RENASCENCE

Dinero (Staked S) Audit Report

Version 2.0

Audited by:

MiloTruck

bytes032

December 21, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Dinero

Dinero is an experimental protocol which capitalizes on the premium blockspace market by introducing:

1. An ETH liquid staking token ("LST") which benefits from staking yield and the Dinero protocol
2. A decentralized stablecoin (DINERO) as a medium of exchange on Ethereum
3. A public and permissionless RPC for users

2.2 Overview

Project	Dinero (Staked S)
Repository	ss
Commit Hash	0f4bcb26727d...
Mitigation Hash	f8e0bf0e1096...
Date	17 December 2024 - 21 December 2024

2.3 Issues Found

Severity	**Count**
High Risk	2
Medium Risk	1
Low Risk	1
Informational	1
Total Issues	**5**

3 Findings Summary

ID	Description	Status
H-1	Share price is temporarily inflated on `redeem()`	Resolved
H-2	S token rewards received from `SFC.claimRewards()` are not handled	Resolved
M-1	Exceeding the validator's delegated stake limit could DOS the `SS` vault	Resolved
L-1	Enforcing a minimum number of shares per deposit is sub-optimal	Resolved
I-1	Minor code improvements	Resolved

4 Findings

High Risk

[H-1] Share price is temporarily inflated on `redeem()`

Context:

- SS.sol#L293
- SS.sol#L197-L200

Description: Whenever a user calls `redeem()` to redeem a certain amount of shares for assets, the shares are burned:

```
_burn(_owner, _shares);
```

However, `totalSStaked`, which is used as `totalAssets()` for vault, is not decreased:

```
function totalAssets() public view override returns (uint256) {
    SSStorage storage $ = _getSSStorage();
    return $.totalSStaked;
}
```

Since the total supply of the vault decreases but `totalAssets()` remains constant, the vault's share price will temporarily be inflated when `redeem()` is called. The share price only returns to normal when the corresponding amount of assets is subtracted from `totalSStaked` in `completeWithdrawal()`.

As such, if any user deposits while there are withdrawals pending, they will receive less shares than expected, causing a loss of funds.

For example:

- Assume the vault has the following state:
  - `totalSStaked = 100e18`
  - `totalSupply = 100e18`
- Alice calls `redeem()` to redeem `20e18` shares:
  - `assets = shares * totalSStaked / totalSupply = 20e18 * 100e18 / 100e18 = 20e18`
  - `totalSupply = 100e18 - 20e18 = 80e18`
- The vault's share price is now `totalSStaked / totalSupply = 100e18 / 80e18 = 1.25`, which is inflated from the correct share price of `1`.
- Bob calls `depositNative()` to deposit `10e18` assets:
  - `shares = assets * totalSupply / totalSStaked = 10e18 * 80e18 / 100e18 = 8e18`
  - `totalSStaked = 100e18 + 10e18 = 110e18`
  - `totalSupply = 80e18 + 8e18 = 88e18`

- After the withdrawal period, Alice `redeem()` to complete her withdrawal:
  - `totalSStaked = 110e18 - 20e18 = 90e18`
- Now, Bob's `8e18` shares are worth:
  - `assets = shares * totalSStaked / totalSupply = 8e18 * 90e18 / 88e18 = ~8.18e18`

As seen in the example above, Bob's initial deposit of `10e18` assets has decreased to `~8.18e18` assets, causing a loss of funds.

Recommendation: In `redeem()`, `totalSStaked` should be decreased by `assets` whenever shares are burned:

```diff
  // Then state changes
  $.withdrawalRequests[_receiver][wrID] = assets;
  $.pendingUndelegations += assets;
+ $.totalSStaked -= assets;
  _burn(_owner, _shares);
```

To accommodate this change, `completeWithdrawal()` should not subtract from `totalSStaked`:

```diff
  // Then state changes
  delete $.withdrawalRequests[msg.sender][_wrID];
  $.pendingUndelegations -= assets;
- $.totalSStaked -= assets;
```

Additionally, `getActualStake()` should simply return `$.totalSStaked`:

```diff
  function getActualStake() public view returns (uint256) {
      SSStorage storage $ = _getSSStorage();
-     return $.totalSStaked - $.pendingUndelegations;
+     return $.totalSStaked;
  }
```

Dinero: Fixed in commit 3b13a5b.

Renascence: Verified, the recommendation was implemented. Additionally, `getActualStake()` was removed and replaced with `totalAssets()` for the check in `redeem()`.

[H-2] S token rewards received from `SFC.claimRewards()` are not handled

Context:

- SS.sol#L99-L103

Description: In the `collectPendingRewards()` modifier, if the validator's status is not `OK_STATUS`, `SFC.claimRewards()` is called to directly transfer pending rewards to the contract:

```
if (status == OK_STATUS) {
    sfc.restakeRewards($.validatorId);
} else {
    sfc.claimRewards($.validatorId);
}
```

However, the claimed S tokens are not handled. The `SS` vault was not designed to be able to hold S tokens, since all of the vault's assets are delegated to the validator on deposit. Therefore, the S tokens received from `claimRewards()` cannot be withdrawn through `redeem()` and `completeWithdrawal()`.

As a result, if the validator's status is not `OK_STATUS`, any rewards claimed will be permanently stuck in the `SS` vault.

Recommendation: Add functionality to the `SS` vault to handle S tokens in the contract. One possible solution would be to add a function that allows withdrawals to be taken from the contract's S token balance, instead of undelegating from the validator.

Dinero: Fixed in commit 2e0aeca by adding emergency withdrawal method for the owner.

Renascence: If rewards received from `claimRewards()` are not meant to be distributed to depositors, `totalSStaked` should only be increased when `restakeRewards()` is called:

```
  if (pendingRewards > 0) {
      if (status == OK_STATUS) {
          // Check for delegation limit by trying to restake first
          try sfc.restakeRewards($.validatorId) {} catch {}
+         $.totalSStaked += pendingRewards;
      } else {
          sfc.claimRewards($.validatorId);
      }

-     $.totalSStaked += pendingRewards;

      emit RewardsCollected(pendingRewards);
  }
```

Otherwise, the vault's share price would increase even when `claimRewards()` is called, making the vault insolvent as the S token in the vault cannot be withdrawn by users.

Dinero: Fixed in commit f8e0bf0.

Renascence: Verified, an `emergencyWithdraw()` function was added to allow S token in the vault to be withdrawn by the owner. Once the validator's status is no longer `STATUS_OK`, the vault will no longer distribute rewards to depositors. Instead, the owner simply retrieves the remaining rewards.

Medium Risk

[M-1] Exceeding the validators delegated stake limit could DOS the `SS` vault

Context:

- SFC.sol#L637-L639
- SFC.sol#L616-L621
- SS.sol#L92-L101
- SS.sol#L255-L256

Description: In `SFC._delegate()`, the total amount delegated to a validator is restricted by `_checkDelegatedStakeLimit()`:

```
if (!_checkDelegatedStakeLimit(toValidatorID)) {
    revert ValidatorDelegationLimitExceeded();
}
```

```
/// Check whether the self-stake covers the required fraction of all delegations for
the given validator.
function _checkDelegatedStakeLimit(uint256 validatorID) internal view returns (bool) {
    return
        getValidator[validatorID].receivedStake <=
        (getSelfStake(validatorID) * c.maxDelegatedRatio()) / Decimal.unit();
}
```

This enforces that each validator can have only up to 15 times their self-staked amount delegated to it.

However, this limit could cause functions in the `SS` vault to revert, resulting in DOS.

Firstly, the `collectPendingRewards()` modifier attempts to restake the validator's rewards as long as its status is `STATUS_OK` and there are pending rewards:

```
(uint256 status, , , , , , ) = sfc.getValidator($.validatorId);
uint256 pendingRewards = sfc.pendingRewards(
    address(this),
    $.validatorId
);

if (pendingRewards > 0) {
    if (status == OK_STATUS) {
        sfc.restakeRewards($.validatorId);
    } else {
```

Since the delegated stake limit is not checked, if the amount delegated to the validator exceeds the limit after restaking, attempting to restake rewards through `SFC.restakeRewards()` will always revert. As a result, as the `collectPendingRewards()` modifier is called by all of the vault's functions, all functions will be DOSed.

Secondly, calling `SFC.delegate()` in `depositNative()` will also revert if the delegated stake limit is already exceeded:

```
// External call first (CEI pattern)
$.sfc.delegate{value: msg.value}($.validatorId);
```

Note that to reach the delegated stake limit, an attacker can delegate to the vault's `validatorId` by directly calling `SS.delegate()`, instead of depositing through the vault.

Recommendation: In `collectPendingRewards()`, `SFC.restakeRewarsd()` should only be called if restaking the pending rewards does not exceed the delegated stake limit for the validator.

Dinero: Fixed in commit 1e4413a by adding a try-catch clause to prevent unsuccessful restake due to reaching delegation limit.

Renascence: Consider adding the following checks on `gasleft()`, which prevents `restakeRewards()` from being skipped when it reverts due to running out of gas:

```
uint256 gasBefore = gasleft();
try sfc.restakeRewards($.validatorId) {} catch {
    uint256 gasAfter = gasleft();
    if (gasAfter * 64 <= gasBefore) revert InsufficientGas();
}
```

Dinero: Added in commit f8e0bf0.

Renascence: Verified, if the delegated stake limit is reached, `restakeRewards()` will be skipped and the `collectPendingRewards()` modifier no longer reverts. Note that this means the vault will no longer claim/distribute rewards once the delegated stake limit is reached.

Low Risk

[L-1] Enforcing a minimum number of shares per deposit is sub-optimal

Context:

- SS.sol#L57-L58
- SS.sol#L249-L251

Description: On every deposit, `depositNative()` checks that the number of shares minted to the user is not less than `MIN_SHARES`:

```
/// @notice Minimum shares that must be minted for a deposit
uint256 public constant MIN_SHARES = 1e6;
```

```
uint256 shares = previewDeposit(msg.value);
if (shares == 0) revert ZeroShares();
if (shares < MIN_SHARES) revert InsufficientShares();
```

However, since the vault's share price constantly increases over time, the minimum amount of assets needed for a deposit is not fixed and will also increase over time.

Additionally, the `MIN_SHARES` check is only enforced in `depositNative()`. As such, it is possible for a user to hold less shares than `MIN_SHARES` if their withdrawal leaves less than `1e6` shares remaining.

Recommendation: If enforcing a minimum number of shares per deposit is not necessary, consider removing the `shares < MIN_SHARES` check.

Alternatively, consider enforcing a minimum amount of assets per deposit, instead of shares.

Dinero: Fixed in commit 33bfb32.

Renascence: Verified, the `MIN_SHARES` check has been removed.

Informational

[I-1] Minor code improvements

Context:

1. SS.sol#L18, SS.sol#L155-L164
2. SS.sol#L137
3. SS.sol#L148-L153
4. SS.sol#L236-L245
5. SS.sol#L246
6. SS.sol#L340-L343

Description:

1. SS.sol#L18 - The contract does not have to inherit `ERC20Upgradeable` as it is already inherited by `ERC4626Upgradeable`. Consider removing `ERC20Upgradeable` in this line. Additionally, this also allows the `decimals()` function to be removed.
2. SS.sol#L137 - This check can be removed as it is already checked in `__Ownable_init()`.
3. SS.sol#L148-L153 - `_authorizeUpgrade()` can be declared as `view`.
4. SS.sol#L236-L245 - `depositNative()` should be declared as `external`.
5. SS.sol#L246 - Checking `msg.value == 0` is unnecessary as it is implicitly enforced by the `shares == 0` check below.
6. SS.sol#L340-L343 - Having a fallback function that reverts is redundant and can be removed.

Dinero: All issues have been fixed in commit 0500f53.

Renascence: Verified, all issues have been fixed as recommended.