NodeSet

Constellation Contract Review

Security Assessment Report

Version: 2.1

October, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the NodeSet smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the NodeSet smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the NodeSet smart contracts in scope.

Overview

NodeSet's Constellation is a decentralised staking protocol built on top of Rocket Pool, designed to optimise Ethereum staking for both individual and institutional users. It introduces tokenized staking positions (`xrETH` and `xRPL`) and a managed node operation system through a SuperNode concept.

This review is primarily focused on the core contracts of the Constellation protocol, including the `WETHVault`, `RPLVault`, `OperatorDistributor`, `SuperNodeAccount`, and associated utility contracts.

Particular attention was given to the protocol's unique features such as the liquidity management between `ETH` and `RPL`, the merkle claim streaming process, and the delegation of node operations to a decentralised operator set.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the NodeSet repository.

The scope of this time-boxed review was strictly limited to files at commit 0ac1f6a. Retesting activities targeted the commit 6976cd8 *Note: third party libraries and dependencies, such as OpenZeppelin, were excluded from the scope of this assessment.*

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 7 issues during this assessment. Categorised by their severity:

- High: 2 issues.
- Medium: 2 issues.

- Informational: 3 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the NodeSet smart contracts in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
CNST-01	Incorrect Use Of `ETH` Balance Instead Of `WETH` Balance	High	Resolved
CNST-02	Incorrect Fee Calculation Logic In `previewMint()` Leading To Reduced Fees	High	Resolved
CNST-03	Double Counting Issue When Admin Calls `sweepLockedTVL()`	Medium	Resolved
CNST-04	Unchecked Future Timestamp May Lead To Replay Attacks On `setTotalYieldAccrued()`	Medium	Resolved
CNST-05	Uninitialised `UUPSUpgradeable` and `AccessControlUpgradeable`	Informational	Closed
CNST-06	Incorrect Usage Of The `initializer` Modifier Instead Of `onlyInitializing`	Informational	Resolved
CNST-07	Miscellaneous General Comments	Informational	Resolved

CNST-01	Incorrect Use Of ETH Balance Instead Of WETH Balance		
Asset	Constellation/OperatorDistributor.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

In the `rebalanceWethVault()` function on line [**415**], the ETH balance is incorrectly referenced instead of the WETH balance:

```
weth.deposit{value: address(this).balance}();
SafeERC20.safeTransfer(IERC20(address(weth)), address(vweth), address(this).balance);
```

Since all ETH is wrapped to WETH on line [**414**] using the `weth.deposit()` function, `address(this).balance` returns `0`. As a result, when `balanceEthAndWeth < requiredWeth`, no WETH is transferred to the `WETHVault`, which may prevent users from being able to withdraw WETH from the vault.

Recommendations

Substitute `address(this).balance` with `weth.balanceOf(address(this))`.

Resolution

This issue has been fixed in PR#353 by using the balance of WETH instead of balance of ETH.

CNST-02	Incorrect Fee Calculation Logic In `previewMint()` Leading To Reduced Fees		
Asset	`Constellation/WETHVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

There is an inconsistency in how fees are calculated between the `previewMint()` and `previewDeposit()` functions.

Both functions use `getMintFeePortion()` to calculate the fees, however:

- In `previewMint()`, the fees are calculated based on the assets needed to mint the shares, which do not include the fees:

```
function previewMint(uint256 shares) public view virtual override returns (uint256) {
   uint256 assets = super.previewMint(shares);
   return assets + this.getMintFeePortion(assets);
    }
```

- In `previewDeposit()`, the fees are based on the total deposited assets, which already include the fees:

```
function previewDeposit(uint256 assets) public view virtual override returns (uint256) {
    uint256 fee = this.getMintFeePortion(assets);
    return super.previewDeposit(assets - fee);
}
```

As a result, `WETHVault.mint()` mints more shares than `WETHVault.deposit()` for the same asset amount. Users can exploit this difference to reduce fees by opting for one method over the other.

This discrepancy allows users to bypass the intended fee calculation, which consequently leads to a reduction in the protocol's revenue.

Recommendations

Align the fee calculation methodology between `previewMint()` and `previewDeposit()` to ensure consistent fee application by creating a new function that calculates fees in `previewMint()` based on the raw asset value.

Resolution

The development team has fixed this issue in PR#349. A new function `getAdditionalMintFeeToReceive()` has been created that calculates the fees that should be added to an amount of assets that does not already include fees. This function is used in function `previewMint()`.

CNST-03	Double Counting Issue When Admin Calls `sweepLockedTVL()`		
Asset	`Constellation/MerkleClaimStreamer.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

If the admin calls `sweepLockedTVL()`, the `merkleRewards` will be double-counted in `totalAssets()` of both `WETHVault` and `RPLVault`.

The `WethVault.totalAssets()` function has the following implementation:

```
function totalAssets() public view override returns (uint256) {
    OperatorDistributor od = OperatorDistributor(getDirectory().getOperatorDistributorAddress());
    (uint256 distributableYield, bool signed) = this.getDistributableYield();
    uint256 merkleRewards = MerkleClaimStreamer(getDirectory().getMerkleClaimStreamerAddress()).getStreamedTvlEth();
    return (
        uint256(
            int(IERC20(asset()).balanceOf(address(this)) + od.getTvlEth() +  merkleRewards)  +
                (signed ? -int(distributableYield) : int(distributableYield))
        )
    );
}
```

However, when the admin calls `sweepLockedTVL()`, the `priorEthStreamAmount` is not updated. Despite this, the function still sends this amount to the `OperatorDistributor`, whose TVL includes both the Operator's `ETH` balance and the `priorEthStreamAmount` as part of the `totalAssets()` of `WETHVault`. This leads to double-counting of the `merkleRewards`.

The same issue applies to `priorRplStreamAmount` and the `RPLVault`. This amount is also not updated, resulting in the double counting of `merkleRewards` in `RPLVault.totalAssets()`.

Recommendations

Set the variables `priorRplStreamAmount` and `priorEthStreamAmount` to zero when the admin calls `sweepLockedTVL()`.

Resolution

The development team has fixed this issue in PR#373. The prior amounts are now updated inside the function `sweepLockedTVL()` using a new internal function `_updatePriorStreamAmounts()`. Additionally, the `sweepLockedTVL()` now can only be called by the Protocol. This function is also called in the function `setStreamingInterval()`, when the streaming interval is update.

CNST-04	Unchecked Future Timestamp May Lead To Replay Attacks On `setTotalYieldAccrued()`		
Asset	`Constellation/PoAConstellationOracle.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `setTotalYieldAccrued()` function uses the timestamp `sigData.timeStamp` to prevent outdated updates. However, it does not check for timestamps set far in the future:

```
require(sigData.timeStamp > _lastUpdatedTotalYieldAccrued, 'cannot update oracle using old data');

// ...(snip)...

_lastUpdatedTotalYieldAccrued = block.timestamp;
```

While this ensures that the new timestamp is greater than the last update, it does not check whether the timestamp is realistic or whether it is set unreasonably far into the future.

As the signature is tied to the timestamp, if the signature was generated with a timestamp set very far ahead into the future, anyone in possession of said signature would be able to replay the same request multiple times, so long as `sigData.timeStamp > block.timestamp`.

Recommendations

Add a maximum allowed time difference between the current block timestamp and the signature timestamp.

Alternatively, consider using a nonce-based system instead of, or in addition to, timestamps to prevent replay attacks.

Resolution

This issue has been addressed by adding the following require statement:

```
require(sigData.timeStamp <= block.timestamp, 'cannot update oracle using future data');
```

CNST-05	Uninitialised `UUPSUpgradeable` and `AccessControlUpgradeable`
Asset	`External/NodeSetOperatorRewardDistributor.sol, External/Treasury.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The contract `Treasury` inherits the abstract contract `UUPSUpgradeable` and `AccessControlUpgradeable`. However, these contracts are not initialised inside the `Treasury` child contract.

The contract `NodeSetOperatorRewardDistributor` inherits the abstract contract `UUPSUpgradeable` and `AccessControlUpgradeabl` However, these contracts are not initialised inside the `NodeSetOperatorRewardDistributor` child contract.

Recommendations

Initialize the contracts `AccessControlUpgradeable` and `UUPSUpgradeable` inside `Treasury` and `NodeSetOperatorRewardDistribut`

Resolution

This issue is partially fixed. The function `__UUPSUpgradeable_init()` has been added to the `Treasury.initialize()`, however, `UUPSUpgradeable` is still not initialided in `NodeSetOperatorRewardDistributor`.

`AccessControlUpgradeable` is also still not initialised in both `Treasury` and `NodeSetOperatorRewardDistributor`.

The development team has acknowledged the finding and its partial resolution.

CNST-06	Incorrect Usage Of The `initializer` Modifier Instead Of `onlyInitializing`
Asset	`Constellation/Utils/UpgradeableBase.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Most of the contracts inherit the abstract contract `UpgradeableBase`. This contract uses the `initializer` modifier in its `initialize()` function, which is called by the child contract's `initialize()` function as follows:

```
function initialize(address _directory) public override initializer {
   super.initialize(_directory);
   // ...
```

This implementation prevents the child contract from being initialized in a separate call after deployment. Additionally, upgrading to a new implementation is not possible if the same logic is used in the new version.

Recommendations

Replace the `initialize` modifier with the `onlyInitializing` modifier.

Resolution

This issue has been fixed by replacing the `initialize` modifier with the `onlyInitializing` modifier as recommended.

CNST-07	Miscellaneous General Comments
Asset	*
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Unused Custom Errors**
   Related Asset(s): Constellation/Utils/Errors.sol
   There are multiple unused custom errors in `Errors.sol` contract.

```
error NotAContract(address addr);
error BadBondAmount(uint256 expectedBondAmount, uint256 actualBondAmount);
error InsufficientBalance(uint expectedBalance, uint256 actualBalance);
error BadRole(bytes32 role, address user);
error BadSender(address expectedSender);
error LowLevelCall(bool success, bytes data);
error ZeroAddressError();
error BadPredictedCreation(address expected, address actual);
```

   It is recommended that the definition be removed when custom errors are unused.

2. **Multiple Instances Of Unused Imports**
   Related Asset(s): Constellation/MerkleClaimStreamer.sol, Constellation/SuperNodeAccount.sol
   Several contracts in the codebase contain unused imports. This includes:

   - In `MerkleClaimStreamer`:

```
import '../Interfaces/RocketPool/IRocketMerkleDistributorMainnet.sol';
```

   - In `SuperNodeAccount`:

```
import '../Interfaces/RocketPool/RocketTypes.sol';
import '../Interfaces/RocketPool/IRocketNetworkVoting.sol';
import '../Interfaces/RocketPool/IRocketDAOProtocolProposal.sol';
```

   Remove all unused imports from the contracts.

3. **Lack Of Zero Address Checks**
   Related Asset(s): External/Treasury.sol & Constellation/Utils/UpgradeableBase.sol
   There are mutiple instances where address parameters are not checked against the zero address. This occurs in functions that transfer funds or initialise important contract references. Specifically:

   - In `Treasury._claimEthInternal()`:

```
function _claimEthInternal(address payable _to, uint256 _amount) internal {
    (bool success, ) = _to.call{value: _amount}('');
    require(success, 'Failed to transfer ETH to recipient');
    emit ClaimedEth(_to, _amount);
}
```

- In `UpgradeableBase.initialize()`:

```
function initialize(address directoryAddress) public virtual initializer {
    _directory = Directory(directoryAddress);
    __UUPSUpgradeable_init();
}
```

Implement zero address checks in all functions that accept address parameters, especially those involving fund transfers or critical contract initializations.

4. **Redundant External Calls In `transferMerkleClaimToStreamer()`**

   Related Asset(s): Constellation/OperatorDistributor.sol

   In the `transferMerkleClaimToStreamer()`, there are multiple redundant calls to `getDirectory().getMerkleClaimStreamerAddress()`:

```
function transferMerkleClaimToStreamer(uint256 ethAmount, uint256 rplAmount) external onlyProtocol {
    if (ethAmount > 0) {
        (bool success, ) = getDirectory().getMerkleClaimStreamerAddress().call{value: ethAmount}('');
        require(success, 'ETH transfer to MerkleClaimStreamer failed');
    }

    if (rplAmount > 0) {
        SafeERC20.safeTransfer(
            IERC20(_directory.getRPLAddress()),
            getDirectory().getMerkleClaimStreamerAddress(),
            rplAmount
        );
    }
}
```

   Store the result of `getDirectory().getMerkleClaimStreamerAddress()` in a local variable

5. **Redundant `@notice` Tag In Documentation**

   Related Asset(s): Constellation/OperatorDistributor.sol

   In the documentation for `getTvlEth()`, there is a redundant `@notice` tag:

```
/**
 * @notice Returns the total ETH and WETH managed by the contract, including both the ETH+WETH balances of this contract
 * @notice Returns the total ETH and WETH managed by the contract, including both the ETH+WETH balances of this contract
 * and the SuperNode's staked ETH.
 */
```

   The first two lines are identical, with the second line being a continuation of the description.

   Remove the redundant `@notice` tag and merge the description into a single, comprehensive statement.

6. **Redundant WETH Balance Calculation**

   Related Asset(s): Constellation/OperatorDistributor.sol

   In `rebalanceWethVault()`, there's an unnecessary recalculation of the `WETH` balance:

```
uint256 wethBalance = IERC20(address(weth)).balanceOf(address(this));
uint256 balanceEthAndWeth = IERC20(address(weth)).balanceOf(address(this)) + address(this).balance;
```

   The `wethBalance` variable is correctly calculated, but then the same calculation is repeated in the very next line when computing `balanceEthAndWeth`.

   Use the previously calculated `wethBalance` in the `balanceEthAndWeth` calculation.

7. **Redundant External Calls In `stakeRpl()`**

   Related Asset(s): Constellation/OperatorDistributor.sol

   In `stakeRpl()`, there are multiple redundant external calls to `_directory.getRPLAddress()` and `_directory.getRocketNodeStakingAddress()`:

```
function stakeRpl(uint256 _amount) external onlyProtocol {
    SafeERC20.safeApprove(IERC20(_directory.getRPLAddress()), _directory.getRocketNodeStakingAddress(), 0);
    SafeERC20.safeApprove(IERC20(_directory.getRPLAddress()), _directory.getRocketNodeStakingAddress(), _amount);
    IRocketNodeStaking(_directory.getRocketNodeStakingAddress()).stakeRPLFor(
        getDirectory().getSuperNodeAddress(),
        _amount
    );
}
```

Store the results of these external calls in local variables.

8. **Multiple Typos Throughout The Codebase**

   Related Asset(s): *.sol

   A review of the codebase has revealed the presence of multiple (typos) across various contracts and comments. While some of these may be minor, the cumulative effect can impact code readability, maintainability and potentially lead to misunderstandings during development or auditing processes.

   Implement a spell-checking tool to identify and correct typos.

9. **Unnecessary Non-Negativity Check On Unsigned Integer**

   Related Asset(s): Constellation/RPLVault.sol & Constellation/WETHVault.sol

   In multiple locations across the codebase, specifically in `RPLVault` and `WETHVault`, there are unnecessary checks for non-negativity on `uint256` variables:

   - In `RPLVault`, line [**232**];
   - In `WETHVault`, line [**372**];
   - In `WETHVault`, line [**360**].

   Remove all instances of non-negativity checks on unsigned integers.

10. **Redundant Fee Validation In `setTreasuryFee()`**

    Related Asset(s): Constellation/WETHVault.sol

    There are two validation checks for the `_treasuryFee` parameter:

```
function setTreasuryFee(uint256 _treasuryFee) external onlyMediumTimelock {
    require(_treasuryFee <= 1e18, 'Fee too high');
    require(_treasuryFee + nodeOperatorFee <= 1e18, 'Total fees cannot exceed 100%');
    treasuryFee = _treasuryFee;
}
```

    The first `require()` check is redundant because the following `require()` check already covers this case and provides a more comprehensive validation.

    Remove the redundant check and update the function's documentation to reflect that it checks the combined fees.

11. **Misleading Comment For `maxWethRplRatio` In WETHVault**

    Related Asset(s): Constellation/WETHVault.sol

    There is a discrepancy between the comment and the actual value set for `maxWethRplRatio`:

```
maxWethRplRatio = 40e18; // 400% at start (4 ETH of xrETH for 1 ETH of xRPL)
```

    The comment suggests that the ratio is 400%, which would indeed be 4 `ETH` of `xrETH` for 1 `ETH` of `xRPL`. However, the actual value set (`40e18`) represents 4000%, not 400%. This means the correct interpretation is 40 `ETH` of `xrETH` for 1 `ETH` of `xRPL`.

    Update the comment to accurately reflect the implemented ratio.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows.

1. The `Errors` contract has been removed.
2. The unused imports have been removed.
3. The zero address check has been added as recommended.
4. A new local variable `mcsAddress` has been added to store the value of `getDirectory().getMerkleClaimStreamerAddress()`.
5. The development team has removed the redundant `@notice` tag.
6. `wethBalance` is used now in the calculation of `balanceEthAndWeth`.
7. Local variables for RPL Token and `rocketNodeStakingAddress` has been added as recommended.
8. Not all the typos has been addressed. Specifically in :
    - `SuperNodeAccount` line [**187**] : `depoist` should be `deposit`
    - `SuperNodeAccount` line [**350**] : `dmins` should be `Admins`
9. The unnecessary non-negativity check has been removed.
10. The redundant check on the fees has been removed.
11. The value `maxWethRplRatio` has been updated now to `4e18` to match the comment.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `brownie` framework was used to perform these tests and the output is given below.

```
Ran 2 tests for test/tests-fork/MerkleClaimStreamer.t.sol:MerkleClaimStreamerTest
[PASS] test_get_streamed_tvl_eth() (gas: 22930)
[PASS] test_initializeVault() (gas: 19201)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 772.56ms (2.99ms CPU time)

Ran 1 test for test/tests-fork/Directory.t.sol:DirectoryTest
[PASS] test_initialize() (gas: 30773)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 775.11ms (383.13µs CPU time)

Ran 1 test for test/tests-fork/PoAConstellationOracle.t.sol:PoAConstellationOracleTest
[PASS] test_initialize() (gas: 18099)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 777.03ms (259.49µs CPU time)

Ran 1 test for test/tests-fork/PriceFetcher.t.sol:PriceFetcherTest
[PASS] test_rpl_get_price() (gas: 61009)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 783.50ms (1.96ms CPU time)

Ran 5 tests for test/tests-fork/RPLVault.t.sol:RPLVaultForkTest
[PASS] test_deposit() (gas: 460079)
[PASS] test_initializeVault() (gas: 40357)
[PASS] test_mint() (gas: 461724)
[PASS] test_redeem() (gas: 762067)
[PASS] test_withdraw() (gas: 779472)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 803.18ms (51.20ms CPU time)

Ran 8 tests for test/tests-fork/WETHVault.t.sol:WETHVaultForkTest
[PASS] test_deposit() (gas: 569774)
[PASS] test_deposit_revert_insufficient_rpl_coverage() (gas: 821603)
[PASS] test_deposit_rplstaked_not_zero() (gas: 1046627)
[PASS] test_deposit_withdraw_edge_amounts() (gas: 928248)
[PASS] test_initializeVault() (gas: 32236)
[PASS] test_mint_discrepancy_with_deposit_vuln() (gas: 474665)
[PASS] test_redeem() (gas: 1007819)
[PASS] test_withdraw() (gas: 1054971)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 811.57ms (124.09ms CPU time)

Ran 7 tests for test/tests-fork/OperatorDistributor.t.sol:OperatorDistributorForkTest
[PASS] test_initialize() (gas: 26311)
[PASS] test_processMinipool() (gas: 2852700)
[PASS] test_processNextMinipool() (gas: 4958849)
[PASS] test_rebalanceRplStake_case_1() (gas: 470484)
[PASS] test_rebalanceRplStake_case_2() (gas: 470471)
[PASS] test_rebalanceRplVault() (gas: 301816)
[PASS] test_rebalanceWethVault() (gas: 226618)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 827.48ms (108.61ms CPU time)

Ran 3 tests for test/tests-fork/Whitelist.t.sol:WhitelistTest
[PASS] test_addOperator() (gas: 100116)
[PASS] test_initialize() (gas: 18099)
[PASS] test_removeOperator() (gas: 110925)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 862.31ms (17.64ms CPU time)

Ran 8 tests for test/tests-fork/SuperNodeAccount.t.sol:SuperNodeAccountForkTest
[PASS] test_closeDissolvedMinipool() (gas: 4850466)
[PASS] test_createMinipool() (gas: 2906077)
[PASS] test_createMinipool_two_pools() (gas: 4967259)
[PASS] test_initialize() (gas: 24244)
[PASS] test_lazyInitialize() (gas: 26561)
[PASS] test_removeMinipool_firstPool() (gas: 4824592)
[PASS] test_removeMinipool_secondPool() (gas: 4822548)
[PASS] test_stake() (gas: 2980558)
```

```
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 862.35ms (376.94ms CPU time)

Ran 9 test suites in 873.78ms (7.28s CPU time): 36 tests passed, 0 failed, 0 skipped (36 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

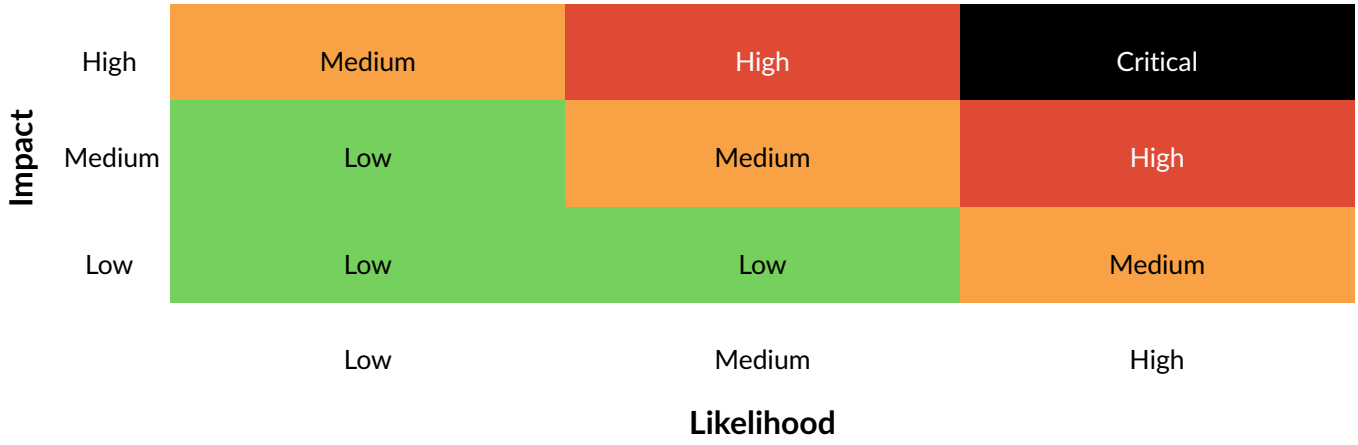

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].