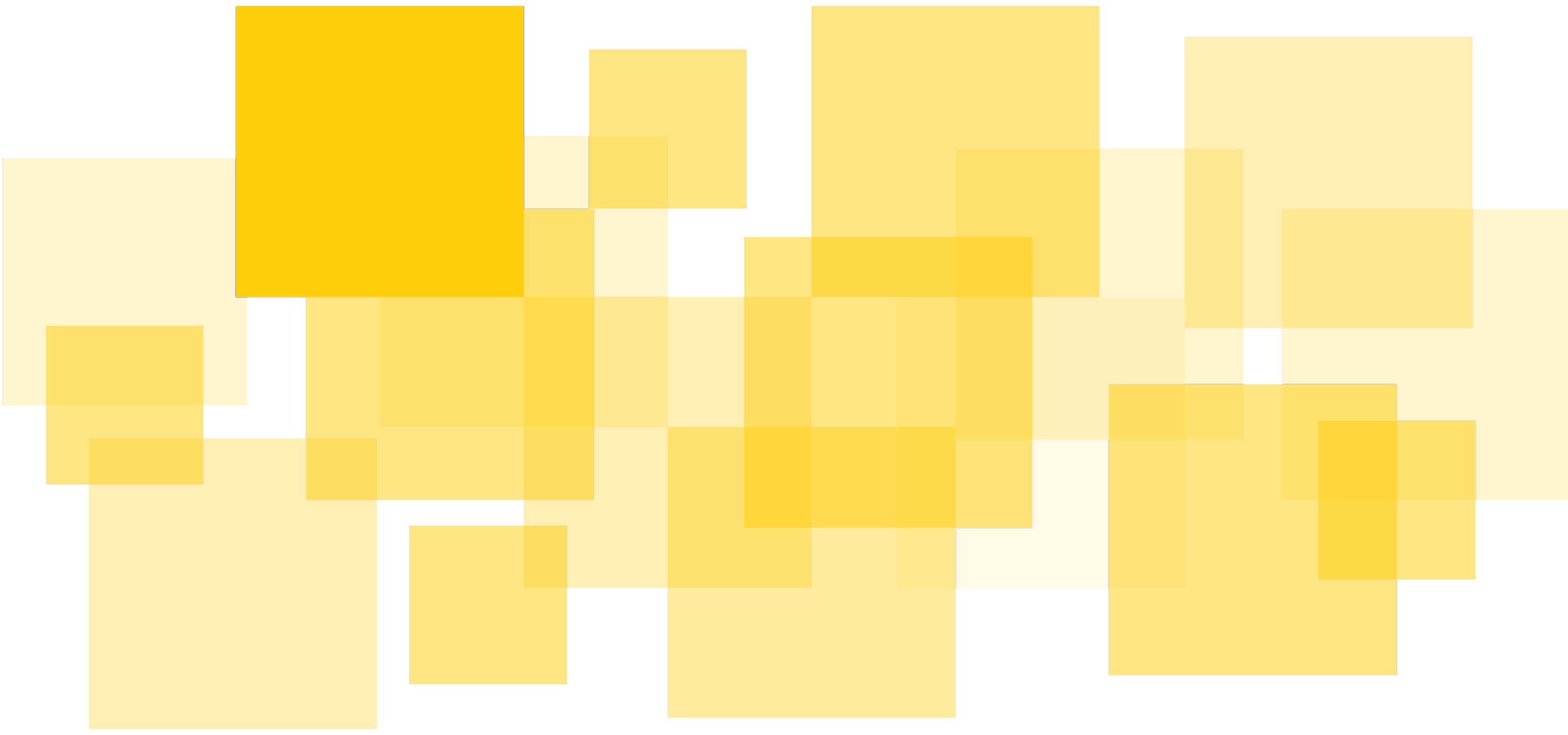
Audit Report

Hatom Liquid Staking

Delivered: July 12, 2023



Prepared for Hatom Labs by Runtime Verification Inc.

runtime verification

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Summary

Runtime Verification, Inc. has audited the smart contract source code of the Hatom Liquid Staking contract. The review was conducted from 2023-04-24 to 2023-05-31 and from 2023-06-27 to 2023-07-07.

Hatom Labs engaged Runtime Verification in checking the security of their Hatom Liquid Staking contract.

The issues which have been identified can be found in the Notable Findings section.

Scope

The audited files are:

- common.rs
- constants.rs
- contract.rs
- delegate.rs
- delegation.rs
- errors.rs
- events.rs
- governance.rs
- migration.rs
- model.rs
- proxies.rs
- rewards.rs
- score.rs
- selection.rs
- storage.rs
- unbond.rs
- undelegate.rs

The review focused mainly on the HatomProtocol/hatom-liquid-staking private code repository.

The audit has focused on the above contract and has assumed the correctness of the libraries and external contracts it uses. The libraries are widely used and assumed to be secure and functionally correct.

The review focused on two commits in the HatomProtocol/hatom-liquid-staking GitHub repository:

- Commit 1: de66c584008de89b5cd484a4d186182599e25210. This is the initial commit provided by Hatom.
- Commit 2: 92194aec4527b510add21258e47c87b716cbcdf7. A later iteration containing fixes for some of the issues highlighted during the audit and other changes. It also

includes a redesign of external contract calls, penalties and other parts of the contract. This was a best-effort audit. See section Commit 2 audit for details.

Assumptions

This is a summary of the assumptions made in this audit. See the Assumptions appendix for more details.

1. We assume the underlying development platform (the MultiversX blockchain, the Rust tool-chain etc.) is safe and reliable.
2. We assume that the underlying libraries (the MultiversX framework/SDK) are safe and reliable.
3. We assume that the contract administrator has a good knowledge of how the protocol works and does not behave in ways that would lead to bad outcomes (though we do point out use-cases in which the contract could rely less on the administrator).

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in the Disclaimer, we have used the following approaches to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the protocol. We also checked security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic, specification and the implementation.

Second, we carefully checked if the code is vulnerable to known security issues and attack vectors.

Third, we regularly participated in meetings with the Hatom team and offered our feedback during ongoing design discussions, and suggested development practices as well as design improvements.

This report describes the intended behaviour of the application under review and then outlines issues we have found, both in the intended behaviour and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter.

Overview

The Hatom Liquid Staking application provides an interface that allows users to stake EGLD and receive staked EGLD (sEGLD) in return. The Liquid Staking contract delegates these EGLD to Staking Provider contracts, which generate rewards in exchange. These rewards are also delegated, increasing the value of sEGLD. When removing EGLD tokens from the Liquid

Staking application, users must wait for a cooldown period before being able to retrieve the tokens.

Staked EGLD tokens represent ownership of a share in the total amount of EGLD delegated through the Liquid Staking contract. For the cooldown period, the Liquid staking contract emits a NFT that represents the ownership of the EGLD.

In the next section we present the callable interfaces and their functionality. The Liquid staking endpoints that call the Staking Provider contracts will be decomposed in their atomic pieces, which, usually, consist of:

1. The main endpoint call, generated by the user
2. The Staking Provider contract endpoint call, which may be split into a success and a failure case
3. The callback for the delegation call, which may also be split into the success and the failure case.

Design and Implementation

In order to differentiate between the initial and final values for a function, we will add an apostrophe before the final values (e.g. $Shares_U$ is the initial value, $'Shares_U$ is the final one).

State model

This section defines the variables that describe the contract state at Commit 1. There are no similar definitions for the contract state at Commit 2.

A value in the following list is defined as the sum of the values in its sub-list, e.g. $EGLD_S = \sum EGLD_S^T + \sum EGLD_S^{cb(T)}$

- Shares
  - $Shares$ is the total amount of existing sEGLD tokens.
  - $Shares_U$ is the amount of shares held by user U
  - $Shares_{LS}$ is the amount of shares held by the Liquid Staking (LS) contract.
    - $Shares_{LS}^{undelegate}$ are the shares held by the contract while one or more transactions for ‘undelegate’ are in progress
- EGLD
  - $EGLD_U$ is the amount of EGLD held by user U
  - $EGLD_{LS}$ is the amount of EGLD held by the LS contract.

        - $Rewards_{LS}$ is the amount of rewards that were retrieved from Staking Provider contracts, but which was not yet reinvested.
        - $ProtocolReserve$ is the amount of rewards the contract keeps for itself
        - $Withdrawable$ and $Withdrawable_{DC}$ are the amount of EGLD that can be retrieved when unbonding.
        - $EGLD_{LS}^{P}$ is the EGLD held by the LS contract for the penalty P
    - $EGLD_{S}$ is the in-flight amount of EGLD held by the system (e.g. in the time between making an async call and the time the called code runs).
        - $EGLD_{S}^{T}$ is the amount held while waiting for a specific call T to run.
        - $EGLD_{S}^{cb(T)}$ is the amount held while waiting for T's callback to run.
- EGLD held by Staking Provider contracts (DC)
    - $Delegated$ is the total amount of EGLD delegated to contracts.
        - $Delegated_{DC}$ is the amount of tokens delegated to a specific Staking Provider contract.
    - $Undelegated$ is the total amount of ELGD that was undelegated from contracts, but can't be removed yet.
        - $Undelegated_{DC}$ is the same, but restricted to a specific contract
            - $Undelegations_{DC}$ is the set of all pending undelegations for the DC contract.
    - $Unbond$ is the total amount of ELGD that was undelegated from contracts, and can be removed.
        - $Unbond_{DC}$ is the same, but restricted to a specific contract
            - $Unbondings_{DC}$ is the set of all pending unbondings for the DC contract.
    - $Rewards$ is the total amount of tokens that can be retrieved as rewards. It does not include the fees that the Staking Provider contract may keep for itself.
        - $Rewards_{DC}$ is the same, but restricted to a specific contract.
- Undelegate NFTs
    - $NFTs$ is the set of all undelegate NFTs.
        - $NFTs_{U}$ is the set of undelegate NFTs held by user U
        - $NFTs_{LS}$ is the set of undelegate NFTs held by the LS contract
            - This is actually the set of NFTs held while unbonding
- Pending requests made by the LS contract
    - For endpoint in $Delegate$, $Undelegate$, $Unbond$, $DelegateTo$, $UndelegateFrom$, $UnbondFrom$, $DelegateTrusted$, $UndelegateTrusted$, $ClaimRewards$, $DelegateRewards$:
        - $Pending_{endpoint}$ - all in-progress delegate requests made to endpoint by the LS contract

- $Pending^{Unhandled}_{endpoint}$ - requests not handled yet by a Staking Provider contract
- $Pending^{Success}_{endpoint}$ - requests successfully handled by a Staking Provider contract, but whose callback did not run yet.
- $Pending^{Failure}_{endpoint}$ - requests successfully handled by a Staking Provider contract, but whose callback did not run yet.

Liquid Staking contract storage

- $State$ (Active/Inactive)
- $LsTokenSupply$
- $CashReserve$
- $RewardsReserve$
- $ProtocolReserve$
- $TotalUndelegated$
- $TotalWithdrawable$
- $MinEgld$ - min_egld_to_delegate
- $UnbondPeriod$
- $MigrationWhitelist$
- $DcData(contract)$ - delegation_contract_data
    - $TotalDelegated$
    - $TotalUndelegated$
    - $TotalWithdrawable$
    - $Admin$
- $Admin$
- $RewardsManager$
- Penalties
    - $Penalties$ is the set of all penalties that are still relevant to the LS contract
- Contracts
    - $SortedContracts$ is a list of contracts sorted by delegation score. This list is used when selecting a contract to be used by various endpoints.
    - $DcData$ holds data for each Staking Provider contract with which the LS contract currently works.
    - $Outdated$ are the Staking Provider contracts marked as outdated.
    - $Blacklisted$ is a list of all contracts blacklisted in the past by the LS contract, but which were not re-whitelisted. A blacklisted contract may not be in $DcData$.
- Ls_token, undelegate_token, undelegate_token_name, unbond_period

Note that, if DC is a Staking Provider contract, then values like $Delegated_{DC}$ should be close, but not equal to values like $DcData(DC).TotalDelegated$ because the former describes the actual state of a Staking Provider contract, while the latter is the LS contract's view of the Staking

Provider contract's state, and there is some time lag between the actual state changes and the LS contract updates.

Other data

- $CurrentTime$ is the current blockchain epoch or block (which one is actually used depends on the context).
- FX is the exchange rate between sEGLD and EGLD
  - $FX = \frac{CashReserve+RewardsReserve}{Shares}$ if $Shares > 0$
  - $FX = 1$ otherwise

Invariant

This contains invariant properties for the Liquid Staking contract together with some properties that should be invariant (or something similar to them should be invariant), but are not, and for which issues were filed.

1. $CashReserve = \sum_{DC \in DcData} DcData(DC).TotalDelegated + \sum_{P \in Penalties} P.egld$
2. $\sum_{DC \in DcData} DcData(DC).TotalDelegated + \sum_{P \in Penalties} P.egld$

   $\geq \sum_{U \in Users} Shares_U \cdot FX + \sum_{Ud \in Pending_{undelegate}} Ud.egld$
   a. The two quantities above are actually almost equal.
   b. This is not preserved when the FX changes (e.g. because rewards are being claimed) while undelegate calls are pending. Even then, it should not be wrong by much.
3. $DcData(DC).TotalDelegated$

   $= Delegated_{DC}$

   $- \sum_{D \in AllPendingDelegate, D.contract=DC} D.egld$

   $+ \sum_{Ud \in AllPendingUndelegate, Ud.contract=DC} Ud.egld$
   a. Where
      $AllPendingDelegate =$

      $Pending^{Success}_{delegate} \cup Pending^{Success}_{delegateTrustedContract}$
      $\cup Pending^{Success}_{delegateTo} \cup Pending^{Success}_{delegateRewards}$

      $AllPendingUndelegate =$

      $Pending^{Success}_{undelegate} \cup Pending^{Success}_{undelegateTrustedContract} \cup Pending^{Success}_{undelegateFrom}$

4. $Delegated_{DC} \geq \sum_{D \in AllPendingDelegate, D.contract = DC} D.egld$
    a. Not always preserved

5. $TotalUndelegated = \sum_{DC \in DcData} DcData[DC].TotalUndelegated$
6. $DcData[DC].TotalUndelegated =$

$$\sum_{NFT \in NFTs, NFT.contract = DC} NFT.egld + \sum_{P \in Penalties, P.contract = DC \land \neg P.unbonded} P.egld$$

    a. Not always preserved.
7. $Undelegated_{DC}$

$$= \sum_{NFT \in NFTs, NFT.contract = DC \land NFT.unbondTime > CurrentTime} NFT.egld$$

$$+ \sum_{P \in Penalties, P.contract = DC \land \neg P.unbonded \land P.unbondTime > CurrentTime} P.egld$$

$$+ \sum_{Ud \in AllPendingUndelegate, Ud.contract = DC} Ud.egld$$

8. $Shares_{LS}^{undelegate} = \sum_{Ud \in Pending_{undelegate}} Ud.shares$

9. $Shares_{LS} = Shares_{LS}^{undelegate}$

10. $EGLD_{LS}^{P} = \sum_{P \in Penalties, P.unbonded \land P.id \notin \{D.PenaltyId | D \in Pending_{delegateTo}\}} P.egld$

11. $TotalWithdrawable = \sum_{DC \in DcData} DcData[DC].TotalWithdrawable$
12. $Withdrawable_{DC} = DcData(contract).TotalWithdrawable$

13. $\sum_{NFT \in NFTs, NFT.time \leq CurrentTime \land NFT.contract = DC} NFT.egld$

$$+ \sum_{P \in Penalties, P.contract = DC \land \neg P.unbonded \land P.time \leq CurrentTime} P.egld$$

$$= Withdrawable_{DC} + Unbond_{DC}$$

$$+ \sum_{Ub \in Pending_{unbond}^{Success} \cup Pending_{unbondFrom}^{Success}, Ub.contract = DC} EGLD_{S}^{cb(Ub)}$$

14. $RewardsReserve \geq \sum_{D \in Pending_{DelegateRewards}} D.egld$
    a. Not always preserved
15. $DC \in Blacklisted \land DC \in DcData$ implies $DC \in Outdated$

16. $DC \in Blacklisted$

    $\wedge (DC \notin DcData \vee DcData[DC].TotalDelegated = 0)$

    Implies

    $Delegated_{DC} = 0$

    a. This is not always preserved
17. $DC \in Blacklisted \wedge DC \in DcData$ implies $DcData[DC].TotalDelegated = 0$
    a. Not always preserved
18. $\forall DC \notin DcData\,.\ Delegated_{DC} = Undelegated_{DC} = Unbond_{DC} = 0$
    a. Partly implied by properties 5 and 7 and by invariant 16
19. $D \in Pending_{delegate} \cup Pending_{delegateTrustedContract} \cup Pending_{delegateTo} \cup Pending_{delegateRewards}$ implies $D.contract \in DcData$
    a. Not always preserved
20. $DcData[DC].TotalUndelegated > 0 \vee Undelegated_{DC} > 0$ implies $DC \in DcData$
21. $DcData[DC].TotalDelegated > 0 \vee Delegated_{DC} > 0$ implies $DC \in DcData$
22. $NFTs_{LS}$ is in bijection with $Pending_{unbond}$ on $NFT.nonce = Ub.NFT.nonce$; also: $NFT.contract = Ub.contract, NFT.egld = Ub.egld$
23. $D \in Pending^{Success}_{delegateTo}$ implies:

    $D.PenaltyId \in Penalties$

    $Penalties[Ub.PenaltyId].unbonded$

    $D.egld = Penalties[D.PenaltyId].egld$
24. $Ub \in Pending_{unbondFrom}$ implies:
    a. $Ub.PenaltyId \in Penalties$
    b. $Ub.contract = Penalties[Ub.PenaltyId].contract$
    c. $Ub.egld = Penalties[Ub.PenaltyId].egld$
    d. $\neg Penalties[Ub.PenaltyId].unbonded$
    e. $Penalties[Ub.PenaltyId].UnbondTime \leq CurrentTime$
    f. Not always preserved
25. $NFT \in NFTs_{LS}$ implies $NFT.unbondTime \leq CurrentTime$
26. The following values are $>= 0$:
    a. $ProtocolReserve$
    b. $DcData(DC).TotalWithdrawable$
27. The following values are > 0
    a. $D.egld$

       $\forall D \in Pending_{delegate} \cup Pending_{delegateTrustedContract} \cup Pending_{delegateTo} \cup Pending_{delegateRewards}$
    b. $Ud.egld$,

       $\forall Ud \in Pending_{undelegate} \cup Pending_{undelegateTrustedContract} \cup Pending_{undelegateFrom}$
    c. $Ud.egld, \forall Ud \in Pending_{unbond} \cup Pending_{unbondFrom}$
    d. $Ud.shares, \forall Ud \in Pending_{unbond}$

e. $P.egld, \forall P \in Penalties$
f. $NFT.egld, \forall NFT \in NFTs$

28. The $SortedContracts$ list
    a. Is sorted by the contract's delegation score
    b. Is in bijection with $DcData - Blacklisted$
29. The contract holds
    a. $EGLD_{LS}$ - EGLD
    b. $Shares_{LS}$ - sEGLD
    c. $NFTs_{LS}$ - NFTs
    d. This is not always preserved

Inferred Invariant Properties

30. $\sum_{NFT \in NFTs_{LS}, NFT.contract=DC} NFT.egld = \sum_{Ub \in Pending^{Success}_{unbond}, Ub.contract=DC} Ub.egld$
    a. Implied by invariant 22
31. $D \in Pending_{undelegate} \cup Pending_{undelegateTrustedContract} \cup Pending_{undelegateFrom}$ implies $D.contract \in DcData$
    a. Implied by invariants 7, 27.b and 20
32. $D \in Pending_{unbond} \cup Pending_{unbondFrom}$ implies $D.contract \in DcData$
    a. Implied by invariants 22 and 33 for the $Pending_{unbond}$ part
    b. Implied by invariants 27.c, 37 and 20 for the $Pending_{unbondFrom}$ part
33. $NFT \in NFTs$ implies $NFT.contract \in DcData$
    a. Implied by invariants 6, 27.f and 20
34. $DC \in Blacklisted$
    $\wedge\ DcData[DC].TotalDelegated = 0$
    $\wedge\ DcData[DC].TotalUndelegated = 0$
    $\wedge\ DcData[DC].TotalWithdrawable = 0$
    Implies
    $Unbond_{DC} = 0$

    a. Implied by invariant 35
35. $Unbond_{DC} \le DcData[DC].TotalUndelegated$
    a. Implied by invariants 6, 13
36. $DC \in Blacklisted$
    $\wedge\ DcData[DC].TotalDelegated = 0$
    $\wedge\ DcData[DC].TotalUndelegated = 0$
    Implies
    $Undelegated_{DC} = 0$
    a. Implied by invariants 3, 4, 6, 7

37. $\sum_{Ub \in Pending_{unbondFrom}, Ub.contract=DC} Ub.egld > 0$ implies $DcData[DC].TotalUndelegated > 0$
    a. Implied by invariants 6 and 24
38. The following values are $>= 0$:
    a. $CashReserve$
        i. Implied by invariants 1, 27.e and 38.d
    b. $RewardsReserve$
        i. Implied by invariants 14 and 27.a
    c. $DcData(contract).TotalDelegated$
        i. Implied by invariants 3, 4 and 27.c
    d. $DcData(contract).TotalUndelegated$
        i. Implied by invariants 6, 27.e and 27.f
    e. $TotalUndelegated$
        i. Implied by invariants 5 and 38.d
    f. $TotalWithdrawable$
        i. Implied by invariants 11 and 26.b
39. $EGLD_{LS}$

$$= Rewards_{LS} + ProtocolReserve + \sum_{DC \in DcData} DcData(DC).TotalWithdrawable$$

$$+ \sum_{P \in Penalties, P.unbonded \wedge P.id \notin \{D.PenaltyId | D \in Pending_{delegateTo}\}} P.egld$$

    a. Implied by Invariant 10, Invariant 11 and the definition for $EGLD_{LS}$

Notable findings

A01 - Keeping contracts outdated forever

[Severity: High | Difficulty: Low | Category: Security]

A user (U) can keep some/many/all contracts outdated forever if all Staking Provider contracts have liquidity caps and the relevant ones are in the same shard as the Liquid Staking contract.

Scenario:

1. While this makes sense to user U
    a. In a single transaction:
        i. Take DC to be the Staking Provider contract with the maximum difference between cap and total_value_locked.
        ii. Continue only if DC is in the same shard, otherwise exit the loop.
        iii. U finds the max quantity of EGLD that DC can take - MaxE.
        iv. U adds a small quantity E of EGLD directly to DC.

        v. U asks the Liquid Staking contract to add MaxE to DC. There is only one contract that can take this amount, DC. Adding liquidity will fail, which marks the contract as outdated.
    b. In another transaction (cleaning up):
        i. U asks DC to undelegate E.
        ii. U withdraws all of his unbonded tokens from DC (if any)
2. The admin updates the data for the outdated DCs in an attempt to fix this issue.
3. Repeat from item 1 forever.

Github issue.

Comments

The Hatom team pointed out that the attacker would need to delegate at least 1 EGLD every 30 minutes (the data refresh interval) for each Staking Provider contract that is being kept outdated, and this will be locked for 10 days. This means that the user needs to lock 480 EGLD (~17-18K USD) per contract. Also, they expect to have contracts that are not capped.

Suggestions

1. Set the $DcData[DC].cap$ value to something smaller than the actual cap, so that U has to delegate a larger quantity of EGLD to DC
2. Instead of outdating the contract, if the $DcData[DC].TotalValueLocked - DcData[DC].cap$ value is large enough, one could set $DcData[DC].TotalValueLocked$ to something like $\frac{DcData[DC].TotalValueLocked + DcData[DC].cap}{2}$ or $DcData[DC].TotalValueLocked - Constant$. This would not fully solve the problem, but it would probably make it less bad.

A02 - Blacklisting is hard if a contract does not have delegated tokens

[Severity: - | Difficulty: - | Category: -]

Any attempt to blacklist a contract without delegated tokens will start with an undelegate_from call. If the contract has no delegated tokens, this call will fail in the require_sufficient_egld check. This will prevent blacklisting such contracts.

Github issue.

Suggestions

1. The Hatom team suggested using the migration feature to delegate some amount to the contract, after which it can be blacklisted.
2. Note that delegating some amount (e.g. 1 EGLD) to each contract when whitelisting would not solve the problem, as it can be removed by undelegations.

Status

Fixed at Commit 2.

A03 - unbond_from sends EGLD to caller

[Severity: High | Difficulty: High | Category: -]

Issue found by the Hatom team during audit discussions.

Unbond_from should, instead, leave the EGLD in the contract, in order to be sent to another Staking Provider contract in delegate_to, in the same way as unbond_from_cb.

The “High” difficulty rating is misleading because, although this requires an administrator action, it is fairly difficult for the administrator to avoid taking this action.

Github issue. Fix.

Status

Fixed at Commit 2

A04 - Whitelisting is sometimes not possible after blacklisting

[Severity: High | Difficulty: Low | Category: Security]

This is how blacklisting works:

1. The admin calls undelegate_from, asking for the entire delegation_contract_data(..).total_delegated amount to be undelegated.
2. This causes the contract to be marked as blacklisted and outdated. This means that nobody can call delegate, undelegate and change_delegation_contract_params. The latter means that nobody can change the outdated bit.
3. After some time, the admin can call unbound_from for the entire amount. Usually (though not always) this will cause total_delegated, total_undelegated and total_withdrawable to be 0. If that's the case, then the contract is deleted from delegation_contract_data.
4. If that's not the case, then it must be that there are some users that have undelegated their tokens, but didn't unbound them yet. After those users unbound their tokens, the total_* amounts above become 0 and the contract is deleted.
5. The contract can be re-added with whitelist_delegation_contract.

There may be a user which, for whatever reason, has undelegated tokens, but does not remove them In step 4 above, as expected. If that happens, then the Staking Provider contract will never be removed from delegation_contract_data, so it will never be re-whitelisted.

Github issue.

Suggestions

1. Have an endpoint for whitelisting blacklisted contracts.

Status

Fixed at Commit 2

A05 - Using withdraw can cause data inconsistencies

[Severity: High | Difficulty: High | Category: Protocol Invariants]

Withdraw is admin-only, so the issues mentioned below only make sense if the admin is likely to behave in certain ways.

The function allows the admin to transfer various tokens, including EGLD and shares to some account.

When withdrawing EGLD, the function does not update the various values that track the amount of EGLD that the contract has, e.g. protocol_reserve, leading to data inconsistencies.

If it is used for withdrawing shares, the function can cause loss of funds for users in the following scenario:

1. A user calls undelegate, sending some shares.
2. undelegate sends the request to a contract in another shard.
3. The admin attempts to withdraw the shares sent by the user (unlikely).
4. The undelegate callback crashes because it does not have enough shares to burn.

Github issue. Fix.

Status

Fixed at Commit 2

Concurrency issues

All of these issues are caused by having data validations in the Liquid Staking contract endpoints, before the endpoint calls a Staking Provider contract, while the data updates happen in that call's callback. In some cases, this means it is possible to run the callback with invalid data.

Suggestions

1. The Liquid Staking contract could keep track of various quantities involved in the calls, e.g. it could keep track of the amount of EGLD that is "in flight" when delegating and it could take it into account when validating data.

A06 - Concurrent unbond_from may lead to user funds loss and contract removal

[Severity: High | Difficulty: High | Category: Protocol Invariants]

This requires two admin actions, so the admin can avoid it.

Two concurrent unbond_from calls for the same penalty will withdraw unbonded tokens that belong to users, preventing these users from withdrawing their tokens. This may also remove the contract in some cases (i.e. when all EGLD in the contract is unbonded and the penalty is exactly half of the available amount).

Github issue, issue.

Suggestions

1. unbond_from_cb could skip most updates if self.penalties(penalty_id).unbonded is true.
2. The Hatom team suggested having an ongoing_unbond_from storage and checking it in the unbond_from endpoint.

Status

Fixed at Commit 2

A07 - Concurrent delegate_rewards may delegate more rewards than available

[Severity: High | Difficulty: High | Category: Protocol Invariants]

This requires two admin actions, so the admin can avoid it.

Two concurrent delegate_rewards calls will transfer more funds than expected because the amount of available rewards is updated only in the callback.

Github issue.

Suggestions

1. The Hatom team suggested having an ongoing_rewards_delegation storage and checking it in the delegate_rewards endpoint.

Status

Fixed at Commit 2

A08 - Concurrent delegate_to may delegate more than intended

[Severity: High | Difficulty: High | Category: Protocol Invariants]

This requires two admin actions, so the admin can avoid it.

Delegate_to should not be called twice for the same penalty_id, at least not until the first call is fully finished, since it will probably delegate the penalty amount twice.

Github issue.

Status

Fixed at Commit 2

A09 - Tokens stuck in a blacklisted contract

[Severity: High | Difficulty: Low | Category: Protocol Invariants]

Scenario:

1. A user U attempts to delegate N EGLD to a Staking Provider contract DC (selected automatically).
2. Before the callback runs, the admin attempts to blacklist the DC contract and calls undelegate_from with the entire amount.
3. delegate_cb for the first call runs and updates self.delegation_contract_data(DC).total_delegated
4. undelegate_from_cb runs and marks the contract as blacklisted.

In this case, there is no way to

1. Remove the N EGLD mentioned above from DC
2. Whitelist the DC contract.

Most likely, all delegate endpoints (delegate, delegate_trusted_contract, delegate_to and delegate_rewards) have the same issue when running concurrently with undelegate_from.

Github issue.

Suggestions

1. The Hatom team suggested that the admin could pause the Liquid Staking contract, and wait until all in-progress calls finish. After that, the admin could safely run undelegate_from.
2. Note that even if a contract could be re-whitelisted without removing it first, the admin may not want to do so before the reason that caused it to be blacklisted is no longer valid.

Status

Fixed at Commit 2

Informative findings

A10 - Delegating to a blacklisted or outdated contract

[Severity: Low | Difficulty: High | Category: Protocol Invariants]

For users in the migration whitelist, the Liquid Staking contract skips the normal checks when delegating to the corresponding contract. This means that it is possible to delegate EGLD to blacklisted or outdated contracts.

Github issue.

Comments

Delegating to blacklisted contracts requires inconsistent behaviour from admins. Delegating to outdated contracts does not require inconsistent admin behaviour.

The migration whitelist is supposed to be used in very specific contexts, for a limited time, in which contracts should not get blacklisted, and in which delegating to outdated contracts should be fine. To be specific, it should only be used after a Staking Provider contract is being added, allowing the provider to migrate liquidity through the Liquid Staking contract.

Suggestions

1. Add a check that the migration contract is not blacklisted when delegating to it.

Status

This issue is mostly fixed at Commit 2 - whitelisted users can't delegate to blacklisted contracts anymore, however, they can still delegate to outdated contracts.

A11 - Rounding error larger than needed for share <-> egld transformations

[Severity: Low | Difficulty: - | Category: Best practices]

See the Rounding Errors section for more information.

Example: If $Shares = 489556478701587097 06$, $CashReserve = 601338163526516253 53$ and the users wants to undelegate 1000 sEGLD (i.e. $S = 1000 \cdot WAD$), then the rounding error is $-\ 999$ (i.e. $-\ 999 \cdot \frac{1}{WAD}$ EGLD).

Github issue.

A12 - Rewards generated by a user's EGLD may be received by another

[Severity: Medium | Difficulty: - | Category: -]

Any user can call claim_rewards_from. However, this does not happen automatically when calling endpoints like undelegate. This means that if a user U calls undelegate while some rewards could be retrieved by claim_rewards_from, then those rewards are not included when calculating how much EGLD to send to U, although part of those rewards were generated by the U's tokens.

In the extreme, if the contract has a single user U, some rewards R are available in the DC contracts, and U undelegates all his shares, then U will not get the R rewards, and the first user to delegate some EGLD will get them.

Github issue.

Suggestions

1. The LS contract could require that the rewards were claimed in the current epoch when undelegating.
2. The current solution could also work if rewards are claimed automatically each epoch, and one epoch's rewards are small enough that it's not worth the trouble.

Status

This is solved at Commit 2 by allowing users to claim and delegate rewards.

A13 - The penalty system assumes two working Staking Provider contracts

[Severity: Informative | Difficulty: - | Category: -]

The administrator can undelegate and unbond some EGLD from a contract and then delegate it to another contract. In order for this to work, there should be at least one Staking Provider contract besides the one being penalized that can receive these EGLD tokens.

Github issue.

A14 - Not handling the case when a Staking Provider contract loses funds

[Severity: Informative | Difficulty: - | Category: -]

It is not clear whether a Staking Provider contract can lose user funds (e.g. by incorrect accounting, getting stuck, or the funds getting stolen). It's also not clear whether it can happen that only a small number of such contracts lose funds (perhaps it could happen to, say, a contract, after which the others are stopped until the issue is fixed).

In the following, let us assume that it can happen for a single Staking Provider contract, or a small number of such contracts.

If that's the case, then the admin can try to withdraw funds from that contract. However, if that's not possible, then the Liquid Staking contract will think it has more EGLD available than it actually has. If all users try to withdraw their tokens, then the last users to do this will not be able to withdraw anything. This will probably cause most/many users to attempt to withdraw their tokens, in order to not be the last ones to do so.

Github issue.

Suggestions

1. Allow the admin to recompute the share value by, e.g., changing the cash_reserve. This could be done in a safe way by, e.g., allowing the admin to declare a Staking Provider contract as a "lost cause", which would, among other things, remove its entire total_delegated value (or only part of it, or perhaps other values like total_undelegated should be removed) from the cash_reserve.
   This suggestion requires more design work to make it reasonable.

A15 - Trusted contracts should be in the same shard

[Severity: High | Difficulty: High | Category: -]

This is about admin actions, so the admin can avoid issues if careful.

Various callbacks make same-shard calls to trusted contracts. It would be safer if add_trusted_contract checked that the trusted_contract is in the same shard as the Liquid Staking contract.

Github issue.

Suggestions

1. add_trusted_contract should check that the trusted_contract is in the same shard as the Liquid Staking contract.

Status

Fixed at Commit 2 by removing trusted contracts.

A16 - Missing check when setting roles for undelegate_token

[Severity: Informative | Difficulty: - | Category: Best practices]

The endpoints mentioned below can be called only by admins, so the admin can avoid issues if careful.

Set_undelegate_token_roles does not require that undelegate_token is set, while set_ls_token_roles requires that ls_token is set.

Github issue.

Suggestions

1. Check that undelegate_token is set in set_undelegate_token_roles

Status

Fixed at Commit 2

A17 - Inconsistent handling of total_fee in init

[Severity: Informative | Difficulty: - | Category: Best practices]

This is about admin actions, so the admin can avoid issues if careful.

Init sets total_fee to 0 if non-empty, instead of leaving it unset.

At the same time, set_total_fee allows only non-zero values, and set_state_active requires a non-empty total_fee, which is inconsistent with what init does.

Github issue.

Suggestions

1. Do not set total_fee in init.

Status

Fixed at Commit 2

A18 - Some endpoints do not produce events

[Severity: Informative | Difficulty: - | Category: Best practices]

add_trusted_contract and remove_trusted_contract do not produce events.

Github issue.
unbond_cb, on success, if there is not enough egld, does not emit events.

Github issue.

Suggestions

1. Generate events in add_trusted_contract and remove_trusted_contract.
2. Generate events for the missing case in unbond_cb.

Status

The two issues mentioned above were fixed at Commit 2

A19 - Panic in the unbond_from_cb callback

[Severity: Medium | Difficulty: - | Category: Protocol Invariants]

In theory, the following should never happen. However, this code looks unsafe.

In the unbond_from_cb callback, on success, if there are not enough EGLD for the given penalty, the contract panics. This means that the EGLD received from the Staking Provider contract (if any) are, practically, lost.

Github issue.

Suggestions

1. unbond_from_cb should behave in a similar way to unbond_cb. It should emit an event instead of panicking.

Status

Fixed at Commit 2

A20 - Staking Provider contract selection could use more precise probabilities

[Severity: Informative | Difficulty: - | Category: -]

When selecting a Staking Provider contract for delegation or undelegation, the Liquid Staking contract will assign probabilities according to a per-contract int weight, such that the sum of all weights equals BPS (10000). Then the contract generates a random number between 0 and BPS inclusively, and checks in which bucket that number is. The buckets are closed intervals at the left, and open at the right, e.g. $[start,\ start + weight)$. This means that BPS is not part of any bucket, but is assigned by the code to the last bucket, increasing its probability by $\frac{1}{10000}$.

Github issue.

Suggestions

1. Generate the random number between 0 and BPS-1 inclusively. The code may look better, too.

Status

Fixed at Commit 2

Refactoring

A21 - Duplicated code in unbond and unbond_cb

[Severity: Informative | Difficulty: - | Category: Best practices]

There is some common code in unbond and unbond_cb - it may be worth extracting to a function. This also applies to unbond_from and unbond_from_cb.

Github issue.

Suggestions

1. Extract a function with the common code.

Status

Fixed at Commit 2

A22 - Duplicated code for updating contract score

[Severity: Informative | Difficulty: - | Category: Best practices]

There are two very similar pieces of code that recompute the score for a Staking Provider contract (in delegation.rs and governance.rs), then remove/insert the contract in the delegation_contracts_list to keep it sorted.

Github issue.

Suggestions

1. Extract a function with the common code.

Commit 2 audit

Liquid Staking contract storage

- $State$ (Active/Inactive)
- $LsTokenSupply$
- $CashReserve$
- $RewardsReserve$
- $ProtocolReserve$
- $TotalUndelegated$
- $TotalWithdrawable$
- $UnbondPeriod$
- $MigrationWhitelist$
- $NumWhitelistedUsers$
- $DcData(contract)$ - delegation_contract_data
    - $TotalDelegated$
    - $TotalUndelegated$
    - $TotalWithdrawable$
    - $Admin$
    - $PendingToDelegate$
    - $PendingToUndelegate$
- $UndelegationMode$
- $Admin$
- Penalties
    - $Penalties$ is the set of all penalties that are still relevant to the LS contract
        - The Penalty data structure changed at Commit 2

- Contracts
  - $SortedContracts$ is a list of contracts sorted by delegation score. This list is used when selecting a contract to be used by various endpoints.
  - $DcData$ holds data for each Staking Provider contract with which the LS contract currently works.
  - $Outdated$ are the Staking Provider contracts marked as outdated.
  - $Blacklisted$ is a list of all contracts blacklisted in the past by the LS contract, but which were not re-whitelisted. A blacklisted contract may not be in $DcData$.
- Ls_token, undelegate_token, undelegate_token_name, unbond_period

Note that, if DC is a Staking Provider contract, then values like $Delegated_{DC}$ should be close, but not equal to values like $DcData(DC).TotalDelegated$ because the former describes the actual state of a Staking Provider contract, while the latter is the LS contract's view of the Staking Provider contract's state, and there is some time lag between the actual state changes and the LS contract updates.

Invariant

1. The following values are either 0 or greater or equal to $MIN\_DELEGATION\_AMOUNT$, for all known contracts DC and all penalties P:
   a. $DcData(DC).PendingToDelegate$
   b. $DcData(DC).TotalDelegated$
   c. $DcData(DC).PendingToUndelegate$
   d. $DcData(DC).TotalUndelegated$
      i. This is not enforced directly in the code, but it should be true because, due to how undelegation are being made, withdraw_from should return a value that is either 0 or greater or equal to $MIN\_DELEGATION\_AMOUNT$.
   e. $P.egld$
2. The contract holds at least an amount of EGLD equal to the sum of:
   a. $DcData(DC).PendingToDelegate$
   b. $DcData(DC).TotalWithdrawable$
   c. $P.egld$ where $P \in Penalties \land P.withdrawn$
   d. $RewardsReserve$
   e. $ProtocolReserve$

Findings

Also see the Notable findings and Informative findings sections to see which issues from the previous report were fixed and which were not.

B01 - Users can prevent the contract from going into the Open mode

[Severity: High | Difficulty: Low | Category: Core properties]

Any user can prevent the Liquid Staking contract from going into Open mode by delegating + undelegating small amounts to a delegation contract. This works well with contracts that do not have caps, but it could also work with contracts that have caps, where the invested amount is significantly less than the cap.

Github issue.

Example

Let's say that there are 10 contracts with a lot of delegated tokens, and one more contract without a cap (C) and without delegated tokens. The 10 contracts are outdated, and the contract without a cap has no tokens, so nobody can undelegate. However, a user delegates 1 EGLD to C and undelegates it. Most users still can't undelegate, but the last_undelegate timestamp was updated. After a while, C delegates again 1 EGLD and undelegates it. Nobody else can undelegate, but the LS contract still does not go into Open mode. And so on, forever.

Suggestions

One option might be to check how long ago a contract was marked as outdated without being blacklisted. If there is a contract that was marked as outdated a week ago and whose data was never refreshed, then it may mean that the admin is not doing his job, so it may be reasonable to go into the Open mode.

Hatom suggested keeping both conditions:

- if the last undelegation is a long time ago, allow open mode or,
- if the admin is not updating contract data (even if they are not mark as outdated), allow open mode

B02 - Penalties can’t always be redelegated

[Severity: Medium | Difficulty: Low | Category: -]

Normally, there should be some delegation contracts without caps. However, unless that's 100% certain, it is possible that all delegation contracts are full, and that there are withdrawn penalties that can't be delegated. No user funds are lost, so this is mostly fine, but the APR for users would be lower than they would expect from the APR of all delegation contracts, and there is something that may make things better for users, at the cost of increasing contract complexity, see below.

Suggestions

Currently, after a user undelegates, a contract will have some space available, so someone may try to delegate part of the penalty above. This is a race against other users that may also want to delegate. However, it would be possible to either make the undelegate call also delegate part of the penalty, or to skip undelegation and pay the user directly from the penalty (with or without making the user wait for the unbond period).

B03 - Users may not be able to withdraw small amounts of funds when the liquidity is low

[Severity: Medium | Difficulty: Low | Category: Core properties]

If a contract has between MIN_DELEGATION_AMOUNT (inclusive) and 2 * MIN_DELEGATION_AMOUNT (exclusive) EGLD, then these tokens must be undelegated in a single request (the LS contract does not let users withdraw less than MIN_DELEGATION_AMOUNT, and withdrawing an amount greater or equal to MIN_DELEGATION_AMOUNT and less than the total amount of EGLD is also forbidden as it would leave dust in the contract.

When undelegating, a user provides an amount of shares, which is converted into EGLD by the LS contract. The main issue here is that there may be no way to produce exactly the amount of EGLD delegated with the current exchange rate, so withdrawing may be impossible. This may be true even if the exchange rate changes (e.g. due to rewards being collected + delegated).

Quick experiments suggest that a delegated amount of WAD can be withdrawn with ~70% of all possible exchange rates, while WAD+3 cannot be withdrawn with almost any possible exchange rate.

The same applies when withdrawing from a penalty.

Github issue.

Example

Let's say that a delegation contract has MIN_DELEGATION_AMOUNT tokens delegated. The value of MIN_DELEGATION_AMOUNT is represented as 1_000_000_000_000_000_000, i.e. 1 EGLD.

Let's say that a user would like to undelegate 0.9 shares, and that the exchange rate is 1.11111...:

If the user sends 0.9 shares, that is a value of 900_000_000_000_000_000, the LS contract will think that the user tries to undelegate an EGLD value represented as 999_999_999_999_999_999, which is less than MIN_DELEGATION_AMOUNT, so undelegation will fail.

If the user sends a value of 900_000_000_000_000_001, the LS contract will think that the user tries to undelegate 1_000_000_000_000_000_001, which is more than what the delegation contract holds.

This means that, until the exchange rate changes, nobody will be able to withdraw the amount held by the delegation contract. However, even with a different exchange rate, it is possible that the same issue will apply.

Suggestion

Round an amount of EGLD slightly higher than MIN_DELEGATION_AMOUNT to MIN_DELEGATION_AMOUNT in the cases mentioned above.

Assumptions

Due to the removal of trusted contracts and the redesign of async calls and of penalties, most assumptions in the audit for Commit 1 are not needed anymore, see Appendix B: Assumptions.

Appendix A: Endpoint summaries

Delegation score computation

A Stakind Provider contract's score is a fixed-point fractional value between 0 and 1. It is computed as a linear combination of a TVL score (based on the total value locked in the DC contract) and an APR score (based on the contract's annual percentage rate).

The TVL score is 1 if the total value locked is below a minimum amount (configurable), 0 if it's above a maximum amount (also configurable) and it decreases linearly from 1 to 0 as the total value locked grows from the minimum to the maximum value.

The APR score is similar: its value is 0 below a minimum value and 1 above a maximum value (both configurable), and it grows linearly between the two.

Contract selection

- For Delegate
  - The LS contract finds a contract DC with the maximum score such that
    - $DC \in DcData - Outdated$
    - $DelegationAmount \leq DcData[DC].cap - DcData[DC].TotalValueLocked$
    - DC's fee is not "too high"
  - The LS contract just uses DC, unless configured to pick randomly.
  - If configured to pick randomly:
    - The LS contract picks all DC contracts satisfying the constraints above and which are also above a certain percentage of DC's score.
    - The LS contract computes a weight for each of the selected contracts, then picks a random contract, with probabilities proportional to the weights.
- For Undelegate
  - The same as above, except for using the minimum score instead of the maximum one. Also, the $DelegationAmount$ constraint should be replaced with:

            - $DcData[DC].TotalDelegated - UndelegationAmount = 0$
              $\vee\ DcData[DC].TotalDelegated - UndelegationAmount >= MinEgld$

Operation: delegate

Allows a user to stake EGLD in exchange for sEGLD. Delegates the EGLD based on the current configuration of the delegation algorithm. If the delegation is successful, the callback updates the storage and mints sEGLD to the caller. Otherwise, it returns the EGLD back to the caller and sets the Delegation smart contract data as outdated. Notice that the smart contract data will be outdated until it is updated.

User call

Notations

- U := the caller of the delegate endpoint
- E := EGLD sent to delegate
- DC := $MigrationWhitelist[U]$ *or else use contract selection*
    - Contract selection adds the following preconditions:
        - $DC \in DcData - Outdated$
        - $E \leq DcData[DC].cap - DcData[DC].TotalValueLocked$
- D := $Delegate(U, DC, E)$

Transition

delegate() [EGLD payment]

Pre-conditions

- $State = Active$
- $E \geq MinEgld$
- There is a DC contract picked as above

Updates

- State
    - $'EGLD_U = EGLD_U - E$
    - $'EGLD_S^D = E$
    - $'Pending_{Delegate}^{Unhandled} = Pending_{Delegate}^{Unhandled} + D$

Invariant

- All invariant properties are preserved

Staking Provider contract, success

Notations

- $E := EGLD_S^D$

Transition

delegate() [E=EGLD payment, $D \in Pending_{Delegate}^{Unhandled}$ is the pending request]

Pre-conditions

- $E \leq DC.cap - DC.TotalValueLocked$

Updates

- State
  - $'EGLD_S^D = 0$
  - $'Delegated_{DC} = Delegated_{DC} + E$
  - $'DC.TotalValueLocked = DC.TotalValueLocked + E$
  - $'Pending_{Delegate}^{Unhandled} = Pending_{Delegate}^{Unhandled} - D$
  - $'Pending_{Delegate}^{Success} = Pending_{Delegate}^{Success} + D$

Invariant

- 19 may not be preserved
- All other invariant properties are preserved

Callback, success

Notations

- $NewShares = \frac{E}{FX}$

Transition

delegate_cb(U, DC, E) [$D \in Pending_{Delegate}^{Success}$ is the pending request]

Pre-conditions

- $NewShares > 0$

Updates

- State

   - $'Shares = Shares + NewShares$ (mints new shares)
   - $'Shares_U = Shares_U + NewShares$
   - $'Pending^{Success}_{Delegate} = Pending^{Success}_{Delegate} - D$
- Storage
   - $'DcData(DC).TotalDelegated = DcData(DC).TotalDelegated + E$
   - $'CashReserve = CashReserve + E$

Invariant

- 17 may not be preserved
- 21 is preserved because it is impossible to have a delegate call starting while blacklisting a contract, and pending long enough that the contract is removed before the callback.
- All other invariant properties are preserved

Staking Provider contract, failure

Notations

- $E := EGLD^D_S$

Transition

delegate() [EGLD payment, $D \in Pending^{Unhandled}_{Delegate}$ is the pending request]

Updates

- State
   - $'EGLD^D_S = 0$
   - $'EGLD^{cb(D)}_S = E$
   - $'Pending^{Unhandled}_{Delegate} = Pending^{Unhandled}_{Delegate} - D$
   - $'Pending^{Failure}_{Delegate} = Pending^{Failure}_{Delegate} + D$

Invariant

- All invariant properties are preserved

Callback, failure

Notations

- $E := EGLD^{cb(D)}_S$

Transition

delegate_cb(U, DC, E) [$D \in Pending^{Failure}_{Delegate}$ is the pending request]

Updates

- State
  - $'EGLD_U = EGLD_U + E$
  - $'EGLD^{cb(D)}_S = 0$
  - $'Pending^{Failure}_{Delegate} = Pending^{Failure}_{Delegate} - D$
- Storage
  - $'Outdated = Outdated + DC$

Invariant

- All invariant properties are preserved

Operation: delegate_trusted_contract

This is similar to operation:delegate with the following changes:

- The caller must be a trusted contract.
- The LS contract reserves some gas for calling trusted contract endpoints
- On success, the callback calls an endpoint of the trusted contract
  - If that endpoint fails, the callback ends with panic.
- On failure, the callback calls another endpoint of the trusted contract
  - If that endpoint fails, the callback ends with panic.

Operation: delegate_to

Allows the admin to delegate a Penalty to a new Staking Provider based on the current configuration of the delegation algorithm (avoiding the penalized Delegation smart contract). If the delegation is successful, the callback updates the storage and deletes the penalty from the storage. Otherwise it sets the Delegation smart contract data as outdated. Notice that the smart contract data will be outdated until it is updated.

UserCall

Notations

- U := the caller of the delegate endpoint
- P := $Penalties[PenaltyId]$
- E := $P.egld$
- DC := $MigrationWhitelist[U]$ $or\ else\ use\ contract\ selection\ such\ that\ DC \neq P$

    - Contract selection adds the following preconditions:
        - $DC \in DcData - Outdated$
        - $E \leq DcData[DC].cap - DcData[DC].TotalValueLocked$
- $D := DelegateTo(U, DC, E, PenaltyId)$

Transition

delegate_to(PenaltyId)

Pre-conditions

- $U = Admin$
- $State = Active$
- $PenaltyId \in Penalties$
- $P.unbonded$
- There is a DC contract picked as above

Updates

- State
    - $'EGLD^{P}_{LS} = EGLD^{P}_{LS} - E$
    - $'EGLD^{D}_{S} = E$
    - $'Pending^{Unhandled}_{DelegateTo} = Pending^{Unhandled}_{DelegateTo} + D$

Invariant

- 29 does not hold for concurrent calls.
- All other invariant properties are preserved

Staking Provider contract, success

Notations

- $E := EGLD^{D}_{S}$

Transition

delegate() [EGLD payment, $D \in Pending^{Unhandled}_{DelegateTo}$ is the pending request]

Pre-conditions

- $E \leq DC.cap - DC.TotalValueLocked$

Updates

- State

- $'EGLD_S^D = 0$
- $'Delegated_{DC} = Delegated_{DC} + E$
- $'DC.TotalValueLocked = DC.TotalValueLocked + E$
- $'Pending_{DelegateTo}^{Unhandled} = Pending_{DelegateTo}^{Unhandled} - D$
- $'Pending_{DelegateTo}^{Success} = Pending_{DelegateTo}^{Success} + D$

Invariant

- 16 - may not be preserved
- All invariant properties are preserved

Callback, success

Transition

delegate_to_cb(U, DC, E) [$D \in Pending_{DelegateTo}^{Success}$ is the pending request]

Updates

- State
  - $'Pending_{DelegateTo}^{Success} = Pending_{DelegateTo}^{Success} - D$
- Storage
  - $'DcData(DC).TotalDelegated = DcData(DC).TotalDelegated + E$
  - $'Penalties = Penalties - PenaltyId$

Invariant

- 17 may not be preserved
- 21 - preserved because it is impossible to have a delegate_rewards call starting while blacklisting a contract, and pending long enough that the contract is removed before the callback.
- All other invariant properties are preserved

Staking Provider contract, failure

Notations

- $E := EGLD_S^D$

Transition

delegate() [EGLD payment, $D \in Pending_{DelegateTo}^{Unhandled}$ is the pending request]

Updates

- State
    - $'EGLD_S^D = 0$
    - $'EGLD_S^{cb(D)} = E$
    - $'Pending_{DelegateTo}^{Unhandled} = Pending_{DelegateTo}^{Unhandled} - D$
    - $'Pending_{DelegateTo}^{Failure} = Pending_{DelegateTo}^{Failure} + D$

Invariant

- All invariant properties are preserved

Callback failure

Notations

- $E := EGLD_S^{cb(D)}$
- $P := Penalties[D.PenaltyId]$

Transition

delegate_to_cb(U, DC, E) [$D \in Pending_{DelegateTo}^{Failure}$ is the pending request]

Updates

- State
    - $'EGLD_S^{cb(D)} = 0$
    - $'EGLD_{LS}^P = EGLD_{LS}^P + E$
    - $'Pending_{DelegateTo}^{Failure} = Pending_{DelegateTo}^{Failure} - D$
- Storage
    - $'Outdated = Outdated + DC$

Invariant

- All invariant properties are preserved

Operation: undelegate

Allows a user to unstake EGLD in exchange for their sEGLD. Un-delegates the EGLD based on the current configuration of the undelegation algorithm. If the undelegation is successful, the callback updates the storage, burns the sEGLD and mints an NFT to the caller. Otherwise, it returns the sEGLD back to the caller and sets the Delegation smart contract data as outdated. Notice that the smart contract data will be outdated until it is updated.

UserCall

Notations

- U := the caller of the undelegate endpoint
- S := sEGLD sent to undelegate
- $E := S \cdot FX$
- DC := $undelegate\ contract\ selection\ result$
    - Contract selection adds the following preconditions:
        - $DC \in DcData - Outdated$
        - $DcData[DC].TotalDelegated - E = 0$
          $\vee\ DcData[DC].TotalDelegated - E >= MinEgld$
- D := $Undelegate(U,\ DC,\ E,\ S)$

Transition

undelegate() [sEGLD payment]

Pre-conditions

- $State\ =\ Active$
- $S\ >\ 0$
- $E\ \geq MinEgld$
- There is a DC contract picked as above

Updates

- State
    - $'Shares_U = Shares_U - S$
    - $'Pending^{Unhandled}_{Undelegate} = Pending^{Unhandled}_{Undelegate} + D$

Invariant

- All invariant properties are preserved

Staking Provider contract, success

Transition

undelegate(E) [$D \in Pending^{Unhandled}_{Undelegate}$ is the pending request]

Pre-conditions

- $E \le Delegated_{DC}$

Updates

- State
  - $'Delegated_{DC} = Delegated_{DC} - E$
  - $'Undelegations_{DC} = Undelegations_{DC} + Undelegation(CurrentEpoch,\ E)$
  - $'DC.TotalValueLocked = DC.TotalValueLocked - E$
  - $'Pending^{Unhandled}_{Undelegate} = Pending^{Unhandled}_{Undelegate} - D$
  - $'Pending^{Success}_{Undelegate} = Pending^{Success}_{Undelegate} + D$

Invariant

- 4 may not be preserved for two parallel undelegate calls.
- All other invariant properties are preserved

Callback, success

Transition

undelegate_cb(U, DC, E, S) $[D \in Pending^{Success}_{Undelegate}$ is the pending request]

Pre-conditions

Updates

- State
  - $'Shares = Shares - S$ (burns)
  - $'NFTs_U = NFTs_U + NFT(DC,\ E,\ S,\ CurrentEpoch,\ CurrentEpoch + UnbondPeriod)$ (mints)
  - $'Pending^{Success}_{Undelegate} = Pending^{Success}_{Undelegate} - D$
- Storage
  - $'DcData(DC).TotalDelegated = DcData(DC).TotalDelegated - E$
  - $'DcData(DC).TotalUndelegated = DcData(DC).TotalUndelegated + E$
  - $'TotalUndelegated = TotalUndelegated + E$
  - $'CashReserve = CashReserve - E$

Invariant

- It’s not clear whether 16 is preserved or not. However, currently there are better examples where 16 is not preserved, and fixing those will probably also make it clear that 16 is also preserved here.

- All other invariant properties are preserved

Staking Provider contract, failure

Transition

undelegate(E) [$D \in Pending^{Unhandled}_{Undelegate}$ is the pending request]

Updates

- State
  - $'Pending^{Unhandled}_{Undelegate} = Pending^{Unhandled}_{Undelegate} - D$
  - $'Pending^{Failure}_{Undelegate} = Pending^{Failure}_{Undelegate} + D$

Invariant

- All invariant properties are preserved

Callback failure

Transition

undelegate_cb(U, DC, E, S) [$D \in Pending^{Failure}_{Undelegate}$ is the pending request]

Updates

- State
  - $'Shares_U = Shares_U + S$
  - $'Pending^{Failure}_{Undelegate} = Pending^{Failure}_{Undelegate} - D$
- Storage
  - $'Outdated = Outdated + DC$

Invariant

- 2 may not be preserved if the FX changed since the undelegate call.
- All invariant properties are preserved

Operation: undelegate_trusted_contract

This is similar to operation:undelegate with the following changes:

- The caller must be a trusted contract.
- The LS contract reserves some gas for calling trusted contract endpoints
- On success, the callback calls an endpoint of the trusted contract

    - If that endpoint fails, the callback ends with panic.
- On failure, the callback calls another endpoint of the trusted contract
    - If that endpoint fails, the callback ends with panic.

Operation: undelegate_from

Allows the admin to undelegate an amount of EGLD directly from a given Staking Provider. If the undelegation is successful, the callback updates the storage and creates a Penalty structure. Otherwise, it sets the Delegation smart contract data as outdated. Notice that the smart contract data will be outdated until it is updated.

Additionally, if the administrator asks for all available EGLD to be undelegated, the Staking Provider contract is marked for deletion (i.e. it is blacklisted).

UserCall

Notations

- U := the caller of the delegate endpoint
- $ShouldBlacklist$:= $E = DC.TotalDelegated$
- UD := $Undelegate(U, DC, E, ShouldBlacklist)$

Transition

undelegate_from(DC, E)

Pre-conditions

- $U = Admin$
- $State = Active$
- $DC \in DcData$
- $E \leq DC.TotalDelegated$
- $E \geq MinEgld$
- $ShouldBlacklist$ implies rewards were claimed in the current epoch

Updates

- State
    - $'Pending^{Unhandled}_{UndelegateFrom} = Pending^{Unhandled}_{UndelegateFrom} + UD$

Invariant

- All invariant properties are preserved

Staking Provider contract, success

Transition

undelegate(E) [$UD \in Pending^{Unhandled}_{UndelegateFrom}$ is the pending request]

Pre-conditions

- $E \geq Delegated_{DC}$

Updates

- State
  - $'Delegated_{DC} = Delegated_{DC} - E$
  - $'Undelegations_{DC} = Undelegations_{DC} + Undelegation(CurrentEpoch,\ E)$
  - $'DC.TotalValueLocked = DC.TotalValueLocked - E$
  - $'Pending^{Unhandled}_{UndelegateFrom} = Pending^{Unhandled}_{UndelegateFrom} - UD$
  - $'Pending^{Success}_{UndelegateFrom} = Pending^{Success}_{UndelegateFrom} + UD$

Invariant

- 4 may not be preserved for parallel undelegate/undelegate_from calls.
- All invariant properties are preserved

Callback, success

Notations

- $P = Penalty(unbonded = false, DC,\ E,\ S,\ CurrentEpoch,\ CurrentEpoch + UnbondPeriod)$

Transition

undelegate_from_cb(U, DC, E, S) [$UD \in Pending^{Success}_{UndelegateFrom}$ is the pending request]

Updates

- State
  - $'Pending^{Success}_{UndelegateFrom} = Pending^{Success}_{UndelegateFrom} - UD$
- Storage
  - $'DcData(DC).TotalDelegated = DcData(DC).TotalDelegated - E$
  - $'DcData(DC).TotalUndelegated = DcData(DC).TotalUndelegated + E$
  - $'TotalUndelegated = TotalUndelegated + E$
  - $'Penalties = Penalties + P$
  - If $ShouldBlacklist$

        - $'Blacklisted = Blacklisted + DC$
        - $'Outdated = Outdated + DC$
        - $'SortedContracts = SortedContracts - DC$

Invariant

- 17 - may not be preserved
- 16 - may not be preserved
- All other invariant properties are preserved

Staking Provider contract, failure

Transition

undelegate(E) $[D \in Pending^{Unhandled}_{UndelegateFrom}$ is the pending request]

Updates

- State
    - $'Pending^{Unhandled}_{UndelegateFrom} = Pending^{Unhandled}_{UndelegateFrom} - UD$
    - $'Pending^{Failure}_{UndelegateFrom} = Pending^{Failure}_{UndelegateFrom} + UD$

Invariant

- All invariant properties are preserved

Callback failure

Transition

undelegate_from_cb(U, DC, E, S) $[UD \in Pending^{Failure}_{UndelegateFrom}$ is the pending request]

Updates

- State
    - $'Pending^{Failure}_{UndelegateFrom} = Pending^{Failure}_{UndelegateFrom} - UD$
- Storage
    - $'Outdated = Outdated + DC$

Invariant

- All invariant properties are preserved

Operation: <time passes>

This is not part of the Liquidity Staking contract interface, but it is included here in order to provide a more complete image of the contract by modeling the changes that are expected to occur in the Staking Provider contracts as time passes.

Updates

- State
  - For all $U \in Undelegations_{DC}$
    - If $U.UnbondTime \leq CurrentTime$
      - $'Undelegations_{DC} = Undelegations_{DC} - U$
      - $'Unbondings_{DC} = Unbondings_{DC} + Unbonding(U.contract, U.egld)$
  - $'Rewards_{DC} = Rewards_{DC} + computeRewards(\Delta t, Delegated_{DC})$

Invariant

- All invariant properties are preserved

Operation: unbond

Allows a user to get their staked EGLD after the undelegation or unbond time period has passed. Users must pay with the NFT received at the undelegate interaction. The contract to contract call is only executed if needed, i.e. there might have been a previous unbonding that already brought the EGLD from the Staking Provider smart contract to this liquid staking smart contract. If it is successful, it updates the storage, burns the NFT and sends the EGLD to the user. Otherwise, the NFT is returned back to the user.

On successful completion, if the Staking Provider contract was marked for deletion by the administrator (i.e. it was blacklisted) and it does not hold any tokens delegated by the Liquid Staking contract, then the Staking Provider contract is removed from the internal data structures, except for the blacklist.

UserCall, enough EGLD

This models the case when a previous unbonding brought enough EGLD from the Staking Provider smart contract to send to the user.

Notations

- $E := NFT.EGLD$
- $DC := NFT.Contract$

Transition

unbond() [NFT payment]

Pre-conditions

- $State = Active$
- $CurrentTime \geq NFT.UnbondTime$
- $DcData(DC).TotalWithdrawable \geq E$

Updates

- State
    - $'NFTs_U = NFTs_U - NFT$ (burns)
    - $'Withdrawable_{DC} = Withdrawable_{DC} - E$
    - $'EGLD_U = EGLD_U + E$
- Storage
    - $'DcData(DC).TotalWithdrawable = DcData(DC).TotalWithdrawable - E$
    - $'DcData(DC).TotalUndelegated = DcData(DC).TotalUndelegated - E$
    - $'TotalWithdrawable = TotalWithdrawable - E$
    - $'TotalUndelegated = TotalUndelegated - E$
    - If $DC \in Blacklisted$
      and $'DcData(DC).TotalWithdrawable = 0$
      and $'DcData(DC).TotalUndelegated = 0$
      and $'DcData(DC).TotalDelegated = 0$
        - $'Outdated = Outdated - DC$
        - $'DcData = DcData - DC$

Invariant

- All invariant properties are preserved

UserCall, not enough EGLD

This models the case when the Liquid Staking contract does not have enough EGLD to send to the user, so it calls the Staking Provider contract in order to get all unbonded EGLD available.

Notations

- $E := NFT.EGLD$
- $DC := NFT.Contract$

Transition

unbond() [NFT payment]

Pre-conditions

- $State = Active$
- $CurrentTime \geq NFT.UnbondTime$
- $DcData(DC).TotalWithdrawable < E$

Updates

- State
    - $'NFTs_U = NFTs_U - NFT$
    - $'NFTs_{LS} = NFTs_{LS} + NFT$
    - $'Pending^{Unhandled}_{Unbond} = Pending^{Unhandled}_{Unbond} + Unbond(U, DC, E, NFT)$

Invariant

- All invariant properties are preserved

Staking Provider contract, success

Notations

- $W := Unbond_{DC} = \sum_{U \in Unbondings_{DC}} U.EGLD$

Transition

DC.withdraw() $[UB \in Pending^{Unhandled}_{Unbond}$ is the pending request]

Updates

- State
    - $'Unbondings_{DC} = \emptyset$
    - $'EGLD^{cb(UB)}_S = W$
    - $'Pending^{Unhandled}_{Unbond} = Pending^{Unhandled}_{Unbond} - UB$
    - $'Pending^{Success}_{Unbond} = Pending^{Success}_{Unbond} + UB$

Invariant

- All invariant properties are preserved

Callback, success, enough EGLD

Notations

- $W := EGLD^{cb(UB)}_S$

Transition

unbond_cb(U, DC, E, NFT) $[UB \in Pending^{Success}_{Unbond}$ is the pending request]

Pre-conditions

- $DcData(DC).TotalWithdrawable + W \geq E$

Updates

- State
  - $'NFTs_{LS} = NFTs_{LS} - NFT$ (burns)
  - $'Withdrawable_{DC} = Withdrawable_{DC} + W - E$
  - $'EGLD_{U} = EGLD_{U} + E$
  - $'EGLD_{S}^{cb(UB)} = 0$
  - $'Pending_{Unbond}^{Success} = Pending_{Unbond}^{Success} - UB$
- Storage
  - $'DcData(DC).TotalWithdrawable = DcData(DC).TotalWithdrawable + W - E$
  - $'DcData(DC).TotalUndelegated = DcData(DC).TotalUndelegated - E$
  - $'TotalWithdrawable = TotalWithdrawable - E$
  - $'TotalUndelegated = TotalUndelegated - E$
  - If $DC \in Blacklisted$
    and $'DcData(DC).TotalWithdrawable = 0$
    and $'DcData(DC).TotalUndelegated = 0$
    and $'DcData(DC).TotalDelegated = 0$
    - $'Outdated = Outdated - DC$
    - $'DcData = DcData - DC$

Invariant

- 19 - preserved because it was impossible to make delegate requests since blacklisting, and enough time passed since then.
- All other invariant properties are preserved

Callback, success, not enough EGLD

This models an unexpected case, i.e. when even after getting all unbonded funds from the Staking provider contract, there is not enough EGLD to send to the user. This should not happen, but the Liquidity Staking contract provides reasonable handling in case it does happen.

Notations

- $W := EGLD_{S}^{cb(UB)}$

Transition

unbond_cb(U, DC, E, NFT) [$UB \in Pending_{Unbond}^{Success}$ is the pending request]

Pre-conditions

- $DcData(DC).TotalWithdrawable + W < E$

Updates

- State
  - $'NFTs_{LS} = NFTs_{LS} - NFT$
  - $'NFTs_{U} = NFTs_{U} + NFT$
  - $'Withdrawable_{DC} = Withdrawable_{DC} + W$
  - $'EGLD_{S}^{cb(U)} = 0$
  - $'Pending_{Unbond}^{Success} = Pending_{Unbond}^{Success} - UB$
- Storage
  - $'DcData(DC).TotalWithdrawable = DcData(DC).TotalWithdrawable + W$
  - $'TotalWithdrawable = TotalWithdrawable + W$

Invariant

- All invariant properties are preserved

Staking Provider contract, failure

Transition

DC.withdraw() [$UB \in Pending_{Unbond}^{Unhandled}$ is the pending request]

Updates

- State
  - $'Pending_{Unbond}^{Unhandled} = Pending_{Unbond}^{Unhandled} - UB$
  - $'Pending_{Unbond}^{Failure} = Pending_{Unbond}^{Failure} + UB$

Invariant

- All invariant properties are preserved

Callback failure

Transition

unbond_cb(U, DC, E, NFT) [$UB \in Pending_{Unbond}^{Failure}$ is the pending request]

Updates

- State
  - $'NFTs_{LS} = NFTs_{LS} - NFT$
  - $'NFTs_{U} = NFTs_{U} + NFT$
  - $'Pending^{Failure}_{Unbond} = Pending^{Failure}_{Unbond} - UB$

Invariant

- All invariant properties are preserved

Operation: unbond_from

Allows the admin to unbond a Penalty from a Staking Provider after the undelegation or unbond time period has passed. The contract to contract call is only executed if needed, i.e. there might have been a previous unbonding that already brought the EGLD from the Staking Provider smart contract to this liquid staking smart contract. If it is successful, it updates the storage including the Penalty details.

On successful completion, if the Staking Provider contract was marked for deletion by the administrator (i.e. it was blacklisted) and it does not hold any tokens delegated by the Liquid Staking contract, then the Staking Provider contract is removed from the internal data structures, except for the blacklist.

UserCall, enough EGLD

This models the case when a previous unbonding brought enough EGLD from the Staking Provider smart contract to cover the Penalty amount.

Notations

- U := the caller of the delegate endpoint
- P := $Penalties[PenaltyId]$
- E := $P.egld$
- DC := $P.contract$

Transition

unbond_from(PenaltyId)

Pre-conditions

- $U = Admin$
- $State = Active$
- $PenaltyId \in Penalties$
- $\neg P.unbonded$

- $CurrentTime \geq P.UnbondTime$
- $DcData(DC).TotalWithdrawable \geq E$

Updates

- State
  - $'Withdrawable_{DC} = Withdrawable_{DC} - E$
  - $'EGLD_{LS}^{P} = EGLD_{LS}^{P} + E$
- Storage
  - $'DcData(DC).TotalWithdrawable = DcData(DC).TotalWithdrawable - E$
  - $'DcData(DC).TotalUndelegated = DcData(DC).TotalUndelegated - E$
  - $'TotalWithdrawable = TotalWithdrawable - E$
  - $'TotalUndelegated = TotalUndelegated - E$
  - $'P.unbonded = true$
    - The audited code sends some EGLD instead of setting $P.unbonded$. An issue was filed (see the findings section) and the Hatom team fixed it, so we are working with the fixed code here.
  - If $DC \in Blacklisted$
    and $'DcData(DC).TotalWithdrawable = 0$
    and $'DcData(DC).TotalUndelegated = 0$
    and $'DcData(DC).TotalDelegated = 0$
    - $'DcData = DcData - DC$
    - $'Outdated = Outdated - DC$

Invariant

- 18 - preserved because it is impossible to have a delegate_rewards call starting while blacklisting a contract, and pending long enough that the contract is removed before the callback.
- All invariant properties are preserved

UserCall, not enough EGLD

This models the case when the Liquid Staking contract does not have enough EGLD to cover the Penalty amount, so it calls the Staking Provider contract in order to get all unbonded EGLD available.

Notations

- U := the caller of the delegate endpoint
- P := $Penalties[PenaltyId]$
- E := $P.egld$
- DC := $P.contract$

Transition

unbond_from(PenaltyId)

Pre-conditions

- $U = Admin$
- $State = Active$
- $PenaltyId \in Penalties$
- $\neg P.unbonded$
- $CurrentTime \geq P.UnbondTime$
- $DcData(DC).TotalWithdrawable < E$

Updates

- State
  - $'Pending^{Unhandled}_{UnbondFrom} = Pending^{Unhandled}_{UnbondFrom} + Unbond(U, DC, E, P)$

Invariant

- All invariant properties are preserved

Staking Provider contract, success

Notations

- $W := \sum_{U \in Unbondings_{DC}} U.EGLD$

Transition

DC.withdraw() [$UB \in Pending^{Unhandled}_{UnbondFrom}$ is the pending request]

Updates

- State
  - $'Unbondings_{DC} = \emptyset$
  - $'EGLD^{cb(UB)}_{S} = W$
  - $'Pending^{Unhandled}_{UnbondFrom} = Pending^{Unhandled}_{UnbondFrom} - UB$
  - $'Pending^{Success}_{UnbondFrom} = Pending^{Success}_{UnbondFrom} + UB$

Invariant

- All invariant properties are preserved

Callback, success, enough EGLD

Notations

- $W := EGLD_{S}^{cb(UB)}$

Transition

unbond_from_cb(U, DC, E, P) $[UB \in Pending_{UnbondFrom}^{Success}$ is the pending request]

Pre-conditions

- $DcData(DC).TotalWithdrawable + W \geq E$

Updates

- State
  - $'Withdrawable_{DC} = Withdrawable_{DC} + W - E$
  - $'EGLD_{LS}^{P} = EGLD_{LS}^{P} + E$
  - $'Pending_{UnbondFrom}^{Success} = Pending_{UnbondFrom}^{Success} - UB$
  - $'EGLD_{S}^{cb(UB)} = 0$
- Storage
  - $'DcData(DC).TotalWithdrawable = DcData(DC).TotalWithdrawable + W - E$
  - $'DcData(DC).TotalUndelegated = DcData(DC).TotalUndelegated - E$
  - $'TotalWithdrawable = TotalWithdrawable + W - E$
  - $'TotalUndelegated = TotalUndelegated - E$
  - $P.unbonded = true$
  - If $DC \in Blacklisted$
    and $'DcData(DC).TotalWithdrawable = 0$
    and $'DcData(DC).TotalUndelegated = 0$
    and $'DcData(DC).TotalDelegated = 0$
    - $'DcData = DcData - DC$
    - $'Outdated = Outdated - DC$

Invariant

- 6 may not be preserved for concurrent calls.
- 18 - preserved because it is impossible to have a delegate_rewards call starting while blacklisting a contract, and pending long enough that the contract is removed before the callback.
- 24 may not be preserved for concurrent calls (setting unbonded to true)
- 29 may not be preserved for concurrent calls ($EGLD_{LS}^{P}$ is increased twice)
- All other invariant properties are preserved

Callback, success, not enough EGLD

This models an unexpected case, i.e. when even after getting all unbonded funds from the Staking provider contract, there is not enough EGLD to cover the Penalty amount. This should not happen, but it would be better if the Liquidity Staking contract provided reasonable handling in case it does happen.

Notations

- $W := EGLD_S^{cb(UB)}$

Transition

unbond_from_cb(U, DC, E, P) $[UB \in Pending_{UnbondFrom}^{Success}$ is the pending request]

Pre-conditions

- $DcData(DC).TotalWithdrawable + W < E$

Updates

- State
  - $'Withdrawable_{DC} = Withdrawable_{DC} + W$
  - $'Pending_{UnbondFrom}^{Success} = Pending_{UnbondFrom}^{Success} - UB$
  - $'EGLD_S^{cb(UB)} = 0$
- Storage updates: the contract panics here. An issue was filled. Assuming it was fixed, most likely the following updates will happen:
  - $'DcData(DC).TotalWithdrawable = DcData(DC).TotalWithdrawable + W$
  - $'TotalWithdrawable = TotalWithdrawable + W$

Invariant

- Some invariant properties are not preserved if the contract panics.
- All invariant properties are preserved if the contract does the storage updates mentioned above.

Staking Provider contract, failure

Transition

DC.withdraw() $[UB \in Pending_{Unbond}^{Unhandled}$ is the pending request]

Updates

- State

    - $'Pending^{Unhandled}_{UnbondFrom} = Pending^{Unhandled}_{UnbondFrom} - UB$
    - $'Pending^{Failure}_{UnbondFrom} = Pending^{Failure}_{UnbondFrom} + UB$

Invariant

- All invariant properties are preserved

Callback failure

Transition

unbond_from_cb(U, DC, E, NFT) [$UB \in Pending^{Failure}_{Unbond}$ is the pending request]

Updates

- State
    - $'Pending^{Failure}_{UnbondFrom} = Pending^{Failure}_{UnbondFrom} - UB$

Invariant

- All invariant properties are preserved

Operation: claim_rewards_from

Allows anyone to claim rewards from a given Delegation smart contract.

UserCall

Notations

- U := the caller of the claim_rewards_from endpoint

Transition

claim_rewards_from(DC)

Pre-conditions

- $State = Active$
- $DC \in Contracts$
- Did not claim rewards this epoch

Updates

- State

    - $'Pending^{Unhandled}_{ClaimRewards} = Pending^{Unhandled}_{ClaimRewards} + Claim(U,\ DC)$

Invariant

- All invariant properties are preserved

Staking Provider contract, success

Notations

- $E := Rewards_{DC}$

Transition

DC.claim_rewards() [$C \in Pending^{Unhandled}_{ClaimRewards}$ is the pending request]

Updates

- State
    - $'Rewards_{DC} = 0$
    - $'EGLD^{cb(C)}_{S} = E$
    - $'Pending^{Unhandled}_{ClaimRewards} = Pending^{Unhandled}_{ClaimRewards} - C$
    - $'Pending^{Success}_{ClaimRewards} = Pending^{Success}_{ClaimRewards} + C$

Invariant

- All invariant properties are preserved

Callback, success

Notations

- $Pr,\ Rw$:= a split of $EGLD^{cb(C)}_{S}$

Transition

claim_rewards_from_cb(U, DC) [$C \in Pending^{Success}_{ClaimRewards}$ is the pending request]

Updates

- State
    - $'Pending^{Success}_{ClaimRewards} = Pending^{Success}_{ClaimRewards} - C$
    - $'EGLD^{cb(C)}_{S} = 0$

    - $'Rewards_{LS} = Rewards_{LS} + E$
- Storage
    - $'RewardsReserve = RewardsReserve + Rw$
    - $'ProtocolReserve = ProtocolReserve + Pr$

Invariant

- All invariant properties are preserved

Staking Provider contract, failure

Transition

DC.claim_rewards() [$C \in Pending^{Unhandled}_{ClaimRewards}$ is the pending request]

Updates

- State
    - $'Pending^{Unhandled}_{ClaimRewards} = Pending^{Unhandled}_{ClaimRewards} - C$
    - $'Pending^{Failure}_{ClaimRewards} = Pending^{Failure}_{ClaimRewards} + C$

Invariant

- All invariant properties are preserved

Callback failure

Transition

claim_rewards_from_cb(U, DC) [$C \in Pending^{Failure}_{ClaimRewards}$ is the pending request]

Updates

- State
    - $'Pending^{Failure}_{ClaimRewards} = Pending^{Failure}_{ClaimRewards} - C$

Invariant

- All invariant properties are preserved

Operation: delegate_rewards

Allows the admin to delegate part of the EGLD rewards to a staking provider based on the current configuration of the delegation algorithm. If the delegation is successful, the callback updates the storage. Otherwise, it sets the Delegation smart contract data as outdated. Notice that the smart contract data will be outdated until it is updated.

UserCall

Notations

- U := the caller of the delegate_rewards endpoint
- DC := $MigrationWhitelist[U]$ *or else use contract selection*
    - Contract selection adds the following preconditions:
        - $DC \in DcData - Outdated$
        - $DelegationAmount \leq DcData[DC].cap - DcData[DC].TotalValueLocked$
- DR := $DelegateRewards(U, DC, E)$

Transition

delegate_rewards(E)

Pre-conditions

- $U = Admin \vee U = RewardsManager$
- $State = Active$
- $E \leq RewardsReserve$
- $E \geq MinEgld$

Updates

- State
    - $'Rewards_{LS} = Rewards_{LS} - E$
    - $'EGLD_{S}^{DR} = E$
    - $'Pending_{DelegateRewards}^{Unhandled} = Pending_{DelegateRewards}^{Unhandled} + DR$

Invariant

- 14 - not preserved for concurrent calls
- 29 - not preserved for concurrent calls
- All other invariant properties are preserved

Staking Provider contract, success

Notations

- E := $EGLD_{S}^{DR}$

Transition

delegate() [E = EGLD payment, $DR \in Pending^{Unhandled}_{DelegateRewards}$ is the pending request]

Pre-conditions

- $E \le DC.cap - DC.TotalValueLocked$

Updates

- State
    - $'EGLD^{DR}_{S} = 0$
    - $'Delegated_{DC} = Delegated_{DC} + E$
    - $'DC.TotalValueLocked = DC.TotalValueLocked + E$
    - $'Pending^{Unhandled}_{DelegateRewards} = Pending^{Unhandled}_{DelegateRewards} - DR$
    - $'Pending^{Success}_{DelegateRewards} = Pending^{Success}_{DelegateRewards} + DR$

Invariant

- 16 - not preserved for concurrent calls.
- 18, 21 - preserved because it is impossible to have a delegate_rewards call starting while blacklisting a contract, and pending long enough that the contract is removed before the callback.
- All other invariant properties are preserved

Callback, success

Transition

delegate_rewards_cb(U, DC, E) [$DR \in Pending^{Success}_{DelegateRewards}$ is the pending request]

Updates

- State
    - $'Pending^{Success}_{DelegateRewards} = Pending^{Success}_{DelegateRewards} - DR$
- Storage
    - $'DcData(DC).TotalDelegated = DcData(DC).TotalDelegated + E$
    - $'RewardsReserve = RewardsReserve - E$
    - $'CashReserve = CashReserve + E$

Invariant

- 17 may not be preserved

- 21 - preserved because it is impossible to have a delegate_rewards call starting while blacklisting a contract, and pending long enough that the contract is removed before the callback.
- All invariant properties are preserved

Staking Provider contract, failure

Transition

delegate() [E = EGLD payment, $DR \in Pending^{Unhandled}_{DelegateRewards}$ is the pending request]

Updates

- State
  - $'EGLD^{DR}_{S} = EGLD^{DR}_{S} - E$
  - $'EGLD^{cb(DR)}_{S} = EGLD^{cb(DR)}_{S} + E$
  - $'Pending^{Unhandled}_{DelegateRewards} = Pending^{Unhandled}_{DelegateRewards} - DR$
  - $'Pending^{Failure}_{DelegateRewards} = Pending^{Failure}_{DelegateRewards} + DR$

Invariant

- All invariant properties are preserved

Callback failure

Transition

delegate_rewards_cb(U, DC, E) [E = EGLD payment, $DR \in Pending^{Failure}_{DelegateRewards}$]

Updates

- State
  - $'Rewards_{LS} = Rewards_{LS} + E$
  - $'EGLD^{cb(DR)}_{S} = 0$
  - $'Pending^{Failure}_{DelegateRewards} = Pending^{Failure}_{DelegateRewards} - DR$
- Storage
  - $'Outdated = Outdated + DC$

Invariant

- All invariant properties are preserved

Operation: whitelist_delegation_contract

Whitelists a Staking Provider Delegation smart contract. From this point onwards, this smart contract will be eligible as a Delegation smart contract based on the state of the delegation algorithm. If possible, the contract is also removed from the blacklist.

UserCall

Notations

- U := the caller of the whitelist_delegation_contract endpoint

Transition

whitelist_delegation_contract(DC, TVL, MaybeCap, A, other_args)

Pre-conditions

- $U = Admin$
- $DC \notin DcData$
- $MaybeCap \leq TVL$ (if $MaybeCap$ has a value)

Updates

- Storage
    - $'Blacklisted = Blacklisted - DC$
    - $'DcData(DC) = DcData(DC) + \{TotalValueLocked = TVL,\ Cap = cap,\ Admin = A, Total... = 0\}$
    - $'SortedContracts = SortedContracts + DC$

Invariant

- All invariant properties are preserved

Operation: change_delegation_contract_params

Updates the data for a given Staking Provider Delegation smart contract. Also recomputes the contract's delegation score.

UserCall

Notations

- U := the caller of the change_delegation_contract_params endpoint

Transition

change_delegation_contract_params(DC, TVL, MaybeCap, other_args)

Pre-conditions

- $DC \in DcData$
- $MaybeCap \leq TVL$ (if $MaybeCap$ has a value)
- $DC \notin Blacklisted$
- $U = DcData(DC).Admin$

Updates

- Storage
  - $'DcData(DC) = DcData(DC)\{TotalValueLocked = TVL,\ Cap = cap,\ Admin = A, ...\}$
  - $'Outdated = Outdated - DC$
  - $'SortedContracts = (SortedContracts - DC) + DC$ (reinsert with the new score)

Invariant

- All invariant properties are preserved

Operation: set_state_active

Activates the Liquid Staking Module state. The activation can only occur iff:

- the unbond period must have been set
- the minimum amount of EGLD to delegate has been set
- the total fee has been set
- the Liquid Staking token has been issued
- the undelegate NFT has been issued
- the delegation score model has been defined

UserCall

Notations

- U := the caller of the set_state_active endpoint

Transition

set_state_active()

Pre-conditions

- $U = Admin$
- $UnbondPeriod > 0$
- $MinEgld > 0$
- (shares and NFT tokens are set up)

- (Scoring params are set)

Updates

- Storage
  - $'State = Active$

Invariant

- All invariant properties are preserved

Operation: set_state_inactive

Deactivates the Liquid Staking Module state.

UserCall

Notations

- U := the caller of the set_state_inactive endpoint

Transition

set_state_inactive()

Pre-conditions

- $U = Admin$

Updates

- Storage
  - $'State = Inactive$

Invariant

- All invariant properties are preserved

Operation: withdraw_reserve

Withdraws a given amount of EGLD from the protocol reserves to an optionally given account.

UserCall

Notations

- U := the caller of the withdraw_reserve endpoint

Transition

withdraw_reserve(E, Dest)

Pre-conditions

- $U = Admin$
- $E \leq ProtocolReserve$

Updates

- State
  - $'EGLD_{Dest} = EGLD_{Dest} + E$
- Storage
  - $'ProtocolReserve = ProtocolReserve - E$

Invariant

- All invariant properties are preserved

Operation: withdraw

Withdraws either EGLD or any ESDT token amount from the protocol to a recipient.

UserCall

Notations

- U := the caller of the withdraw_reserve endpoint

Transition

withdraw(Dest, Token, Value)

Pre-conditions

- $U = Admin$
- $Value > 0$
- The contract has $Value$ of $Token$

Updates

- The call does not update the storage.
- The call updates the state in a way that's complicated to describe.
- Sends $Value$ of $Token$ to $Dest$

Invariant

- 29 does not hold

- Other invariant properties may not hold.

Operation: init

UserCall

Notations

- U := the caller of the withdraw_reserve endpoint

Transition

init(MaybeAdmin)

Updates

- Storage
    - $'Admin = MaybeAdmin\ orElse\ U$ (if not already set)
    - $'State = Inactive$
    - $'MinEgld = 1\ EGLD$ (if not already set)
    - Other values are being set.

Invariant

- All invariant properties are preserved

Operation: set_unbond_period

Sets the length of the cooling period for undelegate/unbond.

Operation:set_min_egld_to_delegate

Sets the minimum EGLD amount for delegations and undelegations. This should be greater or equal to the dust prevention limit in the Staking Provider contracts.

Operation: register_ls_token

Issues the liquid staking token, namely the sEGLD token.

Operation: set_ls_token_roles

Gives Mint and Burn roles for sEGLD to this contract.

Operation: register_undelegate_token

Issues the Undelegate Nft, the token minted at undelegations as a receipt.

Operation: set_undelegate_token_roles

Gives Mint and Burn roles for the Undelegate Nft to this contract.

Operation: set_rewards_manager

Sets the Rewards Manager of the protocol. Besides the admin, the Rewards Manager is allowed to delegate rewards.

Operation: change_delegation_contract_admin

Updates the admin for a given Staking Provider Delegation smart contract, which is entitled to update its data.

Operation: set_total_fee

Sets the total fee, which represents the final fee end users see discounted from their rewards based on the service fee charged by each Staking Provider and by this Liquid Staking protocol. For example, if a Staking Provider has a service fee of 7% and the total fee is set to 17%, the Liquid Staking Protocol will charge a 10% fee from the total rewards.

Operation: set_delegation_score_model_params

Sets the Delegation Score Model parameters used for the computation of the delegation score for each Staking Provider Delegation smart contract. Higher scores imply better chances of being selected at delegations as well as lower chances of being selected for undelegations.

This also recomputes the delegation score for all contracts.

Operation: set_delegation_sampling_model_params

Sets the Delegation Sampling Model parameters used for the random selection between candidates on a computed list of Staking Providers Delegation smart contracts.

Operation: clear_delegation_sampling_model

Clears the Delegation Sampling Model, i.e. removes the sampling from delegation and undelegation candidates.

Operation: add_trusted_contract

Whitelists a trusted contract, i.e. a contract that allows executing investment strategies involving sEGLD.

Operation: remove_trusted_contract

Removes a smart contract address from the whitelist of trusted contracts.

Operation: set_random_oracle

Sets the Random Oracle.

Rounding Errors

In the following, a bar over a variable means the value of that variable as computed by the contract. When used without bars, the variables are assumed to have their mathematically precise value. Rounding errors will be denoted by either greek letters. The floor(X) value will be denoted by $[X]$.

Let us take an example:
Above we defined FX to be $\frac{CashReserve+RewardsReserve}{Shares}$ (if $Shares > 1$, which will be assumed in the following). Let us denote $WAD \cdot FX$ by WFX.

get_exchange_rate() computes an approximation of WFX, which will be denoted by $\overline{WFX}$. The rounding error is $\epsilon_{WFX} = WFX - \overline{WFX}$.

get_exchange_rate

We have $WFX = WAD \cdot FX = \frac{WAD\cdot(CashReserve+RewardsReserve)}{Shares}$.

Then $\overline{WFX} = \left[\frac{WAD\cdot(CashReserve+RewardsReserve)}{Shares}\right] = [WFX]$.

The rounding error is $\epsilon_{WFX} = WFX - \overline{WFX} = WFX - [WFX]$, which means that $\epsilon_{WFX} \in [0, 1)$

egld_to_shares

Given an amount of EGLD E this function computes $\overline{S_E} = \left[\frac{E\cdot WAD}{\overline{WFX}}\right]$. Let $\delta_{S_E} = \frac{E\cdot WAD}{\overline{WFX}} - \left[\frac{E\cdot WAD}{\overline{WFX}}\right]$.
We know that $\delta_{S_E} \in [0, 1)$.

$$\overline{S_E} = \left[\frac{E \cdot WAD}{WFX}\right] = \frac{E \cdot WAD}{WFX} - \delta_{S_E} = \frac{E \cdot WAD}{WFX - \epsilon_{WFX}} - \delta_{S_E} = \frac{E \cdot WAD}{WFX} \cdot \frac{WFX}{WFX - \epsilon_{WFX}} - \delta_{S_E}$$

Let us denote by S_E the value

$$S_E = \frac{E \cdot WAD}{WFX} = \frac{E \cdot WAD}{\frac{WAD \cdot (CashReserve + RewardsReserve)}{Shares}} = \frac{E \cdot Shares}{(CashReserve + RewardsReserve)}$$

Then we have

$$\overline{S_E} = S_E \cdot \frac{WFX}{WFX - \epsilon_{WFX}} - \delta_{S_E} = S_E \cdot \left(1 + \frac{\epsilon_{WFX}}{WFX - \epsilon_{WFX}}\right) - \delta_{S_E} = S_E + S_E \cdot \frac{\epsilon_{WFX}}{WFX - \epsilon_{WFX}} - \delta_{S_E}$$

The total rounding error is

$$\epsilon_{S_E} = S_E - \overline{S_E} = S_E - \left(S_E + S_E \cdot \frac{\epsilon_{WFX}}{WFX - \epsilon_{WFX}} - \delta_{S_E}\right) = \delta_{S_E} - S_E \cdot \frac{\epsilon_{WFX}}{WFX - \epsilon_{WFX}}$$

We get $sup(\epsilon_{S_E})$ when δ_{S_E} approaches 1 and ϵ_{WFX} approaches 0, so we have

$$sup(\epsilon_{S_E}) = 1 - S_E \cdot \frac{0}{WFX - 0} = 1.$$

We get $inf(\epsilon_{S_E})$ when δ_{S_E} approaches 0 and ϵ_{WFX} approaches 1, so we have

$$inf(\epsilon_{S_E}) = 0 - S_E \cdot \frac{1}{WFX - 1} = - \frac{S_E}{WFX - 1}.$$

ϵ_{S_E} cannot reach its infimum and supremum values, so we have $\epsilon_{S_E} \in (-\frac{S_E}{WFX - 1}, 1)$.

Since $WFX \gg 1$ we can approximate $WFX - 1$ by WFX, so we have

$$\frac{S_E}{WFX - 1} \approx \frac{S_E}{WFX} = \frac{S_E}{WAD \cdot FX}.$$

When the Liquid Staking contract starts running, we have $FX = 1$. In principle, FX can only increase (the various rounding errors may decrease it slightly, but that should not be significant), but, most likely, $FX < 2$ for at least a few years. This means that, the maximum value for $\frac{S_E}{WAD \cdot FX}$ is $\frac{S_E}{WAD}$, which is reached at the start of the Liquid Staking contract, and the actual value will be relatively close to the maximum for a few years.

Then we can approximate the rounding error interval for ϵ_{S_E} with $(-\frac{S_E}{WAD}, 1)$.

Note that this is larger than it could be. If the code computes $\overline{S_E} = \left[\frac{E \cdot Shares}{(CashReserve+RewardsReserve)}\right]$ then the rounding error will be in the $[0,\ 1)$ interval.

Next Fx

We will estimate the error for the FX rate after exchanging EGLD for shares (i.e. after delegating some EGLD).

$$WFX' = WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+S_E} = WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+\frac{E \cdot Shares}{(CashReserve+RewardsReserve)}}$$

$$= WAD \cdot \frac{CashReserve+RewardsReserve+E}{\frac{Shares \cdot (CashReserve+RewardsReserve)+E \cdot Shares}{(CashReserve+RewardsReserve)}}$$

$$= WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares \cdot (CashReserve+RewardsReserve)+E \cdot Shares} \cdot (CashReserve + RewardsReserve)$$

$$= WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares \cdot (CashReserve+RewardsReserve+E)} \cdot (CashReserve + RewardsReserve)$$

$$= WAD \cdot \frac{CashReserve+RewardsReserve}{Shares}$$

$$= WFX$$

$$\overline{WFX'} = \left[WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+\overline{S_E}}\right]$$

Let $\delta_{\overline{WFX'}} = WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+\overline{S_E}} - \left[WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+\overline{S_E}}\right]$. As before, $\delta_{\overline{WFX'}} \in [0,\ 1)$.

$$\overline{WFX'} = WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+\overline{S_E}} - \delta_{\overline{WFX'}}$$

$$= WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+S_E-\epsilon_{S_E}} - \delta_{\overline{WFX'}}$$

$$= WAD \cdot \frac{CashReserve+RewardsReserve+E}{Shares+S_E} \cdot \frac{Shares+S_E}{Shares+S_E-\epsilon_{S_E}} - \delta_{\overline{WFX'}}$$

$$= WFX' \cdot \frac{Shares+S_E}{Shares+S_E-\epsilon_{S_E}} - \delta_{\overline{WFX'}}$$

$$= WFX' \cdot \left(1 + \frac{\epsilon_{S_E}}{Shares+S_E-\epsilon_{S_E}}\right) - \delta_{\overline{WFX'}}$$

$$= WFX' + \frac{WFX \cdot \epsilon_{S_E}}{Shares + S_E - \epsilon_{S_E}} - \delta_{\overline{WFX'}}$$

The error for WFX' is

$$\epsilon_{WFX'} = WFX' - \overline{WFX'} = WFX' - \left(WFX' + \frac{WFX \cdot \epsilon_{S_E}}{Shares + S_E - \epsilon_{S_E}} - \delta_{\overline{WFX'}}\right) = \delta_{\overline{WFX'}} - \frac{WFX \cdot \epsilon_{S_E}}{Shares + S_E - \epsilon_{S_E}}$$

We get $sup(\epsilon_{WFX'})$ when

- $\delta_{\overline{WFX'}}$ approaches its upper limit
- ϵ_{S_E} approaches its lower limit (we assume that $\epsilon_{S_E} < Shares + S_E$)

This means that

$$sup(\epsilon_{WFX'}) = sup(\delta_{\overline{WFX'}}) - \frac{WFX \cdot inf(\epsilon_{S_E})}{Shares + S_E - inf(\epsilon_{S_E})} = 1 - \frac{WFX \cdot \left(-\frac{S_E}{WFX-1}\right)}{Shares + S_E - \left(-\frac{S_E}{WFX-1}\right)} = 1 + \frac{\frac{WFX \cdot S_E}{WFX-1}}{\frac{(Shares+S_E)\cdot(WFX-1)+S_E}{WFX-1}}$$

$$= 1 + \frac{WFX \cdot S_E}{(Shares+S_E)\cdot(WFX-1)+S_E} = 1 + \frac{WFX \cdot S_E}{Shares\cdot(WFX-1)+WFX\cdot S_E}$$

As above, we can approximate $WFX - 1$ by WFX, so we have

$$sup(\epsilon_{WFX'}) \approx 1 + \frac{WFX \cdot S_E}{Shares \cdot WFX + WFX \cdot S_E} = 1 + \frac{S_E}{Shares + S_E}$$

We get $inf(\epsilon_{WFX'})$ when

- $\delta_{\overline{WFX'}}$ is minimum
- ϵ_{S_E} approaches its upper limit (we assume that $\epsilon_{S_E} < Shares + S_E$)

This means that

$$inf(\epsilon_{WFX'}) = inf(\delta_{\overline{WFX'}}) - \frac{WFX \cdot sup(\epsilon_{S_E})}{Shares + S_E - sup(\epsilon_{S_E})} = 0 - \frac{WFX \cdot 1}{Shares + S_E - 1} =- \frac{WFX}{Shares + S_E - 1}$$

As above, we can approximate:

$$inf(\epsilon_{WFX'}) \approx - \frac{WFX}{Shares + S_E}$$

In the end, we get

$\epsilon_{WFX'} \in \left(-\frac{WFX}{Shares+S_E-1}, 1+\frac{WFX\cdot S_E}{Shares\cdot(WFX-1)+WFX\cdot S_E}\right)$

And this interval can be approximated by $\left[-\frac{WFX}{Shares+S_E}, 1+\frac{S_E}{Shares+S_E}\right]$

shares_to_egld

Given an amount S of sEGLD, this function computes $\overline{E_S} = \left\lfloor\frac{\overline{WFX}\cdot S}{WAD}\right\rfloor$. Let $\delta_{E_S} = \frac{\overline{WFX}\cdot S}{WAD} - \left\lfloor\frac{\overline{WFX}\cdot S}{WAD}\right\rfloor$.

We know that $\delta_{E_S} \in [0, 1)$.

$$\overline{E_S} = \frac{\overline{WFX}\cdot S}{WAD} - \delta_{E_S} = \frac{(WFX-\epsilon_{WFX})\cdot S}{WAD} - \delta_{E_S} = \frac{WFX\cdot S-\epsilon_{WFX}\cdot S}{WAD} - \delta_{E_S} = \frac{WFX\cdot S}{WAD} - \frac{\epsilon_{WFX}\cdot S}{WAD} - \delta_{E_S}$$

Let E_s be $\frac{WFX\cdot S}{WAD} = \frac{S\cdot(CashReserve+RewardsReserve)}{Shares}$. Then we have $\overline{E_S} = E_s - \frac{\epsilon_{WFX}\cdot S}{WAD} - \delta_{E_S}$. The total rounding error is $\epsilon_{E_S} = \frac{\epsilon_{WFX}\cdot S}{WAD} + \delta_{E_S}$.

We have:

$$min(\epsilon_{E_S}) = \frac{min(\epsilon_{WFX})\cdot S}{WAD} + min(\delta_{E_S}) = \frac{0\cdot S}{WAD} + 0 = 0$$

$$sup(\epsilon_{E_S}) = \frac{sup(\epsilon_{WFX})\cdot S}{WAD} + sup(\delta_{E_S}) = \frac{1\cdot S}{WAD} + 1 = \frac{S}{WAD} + 1.$$

Then $\epsilon_{E_S} \in [0, \frac{S}{WAD}+1)$. This error is very similar in size to the one for egld_to_shares, and can be lowered in a similar way.

Next Fx

We will estimate the error for the FX rate after exchanging shares for EGLD for shares (i.e. after undelegating some shares).

$$WFX'' = WAD\cdot\frac{CashReserve+RewardsReserve+E_S}{Shares+S} = WAD\cdot\frac{CashReserve+RewardsReserve+\frac{S\cdot(CashReserve+RewardsReserve)}{Shares}}{Shares+S}$$

$$= WAD\cdot\frac{(CashReserve+RewardsReserve)\cdot Shares+S\cdot(CashReserve+RewardsReserve)}{(Shares+S)\cdot Shares}$$

$$= WAD\cdot\frac{(CashReserve+RewardsReserve)\cdot(Shares+S)}{(Shares+S)\cdot Shares}$$

$$= WAD\cdot\frac{(CashReserve+RewardsReserve)}{Shares}$$

$$= WFX$$

$$\overline{WFX''} = \left[WAD \cdot \frac{CashReserve + RewardsReserve + \overline{E_S}}{Shares + S} \right].$$

Let $\delta_{\overline{WFX''}} = WAD \cdot \frac{CashReserve + RewardsReserve + \overline{E_S}}{Shares + S} - \left[WAD \cdot \frac{CashReserve + RewardsReserve + \overline{E_S}}{Shares + S} \right]$. As before, $\delta_{\overline{WFX''}} \in [0,\ 1)$.

$$\overline{WFX''} = WAD \cdot \frac{CashReserve + RewardsReserve + \overline{E_S}}{Shares + S} - \delta_{\overline{WFX''}}$$

$$= WAD \cdot \frac{CashReserve + RewardsReserve + E_S - \epsilon_{E_S}}{Shares + S} - \delta_{\overline{WFX''}}$$

$$= WAD \cdot \frac{CashReserve + RewardsReserve + E_S}{Shares + S} \cdot \frac{CashReserve + RewardsReserve + E_S - \epsilon_{E_S}}{CashReserve + RewardsReserve + E_S} - \delta_{\overline{WFX''}}$$

$$= WFX'' \cdot \frac{CashReserve + RewardsReserve + E_S - \epsilon_{E_S}}{CashReserve + RewardsReserve + E_S} - \delta_{\overline{WFX''}}$$

$$= WFX'' \cdot \left(1 - \frac{\epsilon_{E_S}}{CashReserve + RewardsReserve + E_S}\right) - \delta_{\overline{WFX''}}$$

$$= WFX'' - \frac{WFX'' \cdot \epsilon_{E_S}}{CashReserve + RewardsReserve + E_S} - \delta_{\overline{WFX''}}$$

The error for WFX'' is

$$\epsilon_{WFX''} = WFX'' - \overline{WFX''} = WFX'' - \left(WFX'' - \frac{WFX'' \cdot \epsilon_{E_S}}{CashReserve + RewardsReserve + E_S} - \delta_{\overline{WFX''}}\right)$$

$$= \frac{WFX'' \cdot \epsilon_{E_S}}{CashReserve + RewardsReserve + E_S} + \delta_{\overline{WFX''}}$$

We get $sup(\epsilon_{WFX'})$ when

- $\delta_{\overline{WFX''}}$ approaches its upper limit
- ϵ_{E_S} approaches its upper limit

$$sup(\epsilon_{WFX''}) = \frac{WFX'' \cdot sup(\epsilon_{E_S})}{CashReserve + RewardsReserve + E_S} + sup(\delta_{\overline{WFX''}}) = \frac{WFX'' \cdot \left(\frac{S}{WAD} + 1\right)}{CashReserve + RewardsReserve + E_S} + 1$$

$$= \frac{WAD \cdot \frac{CashReserve+RewardsReserve+E_S}{Shares+S} \cdot \left(\frac{S}{WAD}+1\right)}{CashReserve+RewardsReserve+E_S} + 1$$

$$= \frac{WAD \cdot \left(\frac{S}{WAD}+1\right)}{Shares+S} + 1 = \frac{S+WAD}{Shares+S} + 1$$

We get $min(\epsilon_{WFX''})$ when

- $\delta_{\overline{WFX''}}$ is minimum
- ϵ_{E_S} is minimum

$$min(\epsilon_{WFX'}) = \frac{WFX'' \cdot min(\epsilon_{E_S})}{CashReserve+RewardsReserve+E_S} + min(\delta_{\overline{WFX''}}) = \frac{WFX'' \cdot 0}{CashReserve+RewardsReserve+E_S} + 0 = 0$$

This means that

$$\epsilon_{WFX'} \in [0, \frac{S+WAD}{Shares+S} + 1)$$

Egld_to_shares+shares_to_egld

Let's look at the rounding error when first transforming some egld (E) to shares, then the shares back to egld, as it would happen when the user delegates some egld, then immediately undelegates it. Let WFX' and WFX'' be the exchange rates after the two exchanges.

From the above, we have:

$$\epsilon_{WFX'} \in (-\frac{WFX}{Shares+S_E-1}, 1 + \frac{WFX \cdot S_E}{Shares \cdot (WFX-1)+WFX \cdot S_E})$$

$$\epsilon_{WFX''} \in [0, \frac{\overline{S_E}+WAD}{\left(Shares+\overline{S_E}\right)+\overline{S_E}} + 1) = [0, \frac{\overline{S_E}+WAD}{Shares+2 \cdot \overline{S_E}} + 1)$$

$$\epsilon_{S_E} \in (-\frac{S_E}{WFX-1}, 1)$$

$$\epsilon_{E_{\overline{S_E}}} \in [0, \frac{\overline{S_E}}{WAD} + 1)$$

We have:

$$\overline{E_{\overline{S_E}}} = E_{\overline{S_E}} - \epsilon_{E_{\overline{S_E}}} = \frac{\overline{WFX'} \cdot \overline{S_E}}{WAD} - \epsilon_{E_{\overline{S_E}}} = \frac{\left(WFX' - \epsilon_{WFX'}\right) \cdot \left(S_E - \epsilon_{S_E}\right)}{WAD} - \epsilon_{E_{\overline{S_E}}}$$

$$= \frac{WFX'\cdot S_E - WFX'\cdot\epsilon_{S_E} - \epsilon_{WFX'}\cdot S_E + \epsilon_{WFX'}\cdot\epsilon_{S_E}}{WAD} - \epsilon_{E_{\overline{S_E}}}$$

$$= \frac{WFX'\cdot S_E}{WAD} - \frac{WFX'\cdot\epsilon_{S_E} + \epsilon_{WFX'}\cdot S_E - \epsilon_{WFX'}\cdot\epsilon_{S_E}}{WAD} - \epsilon_{E_{\overline{S_E}}}$$

$$= \frac{WFX'\cdot S_E}{WAD} - \frac{WFX'\cdot\epsilon_{S_E} + \epsilon_{WFX'}\cdot S_E - \epsilon_{WFX'}\cdot\epsilon_{S_E}}{WAD} - \epsilon_{E_{\overline{S_E}}}$$

$$= E - \frac{WFX\cdot\epsilon_{S_E} + \epsilon_{WFX'}\cdot S_E - \epsilon_{WFX'}\cdot\epsilon_{S_E}}{WAD} - \epsilon_{E_{\overline{S_E}}}$$

Then we have

$$\epsilon_{\overline{E_{\overline{S_E}}}} = E - \overline{E_{\overline{S_E}}} = \frac{WFX\cdot\epsilon_{S_E} + \epsilon_{WFX'}\cdot S_E - \epsilon_{WFX'}\cdot\epsilon_{S_E}}{WAD} + \epsilon_{E_{\overline{S_E}}}$$

We get $sup(\epsilon_{\overline{E_{\overline{S_E}}}})$ when:

- $\epsilon_{E_{\overline{S_E}}}$ approaches its upper limit
- $\epsilon_{WFX'}$ approaches its upper limit (assuming $S_E > \epsilon_{S_E}$)
- ϵ_{S_E} approaches its upper limit (assuming $WFX > \epsilon_{WFX'}$)

We have

$$sup(\epsilon_{\overline{E_{\overline{S_E}}}}) = \frac{WFX\cdot sup(\epsilon_{S_E}) + sup(\epsilon_{WFX'})\cdot S_E - sup(\epsilon_{WFX'})\cdot sup(\epsilon_{S_E})}{WAD} + sup(\epsilon_{E_{\overline{S_E}}})$$

$$= \frac{WFX\cdot 1 + \left(1 + \frac{WFX\cdot S_E}{Shares\cdot(WFX-1) + WFX\cdot S_E}\right)\cdot\left(S_E - 1\right)}{WAD} + \frac{\overline{S_E}}{WAD} + 1$$

$$= \frac{WFX + S_E - 1 + \frac{WFX\cdot S_E}{Shares\cdot(WFX-1) + WFX\cdot S_E}\cdot\left(S_E - 1\right) + \overline{S_E}}{WAD} + 1$$

As above, we can approximate $WFX - 1$ with WFX and $S_E - 1$ with S_E. We get:

$$sup(\epsilon_{\overline{E_{\overline{S_E}}}}) \approx \frac{WFX + S_E + \frac{WFX\cdot S_E}{Shares\cdot WFX + WFX\cdot S_E}\cdot S_E + \overline{S_E}}{WAD} + 1$$

$$= \frac{WFX+S_E+\frac{S_E}{Shares+S_E}\cdot S_E+\overline{S_E}}{WAD} + 1$$

$$= \frac{WFX+S_E+\frac{S_E\cdot S_E}{Shares+S_E}+\overline{S_E}}{WAD} + 1$$

We can also approximate $\overline{S_E}$ with S_E, and we get:

$$sup(\epsilon_{E_{\overline{S_E}}}) \approx \frac{WFX+2\cdot S_E+\frac{S_E\cdot S_E}{Shares+S_E}}{WAD} + 1 = \frac{WAD\cdot FX+2\cdot S_E+\frac{S_E\cdot S_E}{Shares+S_E}}{WAD} + 1$$

$$= \frac{2\cdot S_E+\frac{S_E\cdot S_E}{Shares+S_E}}{WAD} + 1 + FX = \frac{S_E}{WAD} \cdot \left(2 + \frac{S_E}{Shares+S_E}\right) + 1 + FX$$

This means that

$$sup(\epsilon_{E_{\overline{S_E}}}) < \frac{3\cdot S_E}{WAD} + 1 + FX$$

We get $inf(\epsilon_{E_{\overline{S_E}}})$ when:

- $\epsilon_{E_{\overline{S_E}}}$ is minimum
- $\epsilon_{WFX'}$ approaches its lower limit (assuming $S_E > \epsilon_{S_E}$)
- ϵ_{S_E} approaches its lower limit (assuming $WFX > \epsilon_{WFX'}$)

$$inf(\epsilon_{E_{\overline{S_E}}}) = \frac{WFX\cdot inf(\epsilon_{S_E})+inf(\epsilon_{WFX'})\cdot S_E-inf(\epsilon_{WFX'})\cdot inf(\epsilon_{S_E})}{WAD} + min(\epsilon_{E_{\overline{S_E}}})$$

$$= \frac{WFX\cdot\left(-\frac{S_E}{WFX-1}\right)+\left(-\frac{WFX}{Shares+S_E-1}\right)\cdot S_E-\left(-\frac{WFX}{Shares+S_E-1}\right)\cdot\left(-\frac{S_E}{WFX-1}\right)}{WAD} + 0$$

$$= \frac{-WFX\cdot S_E\cdot\left(\frac{1}{WFX-1}+\frac{1}{Shares+S_E-1}+\frac{1}{(WFX-1)\cdot(Shares+S_E-1)}\right)}{WAD}$$

$$= \frac{-WFX\cdot S_E\cdot\frac{Shares+S_E-1+WFX-1+1}{(WFX-1)\cdot(Shares+S_E-1)}}{WAD}$$

$$=-\frac{WFX}{WFX-1} \cdot \frac{S_E}{WAD} \cdot \frac{Shares+S_E+WFX-1}{Shares+S_E-1}$$

Using the same approximations as before, we get

$$inf(\epsilon_{\frac{E}{S_E}}) \approx -1 \cdot \frac{S_E}{WAD} \cdot \frac{Shares+S_E+WFX}{Shares+S_E}$$

$$= -\frac{S_E}{WAD} \cdot \frac{Shares+S_E+WFX}{Shares+S_E} = -\frac{S_E}{WAD} \cdot \left(1 + \frac{WFX}{Shares+S_E}\right)$$

In most cases, $WFX < Shares + S_E$, which means that in most cases

$$inf(\epsilon_{\frac{E}{S_E}}) > -\frac{2 \cdot S_E}{WAD}$$

Appendix B: Assumptions

The current contract makes various assumptions about administrator behaviour, e.g.:

- The administrator does not call delegate_to concurrently for the same penalty ID.
  - Not needed at Commit 2
- The administrator does not call unbond_from concurrently for the same penalty ID.
  - Not needed at Commit 2
- The administrator does not call delegate_rewards concurrently.
  - Not needed at Commit 2
- For one of the suggested solutions below, the admin must pause the contract and wait for all transactions to finish before blacklisting a contract.
  - Not needed at Commit 2
- The administrator will not blacklist contracts in the migration whitelist.
  - Not needed at Commit 2
- The administrator will use the withdraw endpoint properly.
  - The endpoint was removed at Commit 2
- The administrator will not set the random_oracle field (it is supposed to be used only for testing)
- The administrator will only add trusted contracts that are in the same shard as the Liquid Staking contract.
  - Trusted contracts were removed at Commit 2
- After creation, the administrator will set the total_fee value to a non-null value before activating the contract.
  - Not needed at Commit 2
- The admin pauses the contract before updating

Besides the above, there are other assumptions that the Liquid Staking contract makes, e.g.:

- Delegate_to assumes that there is a Staking Provider contract that can accept tokens. If that's not the case, the tokens remain uninvested. Also, in some cases, users will not be able to withdraw (all) their tokens.

- Staking Provider contracts will not lose user funds.
  - Their documentation promises this, at least until slashing is implemented.
- Callbacks from calls to the same contract run in the same order as the calls
  - The Hatom team reports that this was not always true in their tests, but that it should be fixed soon by MultiversX

Appendix C: Issue Severity/Difficulty Classification

This section contains a snapshot of RVs issue classification system of the time at which this report was authored. An up-to-date version of this classification system can be found at: https://github.com/runtimeverification/security/blob/master/issue-classification.md

---

Issue Severity/Difficulty Classification

Our issues ranking system is based on two axes, *severity* and *difficulty*. Severity covers "how bad would it be if someone exploited this", and is ranked *Informative*, *Low*, *Medium*, and *High*. Difficulty is "how hard is it for someone to exploit this", and is ranked *Low*, *Medium*, and *High*.

This document is *guidance* for security ratings, and is constantly changing. The lead auditor reserves the right to change severity or difficulty ratings as needed for each situation.

In some cases, it makes sense to be more clear with the client about the *recommended action*. Recommended action can be used to make the nature of the vulnerability clear to the client, and can be *fix design*, *fix code*, or *document prominently*.

Other Ranking Systems

Immunefi ranking: https://immunefi.com/severity-updated/

Severity Ranking

Severity refers to how bad it is if this issue is exploited. This means that the *effects* of the exploit affect the severity, but *who can do the exploit* does not.

If a given attack seems to fit multiple criteria here, use the most severe classification.

High Severity

- Permanent deadlock of some or all protocol operations.
- Loss of any non-trivial amount of user or protocol funds.

- Core protocol properties do not hold.
- Arbitrary minting of tokens by untrusted users.
- DOS attacks making the system (or any vital part of the system) unusable.

Medium Severity

- Sensible or desirable properties over the protocol do not hold, but no known attack vectors due to this (“looks risky” feeling).
- Non-responsive or non-functional system is possible, but recovery of user funds can still be guaranteed.
- Temporary loss of user funds, guaranteed to be recoverable via external algorithmic mechanism like a treasury.
- Loss of small amounts of user funds (eg. bits of gas fees) that serve no protocol purpose.
- Griefing attacks which make the system less pleasant to interact with, potentially used to promote a competitor.
- System security relies on assumptions about externalities like “valid user input” or “working monitoring server”.
- Deployments are not verifiable, so that phishing attacks may be possible.

Low Severity

- Slow processing of user transactions can lead to changed parameters at transaction execution time.
- Function reverts on some inputs that it could safely handle.
- Users receive less funds than expected in pure mathematical model, but bounds on this error is very small.
- Users are not protected from obviously bad choices (eg. trading into an asset with zero value).
- System accumulates dust (eg. due to rounding errors) that is unrecoverable.

Informative Severity

- Not following best coding practices. Examples include:
    - Missing input validation or state sanity checks,
    - Code duplication,
    - Bad code architecture,
    - Unmatched interfaces or bad use of external interfaces,
    - Use of outdated or known problematic toolchains (eg. bad compiler version),
    - Domain specific code smells (eg. not recycling storage slots on EVM).
- Gas optimizations.
- Non-intuitive or overly complicated behaviors (which may lead to users and/or auditors mis-understanding the code).
- Lack of documentation, or incorrect/inconsistent documentation.
- Known undesired behaviors when the security model or assumptions do not hold.

Difficulty Ranking

Difficulty refers to how hard it is to actually accomplish the exploit. The things that increase difficulty are how *expensive* the attack is, *who can perform the attack*, and *how much control you need to accomplish the attack*. Note that when analyzing the expense difficulty of an attack, you *must* take into account flash loans.

If an attack fits multiple categories here, because of factors X which makes it severity S1 and Y which makes severity S2, then you need to decide:

- Are *both* X and Y necessary to make the attack happen, then use the higher difficulty.
- If *only* one of X and Y is necessary, then use the lower difficulty.

High Difficulty

- Only trusted authorized users can perform the attack (eg. core devs).
- Performing the attack costs significantly more than how much you benefit (eg. it costs 10x to do the attack vs what is actually won).
- Performing the attack requires coordinating multiple transactions across different blocks, and can be stopped if detected early enough.
- Performing the attack requires control of the network, to delay or censor given messages.
- Performing the attack requires convincing users to participate (eg. bribe the users).

Medium Difficulty

- Semi-authorized (or whitelisted) users can perform the attack (eg. “special” nodes, like validators, or staking operators).
- Performing the attack costs close to how much you benefit (eg. 0.5x - 2x).
- Performing the attack requires coordinating multiple transactions across different blocks, but cannot be stopped if detected early enough.

Low Difficulty

- Anyone who can create an account on the network can perform the attack.
- Performing the attack costs much less than how much you benefit (eg. < 0.5x).
- Performing the attack can happen within a single block or transaction (or transaction group).
- Performing the attack only requires access to a modest amount of capital and a flash-loan system.

Recommended Action

The recommended action classification can be useful to emphasize to clients the nature of a vulnerability. These are ordered from *most* expensive/time-consuming to *least* expensive/time-consuming for the client, so in a sense *fix design* is more severe than *fix code* which is more severe than *document prominently*.

Fix Design

A bug in the design means that even correct implementation will lead to undesirable behavior. This means that code development for that feature *should* be halted, until the design issues are addressed. There is also not much point in reviewing the code for that feature. Often times, this may result in us switching to “Design Consultation”, rather than continuing with “Design Review” or “Code Review”.

Fix Code

A bug in the code means that the implementation does not conform to the design. The design has already been reviewed and deemed safe and operational. A fix for the code should be applied to avoid the bug (maybe adding an extra safety check, or calculating some quantity differently). If working on the code reveals problems in the design, then this should be escalated to *fix design*.

Document Prominently

The protocol design and code work as intended, but may have unintuitive or unsafe behavior when paired with some other financial product (or integration). Then in the documentation (targeted at developers trying to integrate with the product), the client should include the assumptions that this protocol makes of the other (or vice-versa). This can be important for avoiding scenarios where each protocol on its own is safe, but there is a known way to combine them that is unsafe. If the integration problem can be avoided by adding some checks in the code, you should consider (with the client) escalating this to a *fix code*. If the integrations can be made safe altogether by reworking the system a bit, you should consider (with the client) escalating this to a *fix design*.

Examples

Inside Attack

A contract has special admin accounts controlled by the internal devs. They can take a sequence of actions which steal user funds. The client may say “we will not address this, because we trust ourselves and the community trusts us.” This should be classified as *high* severity (loss of user funds), and *high* difficulty (compromise of the core dev team).

Arithmetic Rounding Issue

A contract has some complicated arithmetic that has unbounded rounding errors. Any user can pick inputs to the contract to control how those rounding errors accumulate, which leads to locking up (effectively burning) assets from a protocol treasury. This should be classified as *high* severity (loss of protocol funds). The difficulty rating of this is subjective, because it may depend on current market dynamics and contract state. If at *any* time, the user can manipulate inputs by small amount to achieve arbitrarily large output changes, then it is *low* difficulty. If certain conditions (reasonably believed to be outside the attacker's control) need to be met, *then* they can exploit this with limited changes to inputs, then *medium* difficulty.

Oracle Price Attack

A contract has an oracle feed from a trusted oracle provider which uses DEX calculated prices as part of the input aggregation. A user discovers that manipulating a specific DEXs liquidity pools has an effect on the oracle price feed. The user cannot manipulate the price feed and perform the attack in the same block, because they do not have control over when oracle price updates come in. The user manipulates the price feed by interacting with the DEX liquidity pools, then waits for the price feed to adjust parameters for a lending platform that is relying on it. Once the lending platform has consumed the updated price, the user takes out a loan at more favorable rates from the lending platform. The original capital used to manipulate the liquidity pools is withdrawn. This should be classified as *high* severity (loss of protocol funds), and *high* difficulty (requires large funds locked into liquidity pools over the course of the attack, which spans multiple blocks).

Missing User Protections

A contract collects fees for interaction, but it over-collects fees because it doesn't know ahead how much it will need, then attempts to reimburse the user with the extra at the end. The reimbursement procedure calculates the reimbursement incorrectly, which leads to some funds being locked as dust and the user receiving too few fees afterwards. The amount of dust accumulated is small, and the error does not break any core properties of the protocol. This should be classified as *medium* severity (loss of user funds that serves no protocol purpose), and *low* difficulty (happens automatically for any user).

Weird Transient Dynamics due to Rounding

A protocol has a mechanism for users depositing funds and receiving a proportional amount of the protocol's token. The proportion is determined by how many funds have been deposited thus far, and contains a division. At protocol startup, the denominator in the division is zero, leading to weird rounding issues with the initial allotments of tokens. After startup, this denominator goes up, and the problem goes away. There is no obvious downside to this, other than the initial investors do not get the same exchange rate as expected. This should be classified as *low* severity (no known issues with accumulated rounding errors, but seems weird), and *low* difficulty (will happen no matter what).

Divergence of Ghost Calculation

A protocol has individual interest accumulation, which is calculated and maintained on each interaction the individual has with the protocol. Separately, a *global* interest is tracked as the total interest earned by users, but only calculated at specific events where the global interest is used. It's profitable for a user to interact often with the contract, so their interest is compounded more frequently. Meanwhile, the global interest calculation is not keeping up with the more frequent user-level compounding because it's not maintained as often. No obvious harm comes from this divergence of the global interest from the true interest, but the global interest is used in some other calculations. This is classified as *medium* severity (sensible property about the protocol does not hold, but no known issues), and *low* difficulty (likely to happen on its own).

Integration Attack

Suppose we are asked to audit Uniswap V3, and we know that they will be providing a price oracle based on the current status of their liquidity pools. If the Uniswap V3 pools in question have low liquidity, they are subject to manipulation by people with sufficient capital ("whale attack"). Perhaps this allows a user to manipulate the price enough to withdraw the majority of another protocol's collateral (some protocol that relies on the price feed from Uniswap V3). This is classified as *high* severity (permanent loss of protocol funds), and *low* difficulty (just requires a whale to manipulate Uniswap V3 liquidity pools, and the capital to manipulate the liquidity can be withdrawn). This should also be classified as a *document prominently*, so that the Uniswap team knows to make it very clear to users of their price feed that they must watch for this issue. It can be potentially escalated to a *fix design* if the Uniswap V3 team agrees that they will not provide price feeds for low-liquidity situations, or will provide amended price feeds.