Kelp DAO

rsETH Adapter

Security Assessment Report

Version: 2.0

November, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Kelp and August 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 in scope. 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 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*.

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

Overview

The `GainAdapter` contract is an adapter contract for converting the deposited assets into `rsETH` tokens. The contract is used as a central point for user interactions with multiple gain vaults.

The `GainLendingPool` contract represents a lending pool that is fully compliant with the ERC-4626 standard. This contract receives assets from the `GainAdapter` contract in the form of `rsETH` tokens.

Security Assessment Summary

Scope

The assessment consists of two parts. The first part of the review was conducted on the files hosted on the Kelp DAO repository. While the second part of the review was conducted on the Fractal Protocol repository.

The scope of this time-boxed review was strictly limited to files at commit 685ac41 for the first part and commit ed67adc for the second part.

The retesting of the fixes was conducted at commit 0a3f37c for the first part and commit 9a3d0ed for the second part.

The following files were considered in scope:

1. Part 1
    - `GainAdapter.sol`
    - `WeETHPriceOracle.sol`
    - `WstETHPriceOracle.sol`
    - `METHPriceOracle.sol`
    - `RETHPriceOracle.sol`
2. Part 2
    - `GainLendingPool.sol`

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 13 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Medium: 2 issues.
- Low: 3 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Kelp DAO 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
KRSA-01	Incorrect Shares Calculation Due To Inaccurate Computation of Total Assets	Critical	Resolved
KRSA-02	Failed Deposit When `GainLendingPool` Has No `rsETH` And Non-zero Total Supply	High	Resolved
KRSA-03	Price Changes Result In Stuck Tokens	High	Closed
KRSA-04	Incorrect Modifier Being Used To Pause Deposits	Medium	Resolved
KRSA-05	No Refunding Of Excess `msg.value` During Swap	Medium	Resolved
KRSA-06	Lack Of Access Control On `setGainAdapter()`	Low	Resolved
KRSA-07	Inaccessibility Of `setGainAdapter()`	Low	Closed
KRSA-08	Inaccuracy On `totalAssets()` Calculation	Low	Resolved
KRSA-09	Vault Whitelist Removal Can Be Disrupted	Informational	Resolved
KRSA-10	Uninitialised Implementation Contract	Informational	Resolved
KRSA-11	`depositAsset()` May Be Called With ETH	Informational	Resolved
KRSA-12	Redundant Operation In Fetching `minRSETHAmountExpected`	Informational	Resolved
KRSA-13	Miscellaneous General Comments	Informational	Resolved

KRSA-01	Incorrect Shares Calculation Due To Inaccurate Computation of Total Assets		
Asset	`GainLendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The conversions between shares and assets do not account for assets held externally to `GainLendingPool` when calculating total assets. The issue applies to any conversion between shares and assets including deposits and withdrawals.

The following uses the deposit case as an example. When a user makes a deposit, they are minted shares based on the ratio of the amount of assets deposited to the total assets within the system. However, the `previewDeposit()` function, which computes the user's shares, calls `_getTotalAssets()` which only includes the amount of assets currently stored within the vault.

The function `previewDeposit()` will call `_convertToShares()`, where the total assets is set as `_getTotalAssets()`.

```
function previewDeposit(uint256 assets) public view virtual override returns (uint256) {
    return _convertToShares(assets, MathUpgradeable.Rounding.Down);
}
```

```
function _convertToShares(uint256 assets, MathUpgradeable.Rounding rounding) internal view virtual returns (uint256) {
    return (assets == 0 || _totalSupply == 0) ? _initialConvertToShares(assets, rounding) : assets.mulDiv(_totalSupply,
        ↪  _getTotalAssets(), rounding);
}
```

However, `_getTotalAssets()` does not include all assets held by the vault. The correct accounting for total assets can be seen in the function `totalAssets()`, where assets in the gain adapter are included.

```
function totalAssets() external view override returns (uint256) {
    uint256 ethReservedAmount = gainAdapter.getEthReservedAmount(address(this));
    uint256 rsETHReservedAmount = gainAdapter.getRsETHValueFromETHAmount(ethReservedAmount);
    uint256 internalTotalAssets = _getTotalAssets();

    return internalTotalAssets + rsETHReservedAmount;
}
```

The impact is that the assets belonging to the vault currently being held by the adapter are not included in the share calculations. This results in users receiving more shares per asset than they should upon deposits.

Recommendations

The issue may be resolved by overriding `_getTotalAssets()` to include the assets being held by `GainAdapter` for the vault.

Modifications will also need to be made to `totalAssets()` to account for this.

Resolution

This issue has been solved in commit 8db0fd3. The function `_getTotalAssets()` is overridden on `GainLendingPool` contract to include the assets held by the `GainAdapter` for the vault.

KRSA-02	Failed Deposit When `GainLendingPool` Has No `rsETH` And Non-zero Total Supply		
Asset	`GainLendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A deposit transaction fails under a certain condition.

A deposit transaction involves a user calling either `GainAdapter.depositETH()` or `GainAdapter.depositAssets()` while transferring the ownership of the deposited assets from the user to the `GainAdapter` contract. The called function then calls `GainLendingPool.reserveDeposit()`.

The testing team observed the following edge-case on the `GainLendingPool` contract (referred to as the vault contract) when executing a deposit transaction.

1. The vault contract does not hold any `rsETH` (here, `rsETH` is the supported asset).
2. The vault contract has successfully accepted a deposit and therefore the liquidity pool token's total supply is non-zero.

If the described condition above holds, the deposit transaction reverts. This occurs because the function `BaseUpgradeableERC4626._getTotalAssets()` that is called when executing `previewDeposit()` returns zero. The zero return value is due to the exclusion of the assets still held by the `GainAdapter` contract as the caller. This chain of executions yields a `division by zero operation` in function `BaseUpgradeableERC4626._convertToShares()`.

Consider the following example.

1. Original condition: the vault does not hold any `rsETH`.
2. A deposit transaction `TX1` is received. The `TX1` is successfully executed and the caller receives the newly minted liquidity token. Here, the liquidity token's total supply is non-zero. The deposited asset is held by the `GainAdapter` contract.
3. A deposit transaction `TX2` is received which will revert due to a division by zero.

Recommendations

Override the function `_getTotalAssets()` to include the assets held by the `GainAdapter` contract.

Resolution

This issue has been solved in commit 8db0fd3. The function `_getTotalAssets()` is overridden on `GainLendingPool` contract to include the assets held by the `GainAdapter` for the vault.

KRSA-03	Price Changes Result In Stuck Tokens		
Asset	`GainLendingPool.sol, GainAdapter.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

A price increase of either `rsETH` or the underlying asset prevent tokens being transferred to the `GainAdapter`.

The deposit process has three steps.

1. User: `depositETH()` / `depositAsset()`
2. Operator: `mintRsETH()`
3. Operator: `sendRsETHToVault()`

When `depositETH()` or `depositAsset()` is called the current value of the incoming asset is recorded in ETH and added to `ethReserved[vault]`. However, there is a delay before the operator calls `mintRsETH()` and then `sendRsETHToVault()`.

If there is a price increase of `rsETH` between `mintRsETH()` and `sendRsETHToVault()`, then the ETH value of the `rsETH` will increase above the amount registered in the deposit. As a result, if all the `rsETH` is transferred in `sendRsETHToVault()` then the following line will cause a revert.

```
if (ethReserved[vault] < ethValueOfRsETHAmount) {
    revert NotEnoughETHReserved();
}
```

Consider the following example.

1. `depositETH()` of 100 ETH such that `ethReserved[vault] = 100`
2. `mintRsETH()` at a price of 9:10 such that 90 rsETH tokens are minted
3. rsETH price increases about 1%
4. `sendRsETHToVault()` for 90 rsETH tokens will revert as the current value is 101 ETH which is larger than `ethReserved[vault]`.

The result of this example is that about 1 rsETH must be left in the `GainAdapter` contract and cannot be transferred to `GainLendingPool`.

A similar principle can be applied to price changes of assets relative to rsETH if `depositAsset()` is called with an alternate token.

The impact is that a portion of the rsETH tokens will not be accounted for by any vault and stuck in the contract.

Recommendations

One solution is to perform the deposit, mint and send steps together. That is, expand the logic of `depositETH()` and `depositAsset()` to perform minting and sending operations.

An alternate solution is to combine the deposit and mint steps. If the assets are converted to rsETH during the deposit functions then the rsETH amount can be stored instead of `ethReserved[vault]`. This would allow the exact token amounts to be transferred, to match the stored values.

Resolution

The development team acknowledged the issue with the following statement.

While excess ETH or other asset accumulation in the adapter contract is acknowledged, this design choice ensures protocol stability and does not impact user funds. The protocol guarantees that the ETH equivalent of user deposits will always be honored, regardless of the asset type or price fluctuations.

KRSA-04	Incorrect Modifier Being Used To Pause Deposits		
Asset	`GainLendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `reserveDeposit()` function uses a modifier to allow deposit reservations only if deposits are not currently paused. However, the function currently implements the `ifWithdrawalsNotPaused` modifier which is intended to prevent the function from executing when withdrawals have been paused. This means that if deposits are paused by setting `depositsPaused == true`, deposit reservations will still be allowed.

```
function reserveDeposit(address account, uint256 amountInETH) external nonReentrant ifConfigured ifWithdrawalsNotPaused {
```

Recommendations

Replace the `ifWithdrawalsNotPaused` modifier with `ifDepositsNotPaused`.

Resolution

This issue has been addressed in commit 9a3d0ed. The `reserveDeposit()` function has been updated by replacing the `ifWithdrawalsNotPaused` modifier with the `ifDepositsNotPaused` modifier.

KRSA-05	No Refunding Of Excess `msg.value` During Swap		
Asset	`GainAdapter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When calling `swapAssetForETH()` it is possible that the operator may send `msg.value > assetValueInETH` to account for any price fluctuations during the transaction. However, in this event the excess amount is never refunded to the operator and is locked within the contract.

Recommendations

Add a check to see whether `msg.value > assetValueInETH` and if that is the case refund the excess amount to the operator.

Resolution

The issue has been fixed on commit e5d30e9. The excess ETH is now returned to the caller.

KRSA-06	Lack Of Access Control On `setGainAdapter()`		
Asset	`GainLendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `setGainAdapter()` does not have any access control. This will allow anyone to call this function and set `_gainAdapter` to a malicious address.

The issue is rated low likelihood as the function may only be called once and should be done right after deployment.

Recommendations

Consider adding the `onlyOwner()` modifier to ensure that only `_owner` is able to set this value.

Resolution

This issue has been addressed in commit 9a3d0ed. The `onlyOwner()` modifier has been added to the `setGainAdapter()` function.

KRSA-07	Inaccessibility Of `setGainAdapter()`		
Asset	`GainLendingPool.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The modifier `ifNotConfigured` in function `setGainAdapter()` may prevent the function to be called properly.

This happens if `configurePool()` is called before `setGainAdapter()`, then the function `setGainAdapter()` can no longer be called. Therefore, `reserveDeposit()` reverts with `Unauthorized()`.

Recommendations

Consider removing the modifier `ifNotConfigured`. Alternatively, make sure the function `setGainAdapter()` is called before the function `configurePool()` in the deployment script.

Resolution

The development team acknowledged the issue with the following statement.

This is intentional as we only want the gainAdapter to be set one time. The deployment script is set up so that `setGainAdapter()` is called before `configurePool()`.

KRSA-08	Inaccuracy On `totalAssets()` Calculation		
Asset	`GainLendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The function `GainLendingPool.totalAssets()` takes into account the potentially stale value `GainAdapter.ethReservedAmount`.

`GainAdapter.ethReservedAmount[vault]` is the recorded value of all assets currently held by the `GainAdapter` for a specific vault. It includes the reserved deposits for the `GainLendingPool`.

Each asset is priced in ETH at the time of deposit. Hence, there may be discrepancies between the recorded amount and the current value. Therefore, the return value of `totalAssets()` is not accounting for price fluctuations of underlying assets.

Recommendations

The recommendation of KRSA-01 may be used to resolve this issue.

Alternatively, to get a more accurate result, consider recording separate amounts for each asset and recalculating the current value of each asset class in the `GainAdapter`. This will significantly increase gas consumption.

Resolution

This issue has been solved in commit 8db0fd3. The function `_getTotalAssets()` is overridden on `GainLendingPool` contract to include the assets held by the `GainAdapter` for the vault.

KRSA-09	Vault Whitelist Removal Can Be Disrupted
Asset	`GainAdapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

A malicious actor could repeatedly send a dust amount to satisfy the following condition: `ethReserved[vault] != 0`. This will revert the call to function `removeWhitelistedVault()` to remove the vault from the whitelist because of the check on lines [**432-434**]:

```
if (ethReserved[vault] != 0) {
    revert EthReservedIsNotZero();
}
```

Aditionally, challenges may occur in forcing the value of `ethReserved[vault]` to be zero. That is because `sendRsETHToVault()` takes in an amount of rsETH and converts it to ETH value. Due to the scaling of shares it would need to use the price at a specific block to predict exactly the number of rsETH share to transfer to the vault.

The issue is raised as informational severity as the pauser may pause deposits on the vault contract to prevent this.

Recommendations

Consider adding a minimum amount to deposit to prevent dust amount.

Resolution

The issue has been fixed on commit e87811c. A customisable minimum amount is now enforced to prevent dust amount deposits.

KRSA-10	Uninitialised Implementation Contract
Asset	`GainAdapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Openzeppelin recommends locking implementation contracts upon deployment to prevent them from being taken over by an attacker.

Recommendations

The following should be added to ensure that a deployed contract is not left unlocked:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Resolution

The issue has been fixed on commit fe0b397. The recommended action is implemented.

KRSA-11	`depositAsset()` May Be Called With ETH
Asset	`GainAdapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

There are no restrictions to stop `depositAsset()` from being called with the asset as `ETH_IDENTIFIER`.

Were a user to call `depositAsset()` with `ETH_IDENTIFIER` a revert would occur. That is due to `safeTransfer()`, which performs checks to ensure the token address has code.

Since a revert occurs the issue is raised as informational.

Recommendations

Consider adding a check to `depositAsset()` to ensure the asset is not `ETH_IDENTIFIER`.

Resolution

The issue has been fixed on commit af21266. An additional check is now added in function `depositAsset()` to prevent the issue from happening.

KRSA-12	Redundant Operation In Fetching `minRSETHAmountExpected`
Asset	`GainAdapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

On line [323], the `rsETHAmountToMint` becomes `minRSETHAmountExpected` in `LRTDepositPool.depositAsset()`.

```
lrtDepositPool.depositAsset(asset, depositAmount, rsETHAmountToMint, "");
```

Here is the definition of function `LRTDepositPool.depositAsset()` taken from `ILRTDepositPool.sol`.

```
function depositAsset(
        address asset,
        uint256 depositAmount,
        uint256 minRSETHAmountExpected,
        string calldata referralId
    )
        external;
```

The function `LRTDepositPool.depositAsset()` compares the value of `minRSETHAmountExpected` and the internally-calculated `rsethAmountToMint` in function `LRTDepositPool._beforeDeposit()`.

A value check can be found on lines [561-563] of `LRTDepositPool.sol` as shown below:

```
if (rsethAmountToMint < minRSETHAmountExpected) {
    revert MinimumAmountToReceiveNotMet();
}
```

Note that `rsethAmountToMint` is calculated by calling `LRTDepositPool.getRsETHAmountToMint()` on line [559] of `LRTDepositPool.sol` (as a part of the operation of `LRTDepositPool.depositETH()` function):

```
rsethAmountToMint = getRsETHAmountToMint(asset, depositAmount);
```

The code above is identical to the code on line [304] of `GainAdapter` contract.

```
uint256 rsETHAmountToMint = lrtDepositPool.getRsETHAmountToMint(asset, depositAmount);
```

Hence, the following condition always holds:
`rsETHAmountToMint == minRSETHAmountExpected`, because both values are retrieved using the identical function of `LRTDepositPool.getRsETHAmountToMint()` with identical input parameters. Therefore, it is somewhat redundant.

Recommendations

Consider refactoring the function `mintRsETH()` such that the expected amount to mint becomes a user input. Alternatively, simply use zero as `rsETHAmountToMint` on lines [315, 323] of `GainAdapter.sol` to make sure the check on lines [561-563] of `LRTDepositPool.sol` passes.

Resolution

The issue has been fixed on commit 5e6f061. The `minRsETHAmountExpected` is now an input value.

KRSA-13	Miscellaneous General Comments
Asset	All contracts
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. **No Check For `Address(0)`**
   Related Asset(s): LendingPool.sol
   In `configurePool()` there are no checks for `address(0)` on `newScheduledCallerAddress` and `newUnderlyingAsset` when passing these values. As their respective variables (`scheduledCallerAddress` and `_underlyingAsset`) do not have setter functions to update their value later, if they are accidentally set to `address(0)` this will deem the pool unusable.
   Consider adding a check for `address(0)` for both values.

2. **Unnecessary Check For `msg.sender` in `isBlacklisted`**
   Related Asset(s): GainLendingPool.sol
   In `processWithdrawal()` the following check is made:

   ```
   if (isBlacklisted[msg.sender] || isBlacklisted[account]) {
       revert Errors.Blacklisted();
   }
   ```

   However, since the only allowed caller of this function is the `gainAdapter` the check for `isBlacklisted[msg.sender]` will never return `true` .
   Consider removing the check for `isBlacklisted[msg.sender]` .

3. **Unnecessary Check On `msg.sender`**
   Related Asset(s): GainLendingPool.sol
   In `processWithdrawal()` the following check is made:

   ```
   if (msg.sender != address(gainAdapter) || msg.sender == account) {
       revert Errors.Unauthorized();
   }
   ```

   However, the second check of `msg.sender == account` is unnecessary as only the `gainAdapter` is allowed to call this function. This makes the first check sufficient as any address other than the `gainAdapter` will be denied. Also, since the `gainAdapter` does not maintain an account with the `vault` , the second condition will never return `true` .
   Consider removing the second check for `msg.sender == account` in the `if` statement.

4. **Inaccurate Information Emitted By `WithdrawalRequested` Event**
   Related Asset(s): GainLendingPool.sol
   The event `WithdrawalRequested` as defined in the `TimeLockedERC4626` contract specifies that the first argument is `ownerAddress` , while the emitting code on line [**75**] of `GainLendingPool` contract specifies `msg.sender` as the

first argument. Since `msg.sender` is the `GainAdapter` contract, this means that this argument does not reflect the actual owner of the asset.

Consider replacing `msg.sender` with `account` to increase information accuracy.

5. **Typo**
   Related Asset(s): GainLendingPool.sol
   The word `fors` on line [**40**] may be a typo. Replace with `for`.
6. **Access Control Implementation Could Utilise A Modifier**
   Related Asset(s): GainLendingPool.sol
   The access control code on lines [**45-47**] could be implemented using a modifier for simplicity and readability.
7. **Potential Zero Value For `ethValueOfRsETHAmount` And `assetValueInETH`**
   Related Asset(s): GainAdapter.sol
   In `sendRsETHToVault()` on line [**264**] there is no check to ensure the value returned for `ethValueOfRsETHAmount` is not zero. This is also the case in `swapAssetForETH()` on line [**377**] for `assetValueInETH`.
   Add a check to ensure both values are none zero similar to what is being done for `assetValueInETH` on line [**201**].
8. **Multiple `requestIds`**
   Related Asset(s): GainAdapter.sol
   Lido's `claimWIthdrawalsTo()` allows for multiple `requestIds` (as evident from `UnstakeStETH._claimStEth()`) but `claimStEth()` only allows for a single `requestId` instead.
   So if there are multiple `requestIds` (from `unstakeStEth()` -> `UnstakeStETH._unstakeStEth()`) to process, then the operator needs to call `claimStEth()` multiple times, which can be inconvenient.
   Consider allowing multiple `requestIds` to allow asset claiming in one transaction.
9. **Approval Operation**
   Related Asset(s): GainAdapter.sol
   Consider replacing the `approve()` function on line [**321**] with `safeApprove()`.

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. No fix.
2. Fixed on commit 9a3d0ed.
3. Fixed on commit 9a3d0ed.
4. Fixed on commit 9a3d0ed.
5. Fixed on commit 9a3d0ed.
6. No fix.

7. Fixed on commit e760873.
8. Fixed on commit e760873.
9. Fixed on commit e760873.

Appendix A 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.

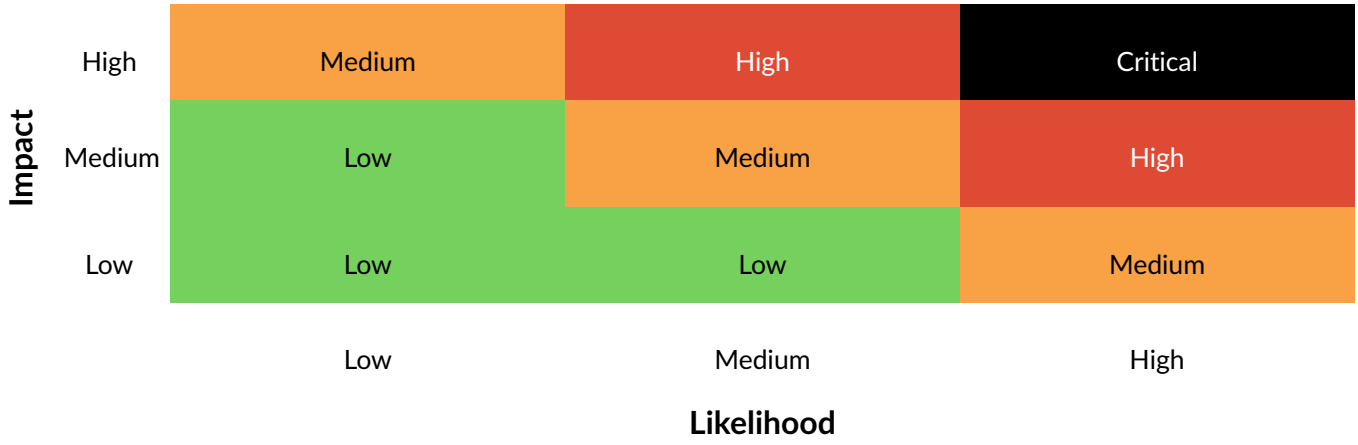

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].