Security Audit Report

Hatom booster-v2
MultiversX smart contract

by ARDA

on March 4, 2025

Table of Content

Disclaimer

The report makes no statements or warranties, either expressed or implied, regarding the security of the code, the information herein or its usage. It also cannot be considered as a sufficient assessment regarding the utility, safety and bugfree status of the code, or any other statements.

This report does not constitute legal or investment advice. It is for informational purposes only and is provided on an "as-is" basis. You acknowledge that any use of this report and the information contained herein is at your own risk. The authors of this report shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Terminology

Code: The code with which users interact.

Inherent risk: A risk for users that comes from a behavior inherent to the code's design.

Inherent risks only represent the risks inherent to the code's design, which are a subset of all the possible risks. **No inherent risk doesn’t mean no risk.** It only means that no risk inherent to the code's design has been identified. Other kind of risks could still be present. For example, the issues not fixed incur risks for the users, or the upgradability of the code might also incur risks for the users.

Issue: A behavior unexpected by the users or by the project, or a practice that increases the chances of unexpected behaviors to appear.

Critical issue: An issue intolerable for the users or the project, that must be addressed.

Major issue: An issue undesirable for the users or the project, that we strongly recommend to address.

Medium issue: An issue uncomfortable for the users or the project, that we recommend to address.

Minor issue: An issue imperceptible for the users or the project, that we advise to address for the overall project security.

Objective

Our objective is to share everything we have found that would help assessing and improving the safety of the code:

1. The **inherent risks** of the code, labelled R1, R2, etc.
2. The **issues** in the **code**, labelled C1, C2, etc.
3. The **issues** in the **testing** of the code, labelled T1, T2, etc.
4. The **issues** in the **other** parts related to the code, labelled O1, O2, etc.
5. The **recommendations** to address each issue.

Audit Summary

Initial scope

- **Repository:** https://github.com/HatomProtocol/hatom-rewards-booster/
- **Commit:** `3fbbcb87e9fdfbf95bda3a6aa9afe3ae7923db33`
- **MultiversX smart contract path:** ./rewards-booster/

Final scope

- **Repository:** https://github.com/HatomProtocol/hatom-rewards-booster/
- **Commit:** `83a2c53192fc89a904c823bc712b29011fd97ed7`
- **MultiversX smart contract path:** ./rewards-booster/

5 inherent risks in the final scope

3 issues in the final scope

19 issues reported in the initial scope and 3 remaining in the final scope:

Severity	Reported			Remaining		
	Code	Test	Other	Code	Test	Other
Critical	0	0	0	0	0	0
Major	4	0	0	0	0	0
Medium	5	3	0	3	0	0
Minor	7	0	0	0	0	0

Inherent Risks

R1: Users won't earn boosted and extra rewards on their pool tokens (i.e., collateral or stake in USH Staking) which are not registered in the Booster.

In order for a user's pool token to be registered in the Booster:

- the Hatom team should have whitelisted his pool token in the Booster,
- the user must do an action that registers his pool token, e.g. staking / unstaking some HTM equivalent, claiming rewards, increasing / decreasing his pool token's position, etc.

As long as a user's pool token is not registered in the Booster, the user won't earn boosted and extra rewards for that token. This is because, at the time the user's pool token is registered, the rewards that the user would have earned if his pool token were registered since the beginning have not been reserved for the user, and thus they might have already been distributed to other users.

R2: Users may not be able to claim rewards as HTM if they claim too late.

This is because the contract has only a limited amount of rewards that can be converted to HTM.

Example: Let's say that if Alice claims now, she would be able to claim rewards as HTM. However, if she rather decides to claim one week later, it is possible that she may not be able to claim rewards as HTM anymore, for instance in the following cases:

- Other users have claimed rewards as HTM during the week, and there are not enough remaining rewards that can be converted to HTM for Alice.
- No other users claimed during the week, but Alice's rewards have increased and may have now exceeded the contract's amount of rewards that can be converted to HTM.

R3: Users might earn less boosted and extra rewards over a period of time depending on when they claim during that period.

This is because the computation of the boosted and extra rewards of a user since his last interaction is based on values that increase / decrease over time:

- the price of the tokens he staked in the Booster,
- the price of the tokens he staked in the USH Staking or the price of the tokens he deposited as collateral in the Controller,
- the staking ratio thresholds of each pool token,
- the duration during which rewards batches were active for each pool token,
- the capping of the boosted and extra compliances to 1.

Example for boosted rewards:

Scenario 1:

- At $t = 0$, Alice has 1 unit of position in the pool A that has a price of 100$ and 1 unit of stake that has a price of 20$. The pool A has a staking ratio threshold of 10% and has a rewards batch.
- At $t = 1$, the price of Alice's position is now 200$ and the price of her stake is now 10$. The pool A now has a staking ratio threshold of 20% and still has a rewards batch.
- At $t = 2$, Alice claims her rewards.

Over the period of time $[0, 2]$, in average, the price of Alice's position is 150$, the price of Alice's stake is 15$, and the staking ratio threshold is 15%. Therefore, over this period, the boosted compliance of Alice is $\max\left(\frac{15\$}{15\% \times 150\$}, 1\right) = \max(0.67, 1) = 0.67$, which means she will earn 67% of the maximum boosted rewards she could have earned for her position in the pool A.

Scenario 2: The same thing as in scenario 1 happens but the only difference is that Alice claims also at $t = 1$.

Over the period of time $[0, 1]$, in average, the price of Alice's position is 100$, the price of Alice's stake is 20$, and the staking ratio threshold is 10%.

Therefore, over this period, the boosted compliance of Alice is $\max\left(\frac{20\$}{10\% \times 100\$}, 1\right) = \max(2, 1) = 1$, thus she will earn 100% of the maximum boosted rewards she could have earned for her position in the pool A.

Over the period of time $[1, 2]$, in average, the price of Alice's position is 200$, the price of Alice's stake is 10$, and the staking ratio threshold is 20%. Therefore, over this period, the boosted compliance of Alice is $\max\left(\frac{10\$}{20\% \times 200\$}, 1\right) = \max(0.25, 1) = 0.25$, thus she will earn 25% of the maximum boosted rewards she could have earned for her position in the pool A.

So over the whole period of time $[0, 2]$, she would have earned only 62.5% of the maximum boosted rewards she could have earned compared to 67% in the first scenario.

R4: Users might earn less boosted and extra rewards in case price oracles malfunction.

This is because the compliance applied to the user's boosted and extra rewards is determined by the relative prices of staked and collateral tokens, which are provided by oracle sources, and there is no guarantee that these sources will not be manipulated, will function continuously, and will provide accurate data.

Here are some sources of errors in prices used in the Booster:

- There is no guarantee that ESDT prices provided by Hatom Oracle to the Booster are accurate, because they are obtained by aggregating prices given by off-chain bots, which can be manipulated, stop functioning or provide inaccurate data. Additionally, although the Oracle may partially mitigate this risk by not providing its price if it is too far from the xExchange safe price, this mitigation mechanism might not always be activated.
- There is no guarantee that AshSwap LP token prices will be accurate as they are based on the liquidity pool's reserves, which can be outdated or manipulated.

- There is no guarantee for a token (e.g. an xExchange farm token) whose value is declared to be equal to another “mirror” token (e.g. the underlying xExchange LP token), that the actual price of this token is equal and will always remain equal to that of the “mirror” token.
- There is no guarantee that prices in the Booster will always be up-to-date as in some situations the last saved price is used instead of querying a fresh price from oracle sources.

R5: Users might lose their unsaved claimable rewards for a rewards batch as soon as 95% of the batch’s total claimable rewards have been saved.

At each block, the claimable rewards $R_C(U)$ for each user U are increased. The claimable rewards can be computed at any time, for any user.

However, they are not automatically saved in storage at each block for all users, since doing so would cost too much gas.

The claimable rewards of a user U are saved in storage as soon as a user (U or anybody) saves them in storage. In practice, the user U doesn’t need to explicitly save them in storage as most of his interactions with the protocol already save them for him under the hood.

Let’s note $R_S(U)$ the rewards saved in storage for a user U. The saved rewards $R_S(U)$ are always less than or equal to the claimable rewards $R_C(U)$.

The Hatom team is allowed to remove a rewards batch as soon as the total saved rewards $\sum_U R_S(U)$ is greater than 95% of the total claimable rewards $\sum_U R_C(U)$.

While a rewards batch is not removed, a user U can claim all his claimable rewards $R_C(U)$. Once a rewards batch is removed, he will only be able to claim his saved rewards $R_S(U)$, and so he won’t be able to claim his unsaved rewards anymore.

Note: When Hatom removes a rewards batch, the compliance of users would continue to be computed as if the rewards batch had not been removed.

Code Issues & Recommendations

Since the code is not open-source, only the remaining issues are published.

C6: Money market interests might not be accrued for too long and lead to inaccurate collateral prices

Severity: Medium **Status:** Won't fix

Location

```
rewards-booster/src/proxies.rs
   get_stored_exchange_rate
```

Description

The price of a collateral token can be significantly underestimated if borrowing interests in the money market have not been accrued for an extended period. This would in turn lead to overestimated users' rewards.

More precisely, when the contract computes the collateral price, it uses an exchange rate from the money market obtained through `get_stored_exchange_rate`. However, this method gives an exchange rate with non-accrued borrowing interests. Therefore, if borrowing interests have not been accrued for an extended period, it can lead to an underestimation of the collateral price. This underestimation can then cause an overestimation of the price integral, potentially inflating users' rewards.

Recommendation

We recommend that `get_price_ratio` first calls the money market endpoint `try_accrue_interest` before calling `get_stored_exchange_rate` to get the exchange rate from the money market. This ensures that borrowing interests are sufficiently up-to-date, providing a more accurate exchange rate for collateral price computation, thus preventing significant overestimations of rewards.

Resolution Notes

The issue has not been fixed.

C7: “unstake” and “on_market_change” might consume too much gas

Severity: Medium **Status:** Won't fix

Description

Current behavior: Gas costs of `unstake` and `on_market_change` endpoints could be too high, which would make transactions fail to be executed or fail to be included in a block. This is because these endpoints perform many iterations:

- A transaction that includes `on_market_change` will iterate over all the money markets of the user, all the money markets’ rewards batches and all the user’s staked j-tokens. Additionally, when it is called through the Controller for liquidation or withdrawing collateral, there are further iterations over all the user’s money markets and rewards batches in the Controller.
- A transaction that includes `unstake` will also iterate over all the money markets of the user, all the money markets’ rewards batches and all the user’s staked j-tokens. Furthermore, it needs to call external price oracles to update each price information.

Expected behavior: Gas costs of `unstake` and `on_market_change` endpoints should be controlled and known to be reasonable in the worst case scenario, i.e. the half-block limit of `300M` gas. Indeed, if `unstake` and `on_market_change` endpoints are too gas consuming, then this will prevent users from withdrawing stake and collateral, and from being liquidated.

Moreover, previous devnet simulations with the Booster and the Controller have shown that the current endpoints are already consuming a significant amount of gas. Thus now that interactions with the Controller in `on_market_change` require to iterate over all the user’s collateral tokens, the overall gas cost might have increased even more in some cases.

Worst consequence: If this gas cost issue exists, an attacker could exploit it by borrowing a huge amount and becoming impossible to liquidate, as any liquidation transaction would run out of gas due to the big gas cost of `on_market_change`.

Recommendation

We suggest doing the following set of system tests once all changes in the contract to account for other issues of the report are finalized, in order to ensure we properly evaluate the gas costs of the endpoints in their final state.

At a high-level, we recommend running a system test of the worst case scenario on devnet in order to check if gas costs are reasonable, i.e. less than the half-block limit (`300M` gas), and sharing these transactions with the auditor. Subsequently, if we observe that gas costs are too high, then we suggest ways to reduce gas costs of `unstake` and `on_market_change`, otherwise it is reasonable to consider that there is no issue.

More precisely, here is the test scenario for `on_market_change`:

- Create a user account, Alice, with collateral across the maximum number of money markets: `MAX_MARKETS_PER_ACCOUNT = 8`.
- Create the maximum number of active rewards batches in the Controller for these money markets: `MAX_REWARDS_BATCHES = 3`.
- Create the maximum number of active rewards batches in the Booster for these money markets: `MAX_REWARDS_BATCHES = 2`.
- Have another user, Bob, who liquidates Alice's account. Bob should also have non-zero collateral and stake in the same money market where the liquidation occurs.
- We check that the overall gas cost of the transaction is at most around `300M` gas.

And here is the test scenario for `unstake`:

- Create a user account, Alice, with collateral across the maximum number of money markets: `MAX_MARKETS_PER_ACCOUNT = 8`.
- Create the maximum number of active rewards batches in the Booster for these money markets: `MAX_REWARDS_BATCHES = 2`.
- Prices and exchange rates are all outdated, so that when `unstake` will be called with the `ReliablePrice` pricing method, it will make the maximal number of external smart contract calls.
- Alice unstakes all her stake.

- We check that the overall gas cost of the transaction is at most around `300M` gas.

Subsequently, in case one test witnesses that the transaction costs significantly more than `300M` gas, the issue can be solved by reducing the number of iterations made by these endpoints until the transaction costs less than `300M` gas. To do so we suggest:

- Re-introducing the `emergency_unstake` approach from a previous version of the Booster, which allows the user to by-pass rewards computation and distribution in case of emergency, if he just wants to withdraw.
- In the Controller, instead of increasing the collateral of the liquidator, rather directly sending him the collateral seized from the liquidated account. This would completely avoid the need to trigger `on_market_change` for the liquidator.

Resolution notes

Multiple optimizations have been implemented, allowing to reduce the gas costs of the worst case scenario to 472M (link to devnet transaction). Although this is a significant improvement, this gas cost remains significantly superior to 300M, i.e. half the limit of a mini-block, hence it might be difficult to include such liquidation transactions when the blockchain is congested.

In addition, the devnet tests were performed using a previous version of the smart contract, which differs a little from the audited version, therefore the gas cost of the worst case scenario using the audited version of the smart contract might differ from the one computed above.

C8: User has no protection against sudden increase of cooldown period for unstaking

Severity: Medium **Status:** Won't fix

Location

`rewards-booster/src/governance.rs`

Description

Current behavior: When users unstake, a cooldown period starts before they can effectively withdraw their funds. At any time, the admin can increase the cooldown period, and the change is immediately effective. Consequently, users have no way to anticipate the change and are forced to accept the longer cooldown period.

Expected behavior: Users should be able to anticipate increases of the cooldown period, as they would make them wait longer than they expected before they can withdraw. Indeed, by being able to anticipate such increases, users could decide to unstake in the meantime if they consider that the new duration is too long.

Worst consequence: The cooldown period is significantly increased, e.g. from 1 day to 15 days, making users wait much longer than anticipated.

Recommendation

At a high-level, we recommend introducing a timelock of 1 day between the time the owner instantiates a change of the cooldown period and the time the change becomes effective.

To do this, we can add the code below, which most importantly:

- Replaces the endpoint `set_cooldown_period` with an endpoint `set_next_cooldown_period` that plans a future change of cooldown period instead of applying it immediately,
- Introduces a new method `update_and_get_cooldown_period`. This method should be called any time we need to retrieve the cooldown period (i.e. in `create_claim`) instead of reading it directly from the storage.

```
const TIMELOCK_COOLDOWN_PERIOD_INCREASE = 24 * 60 * 60

pub struct NextCooldownPeriod {
  pub cooldown_period: u64,
  pub timestamp: u64,
}

#[endpoint(setNextCooldownPeriod)]
fn set_next_cooldown_period(&self, cooldown_period: u64) {
  self.require_admin();
  require!(cooldown_period <= MAX_COOLDOWN_PERIOD);

  self.next_cooldown_period().set(
    NextCooldownPeriod {
      cooldown_period,
      timestamp: current_timestamp +
TIMELOCK_COOLDOWN_PERIOD_INCREASE
    }
  );
}

fn update_and_get_cooldown_period(&self, provider: ManagedAddress) -
> u64 {
  if !self.next_cooldown_period().is_empty() {
    let next_cooldown_period = self.next_cooldown_period().get();
    let current_timestamp = self.blockchain().get_block_timestamp();

    if current_timestamp >= next_cooldown_period.timestamp {
      let cooldown_period = next_cooldown_period.cooldown_period;
      self.provider_data(provider).update(|data|
        data.cooldown_period = cooldown_period
      );
      self.set_cooldown_period_event(provider, cooldown_period);
      self.next_cooldown_period().clear();
    }
  }
  self.provider_data(provider).get().cooldown_period;
}

#[storage_mapper("nextCooldownPeriod")]
fn next_cooldown_period(&self) ->
SingleValueMapper<NextCooldownPeriod>;
```

Finally, we can remove the method `set_cooldown_period_internal`.

Resolution notes

The issue has not been fixed.

Test Issues & Recommendations

Since the code is not open-source, only the remaining issues are published.