Slash Proof Audit Report

Version 2.0

Audited by:

bytes032

alexxander

March 26, 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 Slash Proof

Slash Proof is an on-chain insurance protocol where users purchase coverage that can be received as reimbursement in the event of validators getting slashed. The key user flows of the protocol are:

- Liquidity Providers deposit wstETH into a Vault. In return, they earn fees generated from the purchases of coverage by users.
- Users of the protocol have the option to purchase coverage by paying a premium. The premium can vary based on factors such as the amount of coverage desired and the perceived risk associated with the validator being insured.
- In the event that a validator is slashed and an user has purchased coverage for that validator, they can initiate a claim and receive reimbursement.

2.2 Overview

Project	Slash Proof
Repository	onchain-validator-slashing-protection
Commit Hash	ddce9d8c2459...
Mitigation Hash	3c4f1eebee10...
Date	6 March 2024 - 9 March 2024

2.3 Issues Found

Severity	**Count**
High Risk	0
Medium Risk	4
Low Risk	3
Informational	4
Total Issues	**11**

3 Findings Summary

ID	Description	Status
M-1	Base contracts inherited by the upgradeable `Vault` contract need storage gaps	Resolved
M-2	In `Vault.purchase()` the denomination of `premium` is ambiguous	Resolved
M-3	`Vault.withdrawServiceFee()` should reset `totalServiceToken` and `totalServiceEth`.	Resolved
M-4	Total coverage is not reduced in `Vault.claim()`	Resolved
L-1	Upgradeable contracts `Vault` and `ERC20Mintable` are missing a `_disableInitializers()`	Acknowledged
L-2	`Vault.purchase()` will not refund excessively sent ETH if the coverage purchase is made with `underwritingToken`.	Resolved
L-3	Signature malleability is possible in `VerifySignature.isSignedBySigner()`	Resolved
I-1	Upgradeable contract `Vault` is missing a call to `__ReentrancyGuard_init()`	Resolved
I-2	Function `Vault.updateExpirationTime()` can break trust assumptions.	Resolved
I-3	Code improvements	Resolved
I-4	The value of `vaultStorage.config.feeBasisPoints` in `Vault.sol` should be constrained by an upper bound.	Resolved

4 Findings

Medium Risk

[M-1] Base contracts inherited by the upgradeable `Vault` contract need storage gaps

Context:

- Vault.sol

Description: For upgradeable contracts, there must be a storage gap to allow the addition of new state variables in the future without compromising the storage compatibility with existing deployments. Without a storage gap, if there are any new variables in the `DivaBeaconOracle` contract, they will override variables in the `Vault` contract, since `Vault` extends `DivaBeaconOracle`.

OpenZeppelin Reference here.

Recommendation:

```
@@ -11,6 +11,7 @@ contract DivaBeaconOracle {
    /// @notice Mapping from validator index to validator struct
    mapping(uint256 => BeaconOracleHelper.Validator) public validatorState;

+   uint256[50] __gap;
    /// @notice Prove slashed & status epochs
    function proveValidatorField(
        BeaconOracleHelper.BeaconStateRootProofInfo calldata
        _beaconStateRootProofInfo,
```

Slash Proof: Fixed in commit 60a2bff.

Renascence: Verified, the recommendation has been implemented.

[M-2] In `Vault.purchase()` the denomination of `premium` is ambiguous

Context:

- Vault.sol

Description: Purchasing coverage requires the user to pay an agreed `premium` for the `coverage` that he will receive for the given node. The issue is that it's not clear if the `premium` is denominated in native ETH or in `underwritingToken` (Assumed to be `wstETH`). If the `premium` is agreed upon to be in ETH, then the user cannot pay in `underwritingToken` since the price of `wstETH` is higher than ETH and the user will lose out on funds. If the `premium` is agreed upon to be in `wstETH` then the user can choose to pay the same amount of `premium` but in native ETH, thus receiving an unfair discount.

```
# Vault.sol

function purchase(
    ...
    uint256 premium,
    ...
) external payable nonReentrant onlyNonZero(premium) WhenNotPaused
WhenVaultNotExpired {
    validatePurchaseParams(
        riskScore,
        premium,
        ...
    );
    ...
    bool paidInEth = true;
    if (msg.value < premium) {
        underwritingToken.transferFrom(msg.sender, address(this), premium);
        paidInEth = false;
    }
    if (paidInEth && msg.value > premium) {
        msg.sender.call{value: (msg.value - premium)}("");
    }
```

Recommendation: The denomination of `premium` that's used in the off-chain components should be assumed and the conversion rate supplied in `convertToEthFactor` should be used to convert `premium` either in `underwritingToken` or native ETH.

Slash Proof: Fixed in commit 3c4f1ee.

Renascence: Verified, the recommendation has been implemented.

[M-3] `Vault.withdrawServiceFee()` should reset `totalServiceToken` and `totalServiceEth`.

Context:

- Vault.sol

Description: `Vault.withdrawServiceFee()` is supposed to transfer the accumulated fees to an address specified by the owner. The issue is that `totalServiceToken` and `totalServiceEth` are not reset back to `0`. Firstly, this gives the owner full control over the Vault's funds since he can repeatedly call `Vault.withdrawServiceFee()` and drain the funds. Secondly, assuming a non-malicious owner, there could be a scenario where `Vault.withdrawServiceFee()` is called once during the Vault being active and once after the Vault has expired and users have withdrawn. In such a scenario, `Vault.withdrawServiceFee()` will revert unexpectedly the second time it has been called since the already withdrawn fee will still be accounted for in `totalServiceToken` and `totalServiceEth`.

```
function withdrawServiceFee(address payable withdrawAddress) external onlyOwner {
    underwritingToken.transfer(withdrawAddress, vaultStorage.state.totalServiceToken);
    withdrawAddress.call{value: vaultStorage.state.totalServiceEth}("");
}
```

Recommendation:

```
@@ -364,8 +365,12 @@ contract Vault is
     function withdrawServiceFee(address payable withdrawAddress) external onlyOwner {
-        underwritingToken.transfer(withdrawAddress,
vaultStorage.state.totalServiceToken);
-        withdrawAddress.call{value: vaultStorage.state.totalServiceEth}("");
+        uint256 amountOfEth = vaultStorage.state.totalServiceEth;
+        uint256 amountOfToken = vaultStorage.state.totalServiceToken;
+        vaultStorage.state.totalServiceEth = 0;
+        vaultStorage.state.totalServiceToken = 0;
+        underwritingToken.transfer(withdrawAddress, amountOfToken);
+        withdrawAddress.call{value: amountOfEth}("");
     }
```

Slash Proof: Fixed in commit 3c4f1ee.

Renascence: Verified, the recommendation has been implemented.

[M-4] Total coverage is not reduced in `Vault.claim()`

Context:

- Vault.sol

Description: When a Validator is slashed, the user that has covered that Validator can call `Vault.claim()` and receive a `coverage` payout. The issue is that the state variable `vaultStorage.state.totalCoverage` is not reduced by `coverage`. However, `vaultStorage.state.totalUnderWriterEth` and `vaultStorage.state.totalUnderWriterToken` are reduced, which means future users might be unable to purchase coverage since `Vault.purchase()` might revert unexpectedly since `vaultStorage.state.totalCoverage` has an inflated value and the check for enough available funds for coverage might fail.

```
# Vault.sol

function purchase(
    ...
    uint256 _coverage,
    ...
) external payable nonReentrant onlyNonZero(premium) WhenNotPaused
WhenVaultNotExpired {
    ...
    if (
        totalPossibleCoverage * vaultStorage.config.leverageBasisPoints /
        basisPtsToDecimal
            < vaultStorage.state.totalCoverage + _coverage
    ) {
        revert NotEnoughFundsForCoverage();
    }
    vaultStorage.state.totalCoverage += _coverage;
    ...
}
```

Recommendation:

```
@@ -292,6 +293,7 @@ contract Vault is
     vaultStorage.state.totalUnderWriterToken -= amountOfToken;
     vaultStorage.state.totalUnderWriterEth -= amountOfEth;
+    vaultStorage.state.totalCoverage -=
vaultStorage.state.coveredValidators[claimParams.validatorIndex].coverage;
     // paying out underwriterTokens
```

Slash Proof: Fixed in commit 60a2bff.

Renascence: Verified, the recommendation has been implemented.

Low Risk

[L-1] Upgradeable contracts `Vault` and `ERC20Mintable` are missing a `_disableInitializers()`

Context:

- Vault.sol
- ERC20Mintable.sol

Description: The best practice in contracts that inherit from `Initializable` is to disable the initializers since if left uninitialized they can be invoked in the implementation contract by an attacker. For example, there is a past vulnerability disclosure that demonstrates how initializers getting called in the implementation can lead to contract takeover where the attacker can appoint an owner and would self-destruct the implementation, therefore, bricking the Proxy: OZ post-mortem. Although this issue has been fixed from OZ version 4.3.2 it's still best practice to call `Initializable._disableInitializers()` in a constructor in the implementation.

```
# Initializable.sol

* [CAUTION]
 * ====
 * Avoid leaving a contract uninitialized.
 *
 * An uninitialized contract can be taken over by an attacker. This applies to both a
 proxy and its implementation
 * contract, which may impact the proxy. To prevent the implementation contract from
 being used, you should invoke
 * the {_disableInitializers} function in the constructor to automatically lock it
 when it is deployed:
 *
```

Recommendation:

```
@@ -8,6 +8,9 @@ import {Initializable} from
    "@openzeppelin-contracts-upgradeable/contracts/proxy/utils/Initializable.sol";

    contract ERC20Mintable is Initializable, ERC20Upgradeable, OwnableUpgradeable {
+       constructor() {
+           _disableInitializers();
+       }
        function initialize(string memory name, string memory symbol) public
        initializer {
            __ERC20_init(name, symbol);
            __Ownable_init(msg.sender);
```

```
@@ -86,6 +86,9 @@ contract Vault is
     uint256 conversionFactorDivision;
     uint256 basisPtsToDecimal;

+    constructor() {
+        _disableInitializers();
+    }
     function initialize(
         address _messageSignerAddress,
         address underwritingTokenAddress,
```

Slash Proof: Finding is Acknowledged.

[L-2] `Vault.purchase()` **will not refund excessively sent ETH if the coverage purchase is made with** `underwritingToken`**.**

Context:

- Vault.sol

Description: In the case where `msg.value < premium`, `Vault.purchase()` will transfer the user's tokens as payment for the coverage, however, it doesn't consider giving the user a refund. There could be an edge case where the user has approval towards the `Vault` contract and attempts to purchase coverage with ETH, but miss-inputs a lower value than `premium`. In that case, the user will pay the `premium` in tokens but will not be refunded the `msg.value`. This is a user's error scenario, therefore, the severity is Low but it's best to consider performing a refund since excess tokens cannot be rescued from the contract unless an upgrade is performed.

```
# Vault.sol

bool paidInEth = true;
    if (msg.value < premium) {
        underwritingToken.transferFrom(msg.sender, address(this), premium);
        paidInEth = false;
    }
    if (paidInEth && msg.value > premium) {
        msg.sender.call{value: (msg.value - premium)}("");
    }
```

Recommendation: A possible solution is to refund `msg.value` when coverage is purchased with tokens.

```
@@ -211,6 +212,7 @@ contract Vault is
     bool paidInEth = true;
         if (msg.value < premium) {
             underwritingToken.transferFrom(msg.sender, address(this), premium);
+            msg.sender.call{value: msg.value}("");
             paidInEth = false;
         }
```

Another solution would be to revert if `msg.value > 0` but is not enough to cover the `premium`.

```
@@ -210,6 +211,7 @@ contract Vault is
     vaultStorage.state.totalCoverage += _coverage;
     bool paidInEth = true;
         if (msg.value < premium) {
+            require(msg.value == 0, "Purchasing with tokens, but positive msg.value");
             underwritingToken.transferFrom(msg.sender, address(this), premium);
             paidInEth = false;
         }
```

Slash Proof: Fixed in commit 60a2bff.

Renascence: Verified, the recommendation has been implemented.

[L-3] Signature malleability is possible in `VerifySignature.isSignedBySigner()`

Context:

- VerifySignature.sol

Description: According to Open Zeppelin, depending on how the signature library used calculates the `s` value of the signature (either lower or higher order), there could be signature malleability when using `ercrecover`.

```
# OZ ECDSA.sol

// unique. Appendix F in the Ethereum Yellow paper
(https://ethereum.github.io/yellowpaper/paper.pdf), defines
// the valid range for s in (301): 0 < s < secp256k1n ÷ 2 + 1, and for v in (302): v
{27, 28}. Most
// signatures from current libraries generate a unique signature with an s-value in
the lower half order.
//
// If your library generates malleable signatures, such as s-values in the upper
range, calculate a new s-value
// with 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and
flip v from 27 to 28 or
// vice versa. If your library also generates signatures with 0/1 for v instead 27/28,
add 27 to v to accept
// these malleable signatures as well.
```

Currently, `VerifySignature.isSignedBySigner()` does not consider the order of the supplied `s` value.

```
function isSignedBySigner(bytes memory _signature, bytes32 _hash, address
signer_address)
    public
    pure
    returns (bool)
{
    (bytes32 r, bytes32 s, uint8 v) = splitSignature(_signature);
    return ecrecover(_hash, v, r, s) == signer_address;
}

function splitSignature(bytes memory sig) public pure returns (bytes32 r, bytes32 s,
uint8 v) {
    require(sig.length == 65, "invalid signature length");

    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := byte(0, mload(add(sig, 96)))
    }
}
```

Recommendation: Consider using OpenZeppelin's ECDSA library.

Slash Proof: Fixed in commit 3c4f1ee.

Renascence: Verified, the recommendation has been implemented.

Informational

[I-1] Upgradeable contract `Vault` is missing a call to `__ReentrancyGuard_init()`

Context:

- Vault.sol

Description: The `Vault.sol` contract inherits OZ's `ReentrancyGuardUpgradeable`, however, `Vault.initialize()` is missing a call to `ReentrancyGuardUpgradeable.__ReentrancyGuard_init()`. Currently, the missing call doesn't lead to a vulnerability, however, it is best practice to initialize the reentrancy guard.

Recommendation:

```
@@ -97,6 +97,7 @@ contract Vault is
     __Ownable_init(msg.sender);
+    __ReentrancyGuard_init();

     updateConfig(
         _messageSignerAddress, feeMaster, feeBasisPoints, minimumPremium,
         leverageBasisPoints
     );
```

Slash Proof: Fixed in commit 60a2bff.

Renascence: Verified, the recommendation has been implemented.

[I-2] Function `Vault.updateExpirationTime()` can break trust assumptions.

Context:

- Vault.sol

Description: When users purchase coverage the `expiration` state variable indicates the time period for which they agree to pay a `premium` for certain amount of `coverage`. The problem is that the owner can decrease the `expiration`, therefore, breaking the trust assumptions that existing users have towards the system.

```
function updateExpirationTime(uint256 _expiration) external onlyOwner {
    if (_expiration <= block.timestamp) revert BlockTimeStampGreaterThanExpiration();
    expiration = _expiration;
}
```

Recommendation: Consider not allowing `expiration` to be reduced.

Slash Proof: Fixed in commit d506a0a.

Renascence: Verified, the recommendation has been implemented.

[I-3] Code improvements

Context:

- Vault.sol
- VaultMath.sol

Description: In `VaultMath.computeShares()` the `totalUnderwriterValue` is calculated at the start and there is no need to do the same calculation twice.

```
# VaultMath.sol

function computeShares(
    ...
) internal returns (uint256) {
    uint256 totalUnderwriterValue = (
        wstEthToEth(totalUnderwriterToken, convertToEthFactor,
        conversionFactorDivision)
            + totalUnderwriterEth
    );
    if (totalUnderwriterValue > 0) {
        return wstEthToEth(amount, convertToEthFactor, conversionFactorDivision)
            * totalSharesToken
            / (
                wstEthToEth(totalUnderwriterToken, convertToEthFactor,
                conversionFactorDivision)
                    + totalUnderwriterEth
            );
    } else {
        return 0;
    }
}
```

Recommendation:

```
@@ -40,8 +40,7 @@ library VaultMath {
    return wstEthToEth(amount, convertToEthFactor, conversionFactorDivision)
        * totalSharesToken
        / (
-           wstEthToEth(totalUnderwriterToken, convertToEthFactor,
conversionFactorDivision)
-           + totalUnderwriterEth
+           totalUnderwriterValue
        );
```

Description: In `Vault.coveredValidators()`, `vaultStorage.state.validatorOwners[user-Address].length` and `vaultStorage.state.validatorOwners[userAddress][i]` can be cached.

Recommendation:

```
@@ -377,13 +382,15 @@ contract Vault is
     external
     view
     returns (ValidatorsOfAddress[] memory)
-    {
+    {
+        uint256 validatorsLength =
vaultStorage.state.validatorOwners[userAddress].length;
         ValidatorsOfAddress[] memory allValidators =
-            new
ValidatorsOfAddress[](vaultStorage.state.validatorOwners[userAddress].length);
-        for (uint256 i = 0; i <
vaultStorage.state.validatorOwners[userAddress].length; i++) {
+            new ValidatorsOfAddress[](validatorsLength);
+        for (uint256 i = 0; i < validatorsLength; i++) {
+            uint256 validatorIdx = vaultStorage.state.validatorOwners[userAddress][i];
             allValidators[i] = ValidatorsOfAddress({
-                validatorIndex: vaultStorage.state.validatorOwners[userAddress][i],
-                coverage: vaultStorage.state.coveredValidators[vaultStorage.state.val
idatorOwners[userAddress][i]]
+                validatorIndex: validatorIdx,
+                coverage: vaultStorage.state.coveredValidators[validatorIdx]
             });
         }
         return allValidators;
```

Description: `ClaimParams` is an unused import in `VaultMath.sol`.

Recommendation:

```
@@ -1,7 +1,5 @@
     pragma solidity ^0.8.19;
-    import {ClaimParams} from "../Vault.sol";
-
     library VaultMath {
```

Description: The `FundOrWstEthDoesNotMatchParams`, `NotEnoughShares`, `CoverageGreaterThan32` and `ClaimNotAllowed` errors in `Errors.sol` are not used. Consider removing them.

Recommendation:

```
@@ -2,8 +2,6 @@ pragma solidity ^0.8.16;
    interface Errors {
        error AmountIsZero();
-       error FundOrWstEthDoesNotMatchParams();
-       error NotEnoughShares();
        error StateRootVerificationFailed();
        error ValidatorNotSlashed();
        error WithdrawNotAllowed();
@@ -11,12 +9,10 @@ interface Errors {
        error MintingZeroSharesNotAllowed();
        error BlockTimeStampGreaterThanExpiration();
        error SignerAddressZero();
-       error CoverageGreaterThan32();
        error ClaimNotSendByOwner();
        error AlreadyCoveredOrClaimed();
        error ConversionSignatureExpired();
        error PremiumLowerThanMinimumAllowed();
-       error ClaimNotAllowed();
        error ContractIsPaused();
        error NotEnoughFundsForCoverage();
        error PurchaseSignIsExpired();
```

Slash Proof: Fixes applied in commit 60a2bff.

Renascence: Verified, the recommendations have been implemented.

[I-4] The value of `vaultStorage.config.feeBasisPoints` in `Vault.sol` should be constrained by an upper bound.

Context:

- Vault.sol

Description: Currently, it is possible for the owner to call `Vault.updateConfig()` and set a 100% service fee. It is best practice to constrain the value of `vaultStorage.config.feeBasisPoints` up to a constant value (e.g., 10%) to reduce the trust assumptions users have to make.

```
# Vault.sol

function updateConfig(
    address updatedMessageSignerAddress,
    address updatedFeeMaster,
    uint256 updatedFeeBasisPoints,
    uint256 updatedMinimumPremium,
    uint256 leverageBasisPoints
) public onlyOwner {
    if (updatedMessageSignerAddress == address(0)) revert SignerAddressZero();
    vaultStorage.config.messageSignerAddress = updatedMessageSignerAddress;
    if (updatedFeeMaster == address(0)) revert SignerAddressZero();
    vaultStorage.config.feeMaster = updatedFeeMaster;
    vaultStorage.config.feeBasisPoints = updatedFeeBasisPoints;
    vaultStorage.config.minimumPremium = updatedMinimumPremium;
    vaultStorage.config.leverageBasisPoints = leverageBasisPoints;
}
    }
```

Recommendation: Introduce constant `MAX_FEE` in `Vault.sol`.

```
@@ -79,6 +79,8 @@ contract Vault is
 {
     using VerifySignature for bytes;

+    // example: 10% max fee
+    uint256 constant MAX_FEE = 1000;

@@ -354,6 +357,7 @@ contract Vault is
     vaultStorage.config.messageSignerAddress = updatedMessageSignerAddress;
     if (updatedFeeMaster == address(0)) revert SignerAddressZero();
     vaultStorage.config.feeMaster = updatedFeeMaster;
+    if (updatedFeeBasisPoints > MAX_FEE) revert FeeNotAllowed();
     vaultStorage.config.feeBasisPoints = updatedFeeBasisPoints;
     vaultStorage.config.minimumPremium = updatedMinimumPremium;
     vaultStorage.config.leverageBasisPoints = leverageBasisPoints;
```

Add `FeeNotAllowed` error in `Errors.sol`.

```
@@ -23,4 +23,5 @@ interface Errors {
     error PurchaseSignVerificationFailed();
     error ConversionSignVerificationFailed();
     error VaultExpired();
+    error FeeNotAllowed();
```

Slash Proof: Fixed in commit 3c4f1ee.

Renascence: Verified, the recommendation has been implemented.

5 Centralization Risks

5.1 Claiming owed coverage relies on a signature from the protocol

The `Vault.claim()` function requires a valid `convertFactorSign` signature issued by `vaultStorage.config.messageSignerAddress`, otherwise, the function reverts. The owner of the `Vault` contract has the power to change who `vaultStorage.config.messageSignerAddress` is, therefore, users that have coverage claims must trust the owner that `vaultStorage.config.messageSignerAddress` will always produce a valid signature in order for them to successfully call `Vault.claim()`.

5.2 Pausable functions

The owner of the `Vault` contract can use the `Vault.setPause()` functionality to block the `Vault.claim()` and `Vault.withdraw()` functions. This means the owner has the power to indefinitely hold custody (inside of the `Vault` contract) of any user's deposited funds or owed coverage. The owner must be trusted to use `Vault.setPause()` only in case of emergency.

5.3 Upgradeable contracts

Users who interact with the protocol have to be aware that the `Vault` smart contract is designed to be upgradeable. This means whoever holds owner rights over the upgrade process must be fully trusted.