ETHGas

ETHGas Contract Review

Security Assessment Report

Version: 3.1

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the ETHGas smart contracts and Rust components. The review focused solely on the security aspects of the Solidity and Rust implementations of the components in scope, 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 components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the ETHGas components 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 ETHGas smart contracts in scope.

Overview

ETHGas is a marketplace to source and trade blockspace commitments, alongside the Base Fee itself. It operates as a hybrid exchange - a centralised venue, where the central limit orderbook (CLOB) matches buyers with sellers, alongside a non-custodial smart contract where the collateral is held to backstop validator commitments.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the ETHGas Core Contracts, ETHGas Preconf Commit Boost Module and ETHGas AVS Contracts repositories.

The scope of this time-boxed review was strictly limited to files at commit b4f8943, 8f67c3 and 2e19838 respectively.

Retesting activities were performed on commits dd45911, 54990da and bd28876.

Subsequently, `EthgasPool` contract, equivalent to one from the commit bd28876, has been deployed to mainnet at 0x818EF032D736B1a2ECC8556Fc1bC65aEBD8482c5.

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

Approach

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

For the Rust components, the manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team also utilised the following automated testing tools:

- Clippy linting: https://doc.rust-lang.org/stable/clippy/index.html
- Cargo Audit: https://github.com/RustSec/rustsec/tree/main/cargo-audit
- Cargo Outdated: https://github.com/kbknapp/cargo-outdated
- Cargo Geiger: https://github.com/rust-secure-code/cargo-geiger
- Cargo Tarpaulin: https://crates.io/crates/cargo-tarpaulin

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 11 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Low: 8 issues.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the ETHGas components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
ETHG-01	Restricted Hours Implementation Does Not Handle Cross-Midnight Periods	Medium	Resolved
ETHG-02	Improper Error Handling May Lead To Silent Failures	Low	Resolved
ETHG-03	Inconsistent Input Validation	Low	Resolved
ETHG-04	Use Of `Unwrap` For Error Handling	Low	Open
ETHG-05	Checks-Effects-Interactions Pattern Violation	Low	Resolved
ETHG-06	Unsafe `TransferHelper` Implementation Of Token Operations	Low	Resolved
ETHG-07	Insecure URL Concatenation	Low	Resolved
ETHG-08	Potential Denial Of Service (DoS) Via API Calls	Low	Open
ETHG-09	Possible Panic Due To Unchecked Operation	Low	Resolved
ETHG-10	Secure Handling of Stored JWT Tokens	Informational	Open
ETHG-11	Miscellaneous General Comments	Informational	Closed

ETHG-01	Restricted Hours Implementation Does Not Handle Cross-Midnight Periods		
Asset	`EthgasReward.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `EthgasReward` contract implements restricted hours feature that limits certain operations, such as Merkle root updates, to specific hours of the day. However, the current implementation cannot handle periods that cross midnight (e.g. 10 PM to 2 AM).

This limitation appears in two places:

1. The `setRestrictedHours()` function enforces `_start < _end`, which prevents setting periods that cross midnight
2. The `isInRestrictedHours()` function uses a simple comparison that does not account for cross-midnight periods

This issue affects legitimate use cases where operations should be restricted to overnight hours spanning from evening to early morning.

Recommendations

1. Remove the `require(_start < _end, "Start must be before end")` check from `setRestrictedHours()`.
2. Update `isInRestrictedHours()` to handle cross-midnight periods:

```
function isInRestrictedHours() public view returns(bool) {
    uint8 currentHour = uint8((block.timestamp / 3600) % 24);

    // Handle periods that cross midnight
    if (restrictedHourStart > restrictedHourEnd) {
        // For periods like 22-2, check if current hour is >= start OR <= end
        return currentHour >= restrictedHourStart || currentHour <= restrictedHourEnd;
        } else {
            // For normal periods, check if current hour is between start and end
            return currentHour >= restrictedHourStart && currentHour <= restrictedHourEnd;
        }
}
```

This solution allows for setting restricted hours that cross midnight while maintaining the correct logic for determining if the current time falls within the restricted period.

Resolution

The finding has been resolved in commit bd28876.

ETHG-02	Improper Error Handling May Lead To Silent Failures		
Asset	`bin/ethgas_commit.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `Err(err)` branch of `load_commit_module_config::<ExtraConfig>()` logs the error using `eprintln!()`, but does not return an error, leading to silent failures:

```
Err(err) => {
    eprintln!("Failed to load module config: {err:?}");
}
```

Critical error messages are only logged, but the function continues execution instead of terminating properly.

Recommendations

Ensure all critical errors cause the program to terminate cleanly by returning an error, e.g.:

```
Err(err) => {
    error!("Failed to load module config: {:?}", err);
    return Err(err);
}
```

Resolution

The finding has been resolved in commit 54990da.

ETHG-03	Inconsistent Input Validation		
Asset	`EthgasReward.sol, EthgasGateway.sol, EthgasPool.sol, EthgasStaking.sol, configuration/ACLManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

There is an inconsistent input validation implemented across multiple contracts, particularly for critical parameters such as addresses and access control manager instances.

Lack of thorough validation can result in contracts being initialised with invalid or zero addresses, for example:

- `ACLManager` constructor does not validate supplied addresses before assigning them contract admin, treasurer, timelock or pauser roles
- `EthgasPool` constructor does not validate supplied `_aclManager`, `_weth` or `_token` addresses
- `EthgasGateway` constructor does not validate supplied `_aclManager` or `_addrContract` addresses
- `EthgasReward` constructor does not validate supplied `_aclManager`, `_token` addresses or whether `_token` and `_cap` are equal in size.
- `setAclManager()` function implemented in multiple contracts lacks validation of supplied `_aclManager` address
- `setSupportedToken()` function in `EthgasPool` does not validate supplied `_token` addresses
- `claimRewards()` function in `EthgasReward` does not explicitly check if the merkle root is zero before validation.

Ensuring consistent input validation across all contracts is crucial to prevent potential security risks due to improperly initialised parameters.

Recommendations

Implement consistent input validation across all contracts.

Consider creating a common validation library that could be used across all contracts, ensuring consistency.

Resolution

The finding has been resolved in commit bd28876.

ETHG-04	Use Of `Unwrap` For Error Handling		
Asset	`bin/gen_jwt.rs, bin/ethgas_commit.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The code uses `unwrap()` throughout the codebase for error handling. This pattern appears in multiple critical paths, such as:

- Login flow
- Validator registration
- Metrics initialization

Whilst this code operates in a relatively controlled environment, with specific dependencies and known API endpoints, the extensive use of `unwrap()` means that any error condition will cause the entire service to panic, rather than gracefully degrade or recover.

This could impact the service in its entirety and lead to unexpected crashes.

Recommendations

Consider implementing error handling across the codebase using Rust's safer error handling patterns and best practices, such as:

- **`unwrap_or()` and `unwrap_or_else()`**: Provide a default value or fallback computation when unwrapping
- **`match` statements**: Explicitly handle `Ok` / `Err` or `Some` / `None` cases, providing clear error handling and fallback logic
- **`expect`**: If a panic is truly acceptable, use `expect` with a meaningful error message to clarify the reason for the failure
- **Error propagation with `?`**: For functions returning `Result`, use the `?` operator to propagate errors and handle them at a higher level

Resolution

The finding has been partially resolved in commit 54990da.

There are remaining instances where `expect()` is used, which will still result in panics, but with an informative error message.

ETHG-05	Checks-Effects-Interactions Pattern Violation		
Asset	`libraries/TransferFundHelper.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `TransferFundHelper` library violates the Checks-Effects-Interactions (CEI) pattern in its token transfer handling on line [37]:

```
if (isAdmin) {
        TransferHelper.safeTransfer(tt.token, clientAddress, tt.amount);
        emit Withdrawal(clientAddress, tt);
} else if (currentDailyWithdrawalAmount[tt.token] + tt.amount <= dailyWithdrawalCap[tt.token]) {
        TransferHelper.safeTransfer(tt.token, clientAddress, tt.amount);
        currentDailyWithdrawalAmount[tt.token] += tt.amount;    // @audit updated after external call
        emit Withdrawal(clientAddress, tt);
} else {
        revert ExceedDailyWithdrawalCap();
}
```

Note the daily withdrawal amount being updated after the external transfer call.

Whilst the impact is partially mitigated with only trusted actors being able to whitelist tokens, this pattern violation goes against Solidity's best practices.

Recommendations

Restructure the code to follow the CEI pattern by updating the `currentDailyWithdrawalAmount` before making an external call to `safeTransfer()`.

Resolution

This issue has been addressed in commit 9a2f073 by restructuring the code to follow the CEI pattern. The `currentDailyWithdrawalAmount` is now updated before the external call to `safeTransfer()`.

ETHG-06	Unsafe `TransferHelper` Implementation Of Token Operations		
Asset	`libraries/TransferHelper.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The custom `TransferHelper` library implements custom, "safe" versions of standard token operations (e.g. `safeApprove()` or `safeTransfer()`, but fails to account for some edge cases that the standardized library OpenZeppelin's SafeERC20 explicitly handles.

This is problematic because tokens in the wild do not always conform perfectly to the `ERC-20` specification - some may return `false` instead of reverting on failure, others may not return a value at all.

By not addressing these edge cases, `TransferHelper` could incorrectly conclude that a failing operation was successful, which might lead to situations where funds become locked in a contract, transactions silently fail, or malicious actors exploit these inconsistencies to potentially manipulate token transfers.

Recommendations

Replace the custom `TransferHelper` library with OpenZeppelin's `SafeERC20` library.

Resolution

This issue has been addressed in commit 9a2f073 by replacing the custom `TransferHelper` library with OpenZeppelin's `SafeERC20` library.

ETHG-07	Insecure URL Concatenation		
Asset	`bin/ethgas_commit.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `exchange_api_url` is dynamically built using `format!()`, which can lead to unintended malformed URLs or potential injection vulnerabilities if not properly sanitized.

Note, as `exchange_api_base` that is concatenated is user-controlled and obtained from the config file, likelihood of this being exploited is considered minimal.

Recommendations

Use `Url::parse()` from the `url` crate to ensure the URL is correctly formatted and encoded.

Resolution

The finding has been resolved in commit 54990da.

ETHG-08	Potential Denial Of Service (DoS) Via API Calls		
Asset	`bin/ethgas_commit.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function on line [382] attempts to authenticate using the exchange API `exchange_service.login()`, but does not implement rate limiting or error handling for repeated failures.

If the authentication endpoint is unavailable, the application could flood the API with retry requests, leading to account lockouts or IP bans.

Recommendations

Implement exponential backoff with retry limits, e.g.:

```
// @audit This will retry the login up to 5 times with an increasing delay before failing
let exchange_jwt = retry(Fixed::from_millis(500).take(5), || async {
    exchange_service.login().await.map_err(|err| {
        error!(?err, "Service failed");
        err
    })
}).await?;
```

Resolution

The finding has been partially resolved in 54990da.

One issue to note - `wait_interval_in_second` is declared outside the loop, but only initialised inside the `Ok(config)` branch. If `load_commit_module_config` fails on the first iteration, the variable is used uninitialised in `sleep(Duration::from_millis(...))`.

Provide a default value with declaration to avoid its uninitialised use.

ETHG-09	Possible Panic Due To Unchecked Operation		
Asset	`bin/ethgas_commit.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function on line [**365**] uses `B256::from_str(&eoa)` without handling all parsing errors.

```
Ok(eoa) => {
    match B256::from_str(&eoa) {
        Ok(key) => key,
        Err(_) => {
            error!("EOA_SIGNING_KEY environment variable is not a valid 32-byte hex string");
            return Err(std::io::Error::new(std::io::ErrorKind::Other, "Invalid EOA_SIGNING_KEY").into());
        }
    }
},
```

If the `EOA_SIGNING_KEY` is set but incorrectly formatted, `B256::from_str(&eoa)` will panic.

Recommendations

Use the following approach instead to catch the panic earlier:

```
let key = B256::from_str(&eoa).map_err(|_| {
    error!("Invalid EOA_SIGNING_KEY format");
    std::io::Error::new(std::io::ErrorKind::InvalidData, "EOA_SIGNING_KEY format error")
})?;
```

Resolution

The finding has been resolved in commit 54990da.

ETHG-10	Secure Handling of Stored JWT Tokens
Asset	`bin/ethgas_commit.rs`
Status	**Open**
Rating	Informational

Description

Since the application stores JWT tokens in environment variables, special precautions must be taken to prevent unauthorised access, token leakage, and misuse. JWT tokens are sensitive authentication credentials that, if exposed, could allow an attacker to impersonate the client and perform unauthorised actions on the exchange token.

Key risks of insecurely handling or storing JWT tokens include:

1. **Token Theft (MITM Attack):** If intercepted, an attacker can reuse the token until it expires.
2. **Token Leakage in Logs:** If accidentally logged, the token may be exposed.
3. **Replay Attacks:** A stolen JWT can be used repeatedly until it expires.
4. **Hardcoded Tokens:** Storing JWTs in config files or environment variables can expose them to unauthorised access.
5. **Privilege Escalation:** If a compromised JWT has excessive permissions, it can be misused.

Recommendations

Ensure the following security measures are implemented to reduce risk of compromise of JWT tokens:

1. Use HTTPS (TLS 1.2+) for all API calls to prevent interception.
2. Never log JWTs or expose them in error messages.
3. Use short-lived tokens and refresh them periodically.
4. Store JWTs securely using a secret manager (e.g., AWS Secrets Manager, HashiCorp Vault) instead of environment variables, where possible.
5. Follow the principle of least privilege by ensuring JWTs have minimal required permissions.

ETHG-11	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Magic Numbers In Time Calculations**

   Related Asset(s): libraries/TransferFundHelper.sol

   `TransferFundHelper` uses magic numbers on line [32] for time calculations, which reduces code readability and maintainability:

   ```
   if (block.timestamp - lastWithdrawalTime[tt.token] > 86400) {
   ```

   Using hardcoded values goes against coding best practices.

   Define immutable variables for constant values instead.

2. **Incomplete Documentation And Code Organisation**

   Related Asset(s): *.sol, *.rs

   Several issues related to documentation and comments were noted throughout the codebase:

   - Missing exhaustive NatSpec documentation throughout the codebase
   - Inconsistent commenting style throughout the codebase
   - `TODO` comments left in production code
     - In `EthgasStaking` on line [32]
     - In `ACLManager` on line [6]
   - Outdated or unnecessary comments in `EthgasStaking` on line [146]
   - Commented out code in `EthgasGateway` on line [28]

   Consistent and accurate comments and documentation are essential for improving readability and usability, especially for developers and auditors.

   Consider implementing the following:

   - Add comprehensive NatSpec documentation for all contracts and functions
   - Remove `TODO` comments and implement or document the intended functionality
   - Remove outdated or obsolete comments
   - Establish consistent commenting standards

Recommendations

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

Resolution

The above issues have been addressed in commit b646a0e.

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 `forge` framework was used to perform these tests and the output is given below.

```
Ran 9 tests for test/tests-local/EthgasGatewayTest.t.sol:EthgasGatewayTest
[PASS] testRevertWhen_DepositNotGweiMultiple() (gas: 43545)
[PASS] testRevertWhen_DepositTooLow() (gas: 43493)
[PASS] testRevertWhen_InvalidPubkeyLength() (gas: 37108)
[PASS] testRevertWhen_InvalidSignatureLength() (gas: 34964)
[PASS] testRevertWhen_InvalidWithdrawalCredentialsLength() (gas: 39146)
[PASS] test_InitialState() (gas: 12742)
[PASS] test_PauseUnpause() (gas: 66414)
[PASS] test_UpdateACLManager() (gas: 639499)
[PASS] test_ValidDeposit() (gas: 117999)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 27.77ms (5.79ms CPU time)

Ran 5 tests for test/tests-local/ACLManagerTest.t.sol:ACLManagerTest
[PASS] test_AdminCanGrantOtherRoles() (gas: 71399)
[PASS] test_CheckRoleFunctions() (gas: 30276)
[PASS] test_InitialRoles() (gas: 40806)
[PASS] test_RoleCheckReverts() (gas: 141373)
[PASS] test_TimelockRoleAdmin() (gas: 79466)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 105.79ms (3.09ms CPU time)

Ran 11 tests for test/tests-local/EthgasPoolTest.t.sol:EthgasPoolTest
[PASS] testRevertWhen_ExceedDailyCap() (gas: 51719)
[PASS] test_ConvertEthToWeth() (gas: 47337)
[PASS] test_DailyCapReset() (gas: 134728)
[PASS] test_Deposit(uint256) (runs: 1002, μ: 57872, ~: 57906)
[PASS] test_DepositWithETH() (gas: 67610)
[PASS] test_InitialState() (gas: 25657)
[PASS] test_PauseUnpause() (gas: 43032)
[PASS] test_ServerTransferAnyFund() (gas: 78024)
[PASS] test_ServerTransferFundSingle() (gas: 102522)
[PASS] test_SetDailyWithdrawalCap() (gas: 28523)
[PASS] test_SetSupportedToken() (gas: 43392)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 203.06ms (178.54ms CPU time)

Ran 5 tests for test/tests-local/WETH9.t.sol:WETH9Test
[PASS] test_deposit(uint256) (runs: 1002, μ: 55026, ~: 55267)
[PASS] test_depositAndTransfer() (gas: 151842)
[PASS] test_deposit_withdraw(uint256,uint256) (runs: 1002, μ: 69124, ~: 70112)
[PASS] test_transfer(uint256,uint256) (runs: 1002, μ: 104261, ~: 105261)
[PASS] test_transferFrom(uint256,uint256) (runs: 1002, μ: 159799, ~: 160823)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 237.33ms (513.27ms CPU time)

Ran 2 tests for test/tests-fork/WETH9.fork.t.sol:WETH9ForkTest
[PASS] test_depositAndTransfer() (gas: 105988)
[PASS] test_initialBalance() (gas: 8342)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 296.04ms (617.11µs CPU time)

Ran 10 tests for test/tests-local/EthgasStakingTest.t.sol:EthgasStakingTest
[PASS] testRevertWhen_DepositBelowMinimum() (gas: 24584)
[PASS] testRevertWhen_UnauthorizedAdmin() (gas: 49997)
[PASS] testRevertWhen_WithdrawBeforeLockExpiry() (gas: 135773)
[PASS] testWithdrawAfterLockExpiry() (gas: 128083)
[PASS] test_DepositETH(uint256) (runs: 1002, μ: 139779, ~: 139815)
[PASS] test_DepositToken(uint256) (runs: 1002, μ: 291342, ~: 291372)
[PASS] test_DisableLock() (gas: 159915)
[PASS] test_InitialState() (gas: 25659)
[PASS] test_MultipleDeposits() (gas: 159052)
[PASS] test_PauseUnpause() (gas: 46162)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 535.09ms (592.73ms CPU time)

Ran 6 test suites in 542.11ms (1.41s CPU time): 42 tests passed, 0 failed, 0 skipped (42 total tests)
```

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.

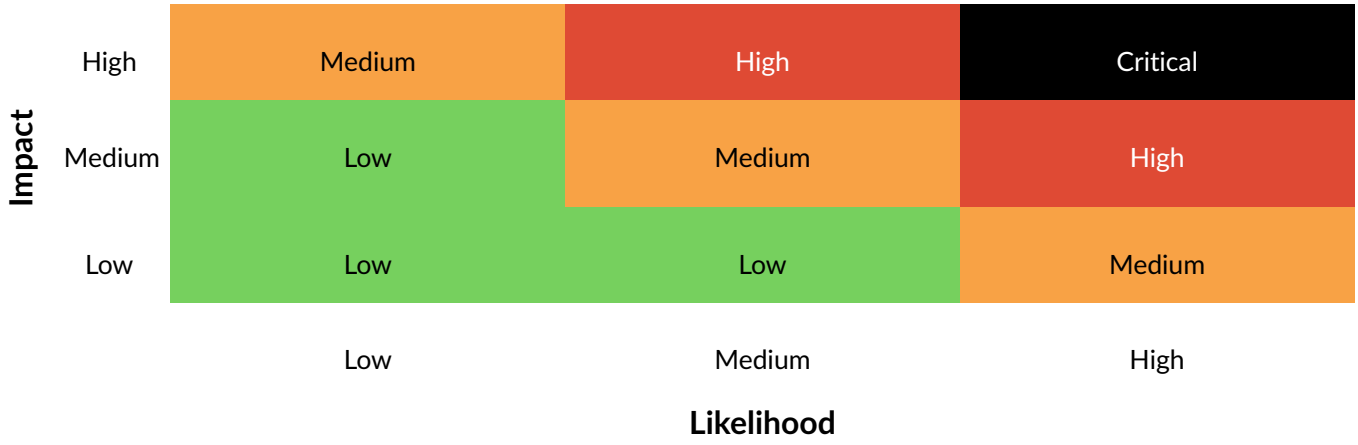

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].