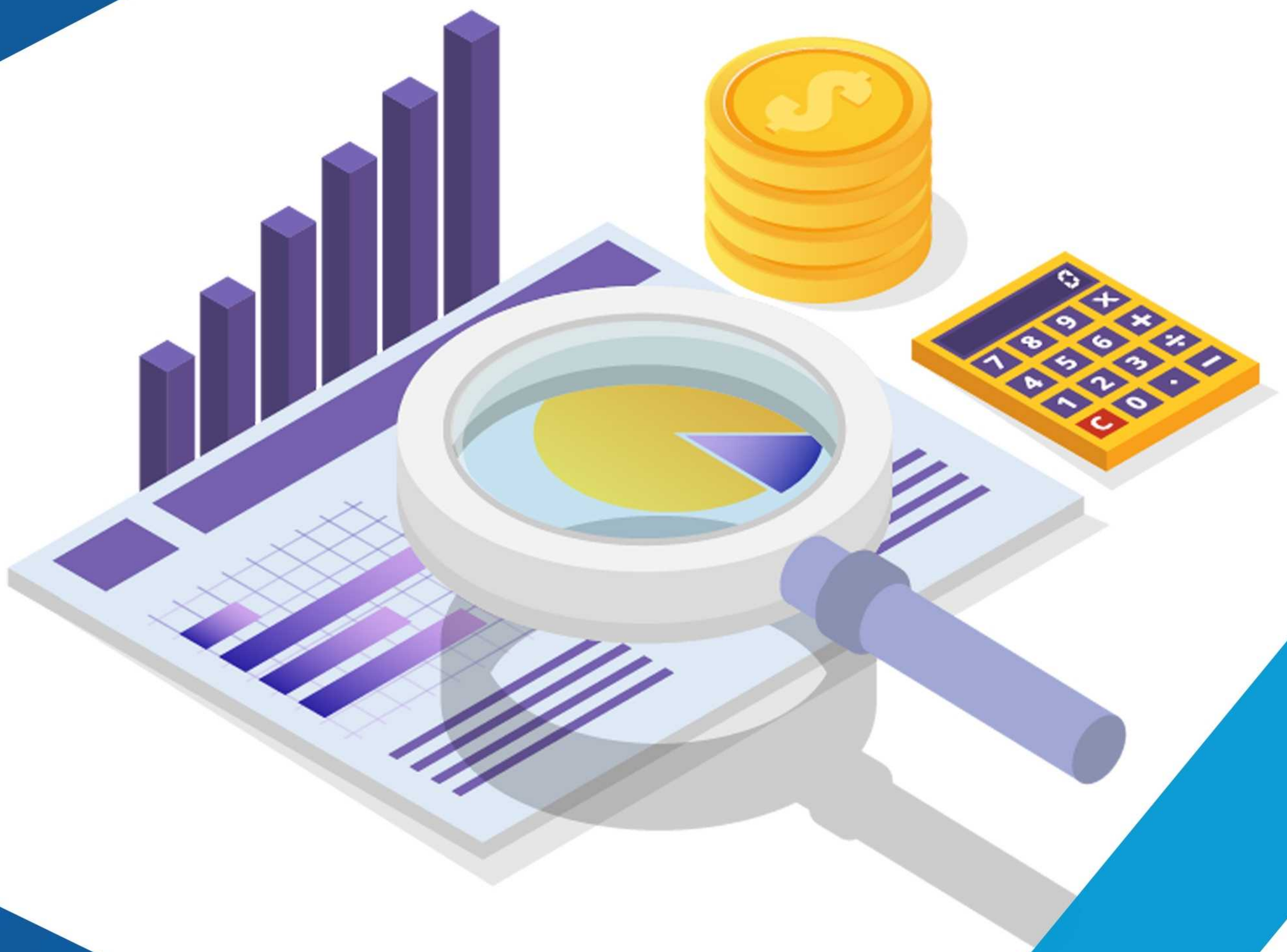
QuillAudits



Audit Report
April, 2021

Contents

Introduction

This audit report highlights the overall security of the HUGO-token contract with commit hash 817bc21. After the audit there were many issues which got closed in commit hash a0b941. With this report, we have tried to ensure the reliability of the smart contract by completing the assessment of their system's architecture and smart contract codebase.

Auditing Approach and Methodologies applied

In this audit, I consider the following crucial features of the code.

- Whether the implementation of token standards.
- Whether the code is secure.
- Whether the code meets the best coding practices.
- Whether the code meets the SWC Registry issue.

The audit has been performed according to the following procedure:

Manual Audit

- Inspecting the code line by line and revert the initial algorithms of the protocol and then compare them with the specification
- Manually analyzing the code for security vulnerabilities.
- Gas Consumption and optimisation
- Assessing the overall project structure, complexity & quality.
- Checking SWC Registry issues in the code.
- Unit testing by writing custom unit testing for each function.
- Checking whether all the libraries used in the code of the latest version.
- Analysis of security on-chain data.
- Analysis of the failure preparations to check how the smart contract performs in case of bugs and vulnerability.

Automated analysis

- Scanning the project's code base with Mythril, Slither, Echidna, Manticore, others.
- Manually verifying (reject or confirm) all the issues found by tools.
- Performing Unit testing.
- Manual Security Testing (SWC-Registry, Overflow)
- Running the tests and checking their coverage.

Audit Details

Project Name: Hugo Finance
Token symbol: HUGO

Languages: Solidity
Platforms and Tools: Remix, VScode and other tools mentioned in the automated analysis section.

Audit Goals

The focus of the audit was to verify that the Smart Contract System is secure, resilient and working according to the specifications. The audit activities can be grouped in the following three categories:

Security
Identifying security related issues within each contract and the system of contract.

Sound Architecture
Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality
A full review of the contract source code. The primary areas of focus include:

- Accuracy
- Readability
- Sections of code with high complexity

Issue Categories

Every issue in this report was assigned a severity level from the following:

High severity issues

Issues on this level are critical to the smart contract's performance/ functionality and should be fixed before moving to a live environment.

Medium severity issues

Issues on this level could potentially bring problems and should eventually be fixed.

Low severity issues

Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	5	0
Next Iteration	0	0	1	0

Manual Audit

SWC Registry test

We have tested some known SWC registry issues. Out of all tests only SWC 102 and 103 are found which is a low priority. We have about it above already.

Serial No.	Description	Comments
SWC-132	Unexpected Ether balance	Pass: Avoided strict equality checks for the Ether balance in a contract
SWC-131	Presence of unused variables	Pass: No unused variables
SWC-128	DoS With Block Gas Limit	Pass
SWC-122	Lack of Proper Signature Verification	Pass
SWC-120	Weak Sources of Randomness from Chain Attributes	Found: No random value used insufficiently (Found in Mythx also)
SWC-119	Shadowing State Variables	Pass: No ambiguous found.
SWC-118	Incorrect Constructor Name	Pass. No incorrect constructor name used
SWC-116	Timestamp Dependence	Pass
SWC-115	Authorization through tx.origin	Pass: No tx.origin found
SWC-114	Transaction Order Dependence	Pass

Serial No.	Description	Comments
SWC-113	DoS with Failed Call	Pass: No failed call
SWC-112	Delegatecall to Untrusted Callee	Pass
SWC-111	Use of Deprecated Solidity Functions	Pass : No deprecated function used
SWC-108	State Variable Default Visibility	Pass: Explicitly defined visibility for all state variables
SWC-107	Reentrancy	Pass: Properly used
SWC-106	Unprotected SELF-DESTRUCT Instruction	Pass: Not found any such vulnerability
SWC-104	Unchecked Call Return Value	Pass: Not found any such vulnerability
SWC-103	Floating Pragma	Fixed
SWC-102	Outdated Compiler Version	Fixed: Latest version is Version 0.8.3. In code 0.8.0 is used
SWC-101	Integer Overflow and Underflow	Pass

High level severity issues

No issues found

Medium level severity issues

No issues found

Low level severity issues

1. **Description** → **SWC 102**: Outdated Compiler Version

```
pragma solidity ^0.8.0;

import './Ownable.sol';

contract HUGO is Ownable {
    /// @notice EIP-20 token name for this token
```

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Remediation
It is recommended to use a recent version of the Solidity compiler which is Version 0.8.3.

Status: Closed

2. Description → SWC 103: Floating Pragma

```
pragma solidity ^0.8.0;

import './Ownable.sol';

contract HUGO is Ownable {
    /// @notice EIP-20 token name for this token
```

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
```

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation
Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.
Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally

Status: Closed

3. **Description: Using the approve function of the token standard [Line 112-115 in HugoToken.sol]**

   The approve function of ERC-20 is vulnerable. Using a front-running attack one can spend approved tokens before the change of allowance value.

```
111     */
112 ▾   function approve(address spender, uint amount) external returns (bool) {
113         _approve(msg.sender, spender, amount);
114         return true;
115     }
116
```

   To prevent attack vectors described above, clients should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. Though the contract itself shouldn't enforce it, to allow backward compatibility with contracts deployed before.

   Detailed reading around it can be found at EIP 20

   Status: Closed

4. **Description: Potential use of "block.number" as source of randomness [line 255, 387, 390 in HugoToken.sol]**

```
253     */
254 ▾   function getPriorVotes(address account, uint blockNumber) public view returns (uint) {
255         require(blockNumber < block.number, "HUGO::getPriorVotes: not yet determined");
256
```

```
385
386 ▾     function _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint oldVotes, uint newVotes) internal {
387 ▾         if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == block.number) {
388             checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
389 ▾         } else {
390             checkpoints[delegatee][nCheckpoints] = Checkpoint(uint32(block.number), newVotes);
391             numCheckpoints[delegatee] = nCheckpoints + 1;
392         }
393
394         emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
395     }
396
```

```
385
386 ▾     function _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint oldVotes, uint newVotes) internal {
387 ▾         if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == block.number) {
388             checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
389 ▾         } else {
390             checkpoints[delegatee][nCheckpoints] = Checkpoint(uint32(block.number), newVotes);
391             numCheckpoints[delegatee] = nCheckpoints + 1;
392         }
```

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

This issue has also been found in automated testing by Mythx attached in the later section.

5. **Description: Prefer external to public visibility level [line 182-187, 211-214, 225-235, 254-286 in HugoToken.sol and 33-35 in Ownable.sol]**

   A function with a **public** visibility modifier that is not called internally. Changing the visibility level to **external** increases code readability. Moreover, in many cases, functions with **external** visibility modifiers spend less gas compared to functions with **public** visibility modifiers. The function definition of **"owner" in Ownable.sol** and the function definition of **“decreaseAllowance”, “delegate”, “delegateBySig”, “getPriorVotes” of HugoToken.sol** is marked as "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as"external" instead

   Recommendations: Use the **external** visibility modifier for functions never called from the contract via internal call. Reading Link.

   Note: Exact same issue was found while using automated testing by Mythx.

Automated Testing

We have used multiple automated testing frameworks. This makes code more secure common attacks. The results are below.

Slither

Slither is a Solidity static analysis framework which runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. After running Slither we got results below.

```
Compilation warnings/errors on HugoToken.sol:
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a c
omment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-I
dentifier: UNLICENSED" for non-open-source code. Please see https://spdx.org for more information.
--> HugoToken.sol
```

Description
No license included in the HugoToken.sol

Recommendation
You need to use a license according to your project. Suggestion to use here would be SPDX license. You can find list of licenses here: https://spdx.org/licenses/

The SPDX License List is an integral part of the SPDX Specification. The SPDX License List itself is a list of commonly found licenses and exceptions used in free and open or collaborative software, data, hardware, or documentation. The SPDX License List includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception.

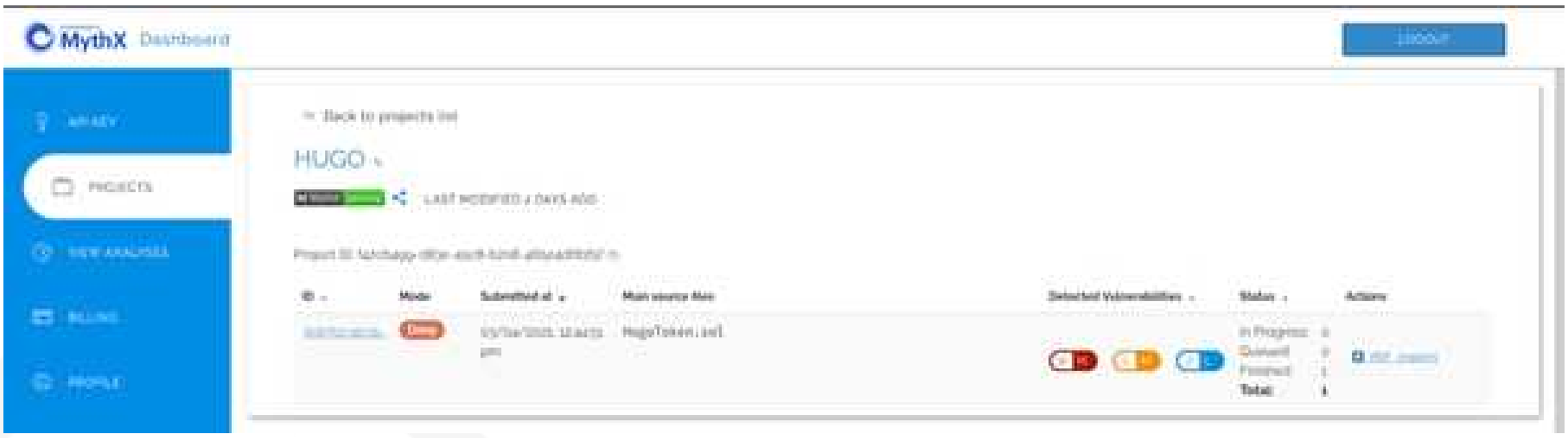
Manticore

Manticore is a symbolic execution tool for the analysis of smart contracts and binaries. It executes a program with symbolic inputs and explores all the possible states it can reach. It also detects crashes and other failure cases in binaries and smart contracts.
Manticore results throw the same warning which is similar to the Slither warning.

Mythx

MythX is a security analysis tool and API that performs static analysis, dynamic analysis, symbolic execution, and fuzzing on Ethereum smart contracts. MythX checks for and reports on the common security vulnerabilities in open industry-standard SWC Registry.

There are many contracts within the whole file. I have separately put them for analysis. Below are the reports generated for each contract separately.



Most of the vulnerabilities generated by Mythx are discussed in Manual sections. Mythx Report contains both Medium and low Severity issues. The pdf copy of the report can be found at this link.

Medium level vulnerability

Function could be marked as external. Already discussed in manual analysis.

Low level vulnerability

- **A floating pragma is set:** Already discussed in manual analysis.

- **Potential use of "block.number" as source of randomness:**
  Already discussed in manual analysis.

- **Loop over unbounded data structure: Line 274**

  uint32 lower = 0;
  uint32 upper = nCheckpoints - 1;
  while (upper > lower) {
  uint32 center = upper - (upper - lower) / 2; // ceil, avoiding
  overflow
  Checkpoint memory cp = checkpoints[account][center];

  Gas consumption in function "getPriorVotes" in contract "HUGO" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the code. Besides, a security audit, please don’t consider this report as investment advice.

Summary

The use of smart contracts is simple and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are a few issues/vulnerabilities to be tackled at various security levels, it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the Hugo team to decide whether any changes should be made.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com