CANTINA

Ton pool contracts

Competition

March 7, 2025

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
3.1 High Risk . . . 4
3.1.1 DoS via `ctx_nominators` can happen . . . 4
3.1.2 Error along manual recover call can result in fund loss . . . 5
3.2 Medium Risk . . . 5
3.2.1 Legacy Stake Lockup Extension Vulnerability in Nominator Contract . . . 5
3.2.2 Force kick does not add `receipt_price` to withdraw amount . . . 8
3.2.3 Nominator can block withdrawals of other nominators due to missing try-catch in `op_controller_accept_withdraws` . . . 8
3.2.4 Incorrect profit distribution in else case in `op_elector_recover_response` . . . 9
3.2.5 `proxy_stake_held_for` gets incorrectly reset in `op_elector_stake_response_fail` causing incorrect calculation of staking times . . . 10

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Chorus One provides enterprise-grade staking integrations for institutions and forward thinking investors, industry-leading research, and support for 50+ protocols.

From Jan 13th to Jan 27th Cantina hosted a competition based on ton-pool-contracts. The participants identified a total of **44** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 2
- Medium Risk: 5
- Low Risk: 23
- Gas Optimizations: 0
- Informational: 14

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 DoS via `ctx_nominators` can happen

Submitted by 0xTheBlackPanther, also found by zubyoz, n4nika, CyberGul and mmujtabaroohani

Severity: High Risk

Context: nominators.fc#L524, nominators.fc#L705

Description: The contract's `ctx_nominators` dictionary can grow unboundedly as any user can become a nominator by meeting the minimum stake requirement. TON contracts have cell count limits, making this an exploitable DOS vector.

Current implementation:

```
() member_stake_deposit(int value) impure {
    throw_unless(error::invalid_stake_value(), value > 0);
    add_member_pending_deposit(value);
    // No limit on total nominators @audit-poc
}
```

An attacker could create multiple addresses, make minimum stake deposits (1_ TON each)_ and easily bloat contract storage with dictionary entries. While storage fees provide some economic protection, they don't prevent arbitrary dictionary growth. This can lead to contract hitting cell count limits. Also this increases gas costs for operations and potential service disruption.

Recommendation: Add nominator limit:

```
int constant MAX_NOMINATORS = 1000;

() member_stake_deposit(int value) impure {
    int current_nominators = count_nominators();
    throw_unless(error::max_nominators(), current_nominators < MAX_NOMINATORS);
    // ... rest of function
}
```

Also implement cleanup:

```
() cleanup_inactive_nominators() impure {
    // Remove nominators with balance < threshold
    // Or inactive > X time
}
```

Alternatively, add emergency mode when approaching limits:

```
() check_storage_limits() impure {
    if (get_cells_count() > CRITICAL_THRESHOLD) {
        enable_emergency_mode();
    }
}
```

Chorus One: Accepted. Chorus One executed a test to evaluate the effect of full contract storage and its implications. When the maximum storage capacity is reached, no new nominators can be added.

Impact Assessment: Very Low.

1. Storage becoming full does not affect the operations of nominators who have already delegated.
2. As nominators exit, they free up storage space, allowing for the addition of new nominators.

Controls:

1. A minimum stake is required to join and remain in the pool. This minimum stake amount can be increased, if required.
2. There is a force exit function to remove wallets. If low delegation wallets become inactive we can exit them from the pool to free up space.

3.1.2 Error along manual recover call can result in fund loss

Submitted by ArsenLupin

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Contract could become unusable and permanently locked if manual recover would not succeed and the msg would be returned back to the elector.

Finding Description: The enough time has passed, and the stake can be recovered from the elector. Nominator calls `op_stake_manual_recover`. However, nominators provides not enough gas:

```
int gas_limit = in_msg~load_coins();
set_gas_limit(gas_limit);
```

It can lead to:

1. Partial execution.
2. Revert along the way which can bounce back to the elector.

Message is sent to the elector. Elector returns stake via `recover_stake`. During the stake recovering the elector sends the message in the bouncable mode (`0x18`). Because of the OOS the following can occur:

1. The msg is bounced back to the elector.
2. Partial execution happens.

So, we end up either in inconsistent state, or in the msg bounced back to the elector.

Once the nominator tries recover the stake one more time, he couldn't do it because the credits in the elector is already deducted and dictionary is deleted. Once the bounceble message reaches the elector, it simply returns.

```
() recover_stake(int op,
                 slice s_addr,
                 int query_id) impure inline_ref {

  (int src_wc, int src_addr) = parse_std_addr(s_addr);

  if (src_wc != - 1) {
    ;; not from masterchain, return error
    return send_message_back(s_addr, op::not_allowed, query_id, op, 0, 64);
  }
  slice ds = get_data().begin_parse();
  (cell elect, cell credits) = (ds~load_dict(), ds~load_dict());
  (slice cs, int f) = credits~udict_delete_get?(256, src_addr); //if found, the credits are deleted.
  ...
```

So, we end up in the situation, where we can't recover the stake and the pool become locked forever, and any further stake is not possible.

Impact Explanation: If the error and exception thrown happens along the call elector-proxy-pool, the msg will be bounced back and the user can never claim the funds back. They will be lost.

If the partial execution happens due to not enough gas provided the pool result in the inconsistent state, without an option of being unlocked.

Likelihood Explanation: High.

Recommendation: Do not allow to provide the arbitrary gas value for the nominators.

Chorus One: Fixed. The option for nominators to call Recover Stake has been removed.

3.2 Medium Risk

3.2.1 Legacy Stake Lockup Extension Vulnerability in Nominator Contract

Submitted by Lamsy, also found by zubyoz

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `get_staking_status()` method incorrectly extends lock periods for legacy stakes due to flawed calculation in `lockup_lift_time`, preventing timely withdrawals and violating agreed staking terms.

Finding Description: The nominator contract's staking status checker uses lockup_lift_time to determine when stakes can be withdrawn. For legacy stakes (stake_held_for = 0), the function incorrectly calculates unlock time using cycle start time (timeSince) as base, potentially extending lock periods beyond agreed duration.

```
if (stake_at <= timeSince) {
    if (stake_held_for == 0) {  // Legacy stake
        return timeSince + stake_lock_duration + params::stake_held_safety();
        // Uses cycle start (timeSince) instead of respecting original unlock time
    }
}
```

Impact Explanation:

- Directly affects fund accessibility.
- Violates staking agreement terms.
- Extends lock periods without authorization.
- Impacts all legacy stakes in system.
- No user workaround available.

Likelihood Explanation:

- Triggers for all legacy stakes where `stake_at < timeSince`.
- Affects every withdrawal attempt for these stakes.
- Condition occurs naturally in normal operation.
- No existing mitigation measures.

**Proof of Concept:

```
;; Constants
const int SAFETY = 300;  // 5 minutes safety margin

;; Path 1: Previous Cycle - New Staking Rules
{
    // Input values
    stake_at = 1000;            // Stake placed at t=1000
    stake_until = 2000;         // Original unlock at t=2000
    stake_held_for = 500;       // New staking rules (>0)

    // Previous cycle
    timeSince = 1100;           // Cycle start
    timeUntil = 1900;           // Cycle end

    // Execution path
    if (1000 <= 1100) {         // True - enters previous cycle check
        if (500 > 0) {          // True - new staking rules
            result = 1900 + 500 + 300 = 2700
        }
    }
}

;; Path 2: Previous Cycle - Legacy Stake
{
    stake_at = 800;             // Earlier stake
    stake_until = 1800;         // Original unlock
    stake_held_for = 0;         // Legacy stake
    stake_lock_duration = 1000; // (1800 - 800)

    timeSince = 1000;
    timeUntil = 1800;

    if (800 <= 1000) {          // True - enters previous cycle
        if (0 > 0) {            // False - legacy path
            // Not reached
        } else {
            result = 1000 + 1000 + 300 = 2300  // ISSUE: Extends beyond stake_until
        }
    }
}

;; Path 3: Current Cycle - New Staking Rules
{
    stake_at = 1500;
    stake_until = 2500;
    stake_held_for = 600;

    // Previous cycle check fails
    timeSince = 1600;
    timeUntil = 2400;

    // Current cycle
    timeSince = 1400;
    timeUntil = 2200;

    if (1500 <= 1400) {        // False - skips current cycle
        // Not reached
    }

    result = 2500 + 300 = 2800 // Falls to default
}

;; Path 4: Current Cycle - Legacy Stake
{
    stake_at = 1200;
    stake_until = 2200;
    stake_held_for = 0;
    stake_lock_duration = 1000;

    // Previous cycle misses
    timeSince = 1300;
    timeUntil = 2100;

    // Current cycle hits
    timeSince = 1100;
    timeUntil = 1900;

    if (1200 <= 1100) {        // False - skips current cycle
        // Not reached
    }

    result = 2200 + 300 = 2500 // Falls to default
}

;; Path 5: Future Stake
{
    stake_at = 2000;
    stake_until = 3000;
    stake_held_for = 400;

    // Both cycles miss
    current_timeSince = 1800;
    current_timeUntil = 2600;

    result = 3000 + 300 = 3300 // Default case
}
```

Recommendation:

```
int lockup_lift_time(int stake_at, int stake_until, int stake_held_for) {
    int stake_lock_duration = stake_until - stake_at;
    var (timeSince, timeUntil) = get_previous_cycle();

    if (stake_at <= timeSince) {
        if (stake_held_for > 0) {
            return timeUntil + stake_held_for + params::stake_held_safety();
        } else {
            // Legacy stake - respect original unlock time
            return min(
                timeSince + stake_lock_duration + params::stake_held_safety(),
                stake_until + params::stake_held_safety()
            );
        }
    }
    // ... rest of function
}
```

Chorus One: Accepted. Works as intended. The elector decides when stake recovery can be run.

3.2.2 Force kick does not add `receipt_price` to withdraw amount

Submitted by Boy2000

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: When a controller tries to force kick a member from the pool, he does not add `receipt_price` to the `withdrawed` amount.

Finding Description:

- nominators.fc#L1515C17-L1515C27:

```
;; Withdraw everything
var (withdrawed, all) = member_stake_withdraw(0);
throw_unless(error::invalid_message(), withdrawed > 0);
throw_unless(error::invalid_message(), all);

;; Forced kick
send_empty_std_message(
    serialize_work_addr(member),
@>    withdrawed,
    send_mode::default(),
    op::force_kick::notification(),
    ctx_query_id
);
```

It's unclear why an operator would want to kick a member, perhaps reducing storage fees, or legal reasons (member is sanctioned etc), in such cases if the `withdrawed` amount is less than tx cost, it will fail. This can happen if member deposits X amount, then withdraws most of it, leaving a small amount behind.

For accounting reasons it is also expected the member to receive back the full amount in a kick event.

Impact Explanation:

- Force kick may fail.
- Withdrawed amount would be less than owned.

Recommendation:

- Use `withdrawed + receipt_price`.
- Don't allow withdraw if remaining balance is < `min_stake`.

Chorus One: Accepted. Works as intended. An administration fee is charged for executing this function.

3.2.3 Nominator can block withdrawals of other nominators due to missing try-catch in `op_controller_accept_withdraws`

Submitted by n4nika, also found by Rhaydden, zubyoz and ArsenLupin

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The controller can accept multiple withdrawals within one transaction which can be seen since `op_controller_accept_withdraws` iterates over a `members` dict, accepting withdrawals for all of them. Since there is no error handling in the `do-until` loop, this allows a nominator to block withdrawals of other nominators.

Finding Description: The code in question can be seen here:

```
() op_controller_accept_withdraws(int value, slice in_msg) impure {

    ;; [...]

    ;; Parse message
    var members = in_msg~load_dict();
    in_msg.end_parse();

    ;; Process operations
    var member = -1;
    do {
        (member, var cs, var f) = members.udict_get_next?(256, member);
        if (f) {
            ;; Accept member's stake
            load_member(member);
            member_accept_withdraw();
            store_member();
        }
    } until (~ f);
    store_base_data();
}
```

Effectively we call `member_accept_withdraw` on every member the controller wants to accept the withdrawal of. Looking at `member_accept_withdraw`, we see that it fails if the member has no pending rewards.

```
() member_accept_withdraw() impure {

    ;; Checks if there are pending withdrawals
    throw_unless(error::invalid_message(), ctx_member_pending_withdraw > 0);
    ;; [...]
}
```

This allows for the following attack:

1. Nominator (`alice`) stakes and unstakes afterwards.
2. `alice` waits until stake is accepted.
3. `alice` requests unstake which seits their `ctx_member_pending_withdraw` to the previously staked value.
4. Controller has multiple unstake requests and batches them into one transaction (including our `alice`'s).
5. `alice` now frontruns the controller's transaction and stakes again.
    - This sets `alice's` `ctx_member_pending_withdraw` to zero which can be seen in nominators.fc#L705-L712.
6. Now the controller's transaction fails because `alice`'s `ctx_member_pending_withdraw` is zero and the error thrown in `member_accept_withdraw` is not handled properly.

Impact: This allows a malicious nominator to consistently block other nominators' withdrawals causing them a loss of funds.

Recommendation: Consider wrapping the call to `member_accept_withdraw` in a `try-catch`.

Chorus One: Fixed by added error handling to avoid failure.

3.2.4 Incorrect profit distribution in else case in `op_elector_recover_response`

Submitted by n4nika, also found by Kral01

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: In the `else` branch in `op_elector_recover_response` we distribute the full value received by the elector (minus fees) instead of only distributing the profits. This then allows nominators to withdraw approximately twice as many coins than they should be allowed to.

Finding Description: In order for this to occur, we need to consider multiple "issues" in the codebase with very low singular impact which, when combined, allow this to be triggered.

Main vulnerability: The most straightforward part is the incorrect distribution done in `op_elector_recover_response`. Here we call `distribute_profits` with `value - fees` while we should only distribute the actual profit (`value - fees - ctx_balance_sent`):

```
var profit = value - fees::stake_fees();
if (profit > 0) {
    distribute_profit(profit);
}
```

Second vulnerability: `op_controller_stake_send` can be called to send more stake to the elector even if `proxy_stake_lock_final` is already set to `true`. For context, this is set to `true` before we send the `recover_stake` message to the elector. Currently it is always reset to `false` when the controller calls `op_controller_stake_send`, allowing for stake to be sent AFTER a recover request was sent. Allowing this makes no sense since we should always wait until we received the response for a pending recover request.

Scenario:

- Coordinator sends stake (send and await confirmation) (`op_controller_stake_send`).
- We wait until `proxy_stake_until < now()` (requirement for `op_stake_recover`).
- Controller sends recover funds message (`R1`) (`op_stake_recover`).
- Controller sends stake (this sets `proxy_stake_lock_final = false`) which now allows a new call to `op_stake_finalize`.
- Controller calls `op_stake_finalize`, this increases `proxy_stake_until` by a small amount.
- Elector responds to `R1` and sends funds to our contract (`op_elector_recover_response`).
- Due to the repeated call to `op_stake_finalize`, the following is now true: `proxy_stake_until > now()`.
- Therefore we enter the `else` branch in `op_elector_recover_response`.
- All the stake gets redistributed instead of only the profit.
- Nominators can withdraw disproportionally large amounts (we distributed more in profits than we have in actual coins).

Impact: This is very unlikely to happen making the likelihood definitely `low`. The direct impact, however, is a direct loss of funds making it clearly `high` putting this at an overall severity of `medium`.

Recommendation: Consider the following:

- Don't allow `op_controller_stake_send` to be called if `proxy_stake_lock_final == true`.
- Properly distribute only the profit, not the stake in the mentioned `else` statement.

Chorus One: Fixed by addeding workflow managenent to allow only a single message to the Elector at a time.

3.2.5 `proxy_stake_held_for` gets incorrectly reset in `op_elector_stake_response_fail` causing incorrect calculation of staking times

Submitted by n4nika, also found by shaflow01

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Since `proxy_stake_held_for` gets set to zero even if a stake fails which was only an update for a previous stake, this leads to incorrect updates of `proxy_stake_until` in `lockup_finalized_time`.

Finding Description: Whenever a `elector::stake::request()` message fails, the elector calls `op_elector_stake_response_fail`. There we then differentiate between two cases.

- The rejected message contained the full staking value we sent.
- The rejected message was an "update" message for a previously sent stake.

This can be seen here:

```
() op_elector_stake_response_fail(int value, slice in_msg) impure {
    ;; [...]
    if (proxy_stake_sent == 0) {
        proxy_stake_at = 0;
        proxy_stake_until = 0;
        proxy_stake_sent = 0;
        proxy_stake_held_for = 0;
        proxy_stake_lock_final = 0;
        on_unlocked();
    }
    proxy_stored_query_id = 0;
    proxy_stored_query_op = 0;
    proxy_stored_query_stake = 0;
    proxy_stake_held_for = 0; ;; [1]
    proxy_stake_lock_final = 0;

    ;; [...]
}
```

Now the problem is, that as seen at `[1]`, we set `proxy_stake_held_for = 0`, even if the rejected stake was only an update. This means we can get the following scenario:

- Send stake to elector (e.g. `100000 TON`).
- Update stake by sending, for example, `10000 TON` more to the elector.
- That update message gets rejected and the elector calls `op_elector_stake_response_fail`.
- This sets `proxy_stake_held_for = 0` while we still have `100000 TON` staked.
- If we now want to recover our funds by calling `op_stake_recover`.
- which calls `stake_lock_finalize`.
- which calculates `proxy_stake_until` based on `proxy_stake_held_for`.

```
int finalized = lockup_finalized_time(proxy_stake_at, proxy_stake_until, proxy_stake_held_for);
```

```
int lockup_finalized_time(int stake_at, int stake_until, int stake_held_for) {
    if (stake_held_for == 0) {
        return stake_until; ;; Legacy
    } else {
        ;; Find previous cycle
        var (timeSince, timeUntil) = get_previous_cycle();
        if (stake_at <= timeSince) {
            return max(timeUntil + stake_held_for, stake_until);
        } else {
            return -1;
        }
    }
}
```

Note that `stake_held_for` being 0 changes the calculation to directly return `stake_until` which doesn't necessarily line up with the schedule in the `elector` contract.

Impact: This causes our effective locktime (returned by `lockup_finalized_time`) to be shorter than expected. In the worst case this causes us to try and recover funds from the elector before they are unlocked, meaning our message will fail and we never get a `op_elector_recover_response`.

Recommendation: Only reset `proxy_stake_held_for` in the if-statement `if (proxy_stake_sent == 0)`.

Chorus One: Mitigation: only reset proxy_stake_held_for when stake gets to zero.