Swell Network

Swell Network – Smart Contracts

Security Assessment Report

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Swell Network 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 Swell Network 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 Swell Network smart contracts.

Overview

Swell Network is a liquid ETH staking protocol that is permissionless and non-custodial. It enables users to participate in proof-of-stake validation with amounts as small as 1 ETH by staking to a node operator with other users to reach the 32 ETH validator requirement.

Unlike other liquid ETH staking projects, the user is able to choose which node operator they stake with, being able to select for lower fees or reliability.

The protocol is vertically integrated, incorporating its own vaults that use various strategies to earn additional income on top of the staking rewards issued.

Users receive two tokens when staking via Swell Network, `swETH` and a `swNFT`. `swETH` is a fungible `ERC-20` that can be used in Swell's own vaults or in the wider DeFi ecosystem to earn further yield. The `swNFT` token is an `ERC-721` that represents the user's claim on the staking rewards relating to their original ETH deposit.

The protocol is controlled by a DAO which has a governance token called `SWELL`. The DAO treasury receives a 5% fee on staking rewards earnt by users on the Beacon chain.

Security Assessment Summary

This review was conducted on the files hosted on the Swell Network repository and were assessed at commit 9c595ed. Specifically, this assessment targeted the following files:

- `swNFTV1.sol`
- `swETH.sol`
- `SWELL.sol`
- `swNFTUpgrade.sol`
- `helpers.sol`

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. 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 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`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 12 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 1 issue.
- Low: 4 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Swell Network 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
SWL-01	Direct Deposits Enable Theft Of A Validator's Funds	High	Resolved
SWL-02	Staking Before Operator Leads To No `swETH` Minted	Medium	Closed
SWL-03	Validators Cannot Receive More Than 32 ETH	Low	Closed
SWL-04	Whitelisted Or Super Whitelisted Users Cannot Be Removed	Low	Closed
SWL-05	Ownership Functionality Can Be Lost Through One Step Transfers	Low	Resolved
SWL-06	Active Validators Cannot Be Deactivated	Low	Closed
SWL-07	Function Returns Are Never Used	Informational	Resolved
SWL-08	swETH Minter Cannot Be Altered Once Set	Informational	Closed
SWL-09	Staking Event Emits Public Key In Hashed Form	Informational	Closed
SWL-10	Incomplete Function Bodies	Informational	Resolved
SWL-11	Miscellaneous `Swell` General Comments	Informational	Resolved
SWL-12	Miscellaneous Gas Optimisations	Informational	Resolved

SWL-01	Direct Deposits Enable Theft Of A Validator's Funds		
Asset	`swNFTUpgrade.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

If a node operator interacts with the deposit contract directly first, it is possible for them to set the withdrawal address to an arbitrary address. Then this node can be added to Swell and used normally. Once deposits are enabled on the Beacon chain, it is possible for this node operator to withdraw all the ETH deposited with this node. In addition to this, it is impossible for the normal withdrawal method specified by `swNFTUpgrade.sol` to work for deposits made to this node.

When a node operator wants to start a new validator on Swell they do so via the `stake()` function. This function calls the `depositContract's` `deposit()` function which allows setting up a new validator on the Beacon chain. The withdrawal address for this validator is set by Swell and is hardcoded as being the address of `swNFTUpgrade.sol`.

The issue stems from the fact that while the withdrawal credentials are broadcasted on each deposit call from the deposit contract, only the first mention of a validator public key triggers the checking and storage of data supplied. Once the deposit data is verified, the withdrawal credentials are stored on the Beacon chain and subsequent deposits to the same public key simply increment the amount of ETH this validator has.

This means that a malicious node operator can directly call the `deposit contract's` `deposit()` function first, supplying their own address as the withdrawal address. Then if they reuse the same node operator's public key when calling via Swell, the Beacon chain code will ignore the correct withdrawal address passed by the Swell `stake()` call as it already recognises the validator node. Apart from the validation node's withdrawal address registered on the Beacon chain, it is difficult to detect this difference on the Ethereum chain as the events emitted via Swell's `stake()` call look the same as for a normal validator.

This results in a node that appears to be a normal Swell validator node but is associated with a different withdrawal address which can be used to drain all related funds once withdrawals are enabled on the Beacon chain. Furthermore, as the withdrawal address does not match the correct Swell withdrawal address, any attempt to withdraw funds from this node via Swell will fail.

Recommendations

A possible solution to this issue is to rely on the off-chain bot used to activate new nodes for Swell deposits to check that the new nodes have the correct withdrawal credentials prior to activating these nodes.

Post merge, other solutions may become viable as then both parts of the deposit transaction will be occurring on the same chain.

Resolution

The development team are aware of the issue and have a `cronjob` bot that checks Beacon Chain deposit and verifies the `withdrawal_credential` before enabling the validator on the platform.

SWL-02	Staking Before Operator Leads To No `swETH` Minted		
Asset	`swNFTUpgrade.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The first staking call to a `validation node` is intended to be done by the `operator`, in which the `operator` sends a 1 or 16 ETH deposit (assuming they are not in the super whitelist). However, no validation is done to check the first user staking is the `operator`. This means a naive end user, Bob, can stake to a new `validation node` and their funds will become stuck, misinterpreted as the operator's deposit.

This is because when an `operator` makes their initial deposit, it is intended as a bond to ensure good behaviour, and so they mint no `swETH`. In addition because Bob lacks the private key related to the validator's public key, this transaction will be rejected on the Beacon chain as there is no way for his signature to match the validator's public key. This will lead to his funds becoming stuck as they will be sent to the deposit contract but not recorded on the Beacon chain and so be unclaimable in the future.

Then as a knock-on effect, this also leads to the operator being able to bypass their required deposit bond as subsequent calls will now always mint swETH, allowing them to sell this to offset their ETH deposit. This breaks behavioural assumptions as now the operator has no funds inside the validator and so has less incentive to fear being slashed. This can therefore risk all funds deposited with this validator.

Bob must still deposit the correct amount of 1 or 16 ETH exactly, else the call to `stake()` will reject this as the initial validator deposit. Bob must also directly call `stake()`, having generated his own incorrect signature and deposit data information. As most users would leverage the user interface that automates this process, the likelihood of this issue with deemed *Low*.

Recommendations

Possible mitigation strategies include:

1. Require the user calling stake to declare if they are the operator, this way a naive user can have their transaction rejected if the operator has not already deposited their bond.
2. Completing the `swDAO` bond requirement code mentioned on line [**386**] and line [**390**], while also requiring a nominal bond deposit for `superWhiteListed` operators to prevent a naive user accidentally setting up an operator's node. If the first operator call requires an extra token it is unlikely to be triggered by accident.
3. Require proof of possession of the validation node's private key on the first operator call. This can be achieved by signing a short message such as the depositing address using the node's related private key, and then verifying the sending address matches this.

Resolution

The development team has clarified that this issue is unlikely to happen for several reasons :

1. There is no way to a naive user to do that through the User Interface. The UI also has wallet signing nonce checks to verify node's owner address for the first stake.
2. The operator's validator public key is not visible for other users.

SWL-03	Validators Cannot Receive More Than 32 ETH		
Asset	`swNFTUpgrade.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Normally a staking validator node can have a maximum effective balance of 32 ETH. This means there is usually no reason to deposit any more than this amount to a validator as the extra ETH will not earn any staking rewards. Because of this, Swell Protocol will revert on attempts to call `stake()` with a validator who already has 32 ETH assigned to them. If a validator has been slashed this can cause problems.

A slashed validator will lose some of their deposited ETH and in this situation the validator's effective deposit can drop below 32 ETH. Due to how the effective balance is determined, this can reduce staking efficiency. In most cases this drop will be to around 95% efficiency, but in the rare case when the ETH balance drops below 16 ETH the validator can be ejected completely.

In both of these slashing cases the validator will want to deposit extra ETH to return their node to 32 ETH. Currently `stake()` does not allow this to happen as when the total amount of deposits hits 32 ETH line [**380**] will cause a revert on subsequent calls of `stake()` to that validator.

While it is still possible for a user to make a direct call to the staking contract's `depositContract.deposit()`, doing so is more complicated and grants the user no `swETH` and so only the `operator` would be incentivized to do this.

Recommendations

Deposits from the `operator` to their own staking node could bypass the 32 ETH cap check, allowing them to top up a slashed validator returning efficiency back to 100% and receiving `swETH` to offset this maintenance cost.

Resolution

The team acknowledged this issue and stated that the suggested changes would contradict the existing reward model and lead to incorrect reward allocation calculations. No action was taken.

SWL-04	Whitelisted Or Super Whitelisted Users Cannot Be Removed		
Asset	`swNFTUpgrade.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `owner` can add a validator to the `whiteList` to reduce the ETH deposit required to make a new validator from 16 ETH to 1 ETH. Similarly, users can be added to the `superWhiteList` and then can create a new validator for no cost at all. These are trusted roles that could lose users income if operated incorrectly, however there is no function to remove a validator from the `whiteList` or the `superWhiteList`.

Adding such functions is recommended because a validator private key could get compromised or a company that represents a validator in the `superWhiteList` may be dissolved. In these instances it makes sense to remove stale or vulnerable validator's public key from their respective whitelist.

Recommendations

Consider adding two functions for removing a validator from the `superWhiteList` and `whiteList`.

Resolution

This issue has been acknowledged by Swell team. The recommended functions will be added when necessary.

SWL-05	Ownership Functionality Can Be Lost Through One Step Transfers		
Asset	`swNFTV1.sol and SWELL.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

`swNFTV1.sol` and `SWELL.sol` inherit from OpenZeppelin's `Ownable.sol` contract. As the ownership transfer in `transferOwnership()` is unilateral, ownership can be accidentally transferred to an uncontrolled address. Once transferred there is no means to reclaim ownership.

While `swNFTV1.sol` makes no direct use of the `Ownable` features, these are used extensively in `swNFTUPgrade.sol` which inherits from `swNFTV1.sol`. The `SWELL.sol` contract does make direct use of `Ownable`, using it as a controller for `minting` and `burning` of `SWELL` tokens.

Recommendations

Change ownership transfer to a propose/accept model in both files.

One convenient way to do this is to replace OpenZeppelin's `Ownable` with Chainlink's `ConfirmedOwnerWithProposal`

Resolution

The development team has clarified that they will use `GnosisSafe` as an extra confirmation step when transfering ownership. In addition to this, the team mentioned that transfering ownership is unlikely to happen.

SWL-06	Active Validators Cannot Be Deactivated		
Asset	`swNFTUpgrade.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A new validation node must be activated by a `bot` before it can receive any deposits from normal users after its initial deposit by the validator operator. Once a validating node is set as active, it is not possible to deactivate it.

This may cause issues if a validating node loses their private key or has it stolen. In such cases, the protocol may want the ability to freeze any additional deposits to this validator.

Recommendations

Consider adding a function to deactivate an existing validator. As this action is less likely than activation, it could be restricted to a DAO role rather than the `bot` used for activations.

Alternatively, a quicker response can be achieved by deactivating the validator in question from appearing in the user interface of the Swell website as most users will likely use this to stake with validators.

Resolution

This issue has been acknowledged by Swell team. The deactivate function will be added when necessary.

SWL-07	Function Returns Are Never Used
Asset	`swNFTUpgrade.sol`
Status	**Resolved:** See commit 28fdb69.
Rating	Informational

Description

In `swNFTUpgrade.sol`, there are four functions that interact with `swETH` owned by staking users. These are `deposit()`, `withdraw()`, `enterStrategy()` and `exitStrategy()` and they return booleans denoting if their action was successful. However, none of these booleans are subsequently used by any other functions.

For functions `deposit()` and `withdraw()`, it is impossible for them to return `false` as the `ERC20` operations they use to measure success will revert on failure. Similarly, this is applicable to the current strategies usable by `enterStrategy()` and `exitStrategy()`.

This makes the booleans returned by these functions pointless as they will always return `true` if the call has not reverted.

Recommendations

Either the return booleans should be used by other functions and modified so that they can return more than one state or they can be removed to reduce code complexity and save gas.

Resolution

In this commit functions `deposit()` and `withdraw()` have been removed, while the `enterStrategy()` and `exitStrategy()` have been updated to check the return value.

SWL-08	swETH Minter Cannot Be Altered Once Set
Asset	`swETH.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `swETH` token has a minter role that is set on construction. Only this address is able to mint `swETH` and this role is handled by the `swNFT.sol` contract. Minting or burning `swETH` will become impossible if the `swNFT` ever has to change address. A new `swETH` token will need to be deployed and all services using it will have to migrate to the new `swETH` token address.

However as `swNFT` uses a proxy contract, it is unlikely the address will ever need to change, making this an informational issue.

Recommendations

A possible method to mitigate this issue would be to enable the minter role to transfer its role to a new address. This way if there ever was a need to swap to a new `swNFT` proxy address, there would be no need to alter the `swETH` implementation.

Resolution

This issue has been acknowledged by Swell team.

SWL-09	Staking Event Emits Public Key In Hashed Form
Asset	`swNFTUpgrade.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

When a user stakes with a `validator`, this is recorded in an event called `LogStake`. The third argument logged is the `pubKey` of the `validator` operating the validation node. Because this argument is indexed and of dynamic length, it is stored as a hash of its original data. This means the original `pubKey` cannot be read from the event directly.

It is possible to determine the original `pubKey` by hashing all known `pubKeys` stored in the `validators` array and comparing the resulting value with the stored hash, but this method puts more burden on the user.

Recommendations

Two possible mitigation strategies are as follows:

1. Add an extra non-indexed field to the event `LogStake` that is a copy of the `pubKey`. This way the indexed hash can be used to search for logs, but the exact value can still be read off from each log.
2. Add a mapping that maps the `pubKey` hash to the original `pubKey`.

Resolution

The development team has clarified that in the `subGraph` the public key is displayed correctly.

SWL-10	Incomplete Function Bodies
Asset	`swNFTUpgrade.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Due to the upgradable nature of some contracts and the fact that Swell Network relies on the ETH Beacon chain staking contract, there are several functions and areas of code that are incomplete. As a result, a full analysis of the protocol could not be completed as the final behaviour of certain functions is undefined and/or unknown.

Affected areas include:

- line [**386**] and line [**390**] of `_stake()` are incomplete, lacking logic for handling the swDAO bond requirements on validator operators.
- `unstake()` has commented out code and is hardcoded to revert currently.
- `batchAction()` passes empty byte arrays to the `strategy` params field.
- The mapping `opRate` is not used anywhere despite having a function `setRate()` which assigns its value.
- The variable `fee` is not used anywhere despite having a function `setFee()` which assigns its value.
- The variable `feePool` is not used anywhere despite being set on construction and having a function `setFeePool()` which assigns its address.

Recommendations

The development team should be careful when implementing these features to ensure they do not introduce new flaws to the protocol. If possible, seek external security reviews before deploying new features.

Resolution

This issue has been acknowledged by Swell team. The team will do an audit review before every new feature.

SWL-11	Miscellaneous `Swell` General Comments
Asset	`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. **helpers.sol**
   (a) Typo: On line [**70**] `"outout"` should read `"output"`.
2. **swNFTv1.sol**
   (a) Several variables are commented out with the note `"not used"` on line [**34**], line [**43**] and line [**45**]. These should be removed if not needed.
3. **swNFTUpgrade.sol**
   (a) On line [**384**] and line [**388**] the word `ether` is missing after the amounts 16 and 1 respectively. This omission does not change the behaviour of the function however.
   (b) Typo: On line [**167**] `"Renonce"` should read `"Renounce"`
   (c) Typo: On line [**253**] `"bactch"` should read `"batch"`.
   (d) Typo: On line [**364**] `"validatator"` should read `"validator"`
   (e) When doing multiple operations on the same line it is best to specify the intended order of operations using brackets. The if clause on line [**384**] should become `if ((!whiteList[pubKey]) && (validatorDeposits[pubKey] < 16))` and line [**388**] should become `if (whiteList[pubKey] && (validatorDeposits[pubKey] < 1))`.

Recommendations

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

Resolution

The development team has acknowledged issues detailed above and has created a ticket to fix that.

SWL-12	Miscellaneous Gas Optimisations
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section describes general observations made by the testing team in which contract callers are able to generate potential gas savings:

1. **Repeated checks:** In several functions, redundant checks are present. These checks should only occur once per call to save gas. Affected functions include:
   - `swNFTUpgrade.updateIsValidatorsActive()` calls `swNFTUpgrade.updateIsValidatorActive()` which performs the same checks that the owner is the correct `bot` role and also that the contracts are `unpaused` repeatedly.
   - `swNFTUpgrade.updateIsValidatorActiveAndSetRate()` calls `swNFTUpgrade.updateIsValidatorActive()` which performs the same checks that the owner is the correct `bot` role and also that the contracts are `unpaused`.
   - `swNFTUpgrade.addSuperWhiteLists()` calls `swNFTUpgrade.addSuperWhiteList()` which repeats the `onlyOwner` check on each call.
   - `swNFTUpgrade.addWhiteLists()` calls `swNFTUpgrade.addWhiteList()` which repeats the `onlyOwner` check on each call.

   These functions follow a pattern whereby a single call function is called many times in a loop. One possible solution is to merge the single and multi-call functions so that the logic occurs directly inside the multi-call function. The single call functions can then be removed. This would reduce gas usage in multi-loop calls and only slightly increase the gas usage if the array is iterated once as the single call functions originally did.
2. **Unnecessary math checks:**
   - In `swNFTUpgrade.withdraw()` the check on line [**208**] means we can perform the arithmetic operations on line [**209**] in an unchecked block to save gas.
3. **Calling length in loops:**
   - In several functions of `swNFTUpgrade.sol`, the limit of a loop is the length of an array. To save gas this length should be called once and stored in a local variable rather than called every iteration of the loop. This is seen on line [**97**], line [**112**], line [**143**], line [**257**] and line [**293**].
4. **Public functions never called internally:**
   - Several functions of `swNFTUpgrade.sol` are marked as public visibility but are never called by any other functions and so can be marked as external visibility to save gas when called. This affects `tokenURI()`, `pause()` and `unpause()`.

Recommendations

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

Resolution

The development team has acknowledged issues detailed above and has created a ticket to fix that.

Appendix A Test Suite

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

```
test_toHex                                PASSED  [5%]
test_pubKeyToString                       PASSED  [11%]
test_bytesToBytes16                       PASSED  [17%]
test_initializer                          PASSED  [23%]
test_zero_address_reverts                 PASSED  [29%]
test_zero_amount_reverts                  PASSED  [35%]
test_stake_not_whitelisted                PASSED  [41%]
test_stake_event_arg_hashed               XFAIL   (Index...)[
test_stake_not_whitelisted_wrong_amount   PASSED  [52%]
test_stake_super_whitelist_user           PASSED  [58%]
test_stake_non_operator_vuln              XFAIL   (Non-...)[
test_deposit_before_stake_vuln            XFAIL   (In...)[
test_deposit                              PASSED  [76%]
test_withdraw                             PASSED  [82%]
test_enterStrategy                        PASSED  [88%]
test_exitStrategy                         PASSED  [94%]
test_batch_action                         PASSED  [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.

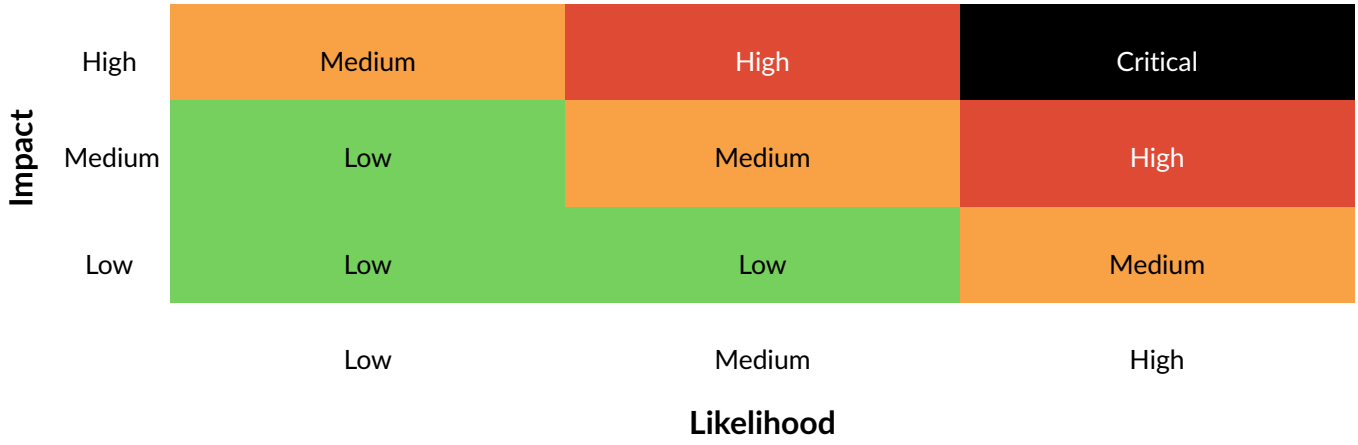

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].