NEAR Protocol

NEARCore Security Review

Version: 1.0

October, 2019

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 was commercially engaged to perform a time-boxed security review of NEARCore, the official reference implementation of NEAR, developed in Rust.

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 the NEARCore client 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 NEARCore Blockchain client.

Overview

NEARCore is the official implementation of NEAR protocol, a next-generation blockchain network focussed on sharding, developed in Rust.

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.

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

Security Assessment Summary

Assessment Scope

This review was initially conducted on the public repository `nearprotocol/nearcore` at commit ce95d67, which contains the following folders:

```
└── nearcore
    ├── async-utils
    ├── chain
    ├── core
    ├── docs
    ├── near
    ├── protos
    ├── pynear
    ├── runtime
    ├── scripts
    ├── target
    ├── tests
    └── test-utils
```

This security assessment targeted exclusively the following components of this repository:

- `chain` : defines the NEAR blockchain logic (types, peer-to-peer stack, serialisation, JSON-RPC definition, syncing mechanism, etc.).
- `near` : implements the NEAR node logic (i.e. `main` function, configuration initialisation command, testnet settings, etc.), defines Nightshade (NEAR *proof-of-stake* consensus mechanism) state transitions, validator management logic, etc.
- `core` : contains the core components used by the NEAR nodes, such as cryptographic primitives (i.e. Ed25519 signing), persistent storage functions, merklization helper functions, etc.

Additionally, this review also targeted the custom serialisation/deserialisation format developed by NEAR Protocol, Borsh (which stands for **B**inary **O**bject **R**epresentation **S**erializer for **H**ashing). Version `0.2.2` was assessed as part of this engagement (commit 556ea4c).

Assessment Methodology Overview

The testing team performed a time-boxed manual review of the relevant codebases, focussing on the following:

- Serialization format resilience to denial of service attacks;
- Block processing and validation;
- Signature verification;
- Underflow/overflow in staking calculations;
- External dependency security posture analysis;

- JSON-RPC input validation;
- Cryptographick key management mechanism;
- Usage of `unsafe` Rust code.

Additionally, the testing team leveraged fuzz testing techniques (i.e. fuzzing), which is a process that allows the identification of bugs by providing randomised and unexpected data inputs to software with the purpose of causing crashes (or in Rust, *panics*) and other unexpected behaviours (e.g. memory leaks).

Sigma Prime produced fuzzing targets targeting NEARCore 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.

These fuzzing targets are available on the testing appendix repository (private repository made available to the testing team). Specifically, the following crates/modules/functions were targeted:

- **near-primitives** | Borsh:
  - `block`, `block_header`, `hash`, `receipt`, `transaction`
- **near-primitives** | Serde:
  - `CryptoHashView`
- **near-crypto** | Borsh:
  - `PublicKey`, `Signature`
- **near-crypto** | Serde:
  - `PublicKey`, `Signature`, `SecretKey`
- **near-chain**:
  - `process_block()`
- **near-network**:
  - `bytes_to_peer_message()`

Please note that significant modifications to the `nearcore` codebase (on the testing appendix repository) have been made to optimise the fuzzing effort. Specifically, the signature verification function has been patched to always provide a valid return value, increasing the fuzzing speed and code coverage.

In addition to these fuzzing targets, the testing appendix repository also contains several tests written for the purpose of this assessment, located inside the following file:

- `nearcore/near/tests/sigp_test.rs`

A preliminary review was conducted on the network specification proposed in the Pull Request #12. This specification provides a high level overview of APIs and protocols that comprise the NEAR networking stack. This was reviewed for obvious design flaws or inconsistencies. Although the testing team did not identify any major issues with this design, we recommend a more thorough review of the implementation, once it is complete.

During the course of this assessment the testing team identified a total of fifteen (15) issues during this assessment, of which:

- Three (3) are classified as critical risk,
- Three (3) are classified as high risk,
- One (1) is classified as medium risk,
- Four (4) are classified as low risk,
- Four (4) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Fantom 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
NEAR-01	Insecure Deserialization of Network Data	Critical	Resolved
NEAR-02	Inefficient Serialisation Can Lead to Resource Exhaustion Attacks	Critical	Open
NEAR-03	Insecure Dynamic Memory Allocation	Critical	Open
NEAR-04	Usage of Outdated and Vulnerable Rust Crates	High	Open
NEAR-05	Unencrypted Traffic Between NEARCore Nodes	High	Open
NEAR-06	Unnecessary Extensive Permissions for Private Keys	High	Open
NEAR-07	Insufficient Node Metrics Monitoring	Medium	Resolved
NEAR-08	Unnecessary Unsafe Trait Implementations	Low	Open
NEAR-09	Overflow in Calculation of Return Stake	Low	Open
NEAR-10	Unnecessary Unsafe `TrieUpdate` Iterators in Runtime	Low	Open
NEAR-11	Insufficient Unit Testing	Low	Open
NEAR-12	Unnecessary Unsafe Transmutations	Informational	Open
NEAR-13	Unnecessary Use of Unit Struct in `Action` Enum	Informational	Open
NEAR-14	Fork Choice and Validator Management Preliminary Analysis	Informational	Open
NEAR-15	Compiler Warnings on `master` Branch	Informational	Open

NEAR-01	Insecure Deserialization of Network Data		
Asset	Borsch		
Status	**Resolved:** Fixed in commit a3b8ce4		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

NEARCore uses a custom serialisation format, Borsh (which stands for **B**inary **O**bject **R**epresentation **S**erializer for **H**ashing), developed and maintained by the NEAR protocol team.

Borsh deserialization of invalid `enum` `variant`s triggers a panic rather than an error. This behaviour is exploitable over the network, as nodes need to deserialise `PeerMessage`s.

The testing team developed a proof-of-concept exploit to illustrate this issue (please refer to the branch `evil-borsh` on the testing repository) where malicious nodes are able to *crash* other legitimate nodes via a single network packet.

Recommendations

This vulnerability was fixed by the development team during the course of the security testing engagement in Borsh version `0.2.3`, please refer to this pull request and this particular commit for further details.

NEAR-02	Inefficient Serialisation Can Lead to Resource Exhaustion Attacks		
Asset	Borsch		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Certain objects are able to be `BorshDeserialized` from an empty slice of input data. If these items are to be decoded by `Vec<T>` where the first four bytes are the length. The NEARCore nodes will successfully deserialize as many objects as dictated by the length.

This will take both a large quantity of time and memory to create up to 2^{32} instances of an object, likely overflowing memory.

The following types/objects are affected by this issue:

1. `bool`
2. `Ipv4Addr`
3. `Ipv6Addr`
4. Fixed length arrays
5. Unit Struct (e.g. `struct MyStruct {}`)
6. Unit Enum (e.g. `enum MyEnum {}`)
7. Tag only `Enums`

Recommendations

For the first 5 items in the list the length is fixed. Therefore the number of bytes read from the data source should match exactly the length of bytes in the object. For example, a `bool` has a length of one byte and therefore exactly one byte should be read. If there are 0 bytes remaining to be read an error should be returned.

This solution can be implemented by using `reader.read_exact(&mut result)` which will return an error if the number of bytes to be read does not equal the length of `result`.

This solution should also be applied to:

- `String`
- `PublicKey`
- `Signature`
- `Box`

Although these objects first require the decoding of length or other integer and hence will error if an empty array is passed to them. It would be more efficient to ensure they read bytes equal to the number of bytes given by the length/cryptography algorithms.

The unit struct and unit enum also represent a special case where they are serialized/deserialized into an empty array. As a result the above resource exhaustion condition can be applied on each of these objects. Two solutions are recommended in relation to these objects:

- Prevent derive `BorshDeserialization` and `BorshSerialization` of these objects at compile time;
- Serialise / Deserialise the objects into a single byte e.g. `[0]`

A tag only enum does not have any data associated with it and is represented in Rust as a `u8`. When deserialising the variant index in an enum, it is decoded using `read` which allows reading of 0 or 1 byte hence if an empty array is passed it will read successfully and default to `variant_index = 0`. This will successfully decode the enum as the first item in that enum. The variant index should read exactly one byte and return an error if not enough bytes are supplied.

Note this may cause complications with the Unit Enum if one of the two solutions above are not implemented.

NEAR-03	Insecure Dynamic Memory Allocation		
Asset	Borsch		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In NEARCore's serialisation format (Borsh), the first four bytes of a `String` or `Box` represent the length of the object to be deserialised. For these types the length is then used to dynamically allocate an array of bytes.

An attacker can set the length to the max length 2^{32} which would then attempt to allocate an array of 2^{32} bytes (appoximately 4 GB) when deserializing. A NEARCore node with less than 4 GB of free memory would instantly go offline due to insufficient available memory. Effectively, 4 bytes of input data can therefore be used to temporarily take up more than 4 GB of memory.

This operation could also be reproduced to fill out all available memory on a NEARCore node and therefore cause nodes to unexpectedly terminate execution.

Here is an example of bytes representing a serialised `String` that would cause the behaviour described above:

- `[255, 255, 255, 255]`.

Recommendations

Modify the deserialisation format to read one byte at a time and appending to a list to only take up memory equivalent to the number of input bytes.

NEAR-04	Usage of Outdated and Vulnerable Rust Crates		
Asset	NEARCore & Borsh		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The following Rust crates are used by several dependencies in NEARCore and are known to be affected by publicly disclosed vulnerabilities:

- `serde_cbor`: Versions prior to `0.10.2` do not properly check if semantic tags were nested excessively during deserialization. This allows an attacker to craft small CBOR documents that cause a stack overflow. Refer to this release description and to this particular commit for further details.
- `sodiumoxide`: Versions prior to `0.2.5` have a `PartialEq` implementation for `generichash::Digest` that compares `itself` to `itself`. As a result `Digest::eq` always returns `true` and `Digest::ne` always returns `false`. Refer to this pull request for further details.
- `memoffset`: Versions prior to `0.5.0` are affected by a vulnerability that causes traps and/or memory unsafety by zero-initializing references. They also could lead to uninitialized memory being dropped if the field for which the offset is requested was behind a deref coercion, and that deref coercion caused a panic. The version used by NEARCore dependencies is `0.2.1`. Refer to this issue for further details.
- `spin`: Versions prior to `0.5.2` are affected by wrong memory orderings which can violate mutual exclusion assumptions inside the `RwLock` implementation, potentially allowing for two writers to acquire the lock at the same time. Refer to this issue for more information.

While the last two items above do not seem to be directly exploitable, the `serde_cbor` and `sodiumoxide` vulnerabilities could have a dramatic impact for NEARCore nodes if exploited by attackers over the wire.

Recommendations

Upgrade the various crates and external dependencies to use the latest versions of these 4 crates.

NEAR-05	Unencrypted Traffic Between NEARCore Nodes		
Asset	NEARCore		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

NEARCore nodes exchange network traffic in cleartext. The testing team intercepted relevant network packets and observed the lack of any encryption mechanism.

Consensus-related objects shared by NEARCore nodes are therefore prone to *Man-in-the-Middle* attacks, whereby malicious actors with adequate network positioning can intercept and alter the content of TCP packets to potentially cause consensus divergences.

Recommendations

Implement strong transport encryption for all messages exchanged by NEARCore peers. We recommend using the latest version of the Transport Layer Security protocol (`TLS v1.3`). The Rustls Rust library could be leveraged for this purpose *(note that this library has not been security reviewed by Sigma Prime):*

NEAR-06	Unnecessary Extensive Permissions for Private Keys		
Asset	NEARCore		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `near` binary allows users to generate eliptic curves private/public keys pairs Ed25519 for nodes and validators, using the `init` argument.

Issuing this command will create the `/.near` directory, with the following `json` files:

- config.json
- genesis.json
- node_key.json
- validator_key.json

The `node_key.json` and `validator_key.json` contain Ed25519 public and private keys. These key pairs are created and stored with unnecessary extensive file permissions. Indeed, these keys are readable by any user of the system (i.e. `-rw-r-r-` or `0644` permission bits).

Recommendations

Change permissions on private key file generation to `-rw----` (i.e. `0600`).

Refer to the following *Pull Request* from a different Proof-of-Stake Blockchain implementation for an example:

https://github.com/sigp/lighthouse/pull/551

NEAR-07	Insufficient Node Metrics Monitoring		
Asset	NEARCore		
Status	**Resolved:** Fixed in Pull Request #1416		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The NEARCore codebase does not include a way to track relevant metrics for nodes. It is therefore difficult to perform large scale resilience testing and track the effect of various types of failures at the infrastructure and network level.

The testing team has therefore built a monitoring system for NEARCore nodes using Prometheus and Grafana (open source analytics and monitoring solutions). At the time of writing, the following metrics are supported:

- Number of blocks processed;
- Number of blocks produced;
- Block height;
- Number of validators;
- Number of peers;
- Average block processing time;
- Actions processed;
- Active stake validators.

The Pull Request associated with these monitoring metrics is #1416.

Recommendations

We recommend expanding the scope of this monitoring solution by tracking more metrics in Prometheus and adding more dashboards in Grafana.

Furthermore, the development team should also consider wrapping these metrics in a Rust feature that can be turned on or off.

NEAR-08	Unnecessary Unsafe Trait Implementations		
Asset	NEARCore		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There are three `unsafe` implementations of the `Send` and `Sync` traits for the `SyncNetworkRecipient` and `chain::Error` types, which are not required.

Unsafe implementations of Send and Sync have the ability to introduce data races into Rust programs, violating the compiler's guarantee to prevent them statically. A data race in the `sync` module would likely cause the node to behave erratically during syncing, and could cause the node to crash if its data stores became internally inconsistent.

Fortunately, the `SyncNetworkRecipient` type is actually `Sync` because the concrete type stored in the `Box<dyn Sender<M>>` field of `actix`'s recipient is an `actix::AddressSender`, which implements `Sync`. However, the `unsafe` implementation is still fragile and inadvisable, because it relies on internal details of the `actix` library which are subject to change.

Please refer to the following relevant sections of the codebase:

- `SyncNetworkRecipient`: `nearcore/chain/client/src/sync.rs:47`
- `chain::Error`: `nearcore/chain/chain/src/error.rs:141`

Recommendations

Two fixes for `SyncNetworkRecipient` are therefore possible:

- Add a `Sync` bound to the upstream `Sender` trait in `actix`, making the trait object in `Recipient` `Sync`;
- Remove the `Sync` bound from the `SyncNetworkAdapter` trait, as no other code seems to be currently relying on this property.

The second solution is preferable for its simplicity.

In the case of the `chain::Error` implementations, they can be deleted without affecting any of the surrounding code.

NEAR-09	Overflow in Calculation of Return Stake		
Asset	NEARCore		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When performing an epoch transition, NEARCode nodes calculate an amount of stake to be returned to each validator based on stake changes in previous epochs. A check is made on line [309] of `runtime.rs` that would ensure that the calculation of the return stake does not overflow, but in the case where the check fails, execution continues. The test case `stake_invariant` triggers this overflow (causing a panic when run in `debug` mode).

Fortunately, due to the wrapping behaviour of addition and subtraction, when run in release mode the overflow does not affect the final values of the account's stake and amount (line [313] and line [314]). If the calculation on line [310] overflows, then the calculations on line [313] and line [314] will also overflow, with the overflows cancelling each other out.

This prevents the erroneous creation of new tokens out of thin air, which might have been possible if different calculations had been performed with the overflowed value.

Recommendations

Even though this overflow is not exploitable in `release`, we recommend that this reliance on the wrapping behaviour be removed. This could be done either by returning an error when the calculation would overflow, or making changes elsewhere to make the staking invariance error unreachable.

NEAR-10	Unnecessary Unsafe `TrieUpdate` Iterators in Runtime		
Asset	NEARCore		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In the `ext` crate (`nearcore/runtime/runtime/src/ext.rs` on line [107] and line [120], additional references are created from a mutable reference to a `TrieUpdate` using `unsafe` .

This introduces the possibility of undefined behaviour when *safe* methods of this library are called in an unexpected sequence. Routing around the type system to save a few cycles is risky and unnecessary, and could be avoided by using concurrency control on `TrieUpdate` .

Recommendations

Implement concurrency control on `TrieUpdate` via `RwLock` or `Mutex` to provide mutability through a shared reference.

NEAR-11	Insufficient Unit Testing		
Asset	NEARCore		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

During this security assessment, the testing team identified that only a very limited number of unit tests have been implemented by the NEARCore development team.

Unit tests detect changes that may break design and implementation assumptions and help with maintaining and updating the codebase. Unit testing reduces defects in the newly developed features and minimizes the probability of introducing new bugs when updating existing features.

Recommendations

For each crate developed for NEARCore, consider writing extensive tests to cover all edge cases the development team can think of.

NEAR-12	Unnecessary Unsafe Transmutations
Asset	NEARCore
Status	Open
Rating	Informational

Description

In the `codec` crate, (`nearcore/chain/network/src/codec.rs`), the unsafe `mem::transmute` on line [48] is unnecessary and could be replaced by a `u32::from_le_bytes` .

Recommendations

Implement the suggestion above, i.e. replacing `mem::transmute` by `u32::from_le_bytes` in the `codec` crate on line [48].

NEAR-13	Unnecessary Use of Unit Struct in `Action` Enum
Asset	NEARCore
Status	Open
Rating	Informational

Description

The enum `Action` defined in `nearcore/core/primitives/src/transaction.rs` on line [37] contains the tag `CreateAccount` which itself is composed of a unit struct (i.e. empty struct in Rust) called `CreateAccountAction`. This unit struct does not provide any functionality but may cause unnecessary complications in deserialization.

Recommendations

Consider removing the `CreateAccountAction` unit struct and leaving `CreateAccount` as a `tag` only.

NEAR-14	Fork Choice and Validator Management Preliminary Analysis
Asset	NEARCore
Status	Open
Rating	Informational

Description

The validator manager is responsible for maintaining the committee composition across multiple forks. It enables a mapping of a block hash to information about the validators at that specific block hash.

Every time a block is processed by `process_block` in `chain.rs`, the `update_head()` function is called, which will update the head if its total weight (previously verified) is greater than the weight of the current head.

Block weight is computed by the `compute_block_weight()` function in `runtime.rs`. The weight of a block is the length of the chain it's on, plus the sum of the number of approval signatures in each block of the chain.

The approval signatures of a block are approvals for the previous block.

While reviewing the NEARCore codebase, the testing team identified a potential attack vector that consists in creating a fork, adding a significant number of validators and overpowering the original chain. This however seems to be a feature/property of the fork choice algorithm used, rather than an implementation issue.

Recommendations

Review of the fork choice design is out of the scope of this assessment. Validating the exploitability of this potential attack vector would require a deep dive into how quickly validators can be added and start making attestations. This analysis was not performed by the testing team due to the time-boxed nature of this assessment.

This issue is raised an *informational* point to make sure the development team is aware of the limitations of this assessment.

NEAR-15	Compiler Warnings on `master` Branch
Asset	NEARCore
Status	Open
Rating	Informational

Description

The following warnings are raised by the Rust compiler when building the NEARcore packages:

```
warning: field is never used: ‘nodes‘
  --> core/store/src/trie/mod.rs:26:5
   |
26 |     nodes: Vec<(CryptoHash, Vec<u8>)>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

warning: struct is never constructed: ‘TrieMemoryPartialStorage‘
   --> core/store/src/trie/mod.rs:346:1
    |
346 | pub struct TrieMemoryPartialStorage {
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: method is never used: ‘from_recorded_storage‘
   --> core/store/src/trie/mod.rs:580:5
    |
580 |     fn from_recorded_storage(partial_storage: PartialStorage) -> Self {
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

These warnings should be addressed by the development team.

Recommendations

It is good practice to keep the `master` branch free of all warnings.

The testing team understands that NEARCore is still going through significant changes. However, we recommend fixing these warnings. Specifically, for the warnings above, the unused fields, structs and methods should simply be removed.

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.

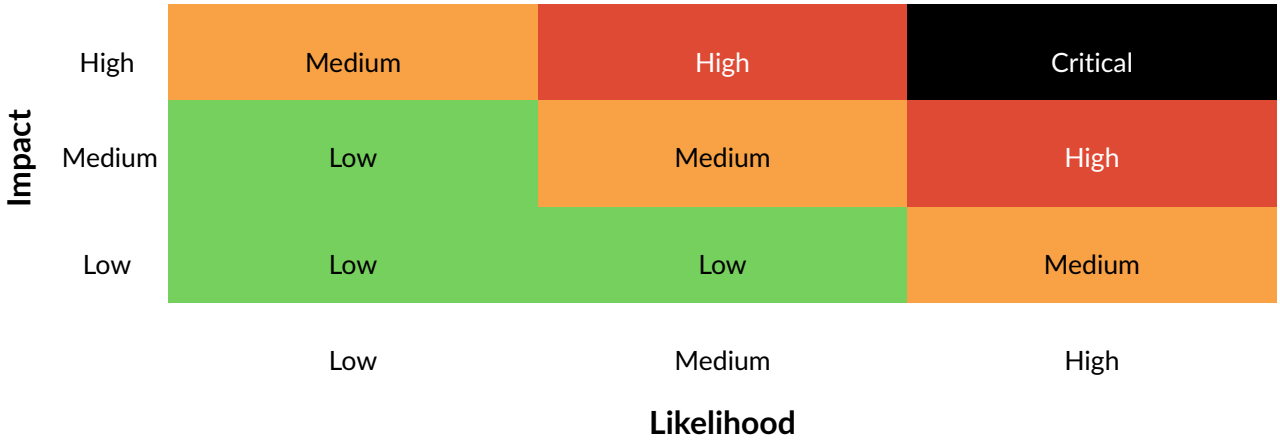

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.