.api.addr_humanize(&deps.api.self_address())?
, denom.clone())?.amount;

    if balance < amount {
        return Err(ContractError::InsufficientFunds { balance, required:
amount });
    }

    let msg = BankMsg::Send {
        to_address: recipient_addr.into(),
        amount: vec![Coin {
            denom,
            amount,
        }],
    };

    Ok(Response::new().add_message(msg))
}
```

3.2 Enhancement of migrate_funds Function in Launchpad Contract to Handle Unredeemed Fees.

Location

contracts/launchpad/src/contract.rs#L493-L551

description :

n the `migrate_funds` function of the `launchpad` contract, the function is responsible for transferring all collected `funding_tokens` and 200M of `vesting_tokens` to the `AMM` and sending 800M of tokens to the `launchpad-vesting` contract. However, the `migrate_funds` function currently does not handle unredeemed fees, which results in these fees remaining locked in the contract. This oversight leaves unredeemed fees unhandled, contradicting the function's goal of finalizing the state of the launchpad contract.

`migrate_funds`

```rust
pub fn migrate_funds(
    deps: DepsMut<SeiQueryWrapper>,
    _env: Env,
    _info: MessageInfo,
) -> Result<Response<SeiMsg>, ContractError> {
    let CurveState {
        funding_precision,
        funding_supply,
        launch_supply,
        ..
    } = CURVE_STATE.load(deps.storage)?;
    let config = CONFIG.load(deps.storage)?;
    let Config {
        team_config,
        funding_token_denom,
        migrate_admin_addr,
        vesting_token_denom,
        launch_token_denom,
        launch_goal,
        ..
    } = config.clone();

    if launch_supply < launch_goal {
        return Err(ContractError::InvalidMigration {});
    }

    let bank_msg = team_config.and_then(|c|
c.to_allocate_bank_msg(launch_token_denom));

    let execute_msg = ExecuteMsgMigrateAdmin::CreatePoolAndMigrate {
        token0_denom: funding_token_denom.clone(),
        token0_precision: funding_precision,
        token1_denom: vesting_token_denom.clone(),
        token1_precision: 6,
    };
    let amm_migrate_msg = WasmMsg::Execute {
        contract_addr: migrate_admin_addr.to_string(),
        msg: to_json_binary(&execute_msg)?,
        funds: parse_sort(vec![
            Coin {
                denom: funding_token_denom,
                amount: funding_supply,
            },
            Coin {
                denom: vesting_token_denom,
                amount: (LAUNCH_TOTAL_SUPPLY -
LAUNCH_NON_MIGRATE_SUPPLY).into(),
            },
        ]),
    };
    let vest_instantiate_msg = config.vest_instantiate_msg()?;
    let vest_instantiate_sub_msg: SubMsg<SeiMsg> =
        SubMsg::reply_on_success(vest_instantiate_msg,
INSTANTIATE_VEST_REPLY_ID);

    // Ensure team allocation is sent first, if it exists
    let res = bank_msg.map_or(Response::new(), |msg|
Response::new().add_message(msg));

    Ok(res
        .add_submessage(vest_instantiate_sub_msg)
        .add_message(amm_migrate_msg)
        .add_attribute("action", "execute_migrate_funds"))
}
```

The `migrate_funds` function should finalize the state of the launchpad contract, ending its lifecycle by transferring all funds .

Recommendation

the `migrate_funds` function should include the handling of unredeemed fees. The function should send these fees to the `fee_admin_addr`.

```
    let fee_total = FEE_TOTAL.load(deps.storage)?;
    if fee_total > Uint128::zero() {
        let fee_admin_addr = CONFIG.load(deps.storage)?.fee_admin_addr;
        let fee_to_send = fee_total;

        FEE_TOTAL.update(deps.storage, |fee_total| -> StdResult<Uint128> {
            Ok(fee_total - fee_to_send)
        })?;

        let fee_msg = BankMsg::Send {
            to_address: fee_admin_addr.into(),
            amount: vec![Coin {
                denom: funding_token_denom.clone(),
                amount: fee_to_send,
            }],
        };

        res = res.add_message(fee_msg);
    }
```