# SMART CONTRACT AUDIT REPORT

BY

## ABDULRASHEED ADEDIRAN

SUBMITTED TO

## QUILLHASH

FOR

## SMART CONTRACT AUDITOR - INTERNSHIP

# Table of Content

# Summary

This audit report presents the results of a detailed review of the QuillHash mock contracts.

**Duration**: 10 days (12th July, 2022 - 22nd July, 2022).

**Github Commit Hash:** `cc15675f8a40b8dd28dfe74255cff59d3c4aa35f`

**Scope**: This audit focused on reviewing the MGGovToken.sol and MockAccessControl.sol contracts from the QuillHash audit mocks repository.

**Objective**: The priority of this audit for each of the reviewed contracts was to;

1. Identify bugs, potential vulnerabilities and demonstrate exploit scenarios.
2. Check for adherence to best practices and optimisation possibilities.

**Audited contracts:**

1. MGGovToken.sol: A governance token.

**Overview**: The codebase was well documented and no critical issues were detected.

Bug Findings:

| **Severity** | High | Medium | Low | Informational |
|---|---|---|---|---|
| **Count** | 0 | 0 | 0 | 4 |

2. MockAccessControl.sol: A CTF kind of challenge.

**Overview**: The smart contract was not properly documented and a few vulnerabilities that potentially expose it to exploitation were detected.

Bug Findings:

| **Severity** | High | Medium | Low | Informational |
|---|---|---|---|---|
| **Count** | | 1 | 1 | 3 |

Artifacts generated from analysis of both contracts with automated tools (Slither and Mythril) have been included in later sections of this report.

### Types Of Severities

Each issue found during the review was assigned a severity level:

**Informational:** Issues that do not pose an immediate risk but are relevant to security best practices.

**Low:** Risks with relatively small impact and are not considered important by the client or can remain unfixed till a later date.

**Medium:** Moderate financial/reputational impact that results from exploiting errors or deficiencies in the smart contract. Issues marked with medium severity should be fixed.

**High:** Serious financial, reputational, or legal risk implications that result from exploiting vulnerabilities in smart contracts. Issues with high severity levels are best fixed before mainnet deployment.

## EVALUATED VULNERABILITIES

Use of best practices

Code documentation

Gas efficiency

Use of tx.origin

Unchecked external functions

Reentrancy

Oracle manipulation

Frontrunning

Timestamp dependence

Insecure arithmetic

Denial of service

Force feeding and other vulnerabilities

## Techniques and Methods

This review was carefully conducted to ensure that each contract's implementation is consistent with the intended functionality, and without unintended edge cases.

### Structural analysis

Each contract was analyzed for conformity to the Solidity structural style guide and Ethereum Natural Language Specification Format (NatSpec) documentation standards.

### Automated tests

Several automated tests were conducted on each contract's functional units with Hardhat-toolbox (Chai, Mocha and Hardhat-Chai-Matchers) for bugs, vulnerabilities and errors.

### Static analysis

Static analysis was conducted on each contract with security analysis tools (Slither and Mythril) to find common vulnerabilities and evaluate code quality.

### Manual analysis

Manual analysis was conducted to check for optimisation possibilities, verify vulnerabilities found and find new vulnerabilities that may have gone undetected by the automated tools.

### Tools and platforms used

Hardhat, Remix, Chai, Mocha, Waffle, Solhint, Mythril, Slither,

# Findings

1. **Contract: MGGovToken.sol**

- **Issue: SPDX license identifier not provided in source file**
- Description: While the absence of a license identifier will not affect the contract's functionality, it is important to have it explicitly defined as it specifies how the code should or should not be used.
- Severity: Informational
- Recommendation: Specify the license used on the first line of the code to ensure compliance with licensing rules.

- **Issue: Compiler version is outdated**
- Description: Using an outdated compiler version (0.6.12) prevents access to updated Solidity security checks. It also exposes the contract to known hacks that have been fixed in later Solidity versions.
- Severity: Informational
- Recommendation: Consider using the latest Solidity compiler version that is trusted and has been thoroughly tested.

- **Issue: `mint()` and `burn()` declared as `public`**
- Description: Declaring a function that is only called externally with the `external` visibility keyword saves gas each time the function is executed. When a function declared as public is called, the function's argument(s) are being copied to memory and this costs gas. With external function calls, arguments are read directly from calldata.
- Severity: Informational
- Recommendation: Functions that are not called internally should be declared with the `external` keyword.

- **Issue**: **Inaccurate comment (contracts/MGGovToken.sol L53)**

  ```
  @notice Delegate votes from `msg.sender` to `delegatee`
  ```

- **Description**: The comment was used to describe function `delegates()`. However, the comment does not reflect what the code currently does. It states that the function delegates votes from `msg.sender` when it does not. The function simply returns the delegates of a given address.
- **Severity**: Informational
- **Recommendation**: Replace the comment with a comment that reflects what the code currently does.

**Functional test:**

```
#mint()
✔ Should mint all tokens to owner and delegate votes to account1 (1139ms)
✔ Should revert if sender is not owner
#burn()
✔ Should burn tokens and deduct delegate's votes of given address (70ms)
✔ Should revert if sender is not owner (42ms)
#delegates()
✔ Should return delegates of given address
#getPriorVotes()
✔ Should revert if votes have not been determined at given block number
✔ Should return prior votes of given address if the given block is valid (44ms)
#events
✔ Should emit `DelegateChanged` when an account changes its delegate
✔ Should emit `DelegateVotesChanged` when a delegate account's vote balance
changes


9 passing (1s)
```

### Static Analysis - Slither:

```
contracts/MGGovToken.sol: Warning: SPDX license identifier not provided in source
file. Before publishing, consider adding a comment containing
"SPDX-License-Identifier: <SPDX-License>" to each source file. Use
"SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see
https://spdx.org for more information.

        - MockGovToken.mint(address,uint256) (contracts/MGGovToken.sol#7-10)
burn(address,uint256) should be declared external:
        - MockGovToken.burn(address,uint256) (contracts/MGGovToken.sol#12-15)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-
could-be-declared-external
```

### Static Analysis - Mythril:

No bugs were detected when the contract was analyzed with Mythril.

2. **Contract: MockAccessControl.sol**

- **Issue: SPDX license identifier not provided in source file**
- Description: While the absence of a license identifier will not affect the contract's functionality, it is important to have it explicitly defined as it specifies how the code should or should not be used.
- Severity: Informational
- Recommendation: Specify the license used on the first line of the code to ensure compliance with licensing rules.

- **Issue: Floating pragma statement**
- Description: The smart contract was deployed with compiler version (^0.8.0). This floating pragma statement is best reserved for libraries and development purposes only as it can potentially expose the smart contract to vulnerabilities from outdated solidity versions. Locking the compiler version ensures that the smart contract does not accidentally get deployed with outdated solidity versions that may introduce bugs.
- Severity: Informational
- Recommendation: Specify the solidity compiler version by locking the pragma statement to a thoroughly tested and trusted compiler version.

- **Issue: `pwn()` uses timestamp for comparisons**
- Description: The `pwn()` function contains a logic where `block.timestamp` was relied upon to evaluate time span for depositing ether. This is very dangerous as miners can manipulate the values to their advantage.
- Severity: Low
- Recommendation: Consider using oracles for time dependent executions.

- **Issue: isContract(address) uses assembly**
- Description: Using `extcodesize` to detect if the caller is a contract is flawed, as a contract will have a code size of zero during deployment. A malicious contract can bypass this check by attacking the vulnerable contract through its constructor thereby gaining access to functionalities that are meant to be inaccessible or protected from unauthorized accounts.

  Exploit scenario:

```
contract AttackMinion {
   constructor( address _minionAddress) payable{
           Minion(_minionAddress).pwn{value: 1 ether/5}();
   }

   receive() payable external{}
}
```

- Severity: Medium
- Recommendation: Use an access modifier on the `pwn()` function to ensure that only authorized accounts can access its functionalities.

- **Issue: The codebase was not properly documented.**
- Description: Absence of comments make testing and readability of smart contracts difficult. In addition to providing more insight to developers, a well documented codebase can also prove useful in giving clarity to users when they interact with the smart contract on the client-side (e.g when signing transactions).
- Severity: Informational
- Recommendation: Consider annotating the codebase with comments that describe the functionalities of the smart contract. It is a good convention to follow the Ethereum Natural Language Specification Format (NatSpec).

**Functional test:**

```
    #pwn()
      ✔ Should revert with error when called with an EOA
      ✔ Should pass when called from a contract's constructor (154ms)
    #retrieve()
      ✔ Should revert with error if sender is not owner
      ✔ Should revert with error if sender has no contribution
      ✔ Should let owner retrieve contributions if balance is > 0 (117ms)

    5 passing (700ms)
```

**Static Analysis - Slither:**

```
——— Starting analysis ———

Minion.pwn() (contracts/MockAccessControl.sol:22-34) uses timestamp for comparisons
      • require(bool,string)(block.timestamp % 120 >= 0 && block.timestamp % 120 <
60,Not the right time) (contracts/MockAccessControl.sol#27)

Minion.isContract(address) (contracts/MockAccessControl.sol:15-21) uses assembly
      • INLINE ASM (contracts/MockAccessControl.sol#17-19)

Pragma version^0.8.0 (contracts/MockAccessControl.sol:1) allows old versions


——— Analysis: 1 succeeded, 0 failed, 0 skipped ———
```

**Static Analysis - Mythril:**

```
==== Dependence on tx.origin ====
SWC ID: 115
Severity: Low
Contract: Minion
Function name: pwn()
PC address: 870
Estimated Gas Usage: 199 - 294
Use of tx.origin as a part of authorization control.
The tx.origin environment variable has been found to influence a control flow
decision. Note that using tx.origin as a security control might cause a situation
where a user inadvertently authorizes a smart contract to perform an action on
their behalf. It is recommended to use msg.sender instead.
--------------------
In file: contracts/MockAccessControl.sol:23
```

```
require(tx.origin != msg.sender, "Well we are not allowing EOAs, sorry")

--------------------
Initial State:

Account: [CREATOR], balance: 0x2, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0
Caller: [ATTACKER], function: pwn(), txdata: 0xdd365b8b, value: 0x0
```

## General Recommendations

Code structure and layout: Consistent code layout aids readability, bug detection and eventually leads to a more secure code. It would be best to restructure the MockGovToken.sol and MockAccessContro.sol contracts to follow the Solidity style guide and structural conventions.

Use of custom errors: Writing checks with `require` statements are quite popular and effective. However, using custom errors may provide a few extra advantages. Unlike `require` statements, custom errors are highly flexible as they allow you to provide more information on a particular error. Consider using custom errors as they are more visible and significantly cheaper to call and deploy.