Secure Code Review of Retrograde Smart Contracts

Retrograde

April 2022

Version: 1.0

Presented by:
BTblock LLC

Corporate Headquarters
BTblock LLC
PO Box 147044
Lakewood, CO 80214
United States

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

EXECUTIVE SUMMARY

Overview

Retrograde engaged BTblock LLC to perform a Secure Code Review of Retrograde Smart Contracts.

The assessment was conducted remotely by the BTblock Security Team. Testing took place on March 28 - April 08, 2022, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the BTblock Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce the risk they pose.

- BT-RETR-01 − Airdrop window can be set in the past and prevent users to claim tokens in simple_airdrop contract
- BT-RETR-02 − Potential overflow in cw20-minter-set contract
- BT-RETR-03 − Sent funds are not returned if they don't match token price in IDO contract
- BT-RETR-04 − Sent funds with denominations other than uusd are lost in IDO contract
- BT-RETR-05 − Edge case: Potential left-over lp_token tokens in convert-lock contract
- BT-RETR-06 − Edge case: Potential left-over retro tokens in IDO contract
- BT-RETR-07 − Reward_unlock_time is allowed to be set in the past in staking contract
- BT-RETR-08 − lp_token is allowed to be reset multiple times
- BT-RETR-09 − Use of unwrap() may create panic
- BT-RETR-10 − execute_add_minter allows the input minter to be the owner in cw20-minterset
- BT-RETR-11 − lp_reserve_ratio in convert-lock contract is not validated

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was very supportive and open to discuss the design choices made

Based on formal verification we conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

BTblock performed a Secure Code Review of Retrograde Smart Contracts. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a private repository at https://github.com/retrogradeprotocol/retrograde-contracts with the commit hash 063c96f041a5947566b5b98227332134f21ee786.

Files included in the code review

```
retrograde-projects/
├── contracts/
│   ├── bonding/
│   │   ├── examples/
│   │   │   └── schema.rs
│   │   ├── src/
│   │   │   ├── testing/
│   │   │   │   ├── contract.rs
│   │   │   │   └── mod.rs
│   │   │   ├── contract.rs
│   │   │   ├── error.rs
│   │   │   ├── lib.rs
│   │   │   ├── msg.rs
│   │   │   └── state.rs
│   │   ├── Cargo.lock
│   │   └── Cargo.toml
│   ├── convert-lock/
│   │   ├── examples/
│   │   │   └── schema.rs
│   │   ├── src/
│   │   │   ├── testing/
│   │   │   │   ├── contract.rs
│   │   │   │   └── mod.rs
│   │   │   ├── contract.rs
│   │   │   ├── error.rs
│   │   │   ├── helpers.rs
│   │   │   ├── integration_tests.rs
│   │   │   ├── lib.rs
│   │   │   ├── mock_querier.rs
│   │   │   ├── msg.rs
│   │   │   └── state.rs
│   │   ├── Cargo.lock
│   │   └── Cargo.toml
│   ├── cw20-minter-set/
│   │   ├── examples/
│   │   │   └── schema.rs
│   │   ├── src/
│   │   │   ├── allowances.rs │
│   │   ├── contract.rs
│   │   │   ├── enumerable.rs
│   │   │   ├── error.rs
│   │   │   ├── lib.rs
```

```
│   │   │   ├── msg.rs
│   │   │   └── state.rs
│   │   └── Cargo.toml
│   ├── ido/
│   │   ├── examples/
│   │   │   └── schema.rs
│   │   ├── src/
│   │   │   ├── testing/
│   │   │   │   ├── contract.rs
│   │   │   │   └── mod.rs
│   │   │   ├── contract.rs
│   │   │   ├── error.rs
│   │   │   ├── lib.rs
│   │   │   ├── mock_querier.rs
│   │   │   ├── msg.rs
│   │   │   └── state.rs
│   │   ├── Cargo.lock
│   │   └── Cargo.toml
│   ├── simple_airdrop/
│   │   ├── examples/
│   │   │   └── airdrop_schema.rs
│   │   ├── src/
│   │   │   ├── contract.rs
│   │   │   ├── crypto.rs
│   │   │   ├── helpers.rs
│   │   │   ├── lib.rs
│   │   │   ├── simple_airdrop.rs
│   │   │   └── state.rs
│   │   ├── tests/
│   │   │   └── integration.rs
│   │   └── Cargo.toml
│   ├── staking/
│   │   ├── examples/
│   │   │   └── schema.rs
│   │   ├── src/
│   │   │   ├── contract.rs
│   │   │   ├── lib.rs
│   │   │   ├── mock_querier.rs
│   │   │   ├── msg.rs
│   │   │   ├── querier.rs
│   │   │   ├── state.rs
│   │   │   └── testing.rs
│   │   ├── Cargo.lock
│   │   └── Cargo.toml
│   └── treasury/
│       ├── examples/
│       │   └── schema.rs
│       ├── src/
│       │   ├── testing/
│       │   │   ├── contract.rs
│       │   │   └── mod.rs
│       │   ├── contract.rs
│       │   ├── error.rs
```

```
│       │   ├── lib.rs
│       │   ├── msg.rs
│       │   └── state.rs
│       └── Cargo.toml
├── packages/
│   └── retrograde/
│       ├── src/
│       │   ├── bond_pricing.rs
│       │   ├── lib.rs
│       │   └── market.rs
│       └── Cargo.toml
├── Cargo.lock
└── Cargo.toml
```

Table 1: Scope

TECHNICAL ANALYSES AND FINDINGS

During the Secure Code Review of Retrograde Smart Contracts, we discovered:

- 4 findings with MEDIUM severity rating.
- 4 findings with LOW severity rating.
- 3 findings with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

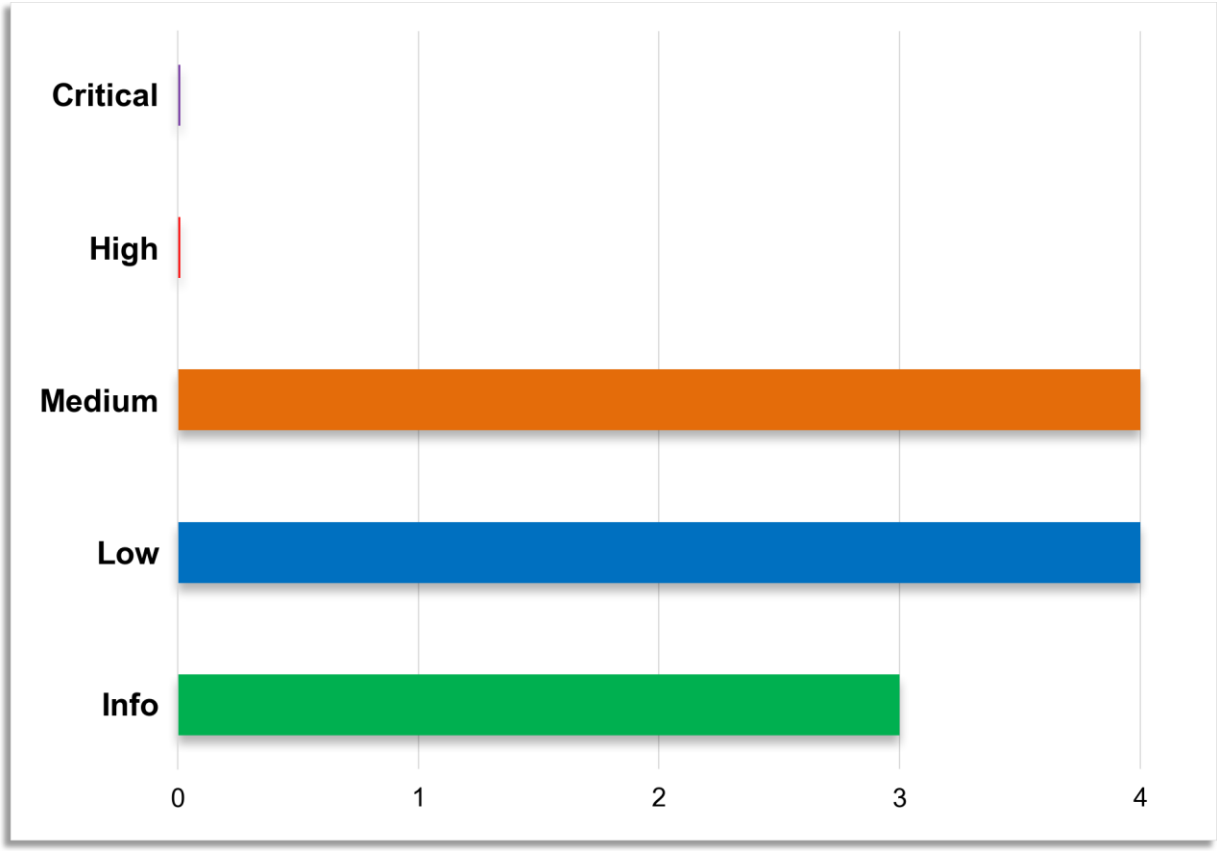

Figure 1: Findings by Severity

Findings

The **Findings** section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description

BT-RETR-01	**Medium**	Airdrop window can be set in the past and prevent users from claiming tokens in simple_airdrop contract
BT-RETR-02	**Medium**	Potential overflow in cw20-minter-set contract
BT-RETR-03	**Medium**	Sent funds are not returned if they don't match token price in IDO contract
BT-RETR-04	**Medium**	Sent funds with denominations other than uusd are lost in IDO contract
BT-RETR-05	**Low**	Edge case: Potential left-over lp_token tokens in convert-lock contract
BT-RETR-06	**Low**	Edge case: Potential left-over retro tokens in IDO contract
BT-RETR-07	**Low**	Reward_unlock_time is allowed to be set in the past in staking contract
BT-RETR-08	**Low**	lp_token is allowed to be reset multiple times
BT-RETR-09	**Informational**	Use of unwrap() may create panic
BT-RETR-10	**Informational**	execute_add_minter allows the input minter to be the owner in cw20-minter-set
BT-RETR-11	**Informational**	lp_reserve_ratio in convert-lock contract is not validated

Table 2: Findings Overview

Technical Analyses

Based on the source code the following account relationship graphs or reference graphs was made to verify the validity of the code as well as confirmation that the intended functionality was implemented correctly and to the extent that the state of the repository allowed.

Further investigations concluded that no critical risks were identified for the application, including:

- No potential panics were detected
- No potential errors regarding wraps/unwraps, expect and wildcards
- No internal unintentional unsafe references

Conclusion

Based on formal verification we conclude that the code implements the documented functionality to the extent of the reviewed code.

Technical Findings

General Observations

During the code assessment it was noted that the code is well structured and concisely split into files. The rust code is well written. However, it looks like there is not consistency in checking whether the deposit/withdraw amount should be greater than 0. For example, bonding and convert-lock contracts do not check if the received amount is greater than 0, but the IDO and simple_airdrop do have such a check.

Moreover, there is an unused parameter `minter.cap` which should be removed. Overall, no critical issues were found.

Airdrop window can be set in the past and prevent users from claiming tokens in simple_airdrop contract

Finding ID: BT-RETR-01
Severity: **Medium**
Status: **Open**

Description

The `instantiate` function in simple_airdrop contract allows `[from_timestamp, to_timestamp]` window to be (accidentally) set in the past if `to_timestamp >= from_timestamp`.

Proof of Issue

File name: simple_airdrop/src/contract.rs
Line number: 30-61

```
    let from_timestamp = msg
        .from_timestamp
        .unwrap_or_else(|| env.block.time.seconds());

    if msg.to_timestamp <= from_timestamp {
return Err(StdError::generic_err(
            "Invalid airdrop claim window closure timestamp",
        ));
    }
```

The code only uses current blocktime if `msg.from_timestamp` is not provided. In the case `msg.from_timestamp` is provided, it is not checked whether it is current or not. The code only checks whether `msg.to_timestamp <= from_timestamp`.

Severity and Impact Summary

Because `config.from_timestamp` is in the past (`env.block.time.seconds() >= config.from_timestamp`), the owner of the contract cannot update the config after the contract is initialized.

File name: simple_airdrop/src/contract.rs **Line number:** 177-180

```
    if env.block.time.seconds() >= config.from_timestamp {
return Err(StdError::generic_err(
                "from_timestamp can't be changed after window starts",
            ));
    }
```

and users cannot claim any tokens after that because `config.from_timestamp` is not valid.

Line number: 237-239

```
    // CHECK :: IS AIRDROP CLAIM WINDOW OPEN ?
    if config.to_timestamp < env.block.time.seconds() {              return
Err(StdError::generic_err("Claim period has concluded"));      }
```

Recommendation

We suggest that the team adds a check to make sure that `msg.from_timestamp` is current.

```
    let from_timestamp = msg
        .from_timestamp
        .unwrap_or_else(|| env.block.time.seconds());

    if from_timestamp < env.block.time.seconds() {
return Err(StdError::generic_err(
            "from_timestamp can't be in the past",
        ));
    }
```

Potential overflow in cw20-minter-set contract

Finding ID: BT-RETR-02
Severity: **Medium**
Status: **Open**

Description

There is a lot of unchecked math in the cw20-minter-set contract. Rust behaves differently in debug mode and release mode on Integer overflow/underflows. In debug mode, Rust adds built-in checks for overflow/underflow and panics when an overflow/underflow occurs at runtime. However, in release (or optimization) mode, Rust silently ignores this behavior by default and computes two's complement wrapping (e.g., 255+1 becomes 0 for an u8 integer).

Proof of Issue

File name: cw20-minter-set/src/contract.rs **Line number:** 308

```
    config.total_supply += amount;
```

File name: cw20-minter-set/src/allowances.rs
Line number: 308

```
    ALLOWANCES.update(
deps.storage,
        (&info.sender, &spender_addr),          |allow|
-> StdResult<_> {              let mut val =
allow.unwrap_or_default();              if let
Some(exp) = expires {                  val.expires =
exp;
            }
val.allowance += amount;
            Ok(val)
        },
    )?;
```

Severity and Impact Summary

Because Rust in release (or optimization) mode silently ignores overflow behavior by default and computes two's complement wrapping, the limit check for the `config.total_supply` will be bypassed.

```
if config.total_supply > limit {     return
Err(ContractError::CannotExceedCap {}); }
```

Or in `execute_increase_allowance` function when the allowance is updated `val.allowance += amount` the amount is wrapped and the function sets incorrect amount.

Recommendation

There are three common ways to deal with arithmetic errors:

1. Replace `+`, `-`, `*`, `/`, `**` with `checked_add`, `checked_sub`, `checked_mul`, and `checked_div` `checked_pow`, respectively.

2. Replace `+`, `-`, `*`, `**` with `saturating_add`, `saturating_sub`, `saturating_mul`, and `saturating_pow`, respectively. This ensures the computed value will stay at the numeric bounds instead of overflowing/underflowing.

3. Alternatively, turn on the overflow-checks in release mode by setting `overflow-checks = true under [profile.release]`

References

1. https://doc.rust-lang.org/book/ch03-02-data-types.html#integer-overflow
2. https://doc.rust-lang.org/cargo/reference/profiles.html#overflow-checks

Sent funds are not returned if they don't match token price in IDO contract

Finding ID: BT-RETR-03
Severity: **Medium**
Status: **Open**

Description

If total `TOTAL_UST_DEPOSITS` is greater than total retro tokens `TOTAL_RETRO_DEPOSITS_PHASE_ONE`, the users who deposited less than the retro token price get no tokens, but their funds are not returned.

For example, if a user deposits 1 UST and there are a total of 1000 UST deposits and 100 retro tokens, the the user can claim nothing and the 1 UST is sent to IDO treasury after that.

Proof of Issue

The total number of retro tokens a user can claim is calculated as follows:

File name: ido/src/contract.rs **Line number:** 274-275

```
let total_retro_deposits: Uint128 =
TOTAL_RETRO_DEPOSITS_PHASE_ONE.load(deps.storage)?;        let
user_claim:  Uint128  =  Decimal::from_ratio(existing_amount,
current_deposits_total) * total_retro_deposits;
```

Severity and Impact Summary

In order to claim one retro token, the `existing_amount` (the deposit amount) should be at least `current_deposits_total/total_retro_deposits`. This won't be problematic if `total_retro_deposits > current_deposits_total` (retro tokens are less valuable than UST), but if retro tokens are more valuable than UST, users who deposited funds that are less than `current_deposits_total/total_retro_deposits` cannot claim any retro tokens.

Recommendation

We recommend that the team either disclose that the IDO contract does not return funds in such cases OR to design an additional phase 5 to return the funds back.

Sent funds with denominations other than uusd are lost in IDO contract

Finding ID: BT-RETR-04
Severity: **Medium**
Status: **Open**

Description

In IDO contract, when a user deposits funds such that the denominations are not uusd, the check in `deposit_ust` function does not return an error, which implies that any such other funds are locked in the contract.

Proof of Issue

File name: ido/src/contract.rs **Line number:** 111-116

```
    let sent_funds: Uint128 = info
        .funds
        .iter()
        .find(|c| c.denom == "uusd")
        .map(|c| c.amount)
        .unwrap_or_else(Uint128::zero);

    // Cannot deposit zero amount     if
sent_funds.is_zero() {          return
Err(ContractError::Std(StdError::generic_err("Deposit
must be larger than zero.")));      }
```

Severity and Impact Summary

Users can accidentally send incorrect types of funds to the IDO contract. The deposit transaction is still executed successfully, and the other funds are locked in the contract since only UST are sent to the Retrograde treasury.

Recommendation

We suggest that the team either return an error or return the funds to the users. For example,

```
let sent_funds: Uint128 = info
    .funds
    .iter()      .map(|item| {
if item.denom != "uusd" {
            Err(StdError::generic_err(format!("Invalid asset provided, only
uusd allowed.")))
        } else {
            Ok(item.amount)
        }

    })
    .last()
    .ok_or_else(|| {
        StdError::generic_err(format!(
            "No uusd assets have been provided."))})??
    .into();
```

Edge case: Potential left-over lp_token tokens in convert-lock contract

Finding ID: BT-RETR-05
Severity: **Low**
Status: **Open**

Description

The total number of `lp_token` tokens a user can claim is calculated as `(reserve_amount/TOTAL_LP_RESERVE) * TOTAL_LP_TOKEN_LOCKED`. This is a decimal number, and it is then converted to `uint128`. Because decimal numbers are always rounded down by `cosmos_sdk uint128`, there can be left-over `lp_tokens` in the contract.

Proof of Issue

File name: convert-lock/src/contract.rs **Line number:** 367-368

```
    let lp_token_share = Decimal::from_ratio(amounts.1,
TOTAL_LP_RESERVE.load(deps.storage)?);     let
lp_token_amount = lp_token_share *
TOTAL_LP_TOKEN_LOCKED.load(deps.storage)?;
```

Severity and Impact Summary

No users can claim any tokens and left-over retro tokens are locked in the contract.

Recommendation

We recommend that the team either implement a round function to return the nearest `u128` to a decimal number or add another function that allows the owner to transfer the leftover retro tokens out of the contract.

Edge case: Potential left-over retro tokens in IDO contract

Finding ID: BT-RETR-06
Severity: **Low**
Status: **Open**

Description

The total number of retro tokens a user can claim is calculated as `(USTDeposits_theirs/USTDeposits_total) * RETRODeposits_total`. This is a decimal number, and it is then converted to `uint128`. Because decimal numbers are always rounded down by `cosmos_sdk uint128`, leftover retro tokens exist, and it gets worse if the `USTDeposits_total > RETRODeposits_total`.

Here is a worst-case scenario: There are 101 users and each deposits 1 UST and there are total 100 retro tokens. Each user can claim (1/101 * 100) = 0.99 which will be rounded down to 0. Thus, no users can claim any tokens and all retro tokens are locked in the contract.

Proof of Issue

The total number of retro tokens a user can claim is calculated as follows:

File name: ido/src/contract.rs **Line number:** 274-275

```
let total_retro_deposits: Uint128 =
TOTAL_RETRO_DEPOSITS_PHASE_ONE.load(deps.storage)?;        let
user_claim:  Uint128  =  Decimal::from_ratio(existing_amount,
current_deposits_total) * total_retro_deposits;
```

Here is our test: There are two users. First user deposited 1 UST and second user deposited 100 USTs and there are 100 retro tokens. The first user will claim (1/101 * 100) = 0.99 which is rounded down to 0.

```
fn  test_claim_retro_rounded_down()  {              let  mut  deps  =
mock_dependencies(&coins(2, "token"));

        let msg = InstantiateMsg {              retro_token:
"retro_token".to_string(),              retrograde_treasury:
"retrograde_treasury".to_string(),              phase_two_length:
Uint64::from(10u32),              phase_three_length: Uint64::from(5u32),
retro_lockup_period: Uint64::from(5u32),
        };          let info = mock_info("owner", &coins(1000, "uusd"));
let user_info = mock_info("user", &coins(1, "uusd"));          let
second_user_info = mock_info("second_user", &coins(100,
"uusd"));                let  res  =  instantiate(deps.as_mut(),  mock_env(),
info.clone(), msg).unwrap();
```

```
        let mut inner_mock_env = mock_env();
inner_mock_env.block.time = Timestamp::from_seconds(9000u64);          let
_res = execute(deps.as_mut(), inner_mock_env.clone(), info.clone(),
ExecuteMsg::StartIDO {start_time: Uint64::from(10000u64)}).unwrap();
inner_mock_env.block.time = Timestamp::from_seconds(10000u64);

        let mut phase_three_env = mock_env();
         phase_three_env.block.time =
Timestamp::from_seconds(inner_mock_env.block.time.seconds() + 10);
let mut phase_four_env = mock_env();
        phase_four_env.block.time =
Timestamp::from_seconds(inner_mock_env.block.time.seconds() + 15);
let mut lockup_expired_env = mock_env();
lockup_expired_env.block.time =
Timestamp::from_seconds(inner_mock_env.block.time.seconds() + 20);

        let _deposit_res = execute(deps.as_mut(), inner_mock_env.clone(),
user_info.clone(), ExecuteMsg::DepositUST {}).unwrap();          let
_second_deposit_res = execute(deps.as_mut(), inner_mock_env.clone(),
second_user_info.clone(), ExecuteMsg::DepositUST {}).unwrap();

        let  claim  =  execute(deps.as_mut(),  lockup_expired_env.clone(),
user_info.clone(), ExecuteMsg::ClaimRetro {}).unwrap();
        // First user gets 0 retro token          let
transfer_cw20_msg = Cw20ExecuteMsg::Transfer {
recipient: "user".to_string(),              amount:
Uint128::from(00u32),
        };          let transfer_execute_msg =
WasmMsg::Execute {              contract_addr:
"retro_token".to_string(),              msg:
to_binary(&transfer_cw20_msg).unwrap(),              funds:
vec![],
        };          assert_eq!(claim.messages[0],
SubMsg::new(transfer_execute_msg));
```

Severity and Impact Summary

No users can claim any tokens and left-over retro tokens are locked in the contract.

Recommendation

We recommend that the team either implement a round function to return the nearest `u128` to a decimal number or add another function that allows the owner to transfer the leftover retro tokens out of the contract.

Reward_unlock_time is allowed to be set in the past in staking contract

Finding ID: BT-RETR-07
Severity: **Low**
Status: **Open**

Description

The function `update_unlock_time` never checks the input `reward_unlock_time`. As a result, the owner can set the unlock time in the past.

Severity and Impact Summary

If the unlock time is set in the past, users can claim reward immediately which is not the intended design.

Recommendation

We recommend that the team validates whether the input `reward_unlock_time` is current or not.

lp_token is allowed to be reset multiple times

Finding ID: BT-RETR-08
Severity: **Low**
Status: **Open**

Description

The owner can set `lp_token` multiple times in `execute_set_lp_token` in the convert-lock contract. The current design should only allow the `lp_token` to be set once because the `TOTAL_LP_TOKEN_LOCKED` is never updated when the `lp_token` is updated.

Proof of Issue

The function never checks whether the `config.lp_token` is already set. It only ensures that the owner can't change `lp_token` after the unlocks have already begun which implies that the owner can change `lp_token` many times before the unlocks time.

File name: convert-lock/src/contract.rs **Line number:** 246-248

```
    if Uint64::from(env.block.time.seconds()) >= deposit_end_time + shortest_lockup_length { return Err(ContractError::Std(StdError::generic_err("Unlocks have already begun. Can't change LP token address."))); }
```

Severity and Impact Summary

A malicious owner can reset `lp_token` but the `TOTAL_LP_TOKEN_LOCKED` is not updated which results incorrect calculations and prevents users from withdrawing `derivative_tokens` and `lp_tokens`.

Recommendation

We recommend that the team add the following code to check if `config.lp_token` is already set.

```
let lp_token = config.lp_token.unwrap_or(Addr::unchecked("")); if lp_token != Addr::unchecked(""){
    Err(ContractError::Std(StdError::generic_err("lp_token is already set.")))
}
```

Use of unwrap() may create panic

Finding ID: BT-RETR-09
Severity: **Informational**
Status: **Open**

Description

Functions `receive_incoming_asset`, `execute_withdraw_user_reward`, and `execute_withdraw_derivative_token` in the convert-lock contract use `DEPOSIT_PERIOD_START_TIME.load(deps.storage)?.unwrap();` and `DEPOSIT_PERIOD_END_TIME.load(deps.storage)?.unwrap();`. They can be called by users before the open deposit period and it will panic because `unwrap()` does not handle the None cases.

Similarly, the function `try_withdraw_token` in the treasury smart contract assumes that the `TOKEN_MANAGER` is not None (`TOKEN_MANAGERS` is not initialized in `instantiate`) and `unwrap()` will panic.

Proof of Issue

File name: convert-lock/src/contract.rs **Line number:** 151-152,306,344

```
    let start_time =
DEPOSIT_PERIOD_START_TIME.load(deps.storage)?.unwrap();     let
end_time = DEPOSIT_PERIOD_END_TIME.load(deps.storage)?.unwrap();
```

File name: treasury/src/contract.rs **Line number:** 190-196

```
    if !TOKEN_MANAGERS
        .load(deps.storage, info.sender.clone())
        .unwrap()
        .contains(&deps.api.addr_validate(&token)?)
        {              return
Err(ContractError::Unauthorized {});          }
```

Severity and Impact Summary

Functions will panic because `unwrap()` does not handle the None cases. `Cosmoswasm` expects a response or an error; panic can lead to unexpected behaviors.

Recommendation

We suggest that the team use `unwrap_or_else()` instead.

execute_add_minter allows the input minter to be the owner in cw20-minterset

Finding ID: BT-RETR-10
Severity: **Informational**
Status: **Open**

Description

The function `execute_add_minter` allows the minter to be the owner. Thus, the `minter.cap` can be set randomly and `owner.cap` in `MINTERS` is therefore different from the cap in `TOKEN_INFO`.

Proof of Issue

File name: cw20-minter-set.rs **Line number:** 495-515

Severity and Impact Summary

Even though `minter.cap` is not used anywhere, the owner should have the same cap in `TOKEN_INFO` and `MINTERS`.

Recommendation

We recommend that the team checks if the input minter is the owner and discard the changes.

lp_reserve_ratio in convert-lock contract is not validated

Finding ID: BT-RETR-11
Severity: **Informational**
Status: **Open**

Description

The whole contract assumes and requires that `lp_reserve_ratio` is less than 1, but there is no such validation to ensure that it is in (0,1) in the `instantiate` function.

Proof of Issue

File name: convert-lock/src/contract.rs **Line number:** 23-41

Severity and Impact Summary

The ratio can be accidentally set to greater than 1 and then all incoming asserts will be reserved.

Recommendation

We suggest that the team check if `lp_reserve_ratio` is in (0,1).