WARBLER

Membership Vaults Audit 11/01/22

Auditors: Dalton, Will

Overview

The Membership Vaults system was audited for two and half weeks from 11/01/22 - 11/16/22.

Scope

The focus was on identifying bugs in Membership Vaults with large economic impact such as loss or theft of funds, miscalculation of rewards, etc. We looked at the following contracts:

- MembershipOrchestrator
- MembershipDirector
- MembershipVault
- MembershipCollector
- MembershipLedger
- GFILedger
- CapitalLedger
- CapitalAssets
- StakedFiduAsset
- PoolTokenAsset
- UserEpochTotals
- Epochs

as well as dependent contracts like AccessControl, Router, and Context.

We used automated and manual processes to audit the system. Automated processes included fuzz tests with Foundry and Echidna. Manual processes included standard code-review with a focus on security, and applying our internal audit and pre-audit checklists to the contracts.

Summary of Findings

Issue Count

No critical vulnerabilities were found. There were a handful of medium severity issues and some low severity issues.

Legend

🛑 High - Loss or theft of funds 🟡 Medium - Logic bugs, griefing attacks, unexpected reverts on critical paths, etc. 🟢 Low / Informational - Gas optimizations, nits, requests for greater clarity, etc.

Contract	🛑	🟡	🟢	Total
All	0	8	13	21
CapitalLedger.sol	0	1	0	1
Epochs.sol	0	0	0	0
ERC20Splitter.sol	0	0	4	4
GFILedger.sol	0	2	1	3
MembershipCollector.sol	0	0	1	1
MembershipDirector.sol	0	0	1	1
MembershipLedger.sol	0	0	2	2
MembershipOrchestrator.sol	0	2	0	2
MembershipVault.sol	0	3	2	5
UserEpochTotals.sol	0	0	2	2

General Comments

View functions don't account for unfinalized epochs

Many view functions don't account for unfinalized epochs and this results in them returning stale values. Examples are

- *getPendingRewardsFor* in MembershipLedger doesn't include pending rewards for unfinalized epochs. This results in lower than expected pending rewards.
- `eligibleAmount` and `nextEpochAmount` returned by *positionOwnedBy* in MembershipVault do not account for the most recent ended epoch if it's unfinalized. This could lead to `elibibleAmount = 0` when it should be non-zero (see membership-vault.md for detailed analysis).

The aforementioned examples cause issues in upstream view functions:

- *currentScore* in MembershipDirector is stale because *getPendingRewardsFor* is stale
- *claimableRewards* in MembershipDirector is forced to do additional calculations for the rewards claimable in non-finalized epochs because it relies on *getPendingRewardsFor*.
- Other examples are *memberScoreOf* and *totalGFIHeldBy*, and *totalCapitalHeldBy* in MembershipOrchestrator.

It's our view from a correctness standpoint that these view fn's should return fully up to date values even if their underlying storage isn't up to date. It's a large burden on callers to take into account eneded but unfinalized epochs and add their own logic to get the correct values. It's either that or they have to execute a tx to force epoch finalization but that defeats the main benefit of using view fn's. in the first place.

This is not something that has to be fixed before launch but the Auditors recommend adhering to this pattern going forward.

Checklists

We have checklists to help the code owners and auditors catch the most common security mistakes. Some redundancy is baked in to the checklists to increase the chance of catching mistakes.

Pre-audit checklist

This checklist is used by the code owners before submitting the contracts for internal audit.

- [ ] Testing and compilation
  - [ ] Code has full branch coverage (can use `forge coverage` for this)
  - [ ] Mainnet forking tests
  - [ ] Contracts compile without warnings
  - [ ] Any `public` function that can be made `external` should be made `external`. This is not just a gas consideration, but it also reduces the cognitive overhead for auditors because it reduces the number of possible contexts in which the function can be called.
- [ ] Documentation
  - [ ] `external` and `public` functions are documented with NatSpec. This makes the auditors' job much easier!
- [ ] Access control
  - [ ] Double checked the permissions on `external` and `public` functions. E.g. "should this function be `onlyAdmin`?"
  - [ ] New roles
    - [ ] Intended behavior of roles are documented, e.g. "the BORROWER_ROLE can perform actions X, Y, Z and can be granted through a governance vote"
    - [ ] An event is emitted whenever the role is assigned or revoked
- [ ] Critical areas for the auditors to focus on are called out
- [ ] Third party integrations
  - [ ] I have assessed the impact of changes (breaking or non-breaking) to existing functions on 3rd party protocols that have integrated with Goldfinch
- [ ] Safe operations
  - [ ] Using SafeERC20Transfer for ERC20 transfers
  - [ ] Using SafeMath for arithmetic for contracts compiled with Solidity version < 8.0
  - [ ] Using SafeCast for casting
  - [ ] No iterating on unbounded arrays or passing them around as params
  - [ ] Arithmetic performs division steps at the end to minimize rounding errors
  - [ ] Not using the built-in *transfer* function
  - [ ] All user-inputted addresses are verified before instantiating them in a contract (eg. `CreditLine(randomAddressParam)`)
  - [ ] Changes follow the checks-effects-interactions pattern, or a reentrancy guard is used.

Audit checklist

This checklist is used by the auditors to help them catch common security missteps. The auditor's analysis included but was not limited to this checklist.

- [ ] Access control
  - [ ] I have double checked the permissions on `external` and `public` functions. E.g. "should this function be `onlyAdmin`?"
  - [ ] New roles

        - [ ] New roles are documented, e.g. “the BORROWER_ROLE can perform actions X, Y, Z and can be granted through a governance vote”
        - [ ] An event is emitted whenever the role is assigned or revoked
- [ ] Library dependencies
    - [ ] Checked release notes for bug fixes on vendored contracts (e.g. minor or patch updates on our vendored OpenZeppelin contracts)
- [ ] Proxies
    - [ ] Changes to upgradeable contracts do not cause storage collisions
- [ ] Safe operations
    - [ ] Using SafeERC20Transfer for ERC20 transfers
    - [ ] Using SafeMath for arithmetic for contracts compiled with Solidity version < 8.0
    - [ ] Using SafeCast for casting
    - [ ] No iterating on unbounded arrays or passing them around as params
    - [ ] Arithmetic performs division steps at the end to minimize rounding errors
    - [ ] Not using the built-in *transfer* function
    - [ ] All user-inputted addresses are verified before instantiating them in a contract (eg. `CreditLine(randomAddressParam)`)
    - [ ] Changes follow the checks-effects-interactions pattern, or a reentrancy guard is used.
- [ ] Are any speed bumps necessary? E.g. a delay between locking a TranchedPool and drawing down.