Security Audit Report

Hatom booster-v2 (3)
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:** `badf2aee6c06bcc9b655a2e9b1657f2c3a0a80df`
- **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

5 issues in the final scope

61 issues reported in the initial scope and 5 remaining in the final scope:

Severity	Reported			Remaining		
	Code	Test	Other	Code	Test	Other
Critical	4	0	0	0	0	0
Major	15	0	0	0	0	0
Medium	23	0	0	5	0	0
Minor	19	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.

C20: Multiple endpoints would run out of gas if "whitelisted_pools" or "whitelisted_stakes" are too big, which would prevent liquidations or withdrawals

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

Description

Current behavior: Several endpoints iterate over the entire set `whitelisted_pools` of pools whitelisted in the observer (collaterals in the Controller or staked tokens in the USH Staking module), and also over the entire set `whitelisted_stakes` of tokens that can be staked in the Booster.

These sets are unbounded in size and so could potentially be so big that some Booster's endpoints would run out of gas (or exceed the limit of allowed storage reads per transaction), and make the transaction fail:

1. `get_effective_account_pool_balances` could run out of gas: this would make staking and unstaking in the Booster fail, as well as `on_market_change` and `on_stake_change`, which would prevent users from being liquidated and from withdrawing their tokens from the Controller and USH Staking module. It would also prevent users from claiming their rewards.
2. `get_effective_balances_for_claim` could run out of gas: this would prevent all users from claiming all their rewards in the Booster.
3. `get_valid_pools` could run out of gas, which would prevent calling some endpoints for managing rewards: `integrate_prices`, `distribute_rewards`, `update_rewards_batches_state`.
4. `get_valid_stakes` could run out of gas, which would prevent calling the endpoint `integrate_prices` that updates price integrals.

5. The iteration over `whitelisted_pools` in `finalize_program` could run out of gas: this would prevent programs from being finalized.

Finally, given that the interactions with the Controller are already consuming a very high amount of gas (close to the 600M hard limit), we should be very cautious about any additional gas cost.

Expected behavior: The aforementioned endpoints should not run out of gas when the sets `whitelisted_pools` and `whitelisted_stakes` increase in size.

Worst consequence: Withdrawals and liquidations start failing because they run out of gas. Users won't be able to get back their funds and bad debt is going to accumulate.

Recommendation

At a high-level, we recommend always avoiding to iterate over all the items of the sets `whitelisted_pools` and `whitelisted_stakes` whose sizes are unbounded.

Instead:

1. In `get_effective_account_pool_balances`:

- We obtain the list of the user's pool tokens. For the Controller's program, we can call the endpoint `get_account_markets` and retrieve the associated collateral tokens, while for the USH Staking module's program, we can call `get_account_stake_tokens`.
- We filter out the user's pool tokens that are not whitelisted in the Booster, i.e. are not in the set `whitelisted_pools`,
- For the Controller's program, we filter out the tokens for which the user has no collateral. This is necessary because `get_account_markets` might have returned markets where the user has debt but no collateral.

Moreover, in the USH Staking module, we need to enforce a limit on the number of staked tokens a user could have, e.g. 8 like the limit on the number of collaterals in the Controller.

2. In `get_effective_balances_for_claim`, if the list `pools` of selected pool tokens is empty, we can return the entire list `whitelisted_pools`. If `pools` is not empty, we just need to verify that it contains only whitelisted tokens and no duplicates, which can be done as follows:

- Initialize a set `visited_tokens` of type `ManagedMapEncoded<TokenIdentifier,bool>`.
- For each token in `pools`, verify that it is not in `visited_tokens`, insert it in `visited_tokens` associated to the value `true` (the value should **not** be set to `false`, as it would remove the token from the map), and verify that it is whitelisted, i.e. that it is in `whitelisted_pools`.

3. In `get_valid_pools`, we do exactly as in `get_effective_balances_for_claim`.

4. In `get_valid_stakes`, we do exactly as in `get_effective_balances_for_claim` and `get_valid_pools`, except that we use the set `whitelisted_stakes` instead of `whitelisted_pools`.

5. In `finalize_program`, another approach has to be taken because we need to ensure that all pool tokens have no rewards batches anymore. Right now, the endpoint iterates over all whitelisted pool tokens. To avoid such iteration, we can proceed as follows:

- We introduce a new `UnorderedSetMapper` `pools_with_rewards_batches` that contains all pool tokens for which `rewards_batches(program_id, pool)` is not empty,
- When we set a rewards batch, we insert the pool token in `pools_with_rewards_batches`,
- When we finalize a rewards batch, if `rewards_batches(program_id, pool)` becomes empty, then we remove the pool token from `pools_with_rewards_batches`.

Then, in the endpoint `finalize_program`, we can now simply check that `pools_with_rewards_batches` is empty.

Resolution notes

Points 2, 3 and 4 have not been solved as the endpoints still iterate over all whitelisted pool tokens or stake tokens.

C21: Some user actions might consume too much gas or make too many storage reads

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

Description

Current behavior: The gas cost and number of storage reads of several user endpoints could be too high, which would make transactions fail to be executed or fail to be included in a block.

1) The most important failures would be for the endpoints `unstake` and `on_pool_change`, as they could prevent users from withdrawing from the Booster, Controller or USH Staking, and could also prevent liquidations in the Controller. These endpoints are costly as they perform many iterations:

- A transaction that includes `on_pool_change` will iterate over all the user's pool tokens, all the associated rewards batches and all the user's staked tokens. Additionally, when it is called through the Controller for liquidating 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 user's pool tokens, all the associated rewards batches and all the user's staked tokens, and update the user's compliance integral for all his pool tokens.

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 multiple staked tokens are added in the Booster and that Hatom Lending Borrowing protocol now integrates new smart contracts for handling the USH token, the overall gas cost might have increased even more in some cases.

2) Other possible failures would occur for user functionalities which require both unstaking and staking positions, which cost significant gas as they perform multiple iterations and can make multiple external smart contract calls to retrieve prices from oracles. These actions are the following: re-allocating stake, claiming rewards from xExchange, and promoting xExchange positions.

Expected behavior: The gas cost and number of storage reads of user endpoints should stay under the half-block gas limit (`300M` gas) and the

storage reads limit because otherwise it might prevent liquidations and stake / collateral withdrawals.

Worst consequence: If liquidation transactions start to run out of gas and fail, users could borrow without fear of being liquidated.

Recommendation

First, we suggest bounding in the USH Staking module the number of pools in which the user can stake, e.g. to 3. Indeed, this number is currently unbounded, and the Booster needs to iterate over all the staked tokens.

Then, we suggest doing the following 2 system tests once all the changes related to the other issues of the report are done, in order to ensure we properly evaluate the gas costs of the endpoints once all the other changes are done.

1) At a high-level, we recommend running a system test on devnet of the worst case scenario for `unstake` and `on_pool_change` in order to check if gas costs are reasonable, i.e. less than the storage reads limit and less than the half-block gas limit (`300M` gas), and sharing the transaction 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_pool_change`, otherwise it is reasonable to consider that there is no issue.

More precisely, here is a test scenario to be run for `on_pool_change`:

- Create a user account, Alice, with collateral and borrows across the maximum number of money markets: `MAX_MARKETS_PER_ACCOUNT = 8`.
- In particular, Alice should have USH borrow, and a maximal number of collaterals registered in the Discount Rate Model
- Create the maximum number of active rewards batches in the Controller for these money markets: `MAX_BATCHES = 3`.
- Create an extra reward batch for each of the user's pool token.
- Create the maximum number of active rewards batches in the Booster for these money markets: `MAX_REWARDS_BATCHES = 2`.
- Alice stakes the maximum number of different tokens: `MAX_ACCOUNT_STAKES = 5`.
- Make Alice register all her collateral.

- Liquidate Alice's account.
- We check that the transaction succeeded and that its overall gas cost is at most around `300M` gas.

Subsequently, in case one test witnesses that one of the limits are reached, the issue can be solved by reducing the number of iterations made by these endpoints. To do so we suggest:

- Reducing `MAX_ACCOUNT_STAKES` to `2`.
- Reducing `MAX_BATCHES` to `2`.
- Reaching out to the auditor if the previous changes are insufficient.

Finally, in case the tests for the Booster pass and that the maximum number of different staked tokens in USH Staking is set to a number smaller than or equal to the maximum of collaterals in the Controller, we do not need to do a devnet test for USH Staking because it would be clear that the worst-case interactions with the Controller are more gas costly than the worst-case interactions with USH Staking.

2) To ensure other user functionalities are not too gas costly, we recommend doing one devnet test for each of the following actions: `cross_reallocate`, `claim_xex_rewards`, and `promote_stake_with_wrapped_egld`. Indeed, these 3 endpoints are more costly than all others, in worst-case scenarios. For each endpoint we can take the same conditions for Alice as in the first devnet test above. Moreover, for `cross_reallocate`, both Alice and her Account Manager should have the maximum number of stake tokens and pool tokens.

Subsequently, in case one test witnesses that the half-block gas limit is reached, the issue can be solved by reducing the number of iterations made by these endpoints, as suggested in the prevous point.

Resolution notes

1) For the endpoints `unstake` and `on_pool_change`, 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.

2) For all other costly functionalities, devnet tests have been performed and successfully witnessed that gas costs are under the half-block gas limit of 300M.

C31: Collateral prices might not include all borrowing interests

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

Location

`rewards-booster/src/wrappers/prices/htoken.rs`
`get_price`

Description

Current behavior: The Booster uses a price of the collateral token that underestimates the real price of the collateral token because it doesn't take into account the borrowing interests that have not been accrued yet in the corresponding money market.

This is because it obtains its price using the endpoint `get_stored_exchange_rate` of the money market, which doesn't include the borrowing interests that have not been accrued yet.

In case the borrowing interests have not been accrued for an extended period of time, the underestimation of the collateral price could become significant. This would in turn lead to an overestimation of users' compliances and so of users' rewards.

Expected behavior: The price of the collateral token used by the Booster should take into account borrowing interests that have not been accrued yet.

Recommendation

Instead of calling the endpoint `getStoredExchangeRate` of the Controller, we suggest calling `getCurrentExchangeRate`, as this one first accrues the interests and so returns an exchange rate that includes all the borrowing interests.

Resolution Notes

The issue has not been fixed.

C35: "claim_rewards" would run out of gas if some lists are too big

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

Location

```
rewards-booster/src/base/rewards.rs
    claim_rewards
```

Description

Current behavior: The lists `account_pool_pending_batch_ids` and `rewards_tokens_to_ignore` could be of any size and beyond a certain size, the endpoint `claim_rewards` would run out of gas or exceed the limit of storage reads per transaction, preventing the user from claiming his rewards.

Note that in practice, `account_pool_pending_batch_ids` is unlikely to ever be too big because it is supposed to be close to the number of active rewards batches for all the users' pool tokens, which is a relatively small number. Similarly, `rewards_tokens_to_ignore` is the list of tokens that the user wants to skip (e.g. if he is frozen for many tokens) and is unlikely to ever be too big.

Expected behavior: The user should be able to claim the rewards he wants, in one or many transactions, regardless of his number of pending batch IDs or the number of tokens that he wants to skip.

Worst consequence: Some users could not claim their rewards.

Recommendation

We recommend modifying the endpoint `claim_rewards` such that the user can specify the list of batch IDs to claim (using a `ManagedVec`). We verify that each batch ID provided is present in `account_pool_pending_batch_ids` and we claim the rewards for those batch IDs. If the list of batch IDs provided by the user is empty, we try to claim all the pending batches.

Moreover, we can remove the argument `rewards_tokens_to_ignore`.

Finally, note that the Account Manager needs to be adapted to interact with the new signature of `claim_rewards`.

Resolution notes

The issue has not been fixed.

C41: User has no protection against sudden increase of cooldown period for unstaking

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

Location

```
rewards-booster/src/base/governance.rs
    set_cooldown_period
```

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.