NEAR Protocol

Initial Smart Contracts

Security Assessment

Version: 2.0

June, 2020

Contents

Introduction

NEAR is a blockchain leveraging a proof-of-stake consensus algorithm and a sharding design to address some of the scalability and performance challenges faced by existing blockchain networks.

Sigma Prime performed several security reviews (in Q4 2019 and Q1 2020) targeting NEARCore, the official reference implementation of NEAR, developed in Rust.

Sigma Prime was commercially engaged to perform a security asssessment of a set of initial smart contracts, developed in Rust and powered by the NEAR platform.

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 software assessed. 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 NEARCore along with a brief overview 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*.

Finally, the Appendix section provides additional documentation, including the severity matrix used to classify vulnerabilities within the initial NEAR smart contracts.

Overview

NEARCore is the official implementation of NEAR protocol, a next-generation blockchain network focussed on sharding, developed in Rust. NEAR Protocol uses an asynchronous sharded runtime.

NEAR allows developers to write, test and deploy scalable and decentralized applications using existing high-level programming languages that compile down to WebAssembly, such as Typescript and Rust, using `near-bindgen`, a Rust library for writing smart contracts.

NEAR also provides an online Integrated Development Environment (IDE), called NEAR Studio, enabling developers to leverage an embedded deployment process.

The following initial NEAR smart contracts, developed in Rust, are included in the scope of this assessment:

- **Fungible token:** Example implementation of a standard interface for fungible tokens allowing for ownership, escrow and transfer, specifically targeting third-party marketplace integration (NEP #21). Similar to the Ethereum ERC20 standard.

- **Staking pool:** Smart contract facilitating the delegation of staking to a single validation node. This contract defines three different roles (keyless contract account, a privileged `owner`, and standard delegator accounts).
- **Lockup:** Smart contract acting as an escrow that locks and holds an owner's tokens for a lockup period. This contract can provide a vesting schedule to enforce a vesting agreement (typically between the NEAR Foundation and an employee).
- **Whitelisting:** Smart contract maintaining a whitelist of the staking pool contracts account IDs that are *officially* approved by the NEAR Foundation.
- **Staking pool factory:** Factory smart contract allowing the automatic deployment and whitelisting of new staking pool contracts. Embeds the staking pool smart contract.

Security Assessment Summary

Assessment Scope

This review was initially conducted on the following commits:

- `fungible-token` : commit add8017
- `lockup` : commit a9a15ad
- `staking-pool` : commit 56e9a9c
- `staking-pool-factory` : commit 4db54f3
- `whitelisting` : commit 4db54f3

Assessment Methodology Overview

The testing team performed a time-boxed manual review of the relevant codebases (see repositories above), focusing on identifying smart contract vulnerabilities such as integer overflows, unexpected panics, denial of service conditions, busines logic flaws, etc.

Additionally, the testing team produced fuzzing targets targeting the smart contracts in scope using libFuzzer, a library part of the LLVM project, enabling coverage-guided fuzz testing. This library focuses on:

- **In-process fuzzing**: the fuzzing engine executes the target many times with multiple data inputs in the same process. It must tolerate any kind of input (empty, huge, malformed, etc);
- **White-box fuzzing**: libFuzzer leverages compiler instrumentation and requires access to the source code;
- **Coverage-guided fuzzing**: for every input/test case, libFuzzer tracks code paths (sections of the code reached), and produces variants of each test case to generate additional input data to increase code coverage.

In addition to these fuzzing targets, the testing team developed several tests written for the purpose of this assessment, located inside the `tests/` folder.

During the course of this assessment the testing team identified a total of eleven (11) issues during this assessment, of which:

- One (1) is classified as critical risk,
- One (1) is classified as high risk,
- Two (2) are classified as medium risk,
- Two (2) are classified as low risk,
- Five (5) are classified as informational.

These issues have all been acknowledged and addressed by the development team.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the the scope of this assessment. 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.

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
NSC-01	Crash from Validator Deletion	Critical	Resolved
NSC-02	Initialisation of Contracts is Vulnerable to Front-running	High	Resolved
NSC-03	Storage Fees Can Lead to a Denial-of-Service Condition	Medium	Resolved
NSC-04	Removal of all Main `PublicKeys`	Medium	Resolved
NSC-05	Allowance Front-Running	Low	Resolved
NSC-06	Terminate Vesting Deficit Includes Storage Fees	Low	Resolved
NSC-07	Missing Account ID Checks for `get_balance`	Informational	Resolved
NSC-08	Transfer to Oneself Possible	Informational	Resolved
NSC-09	Overflows Not Set to Panic	Informational	Resolved
NSC-10	Allowance Can Be Set Larger Than Balance	Informational	Closed
NSC-11	Miscellaneous General Comments	Informational	Resolved

NSC-01	Crash from Validator Deletion		
Asset	`NEARCore`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A validator is able to stake and unstake using the `Stake` action. This allows an account to become a validator. By staking with an amount greater than zero, an account is asking to become a validator (or adjust their stake amount). By staking with zero as the amount, a validator is asking to stop being a validator (i.e. unstake). The process of becoming a validator requires a proportion of their account balance to be locked.

Becoming a validator or exiting from being a validator takes 3 epochs to prevent malicious actions, with the associated funds being locked during this period of time. As a result funds will be locked for 3 epochs before becoming a validator and funds will be locked for 3 epochs after unstaking.

Users are able to delete their own account with the condition that their locked balance is zero.

The following steps will set an accounts locked balance to zero while there is a pending validator proposal:

1. Become a valid staker, i.e. stake above the minimum amount with a valid public key and wait 3 epochs;
2. Unstake (i.e. stake 0);
3. Wait 1 epoch transition;
4. Stake 1 NEAR;
5. Wait 2 epoch transitions.

Currently the locked balance will be set to zero, which allows for the deletion of the account. However, there is an existing validator proposal (i.e. Stake 1 NEAR) which will attempt to be processed on the next epoch transition.

The result is the next epoch transition will attempt to read the staked account from storage which will not exist as it has been deleted. Thus, any node attempting to process this epoch transition will panic and will be unable to restart.

Although this asset, `nearcore`, is out of the scope of this review, the bug was found during the process of testing the contracts and thus has been raised in this report.

Recommendations

We recommend patching the locked amount to handle the case where there is and existing stake 0 call and prevent deletion in this particular scenario.

Resolution

A hotfix has been implemented in commit 9e9db8e.

An additional issue has been raised for a more thorough, long-term fix.

NSC-02	Initialisation of Contracts is Vulnerable to Front-running		
Asset	`lockup` , `fungible-token` & `staking-pool`		
Status	**Resolved:** Resolved in commit 4a45d6c		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There are two actions required to deploy and initialise a contract. The first is the deployment of the code which involves uploading the contract byte code (`.wasm` file). The second is the intialisation function (e.g. `new()` or `default()`), which sets the account state storage based on the code in the contract. The initialisation action may be called from any account.

If these actions are not part of a single transaction, it is possible to front-run the initialisation transaction.

When an attacker witnesses a transaction to deploy the contract they may wait for the execution of the transaction, then front-run the initialisation transaction (i.e. the call to `new()`) with their own parameters.

Front-running of transactions in this scenario is trivial for any block producer as they may decide the order of the transactions inserted into a new block and may always create the following scenario:

1. Deployment transaction;
2. Malicious initialisation transaction;
3. Owner initialisation transaction.

In the case of `lockup` contract, an attacker can front-run the call to `new()` . This will allow the attacker to set the `vesting_schedule` , the `initial_owners_main_public_key` and the transfers to be enabled. The attacker may then transfer the entire balance of the contract to their personal account.

In the case of `staking-pool` contract, an attacker can front-run the call to `new()` . This will allow the attacker to set the `owner_id` parameter, gaining ownership of the contract. The contract is required to be initialised with an initial balance which cannot be attached to the call to `new()` as it is not payable. The attacker will therefore receive all staking rewards fees associated with the contract, as they may set the reward percentage to 100%. Therefore, the fees received from this initial balance will be rewarded to the new owner.

In the case of the `fungible-token` contract, the front runner can here again front-run the `new()` transaction call. The attacker may also set the `owner_id` parameter which will allocate them the entire token supply.

Recommendations

Ensure there is adequate documentation in all cases to warn users against the dangers of using multiple transactions. When deploying new contracts, developers should always initialise contracts in the same transaction.

Resolution

This issue has been resolved by updating the relevant documentation in commit 4a45d6c.

NSC-03	Storage Fees Can Lead to a Denial-of-Service Condition		
Asset	`fungible-token`		
Status	**Resolved:** Resolved in commit 8d8931e		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

In the `fungible-token` contract, transferring tokens to a new account will insert a key value pair into the `accounts` map in `FungibleToken`. The key is a `sha256` hash of the account id and a new `Account` object as the value. The `Account` object contains a `Balance` and another `Map` which is a key pair of a `sha256` hash of an account id and a `Balance`.

This implies a large quantity of storage is required to create a single account. Accounts can be created from the transfer of a single token which may have negligible value depending on the usage. At the cost of the transaction gas and a single token, per account, the attacker can generate a large quantity of storage by creating numerous accounts. As a result the contract will face high storage fees.

Alternatively, an attacker can create numerous entries into the `allowances` map requiring only a single token to create practically infinite allowances. The result is similar in that a large quantity of storage will be used by the contract at the cost of the transaction gas, resulting in high storage fees for the contract.

Note that these accounts only need to have valid IDs (i.e. they do not need to actually exist on the Blockchain).

Extra Account Storage Calculations

The gas used by the transaction was $gas = 6.9 * 10^{12}$.

The price was $price = 5,000$ yocto per gas *(Note: using current testnet prices which are liable to change)*.

The total costs was $total\_cost = gas * price = 3.5 * 10^{16}$ yocto.

The amount of storage increase from the transaction was $storage\_bytes = 365$ bytes.

Storage fees require a minimum of 1 NEAR per 10 KiB of data, which equates to $min\_storage\_requirements = 365 * 10^{24}/(10 * 2^{10}) = 3.6 * 10^{22}$ yocto per new account.

As the cost to the attacker vs the cost to maintain the contract is approximately $cost\_ratio = total\_cost/min\_storage\_requirements = 1/10^{6}$.

The attack is therefore cheap and feasible, assuming the value of a single token is negligible.

Note that the gas rebate is less than the transaction gas fee and so will be negligible relative to the storage fee increase. Furthermore, this attack can be made significantly more efficient by including multiple actions in a single transaction.

Refer to this transaction for details.

As an example, an attacker could create 277,777 accounts at a cost of $3.5 * 10^{16} * 277777 = 9.7 * 10^{21}$ yocto or 0.0097 NEAR.

The contract would be required to maintain a balance of $277,777 * 3.6 * 10^{22} = 10,000$ NEAR.

Extra Allowance Storage Calculations

The gas used by the transaction was $gas = 6.6 * 10^{12}$.

The price was $price = 5,000$ yocto per gas *(Note: using current testnet prices which are liable to change)*.

The total costs was $total\_cost = gas * price = 3.3 * 10^{16}$ yocto.

The amount of storage increase from the transaction was $storage\_bytes = 331$ bytes.

Storage fees required is $min\_storage\_requirements = 331 * 10^{24}/(10 * 2^{10}) = 3.2x10^{22}$ yocto for each new account.

Similarly, the cost ratio is approximately $cost\_ratio = total\_cost/min\_storage\_requirements = 1/10^{6}$. The attack is therefore cheap and feasible, assuming the value of a single token is negligible.

See this transaction for details.

Recommendations

The mainnet gas price is currently set to $100,000$ yocto per gas which would make the cost ratio $1/(5 * 10^{4})$ implying this is still a cheap and feasible attack.

To mitigate this type of attack, the cost of creating a new account can be increased. This may occur by any of the following:

- The contract charging a fee for an account creation;
- The contract burning an additional quantity of gas when creating a new account;
- The contract only allowing insertions of new accounts that exist;
- The network increasing gas price charged;
- Reducing storage fee requirements (could have undesirable side effects).

Resolution

This issue has been resolved in commit 8d8931e.

NSC-04	Removal of all Main `PublicKeys`		
Asset	`lockup`		
Status	**Resolved:** Resolved in commit 9e1ba55		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In the `lockup` contract, main public keys are used by the owner to operate certain owner functions such as adding and removing other keys and transferring funds.

The current owner is able to remove all keys including their own key, leaving the contract with no remaining main keys. This would effectively lock any remaining balance in the contract.

An accidental case where this occurs would be when a user sends two transactions, one to add a new key and the other to remove the existing key. If these two transactions are executed in the wrong order (i.e. the deletion first). The contract will be locked.

Recommendations

To add a level of protection to accidental deletion it is recommended to either:

- Prevent deletion of the signing key;
- Ensure there is at least one main key in the list of keys.

Resolution

This issue has been resolved in commit 9e1ba55.

NSC-05	Allowance Front-Running		
Asset	`fungible-token`		
Status	**Resolved:** Resolved in commit 4ecb328		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Front-running attacks [**?**, **?**] involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions to be executed before the original transaction.

Furthermore, front-running can simply be done by block producers by inserting their transaction before the original transaction when producing a block.

The front-running vulnerability exists in the `set_allowance()` function.

Consider the following scenario:

1. Alice approves Malory to spend 1000 tokens, by calling the `set_allowance()` function on the token contract;
2. Alice wants to reduce the allowance, from 1000 tokens to 500 tokens, and sends a new transaction to call the `set_allowance()` function, this time with `allowance` as `500`.
3. Malory watches the blockchain transaction pool and notices that Alice wants to decrease the allowance. Malory proceeds with sending a transaction to spend the 1000 tokens, which if mined before Alice's second transaction;
4. Alice's second `set_allowance()` happens, effectively providing Malory with an additional allowance of 500 tokens;
5. Malory spends the additional 500 token. As a result, Malory spent 1500 tokens, when Alice wanted her to only spend 500 tokens.

Recommendations

Be aware of the front-running issues in `set_allowance()` and potentially add extended allowance functions which are not vulnerable to the front-running vulnerability.

The functions `increase_allowance()` and `decrease_allowance()` may be used to prevent double spending of the allowance, however decrease will still be vulnerable to front-running the amount to be decreased.

A further method typically used to address this attack vector is to force users to change the allowance to 0 first, before updating it to the desired value (requires two separate transactions). See the `safeApprove()` function in OpenZeppelin's `SafeERC20` contract.

Resolution

This issue has been resolved in commit 4ecb328.

NSC-06	Terminate Vesting Deficit Includes Storage Fees		
Asset	`lockup`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In the `lockup` contract, a vesting schedule is able to be terminated by the NEAR Foundation if a NEAR Foundation `AccountId` is provided, with the remaining unvested balance to be withdrawn by the NEAR foundation. If the contract does not have enough balance to cover the unvested amount, it assumes that the difference is in the staking pool and attempts to withdraw that.

The contract uses the minimum of the remaining unvested amount and `get_account_balance()` to determine how much can be withdrawn by the NEAR Foundation. However, due to the requirements to maintain a minimum balance for storage fees, `get_account_balance()` subtracts the `MIN_BALANCE_FOR_STORAGE` from the actual account balance.

It is possible to reach the case where all amounts have been withdrawn from the staking pool and there is insufficient account balance to cover the remaining unvested amount and `MIN_BALANCE_FOR_STORAGE`. This case can be reached by terminating a contract before any amounts have been vested.

As a result the NEAR Foundation will not be able to withdraw the entire unvested balance. The contract will be stuck in the `TerminationStatus::ReadyToWithdraw` status, and `get_terminated_unvested_balance()` will be equal to the `MIN_BALANCE_FOR_STORAGE`.

To unlock the contract and withdraw the remaining balance there must be a deposit of `MIN_BALANCE_FOR_STORAGE` made to the contract.

Recommendations

Ensure that the development team is aware of this edge case which may require transferring an additional 35 NEAR to finalise termination of the contract.

Resolution

The development team updated the related documentation to mention this edge case in PR #59.

NSC-07	Missing Account ID Checks for `get_balance`
Asset	`fungible-token`
Status	**Resolved:** Resolved in commit ba21c6c
Rating	Informational

Description

There are currently checks to validate the input accounts for `get_allowance()`. These checks prevent contracts accidentally checking the allowance of an invalid account.

The function `get_balance()` does not posses the same checks. Users may accidently check the balance of an invalid account. The return value of the call would be zero.

Furthermore, there is a comment for `get_allowance()` stating how the allowance may have already changed by the time the return value is read by a contract. The same principle also applies to `get_balance()`, however it is not documented.

Recommendations

While these checks and comments are not necessary, it is recommended to have a consistent behaviour across similar functions.

Resolution

This issue has been resolved in commit ba21c6c.

NSC-08	Transfer to Oneself Possible
Asset	`fungible-token`
Status	**Resolved:** Resolved in commit 1aed4d8
Rating	Informational

Description

The functions `transfer()` and `transfer_from()` send an `amount` of tokens from one account to another. In both cases it is valid to transfer the tokens to and from the same account.

In `transfer_from()` this can be done by setting `owner_id` to be the same as `new_owner_id`.

In `transfer()` this can be done by setting `new_owner_id` to be the same as `predecessor_account_id`.

The net change in the accounts balance will be zero and thus there is no known attack vector using this approach.

Recommendations

It is recommended to ensure that `owner_id` and `new_owner_id` represent different accounts.

Resolution

This issue has been resolved in commit 1aed4d8.

NSC-09	Overflows Not Set to Panic
Asset	`fungible-token`
Status	**Resolved:** Resolved in commit 8d8931e
Rating	Informational

Description

The Rust release profile does not have overflow checks on by default. As a result, any overflows (e.g. addition, subtraction, etc.) will simply *wrap*, potentially creating an inconsistent state.

No scenarios could be found to trigger overflows in the `fungible-token` contract.

Recommendations

The release profile should be updated to panic on overflows, that is, to include `overflow-checks = true`.

Resolution

This issue has been resolved in commit 8d8931e.

NSC-10	Allowance Can Be Set Larger Than Balance
Asset	`fungible-token`
Status	**Closed:** c
Rating	Informational

Description

In the `fungible-token` contract, `FungibleToken::set_allowance()` takes an `allowance` as input which gives another account the ability to spend these funds on their behalf.

There are no checks to ensure the current account balance of the sender is greater than the allowance. This may lead a user to believe that they have an allowance larger than what is actually available.

Recommendations

This is the same behaviour as the Ethereum ERC20 token specification. As such no action is required other than the developers be aware of this behaviour.

Resolution

The testing team acknowledged the issue and provided the following response:

"Actually we don't want to enforce it. A scenario, where the owner give unlimited allowance to the exchange and later can actively trade the given token with the exchange without having to increase the allowance after every trade."

NSC-11	Miscellaneous General Comments
Asset	
Status	**Resolved:** See ??
Rating	Informational

Description

This section details miscellaneous findings identified by the testing team that do not have a direct security implication:

- `fungible-token`:
  - line [56] of `fungible-token/src/lib.rs` includes a comment with no message. (i.e. `//`).
  - It should be documented that the contract account should be keyless or there is the potential for modification by the owner.
- `staking-pool`:
  - There is no limit to the owner's reward percentage. They may modify it to 100% and claim all rewards.
  - Any transfers made by a user (i.e. using `send`) without a function call will be treated as rewards and distributed to stakers accordingly.
  - If an account calls `withdraw()` and the same account is deleted before the `transfer()` promise has been executed, the transfer will fail but the changes to `StakingContract` will not be reverted. The result is that the fees will be distributed to the remaining stakers as rewards during the next `internal_ping()`.
  - If a validator is slashed, the remaining funds will be locked in the contract until someone sends a sufficient balance for the contract to have `total_balance >= last_total_balance`.
  - line [27]: `/// A type to distinguish between a balance and a staking shares for better readability.` -> `and staking shares`
  - line [32]: The to do, `// TODO: Revert back to 4 once wasm/wasmer bug is fixed.`, may now be updated.
  - line [72]: `Which will not unlock the funds for 4 epochs.` -> `It will not unlock the funds for 4 epochs.`
  - line [146]: `/// It prevents inflation the price of the share too much.` -> `/// It prevents inflation of the price of the share too much.`
  - line [214]: `/// It's only allowed if the 'unstake' action was not performed in the recent 3 epochs.` -> `in the 3 most recent epochs.`
  - line [275] & line [346]: `Calculated staked amount be positive, ...` -> `Calculated staked amount must be positive, ...`
  - line [383]: `https://github.com/nearprotocol/nearcore/issues/2636`, the associated PR has been merged and note can be removed.

- line [495]: `// Checking if we need there are rewards to distribute.` -> `// Checking, if we need, there are rewards to distribute.`
- line [520]: `// Now buying "stake" shares for the contract owner at the new shares price.` -> `at the new share price.`
- line [532]: `// got any shares or not.` -> `has any shares or not.`
- line [617]: `/// Inner method to get the save the given account for a given account ID.` -> `Inner method to save the ...`

- `lockup`:
  - Type `ProposalId` is `u64` as opposed to `U64`.
  - line [9] of `types.rs`: The to do, `// TODO: Revert back to 4 once wasm/wasmer bug is fixed.`, may now be updated.
  - line [1] of `lib.rs`: `//! A smart contract that allows tokens lockup.` -> `allows tokens to be locked up.`.
  - line [50] of `foundation.rs`: `env::panic(b"There are no termination in progress");` -> `There is no termination in progress`.
  - line [69] of `gas.rs`:

    `/// Gas attached to the inner callback for processing result of the unstake call to the` has a double space after call.
  - line [26] of `owner_callback.rs`:

    `/// Called after a deposit amount was transferred out of this account to the staking pool` is missing a full stop.
  - line [48] of `owners.rs`:

    `// staking pool, but it's on the owner. The contract doesn't care about leftover`

    -> `// staking pool, but it's attributable to the owner. The contract doesn't care about leftovers.`

- `staking-pool-factory`:
  - line [4] of `README.md`: `It allows any user to create an new whitelisted staking pool.` -> `...create a new ...`.
  - line [20] of `README.md`: Missing an extra slash in `// contract`.

Recommendations

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

Resolution

These observations have been addressed by the development team in the following pull requests:

- PR #60
- PR #61
- PR #62
- PR #177

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.

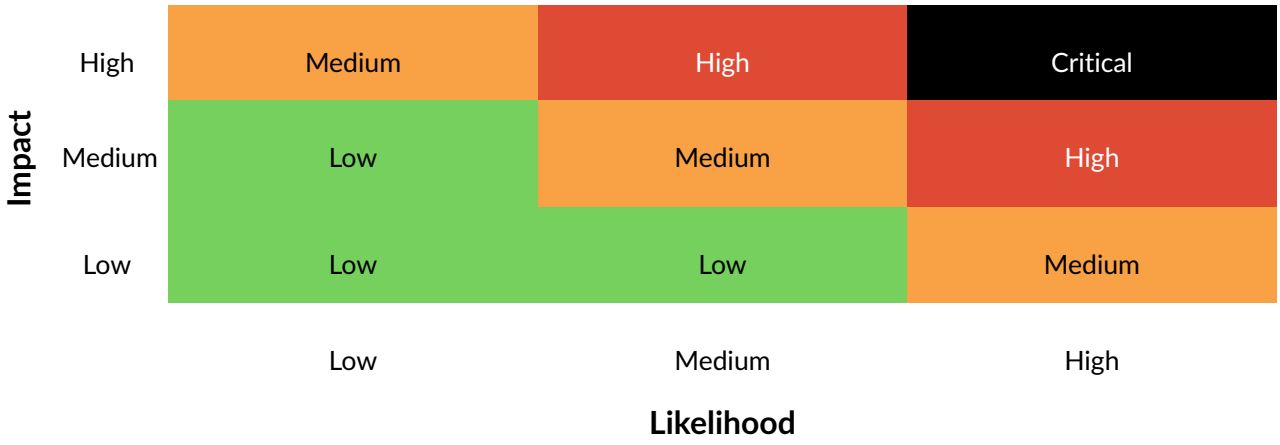

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.