Alongside Finance

Index Smart Contracts

Security Assessment Report

Version: 1.0

May, 2022

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Alongside Finance index 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 Alongside Finance index 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 Alongside Finance smart contracts.

Overview

The Alongside Crypto Market Cap (AMKT) Token is an ERC20 token designed to be fully redeemable for its underlying collateral held in multiple cryptocurrencies and digital assets, serving as an index of the broader crypto market.

The AMKT contracts are forked from WBTC and modified to:

- upgrade contracts to solidity 0.8.7 and utilize OpenZeppelin upgradeable libraries
- support multiple underlying digital assets as collateral
- migrate custody from the protocol team to a third-party institutional custodian
- enable rebalancing of the index portfolio assets on a periodic basis
- generate protocol fees

The AMKT contracts provide a template for managing the minting, burning, and transfer operations of the token. However, significant portions of this activity occur off-chain: utilizing IPFS for data distribution and custodian APIs for the validation, transfer, and rebalancing of portfolio assets. While these off-chain processes were considered throughout the course of the review, they should not be considered within the scope of this security assessment.

Security Assessment Summary

This review was conducted on the files hosted on the Alongside-Finance index-contract repository and were assessed at commit 81f73cb.

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

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts: specifically, 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 all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

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

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 10 issues during this assessment. Categorized by their severity:

- Medium: 2 issues.
- Low: 2 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Alongside Finance index smart contracts. 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
AMKT-01	Lack of Usage of `safeTransferFrom` for External Token Interactions	Medium	Open
AMKT-02	Duplicated Mint Requests May Be Confirmed	Medium	Open
AMKT-03	Merchants Can Mint Requests Over the Supply Ceiling	Low	Open
AMKT-04	OTC Mint Requests Over Merchant Limits Are Allowed	Low	Open
AMKT-05	Limits Only Enforced Per Mint or Burn	Informational	Open
AMKT-06	Inconsistent Indices in `IndexedMapping`	Informational	Open
AMKT-07	Inflation Rate Varies With Frequency of Calls to `_mintToFeeReceiver()`	Informational	Open
AMKT-08	Enums With Consequential First Values Can Cause Unexpected Behaviour	Informational	Open
AMKT-09	Miscellaneous General Comments	Informational	Open
AMKT-10	Impact of Key Compromise for Privileged Roles	Informational	Open

AMKT-01	Lack of Usage of `safeTransferFrom` for External Token Interactions		
Asset	`OTC.sol`		
Status	Open		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

There is a potentially unsafe call to `PaymentToken.transferFrom()` on line [128] of `OTC.sol`.

```
/// @notice Requests IndexToken for PaymentToken
/// @param paymentTokenAmount amount of PaymentToken to deposit
/// @param indexTokenAmount amount of IndexToken to receive
function requestIndexToken(uint256 paymentTokenAmount, uint256 indexTokenAmount)
  external
  onlyUser
  returns (uint256)
{
  PaymentToken.transferFrom(msg.sender, depositAddress, paymentTokenAmount); // reverts on failure
  requests.push(
    Request(msg.sender, paymentTokenAmount, indexTokenAmount, RequestStatus.PENDING, 2**256 - 1, depositAddress)
  );
  uint256 nonce = requests.length - 1;
  emit RequestIndexToken(nonce, msg.sender, depositAddress, paymentTokenAmount, indexTokenAmount);
  return nonce;
}
```

Although this call is expected to revert on failure, this cannot be assumed of an arbirary `paymentToken` which may not fully comply with `ERC20` specifications. As such, failed payments may manage to successfully initiate an index token request.

Recommendations

Use OpenZeppelin's `SafeERC20` wrapper function on line [128] of `OTC.sol`.

Alternatively, new `OTC` deployments for each `paymentToken` should be carefully reviewed and whitelisted by the protocol team. This reduces gas costs on transfers, but retains an avoidable long term risk in the contracts.

AMKT-02	Duplicated Mint Requests May Be Confirmed		
Asset	`Factory.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

There are no requirements in the contracts that prevent a `Merchant` from successully calling `Factory.addMintRequest()` multiple times for the same deposit. It is also possible for the `Issuer` to subsequently confirm the duplicated requests.

Two identical calls can be made by a `Merchant` to `addMintRequest()`, which are both accepted. The values of `nonce` will differ between the requests, and so will `requestHash`. The parameter `txid` is not used for validation, so there is nothing in the contracts to ensure the requests refer to separate transactions.

For the same reason, a `Merchant` may also call `Factory.addMintRequest()` to create new mint requests for previously rejected or cancelled mint requests.

The system therefore relies on the validators to pick up duplicate mint requests to prevent tokens being minted twice in these situations.

Recommendations

The protocol could be changed to require `txid` to be used, although this is a significant design change.

Alternatively, if the team wishes to keep the current design, they should remain keenly aware of this issue and be sure that the `Issuer` validators have a robust system to avoid minting `AMKT` from duplicate requests.

AMKT-03	Merchants Can Mint Requests Over the Supply Ceiling		
Asset	`Factory.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Calls to `Factory.addMintRequest()` do not check that the mint amount is below the `IndexToken.supplyCeiling` threshold. Therefore, a mint request over this limit may be initiated by a `Merchant`.

The pending mint request will eventually fail when `Factory.confirmMintRequest()` is called due to the requirement on line [92] of `IndexToken.mint()`. However, this pattern in asyncronous operations results in a negative experience for the `Merchant` initiating a mint request, who should have already deposited the underlying assets.

The full resolution of the issue after this point is outside the scope of this assessment.

Recommendations

The issue could be mitigated by requiring that mint amounts are below the `IndexToken.supplyCeiling` threshold within the `Factory.addMintRequest()` function.

Alternatively, the team could simply stay alert to this issue and ensure that `IndexToken.supplyCeiling` is never within reach of allowable mint requests.

AMKT-04	OTC Mint Requests Over Merchant Limits Are Allowed		
Asset	`OTC.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Calls to `OTC.requestIndexToken()` do not check that the mint amount is below the `Factory.merchantMintLimit` threshold. Therefore, a mint request over this limit may be initiated by a `Merchant`.

Instead, the `merchantMintLimit` of the `OTC` contract will be applied for all merchants.

Recommendations

The issue could be mitigated by requiring that mint amounts are below the `Factory.merchantMintLimit` of the `OTC` contract within the `OTC.requestIndexToken()` function.

AMKT-05	Limits Only Enforced Per Mint or Burn
Asset	`Factory.sol`
Status	**Open**
Rating	Informational

Description

`merchantMintLimit` and `merchantBurnLimit` are only enforced for each request, therefore a `Merchant` may exceed these limits by submitting multiple requests.

Recommendations

Make sure that this behaviour is understood and intended.

If a total mint limit is preferred, mappings could be created to track the total value in `AMKT` that is currently open per merchant for each request type. These values would need to be incremented and decremented as the processes complete, and they would be tested against `merchantMintLimit` and `merchantBurnLimit`, adding the new request during the test.

AMKT-06	Inconsistent Indices in `IndexedMapping`
Asset	`Members.sol, OTC.sol, IndexedMapping.sol`
Status	Open
Rating	Informational

Description

The `IndexedMapping` is implemented for `Members.merchants` and `OTC.users`. Both of these variables have external functions that retrieve entries by index: `Members.getMerchant()` and `OTC.getUser()`. They also have external functions that retrieve their values in order of index: `Members.getMerchants()` and `OTC.getUsers()`.

The implementation of `IndexedMapping` changes the index values for some entries when other entries are removed.

This behavior could potentially break external applications that might rely on these external functions, if they are unaware of the facts that:

1. the index of a given entry may change
2. the sort order of the returned values may change

Recommendations

`Members.getMerchants()` and `OTC.getUsers()` could be modified to return sorted results. `Members.getMerchant()` and `OTC.getUser()` could be removed entirely.

If these measures are not desired, then the protocol team should clarify in the related documentation that external applications should not rely on the indices and sort orders remaining consistent.

AMKT-07	Inflation Rate Varies With Frequency of Calls to `_mintToFeeReceiver()`
Asset	`IndexToken.sol`
Status	**Open**
Rating	Informational

Description

The fees collected by `IndexToken._mintToFeeReceiver()` are dependent on the total supply of the token, yet also increase the total supply. Therefore, the effective fee rate is compounded whenever `_mintToFeeReceiver()` is called. More frequent calls will increase the effective fee rate.

`_mintToFeeReceiver()` is called with every burn or mint operation of the token. It may also be called via an external function, which `feeReceiver` would be incentivized to call frequently.

Recommendations

Be aware of this issue, particularly when calculating an increase in the fee rate or describing the fee rate to users.

The effective fee rate will be greater than the parameter chosen for `feeRateAnnualBips` due to variable compounding.

AMKT-08	Enums With Consequential First Values Can Cause Unexpected Behaviour
Asset	`Factory.sol, OTC.sol`
Status	Open
Rating	Informational

Description

The user-defined data type `enum` defaults to its first value, so it is good practice to define its first value as a null value that is never used in the code, i.e. a value that will fail any `require` checks performed on the zero-initialized value.

`Factory.sol` and `OTC.sol` have (different) enums called `RequestStatus`, both of which default to a value of `PENDING`.

The testing team did not identify a scenario in the current implementation of the protocol where this could cause an issue.

However, `Factory.sol` is upgradeable and does make checks against `RequestStatus.PENDING` for a given `requestHash` in `validatePendingRequest()`.

```
function validatePendingRequest(Request memory request, bytes32 requestHash) internal pure {
  require(request.status == RequestStatus.PENDING, "request is not pending");
  require(requestHash == calcRequestHash(request), "given request hash does not match a pending request");
}
```

This check could be expected to pass for a zero-initialized `request` struct and its corresponding `requestHash`, which has not yet been added via `addMintRequest` and marked as `PENDING`.

Recommendations

Add a first value `NULL` to both `RequestStatus` enums.

If this is not desired, remain aware of this potential issue if the implementation contract of `Factory.sol` is ever upgraded.

AMKT-09	Miscellaneous General Comments
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team:

1. **Frontrunning Rebalance:** As the protocol grows and the size of the assets in custody increases, it may be possible to monitor calls to `IndexToken.setMethodology()` and thereby predict what the rebalancing trades will be. If those trades are very large in volume, it is conceivable they might affect some asset prices and thus it may be possible to frontrun them for profit.
2. **Meta Transaction Support:** `transfer` and `transferFrom` are not implementing OpenZeppelin's `ContextUpgradeable` (as the orignal functions do) so support for meta transactions would require an implementation upgrade of this contract.
3. **Ownership Transfer:** OpenZeppelin's `Ownable` and `OwnableUpgradeable` allow ownership transfer with a single transaction, which opens up a risk of human error losing owner control of a vital contract. Propose/accept structures of ownership transfer can be used to avoid this issue.
4. **Token Name:** `DeployAMKT.sol` line [43] deploys the AMKT token with the name "Test".
5. **Issuer/Custodian Role Ambiguity:** The confusion over the `Issuer` role being known by the name of another role (`Custodian`) within the contracts seems unnecessarily ambiguous and could potentially cause issues in future through confusion. If the goal is to minimise differences with `WBTC`, one solution would be to refer to the role as (`Custodian`) in the documentation.
6. **Supply Ceiling and Inflation:** The Alongside documentation says, *"The supply ceiling ... MUST not apply to inflation."* Two interpretations of this should be distinguished. Supply ceiling does not block inflation from occurring in the instance where inflation will take the token supply above the supply ceiling. However, the supply ceiling does apply to inflation in the sense that all tokens previously created through inflation are counted towards the supply ceiling.
7. **Unused txid Parameter:** `Factory.addMintRequest()` has the parameter `txid` that is used in the OTC flow, but not in the standard mint flow. This effectively means that merchants can deposit any value they wish into the contract's storage. It is also used in determining `requestHash` values. This is not a desirable state of affairs from a security perspective. It is suggested that a mechanism be added to force `txid` to be empty, unless the sender is on a whitelist, which the `OTC` contract could be added to.
8. **Token Balance:** `OTC.requestIndexToken()` could, for added certainty, check its payment token balance before and after the payment token transfer.
9. **OTC and Shifting Asset Prices:** Unless the OTC system has a way of locking rates on the API side, it seems that a request could potentially be marked `PROCESSING`, and then be unable to pay for all its asset tokens (or have an excess) before ' `finalizeRequest` is called.
10. **Issues in Comments:**
    - `Controller.sol` line [56] and line [65]: technically, the "implementation" contract is the code address, not the proxy addess, and these are both the proxy addresses

- `Controller.sol` line [92] and line [103]: typo "funciton"
- `Factory.sol` line [172]: This comment is wrong. The function sets the address that assets from a burn should be moved to (as per the comment on line [110]).
- `Factory.sol` line [240]: typo "merchnat"
- Many comments say, "@notice Explain to an end user what this does" or, "@dev Explain to a developer any extra details". These should be replaced with the requested explanations.

11. **Issues in Error Messages:**
    - `OTC.sol` line [112]: There is only one validator address so this error message is a little confusing.
    - `OTC.sol` line [201]: This error message should probably be, "Mint request is not approved".
12. **Ignored Return Values:** It is best practice to check return values when they are provided by a function.
    - `DeployAMKT.constructor()` line [33] ignores the return value by `members.setCustodian()`.
    - `DeployAMKTPartTwo.completeDeployment()` line [35] ignores the return value by `controller.setMembers()`.
    - `DeployAMKTPartTwo.completeDeployment()` line [36] ignores the return value by `controller.setFactory()`.
13. **Minimise Function Access:** It is best practice to use the least public permission for a function.
    - `Controller.factoryPause()` should be declared external.
    - `DeployAMKTPartTwo.completeDeployment()` should be declared external.
14. **Zero Address Checks:** Consider adding checks to ensure submitted values are not `address(0)` in the following places:
    - `DeployAMKT.sol` and `DeployAMKTPartTwo.sol` A zero address in any of their function parameters could be a costly mistake in terms of gas.
    - `OTC.sol`: the constructor has no zero address checks.
15. **Contract Check:** The address parameters to `DeployAMKTPartTwo.completeDeployment()` could be checked to ensure that they are smart contracts.
16. **Interactions between Rebalancing and Inflation:** As the `AMKT` token will inflate between rebalances, the numbers in the methodology file will go out of date. Two methods were discussed for addressing this:
    - The systems will take account of the fee mint rate so that all functions dynamically modify the methodology when minting or burning. One possible issue here is that there could be a source of error if the fee rate changes between rebalances.
    - The idea was mentioned of inflation not being applied on mint and burn rates until rebalancing. If this was the case, it might lead to heavy burning just before rebalancing and then heavy minting right after.
17. **Possible Gas Optimisations:**
    - `DeployAMKT.sol` line [43] / line [72] a constant could be used for the supply ceiling. This would also allow for it to be updated in one place.
    - `DeployAMKT.sol` and `DeployAMKTPartTwo.sol` have multiple state variables that seem to be acting only as a form of logging. The events emitted by the contracts could be used instead for this purpose.
    - `Factory.burn()` line [323] is creating a memory variable `txid` that remain an empty `string`. This value literal could be used directly on line [329].
    - `OTC.finalizeRequest()` line [161] reads a variable from storage that is read again in storage on the next line. The memory variable `request` on line [163] could be used for this check. This will be more expensive if many transactions are submitted with the wrong `RequestStatus`, but this presumably will not be the case.

Recommendations

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

AMKT-10	Impact of Key Compromise for Privileged Roles
Asset	`contracts/*`
Status	**Open**
Rating	Informational

Description

The AMKT contracts require several privileged roles to facilitate the minting and burning of AMKT. The testing team recommends reassessing and documenting the abilities and trust assumptions of each of these roles to avoid confusion and bring clarity between specifications, implementation, and documentation.

For example:

- the role name `Custodian` was retained in code from legacy WBTC contracts, while referred as `Issuer` in documentation
- the `Issuer` in documentation seems to have the abilities of several roles in code: `Admin`, `Methodologist`, and `Custodian`
- in the `OTC` contract, it is unclear whether `Validator` is expected to be from the same `Validators` controlling `Issuer`, whether `User` in this contract is expected to be a `Merchant`, and the `OTC` contract must also be a `Merchant`

The `Merchant` role is assumed to be malicious, and the implications of key compromise of other trusted roles is briefly considered here. These findings are not exhaustive and should be considered an overview of the topic:

1. **Issuer**

   The `Issuer` is intended to be set to a multi-sig controlled by a set of validators who observe deposits and withdrawals of underlying assets to trigger minting and burning events.

   A compromise of the Issuer private key could allow an attacker to:

   - set a false `custodianDepositAddress`
   - collude with a `Merchant` to approve mint requests when assets have not been deposited
   - collude with a `Merchant` to approve a mint request for assets from another merchant (using `setCustodianDepositAddress()` to reassign the addresses to validate the malicious mint request and invalidate the real one, or simply reject the real request)
   - maliciously reject valid requests to mint or burn
   - set limits to allow excessive arbitrage or to maliciously block some merchants entirely

2. **OTC Validator**

   The OTC `Validator` can process, reject, or confirm a mint request through the OTC contract. The flow of approval of requests must be co-administered with the `Issuer`.

   If the OTC `Validator` is compromised, it could reject valid requests.

3. **Methodologist**

   The `Methodologist` can set the contents of the methodology file, which defines the underlying composition of 1 AMKT.

   A compromised `Methodologist` could grieve `Merchants` by front running a burn request and setting AMKT value very low just before a burn.

   If a malicious `Merchant` compromises the `Methodologist` then the `Merchant` could:

   - redefine the nominal amount of 1 AMKT to equal most of the value in the vault, and attempt to claim those assets by burning 1 AMKT
   - set a very low AMKT value, mint AMKT at a discount, reset to original methodology, and submit a burn request

4. **Admin**

   `Admin` sets all of the other roles, therefore a compromise of these keys opens all of the previously noted attacks and more:

   - set `feeReceiver` to itself with a high inflation rate, add itself as a `Merchant`, and submit a burn request.
   - upgrade any of the upgradable contracts to behave maliciously. For example, it could add an unrestricted mint method to `IndexToken`.

Recommendations

Ensure that the comments are understood and acknowledged.

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.

```
test_config                     PASSED  [1%]
test_mint                       PASSED  [2%]
test_burn                       PASSED  [4%]
test_factoryPause               PASSED  [5%]
test_addMintRequest             PASSED  [6%]
test_cancelMintRequest          PASSED  [8%]
test_confirmMintRequest         PASSED  [9%]
test_rejectMintRequest          PASSED  [10%]
test_burn                       PASSED  [12%]
test_confirmBurnRequest         PASSED  [13%]
test_pause                      PASSED  [14%]
test_unpause                    PASSED  [16%]
test_setCustodianDepositAddress PASSED  [17%]
test_setMerchantDepositAddress  PASSED  [18%]
test_setMerchantMintLimit       PASSED  [20%]
test_setMerchantBurnLimit       PASSED  [21%]
test_setCustodian               PASSED  [22%]
test_addMerchant                PASSED  [24%]
test_removeMerchant             PASSED  [25%]
test_deployment                 PASSED  [27%]
test_requestIndexToken          PASSED  [28%]
test_processRequest             PASSED  [29%]
test_finalizeRequest_confirmed  PASSED  [31%]
test_claimIndexToken            PASSED  [32%]
test_addUser                    PASSED  [33%]
test_removeUser                 PASSED  [35%]
test_finalizeRequest_rejected   PASSED  [36%]
test_toggleRestriction          PASSED  [37%]
test_mint                       PASSED  [39%]
test_burn                       PASSED  [40%]
test_mintToFeeReceiver          PASSED  [41%]
test_pause                      PASSED  [43%]
test_unpause                    PASSED  [44%]
test_transfer                   PASSED  [45%]
test_transferFrom               PASSED  [47%]
test_setMethodologist           PASSED  [48%]
test_setMethodology             PASSED  [50%]
test_setFeeRate                 PASSED  [51%]
test_setFeeReceiver             PASSED  [52%]
test_setMinter                  PASSED  [54%]
test_setSupplyCeiling           PASSED  [55%]
test_config                     PASSED  [56%]
test_mint                       PASSED  [58%]
test_burn                       PASSED  [59%]
test_factoryPause               PASSED  [60%]
test_addMintRequest             PASSED  [62%]
test_cancelMintRequest          PASSED  [63%]
test_confirmMintRequest         PASSED  [64%]
test_rejectMintRequest          PASSED  [66%]
test_burn                       PASSED  [67%]
test_confirmBurnRequest         PASSED  [68%]
test_pause                      PASSED  [70%]
test_unpause                    PASSED  [71%]
test_setCustodianDepositAddress PASSED  [72%]
test_setMerchantDepositAddress  PASSED  [74%]
test_setMerchantMintLimit       PASSED  [75%]
test_setMerchantBurnLimit       PASSED  [77%]
test_toggleRestriction          PASSED  [78%]
test_mint                       PASSED  [79%]
test_burn                       PASSED  [81%]
test_mintToFeeReceiver          PASSED  [82%]
```

```
test_pause                      PASSED  [83%]
test_unpause                    PASSED  [85%]
test_transfer                   PASSED  [86%]
test_transferFrom               PASSED  [87%]
test_setMethodologist           PASSED  [89%]
test_setMethodology             PASSED  [90%]
test_setFeeRate                 PASSED  [91%]
test_setFeeReceiver             PASSED  [93%]
test_setMinter                  PASSED  [94%]
test_setSupplyCeiling           PASSED  [95%]
test_duplicateMintRequest       XFAIL   [97%]
test_mintToFeeReceiver_rate     XFAIL   [98%]
test_requestIndexToken          XFAIL   (unsaf...)[100%]
```

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.

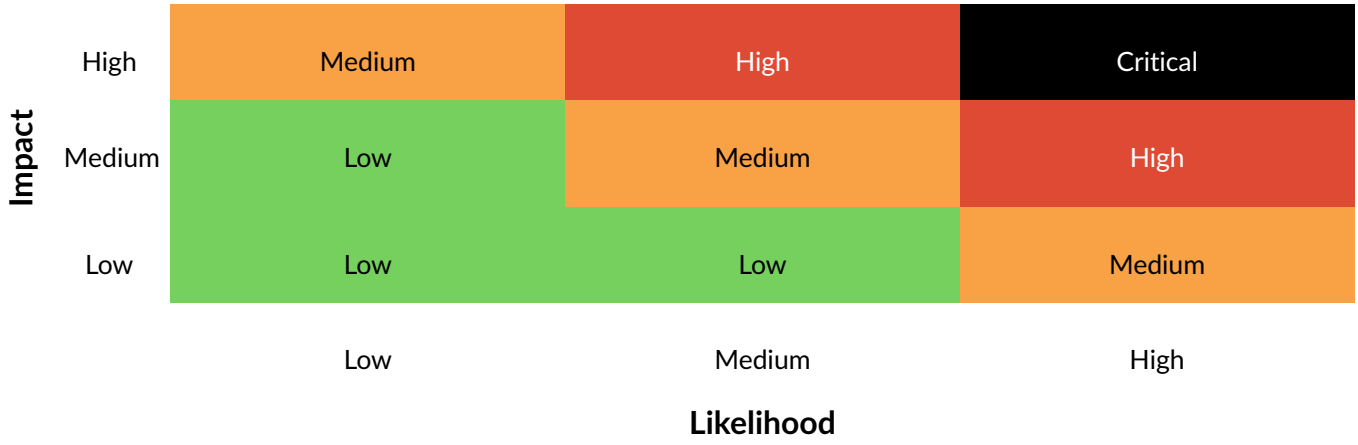

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].