Omni

rlUSD Bridge Contract

Security Assessment Report

Version: 2.0

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Omni 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 Omni 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 Omni smart contracts in scope.

Overview

This engagement focuses on the rlUSD bridge contracts, allowing users to easily bridge their rlUSD tokens across different chains. The bridge features two main components: the `Bridge` contract itself containing all the cross-chain logic and an optional `Lockbox` that can be used to wrap the rlUSD tokens before bridging.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to files at commit 8855575.

The fixes of the identified issues were assessed at commit 6872365.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

For the Solidity components, 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 the Solidity components of the 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 12 issues during this assessment. Categorised by their severity:

- Medium: 2 issues.
- Low: 1 issue.
- Informational: 9 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Omni 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
OMRL-01	Paused Token Leads to Loss of Funds	Medium	Resolved
OMRL-02	Insufficient Gas For `receiveToken()`	Medium	Resolved
OMRL-03	Risks Of Source And Destination Chain Configuration Mismatch	Low	Closed
OMRL-04	Centralisation Risk For Bridge Address Configuration	Informational	Resolved
OMRL-05	Use A Caller-Determined Refund Recipient	Informational	Resolved
OMRL-06	No Mechanism For Deleting Routes	Informational	Resolved
OMRL-07	Missing Check For Routes	Informational	Resolved
OMRL-08	No Support For Rebasing Or Fee Tokens	Informational	Closed
OMRL-09	Unnecessary Approval For Lockbox	Informational	Resolved
OMRL-10	New EVM Version May Be Unsupported	Informational	Closed
OMRL-11	Pausing and Unpausing Have The Same Role	Informational	Resolved
OMRL-12	Optimising Redundant Condition Checks For `lockbox_`	Informational	Resolved

OMRL-01	Paused Token Leads to Loss of Funds		
Asset	`Bridge.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

If the minting or transferring functionality of the wrapped token is paused, the bridging process may fail, resulting in locked funds. The issue occurs during each call of `mint()` and `transfer()`.

During bridging, the internal function `_receiveToken()` is called to finalise the withdrawal. If a lockbox is present, it first mints the wrapped token to the `Bridge` contract and then calls `withdrawTo()` in the lockbox:

Bridge.sol

```
269 if (_lockbox == address(0)) {
        // If no lockbox is set, just mint the wrapped tokens to the recipient.
271     ITokenOps(_token).mint(to, value); // @audit reverts if token is paused
    } else {
273     // If a lockbox is set, mint the wrapped tokens to the bridge contract.
        ITokenOps(_token).mint(address(this), value); // @audit reverts if token is paused
275
        // Attempt withdrawal from the lockbox, but transfer the wrapped tokens to the recipient if it fails.
277     try ILockbox(_lockbox).withdrawTo(to, value) { } // @audit triggers catch if token is paused
        catch {
279         _token.safeTransfer(to, value); // @audit reverts if token is paused
            emit LockboxWithdrawalFailed(_lockbox, to, value);
281     }
    }
```

When `receiveToken()` is called on the destination chain, it internally invokes `_receiveToken()` to mint wrapped tokens. The tokens are either sent directly to the recipient or minted to the `Bridge` contract, depending on whether a lockbox is present.

However, if the minting functionality in the wrapped token contract is paused, as enforced by `_update()` in `StablecoinUpgradeable.sol` then the transaction will revert at lines [**271**] or [**274**] in the `Bridge` contract.

StablecoinUpgradeable.sol

```
162 function _update(address from, address to, uint256 value)
        internal override(ERC20Upgradeable, ERC20PausableUpgradeable)
164     whenAccountNotPaused(from)
        whenAccountNotPaused(to)
166     whenAccountNotPaused(_msgSender())
    {
168     ERC20PausableUpgradeable._update(from, to, value);
    }
```

Source: Ripple RLUSD-Implementation

Furthermore, inside the `Lockbox`, the `withdrawTo()` function invokes the internal `_withdraw()` function. This function first burns the wrapped token from the `Bridge` using `clawback()` and then transfers the underlying token to the user:

Lockbox.sol

```
143 function _withdraw(address from, address to, uint256 value) internal {
        ITokenOps(wrapped).clawback(from, value);
145     token.safeTransfer(to, value);
    }
```

If burning is paused in the wrapped token contract, the `clawback()` function will revert:

StablecoinUpgradeable.sol

```
162 function _update(address from, address to, uint256 value)
        internal override(ERC20Upgradeable, ERC20PausableUpgradeable)
164     whenAccountNotPaused(from)
        whenAccountNotPaused(to)
166     whenAccountNotPaused(_msgSender())
    {
168     ERC20PausableUpgradeable._update(from, to, value);
    }
```

Source: Ripple RLUSD-Implementation

As a result, the `catch` branch in `_receiveToken()` will be executed to directly transfer the wrapped token to the user. However, if transferring the wrapped token to `to` is also paused, it will revert again, causing the withdrawal to fail. Since the bridging transaction in `OmniPortal` is non-retryable, this failure results in locked funds.

Due to the non-retriable design of `OmniPortal`, this revert does *not* roll back the entire `xsubmit()` transaction in `OmniPortal`. As a result, the withdrawal process fails, and the locked tokens become unrecoverable, while the bridging is considered as successful.

Recommendations

A solution for the calls to `mint()` reverting is to handle such cases using a try-catch mechanism and implement a retrieval process, allowing users to claim their tokens later.

```
if (_lockbox == address(0)) {
    // If no lockbox is set, just mint the wrapped tokens to the recipient.
    try ITokenOps(_token).mint(to, value) {}
    catch {
       failed[to] += value;
    }
} else {
    // If a lockbox is set, mint the wrapped tokens to the bridge contract.
    try ITokenOps(_token).mint(address(this), value) {}
    catch {
       failed[to] += value;
       return;
    }
```

Additionally, for the `transfer()` calls, should be tracked when a fail occurs across all branches and implement a retry mechanism, allowing users to claim their funds later. However, consideration must be taken for non-standard ERC20 token implementations as they may not have `returndata` or return `false` rather than reverting if a transfer fails.

These approaches ensure that failed withdrawals are properly tracked, allowing users to later reclaim their tokens when the issue is resolved.

Resolution

This issue was resolved in commits 205bbad and 6872365 by tracking failed withdrawals and allowing users to claim them later.

OMRL-02	Insufficient Gas For `receiveToken()`		
Asset	`Bridge.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When sending tokens to a bridge with a lockbox, `180_000` gas is given to the `receiveToken()` function. However, this is insufficient in certain scenarios. Specifically, this is insufficient in cases where the `try` statement in `_receiveToken()` fails at the last possible moment and the `catch`-body has to be executed even though a lot of gas has already been consumed.

For example, when the test `test_receiveToken_succeeds_insolvent_lockbox` is modified to use a `180_000` gas stipend, it will revert since there was not enough gas remaining to execute the `catch`-body. A similar scenario where gas may run out is if the underlying token has paused transfers.

In case `receiveToken()` runs out of gas, the cross chain call fails and can not be retried, meaning that the tokens for this transfer are lost.

Bridge.sol

```
264 function _receiveToken(address to, uint256 value) internal {
        // Cache addresses for gas optimization.
266     address _token = token;
        address _lockbox = lockbox;
268
        if (_lockbox == address(0)) {
270         // If no lockbox is set, just mint the wrapped tokens to the recipient.
            ITokenOps(_token).mint(to, value);
272     } else {
            // If a lockbox is set, mint the wrapped tokens to the bridge contract.
274         ITokenOps(_token).mint(address(this), value);

276         // Attempt withdrawal from the lockbox, but transfer the wrapped tokens to the recipient if it fails.
            try ILockbox(_lockbox).withdrawTo(to, value) { }
278         catch {
                _token.safeTransfer(to, value);
280             emit LockboxWithdrawalFailed(_lockbox, to, value);
            }
282     }

284     emit TokenReceived(xmsg.sourceChainId, to, value);
    }
```

Recommendations

Consider increasing the gas stipend. Keep in mind that different token implementations will consume different amounts of gas. Additionally, different chains may implement different gas pricing for opcodes, and future chain upgrades may also change opcode pricing. As such, it may be useful to implement mechanisms that allow an admin to change the gas stipends.

Resolution

This issue has been resolved in PR #3074 by increasing default gas limits and allowing them to be set during deployment.

OMRL-03	Risks Of Source And Destination Chain Configuration Mismatch		
Asset	`Bridge.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is a concern that the settings on the source chain may not be properly synchronised with the configurations of the destination chain. This misalignment can lead to two potential issues:

1. **Incorrect Lockbox Absence on the Source Chain:**
   - If the source chain is incorrectly configured to assume that the destination chain does *not* have a lockbox, while in reality, a lockbox exists on the destination chain, issues arise.
   - When a user bridges tokens, the `RECEIVE_DEFAULT_GAS_LIMIT` is forwarded to the Bridge contract on the destination chain.
   - Since the destination chain does have a lockbox, the allocated gas may be insufficient, causing the transaction to revert.
   - Due to the Omni implementation, a failed bridge transaction is not retriable, potentially resulting in a loss of funds.
2. **Incorrect Lockbox Presence on the Source Chain:**
   - If the source chain is wrongly configured to assume that the destination chain *has* a lockbox when it actually does not (or if it is paused), another issue occurs.
   - When a user bridges tokens, they are charged `RECEIVE_LOCKBOX_GAS_LIMIT` on the source chain, assuming that a lockbox exists on the destination chain.
   - On the destination chain, due to the absence or paused state of the lockbox, the user receives wrapped tokens instead of the expected underlying tokens.
   - This results in the user being overcharged—they paid a higher gas fee expecting to receive the underlying token but ended up with wrapped tokens instead.

Additionally, the second scenario can occur even *without* an administrative misconfiguration. If the lockbox on the destination chain is paused but the source chain settings are not updated accordingly, any token bridged during this period would lead to the user being overcharged.

Recommendations

One possible solution is to ensure that the stipulated gas is sufficient to cover both scenarios, whether a lockbox is present or not.

Resolution

The development team has closed the issue with the following comment:

> *"OMRL-03 is not addressed as this is dependent on the bridge operator misconfiguring it. While a valid risk, we would rather preserve the gas efficiency of targeted gas values. Operators who do not require this efficiency can simply set an equal default value for calls to a bridge with or without a lockbox and opt into avoiding the misconfiguration issue entirely, so this was partially corrected in [PR #3074]."*

OMRL-04	Centralisation Risk For Bridge Address Configuration
Asset	`Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

There is a centralisation risk where a compromised admin could set the bridge address of a remote chain to a malicious address under their control:

```
routes[a remote chain].bridge = malicious address
```

This allows the malicious address to invoke `xcall()` on the remote chain, which then calls the bridge contract on the current chain. Since the bridge address for the remote chain is now set to a malicious address, the following check would pass:

Bridge.sol

```
62  modifier onlyBridge() {
        if (msg.sender == address(omni)) {
64          if (_routes[xmsg.sourceChainId].bridge != xmsg.sender) revert Unauthorized(xmsg.sourceChainId, xmsg.sender);
        } else {
66          revert Unauthorized(uint64(block.chainid), msg.sender);
        }
68      _;
    }
```

As a result, the malicious address could receive the underlying token (or its wrapped version) without burning or locking any tokens on the source chain.

Recommendations

To mitigate this, a safer approach would be to implement a two-step verification process for setting routes. In this setup, the admin role would only be able to request route configuration changes, while a separate role would be responsible for executing the request.

Resolution

This issue has been addressed in commits 205bbad and 6872365 by implementing a two-step process for configuring routes.

OMRL-05	Use A Caller-Determined Refund Recipient
Asset	`Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When sending a token, instead of directly refunding the extra fee to `msg.sender`, it may be better to allow the caller to specify a separate refund recipient address. This approach would be particularly beneficial when another DeFi protocol interacts with the `Bridge`, as direct refunds to the protocol itself could complicate accounting, especially when multiple users interact with it simultaneously.

Bridge.sol

```
246 function _omniTransfer(uint64 destChainId, address to, uint256 value) internal {
        Route memory route = _routes[destChainId];
248     bytes memory data = abi.encodeCall(Bridge.receiveToken, (to, value));
        uint256 fee = xcall(
250         destChainId, route.bridge, data, route.hasLockbox ? RECEIVE_LOCKBOX_GAS_LIMIT : RECEIVE_DEFAULT_GAS_LIMIT
        );
252
        if (msg.value < fee) revert InsufficientPayment();
254     if (msg.value > fee) msg.sender.safeTransferETH(msg.value - fee);

256     emit TokenSent(destChainId, msg.sender, to, value);
    }
```

Having a separate refund recipient would also align more closely with the design of Arbitrum and Optimism, where refunds are sent to a designated address. This is especially relevant when transactions are triggered from another chain by a contract. In such cases, aliasing occurs, meaning the refund would currently be sent to the aliased address.

Recommendations

Ensure the above comments are understood and consider changes if desired.

Resolution

This issue has been fixed in PR #3096 by allowing a separate refund recipient to be set.

OMRL-06	No Mechanism For Deleting Routes
Asset	`Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the `setRoutes()` function, there is a check on line [177] to prevent `route.bridge` being set to `address(0)`. This effectively prevents routes from being deleted. However, being able to delete a network may be useful when a chain is being deprecated.

Bridge.sol

```
174 function setRoutes(uint64[] calldata chainIds, Route[] calldata routes) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (chainIds.length != routes.length) revert ArrayLengthMismatch();
176     for (uint256 i = 0; i < chainIds.length; i++) {
            if (routes[i].bridge == address(0)) revert ZeroAddress();
178         _routes[chainIds[i]] = routes[i];
            emit RouteConfigured(chainIds[i], routes[i].bridge, routes[i].hasLockbox);
180     }
    }
```

Consider implementing a mechanism to delete routes and deprecate chains. When doing so, ensure tokens in transit are not lost, i.e., simply setting `route.bridge` to zero would cause any incoming transfers from that chain to be lost.

Recommendations

Ensure the above comments are understood and consider changes if desired.

Resolution

This issue has been fixed in PR #3096 by allowing routes to be set to zero.

OMRL-07	Missing Check For Routes
Asset	`Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the `getRoute()` function, there is no check that the `route` for a given `destChainId` exists. As a result `(address(0), false)` will be returned instead of reverting when `_routes[destChainId]` does not exist.

Bridge.sol

```
121 function getRoute(uint64 destChainId) external view returns (address bridge, bool hasLockbox) {
        Route memory route = _routes[destChainId];
123     return (route.bridge, route.hasLockbox);
    }
```

The same issue is present in the `bridgeFee()` function, where the fee for `RECEIVE_DEFAULT_GAS_LIMIT` is returned by default for a non-existent route.

Bridge.sol

```
131 function bridgeFee(uint64 destChainId) external view returns (uint256 fee) {
        Route memory route = _routes[destChainId];
133     return feeFor(
            destChainId,
135         abi.encodeCall(Bridge.receiveToken, (TypeMax.Address, TypeMax.Uint256)),
            route.hasLockbox ? RECEIVE_LOCKBOX_GAS_LIMIT : RECEIVE_DEFAULT_GAS_LIMIT
137     );
    }
```

Recommendations

Ensure the above comments are understood and consider adding these checks if desired.

Resolution

This issue has been fixed in PR #3096 by implementing the missing checks.

OMRL-08	No Support For Rebasing Or Fee Tokens
Asset	`Bridge.sol, Lockbox.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The bridge does not support fee-on-transfer or rebasing tokens. For example, in case of a fee-on-transfer token, the below snippet would mint `value` amount of wrapped tokens even though the `Lockbox` only received `value - fee` tokens.

Lockbox.sol

```
132 function _deposit(address from, address to, uint256 value) internal {
        token.safeTransferFrom(from, address(this), value);
134     ITokenOps(wrapped).mint(to, value);
    }
```

Seeing as the development team does not intend on using these types of tokens, there is no direct impact and this issue is rated as informational.

Recommendations

Ensure the above comments are understood.

Resolution

The development team has closed the issue with the following comment:

> *"OMRL-08 is not addressed as we are not concerned about rebasing or fee-on-transfer tokens."*

OMRL-09	Unnecessary Approval For Lockbox
Asset	`Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The approval on line [**107**] in `Bridge.sol` is not required and can be removed. In general, lingering approvals are discouraged as they may be exploited in case of a compromise.

This approval *would* be needed if the wrapper token's `clawback()` implementation required an allowance to be spent. However, for the current implementation in StablecoinUpgradeable that is not the case. As such, the approval is currently not needed.

Bridge.sol

```
function initialize(address admin_, address pauser_, address omni_, address token_, address lockbox_)
    external
    initializer
{
    //...

    // Give lockbox relevant approvals to handle deposits and withdrawals if necessary.
    if (lockbox_ != address(0)) {
        ILockbox(lockbox_).token().safeApproveWithRetry(lockbox_, type(uint256).max);
        token_.safeApproveWithRetry(lockbox_, type(uint256).max); //@audit this approval is not required
    }
}
```

Recommendations

Ensure the above comments are understood and consider removing the approval.

Resolution

This issue has been fixed in PR #3096 by removing the approval.

OMRL-10	New EVM Version May Be Unsupported
Asset	`*.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Solidity version `0.8.26` is used, this means that opcodes such as `PUSH0` and `MCOPY` will be used in the resulting bytecode. If the target network does not support these opcodes, it will result in reverts.

Seeing as the development team plans to deploy on Ethereum mainnet, Optimism, Base and Arbitrum—which all support these opcodes—this is not an issue. However, care must be taken if these contracts are deployed on other networks in the future.

Recommendations

Ensure the above comments are understood. In case a target network does not support any of these opcodes, consider lowering the target EVM version during compilation.

Resolution

The development team has closed the issue with the following comment:

> *"OMRL-10 is not addressed as Omni only supports chains that support Solidity 0.8.26, which was understood in the finding."*

OMRL-11	Pausing and Unpausing Have The Same Role
Asset	`Bridge.sol, Lockbox.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The role required to perform deposits and withdrawals for the `Lockbox` is the same role required for unpausing. It is best practice to have two separate roles, a pauser and an unpauser, such that they can be set to different addresses. This allows for an easier to access and quicker response for pausing the protocol, while unpausing can require a more heavily guarded and timelocked multisig.

Similarly, `Bridge` also only has one role for both pausing and unpausing.

Recommendations

Ensure the above comments are understood and consider changes if desired.

Resolution

This issue has been fixed in PR #3096 by implementing an unpauser role.

OMRL-12	Optimising Redundant Condition Checks For `lockbox_`
Asset	`Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The following two conditions are similar and could be merged together to improve code clarity and efficiency.

Bridge.sol

```
102 if (lockbox_ != address(0)) lockbox = lockbox_;

104 // Give lockbox relevant approvals to handle deposits and withdrawals if necessary.
    if (lockbox_ != address(0)) {
106   ILockbox(lockbox_).token().safeApproveWithRetry(lockbox_, type(uint256).max);
      token_.safeApproveWithRetry(lockbox_, type(uint256).max);
108 }
```

Recommendations

Consider merging the two conditions.

Resolution

This issue has been fixed in PR #3096 by merging the redundant checks.

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.

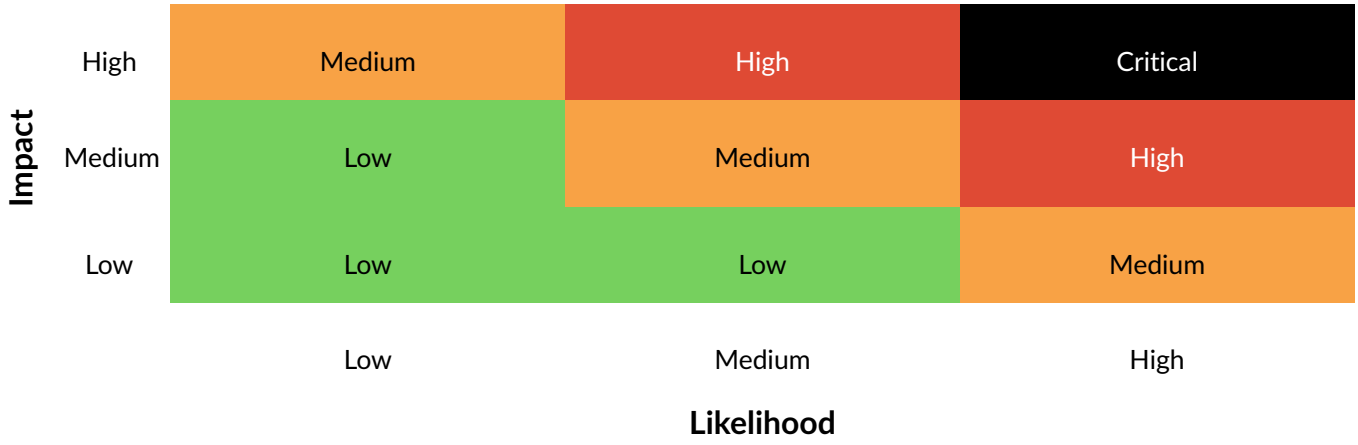

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].