QUILLAUDITS

SECURITY ASSESSMENT REPORT

MGGovToken

Minion

MockStakingRewards

Overview

Audit summary

Delivery Date	21st of June, 2022
Method Of Audit	Manual review, Automated testing
Timeline Engaged	14th of June - 21st of June, 2022

Vulnerability summary

Total Issues	**13**
Total Critical	**0**
Total Major	**1**
Total Medium	**3**
Total Minor	**2**
Total Informational	**7**

Automated Testing Result

First Contract

Project Name	MockGovToken
Description	Creation of a Governance through delegation.
Codebase	Github
Test commit	MockGovTokenTest.ts

Second Contract

Project Name	MockAccessControl
Description	A CTF kind of challenge
Codebase	Github
Test commit	AccessControlTest.ts

Third Contract

Project Name	MockStakingRewards
Description	A staking rewards contract.
Codebase	Github
Test commit	MockStakingRewardsTest.ts

Scope Of Audit

The audit report was conducted as a detailed analysis of three smart contracts for quality, security and correctness.

Checked Vulnerabilities:

- Re-entrancy
- Storage Layout/ Variable packing
- Dependence on block.timestamp
- Gas Limit and use of for Loops
- Comment format
- Use of tx.origin
- Solidity Version
- Overflow and underflow errors
- ERC20 API violation
- Malicious libraries
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked maths
- Unsafe type inference
- Incorrect use of function visibility

Techniques and Methods:

The audit of the smart contract was taken to ensure overall quality of code, use of best practices, efficient use of gas and security from vulnerabilities. The following techniques were used to review all the smart contracts:

Structural Analysis:

The design structure of the smart contract was analysed to ensure the structure would not cause any future problems

Manual Analysis and Testing:

Manual code review was done, automated testing with slither and functional testing with hardhat, chai and remix were also done to estimate the level of security.

CONTRACT 1: MGGovToken - Governance Token

Imports:
"@pancakeswap/pancake-swap-lib/contracts/token/BEP20/BEP20.sol
";

Functions:

- mint (address _to, uint256 _amount)
- burn (address _from, uint256 _amount)
- delegates (address delegator)
- delegate (address delegatee)
- delegateBySig (address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r, bytes32 s)
- getCurrentVotes (address account)
- getPriorVotes (address account, uint256 blockNumber)
- _delegate (address delegator, address delegatee)
- _moveDelegates(address srcRep, address dstRep, uint256 amount)
- _writeCheckpoint (address delegatee, uint32 nCheckpoints, uint256 oldVotes, uint256 newVotes)
- Safe32 (uint256 n, string memory errorMessage)
- getChainId()

Events:

- event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate)
- event DelegateVotesChanged(address indexed delegate, uint256 previousBalance, uint256 newBalance);

FINDINGS:

1. **Coding Style**

Type	Severity	Location
Coding style	Informational	The whole contract

Description: Coding style and contract layout issues influence readability. According to solidity official docs, the best practice to organise your layout of contract is below:

- Pragma statements
- Import statements
- Interfaces
- Libraries
- Contracts

Inside each contract, library or interface, use following order:

- State variables
- Events
- Function Modifiers
- Struct, Arrays or Enums
- Constructor
- Fallback — Receive function
- External visible functions
- Public visible functions
- Internal visible functions
- Private visible functions

Recommendation: Consider following the solidity guidelines on contract structure layout and commenting for all files.

2. Incorrect Public Visibility

Type	Severity	Location
Gas Optimization	Informational	MGGovToken.sol L6 MGGovToken.sol L6

Description: The functions never called throughout the contract should be limited to the **external** visibility instead of **public**.
This would improve the overall gas optimization of the contract.
The functions `mint()` and `burn()` are marked as **public** but never called throughout the contract.

Recommendation: The functions `mint()` and `burn()` should be marked as **external**

3. Variable Tight Packing

Type	Severity	Location
Gas Optimization	Informational	MGGovToken.sol L24

Description: The variables in struct `checkPoint` are slotted between two uint256 variables unoptimized.

Recommendation: The variables in struct `Checkpoint` could be packed such that they both fit into a single 32-bytes slot.
The variable `votes` in the struct could be changed from a ***uint256*** to a ***uint224*** to accommodate the ***uint32*** variable of `fromBlock` resulting in saving of gas consumption related to usage of one extra slot for storage.

4. Solidity Compiler Version

Type	Severity	Location
Implementation	Minor	MGGovToken.sol L1

Description: Slither detected the solidity compiler version of the contract to be an old version - 0.6.12. Older versions tend to have security-related bugs that would have been fixed in the newer versions.

Recommendation: When deploying contracts, you should use the latest released version of Solidity. Apart from exceptional cases, only the latest version receives security fixes. This reduces the chances of security holes and problems being generated in the bytecode.

5. Syntactic Error

Type	Severity	Location
Performance	Medium	MGGovToken.sol L97 MGGOVToken.sol L64

Description: The functions `delegate()` and `delegateBySig()` both call the internal function `_delegate()` with the return statement. However, the function `_deletgate()` does not return any value. This could throw an error.

Recommendation: The function `_delegate()` should be called in both functions `delegate()` and `delegate()` without using the return keyword.

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

Functions:

- isContract(address account)
- pwn()
- verify(address account)
- retrieve()
- timeVal()

FINDINGS:

1. **`pwn` function can be accessed by contracts**

Type	Severity	Location
Performance	Medium	MockAccessControl.sol L22-L34

Description: The function `pwn()` is not supposed to be accessible by EOAs or Contracts, and to be able to set the mapping `pwned` with an address to true, the address also has to have contributed at least 1 ether, whilst not making contributions of more than 0.2eth and less than 0.1 eth at once.
The function can be accessible by a hacker by them creating a hacker contract, and setting a for loop of at least 5 loops in the constructor and calling the `pwn` function with 0.2eth as its `msg.value`, while also taking note of the time frame where the time Minion's contract's timeVal's value satisfies the condition that the modulus of 120 of the timeVal's value should be between the range of 0 and 60.

Although, this vulnerability is not causing any harm to the contract, because the retrieve function which transfers out funds ensures that only the contract owner can call it, It is still a possible exploit.

Recommendation: A modifier `onlyOwner` should be used instead, on the function `pwn()` to ensure that the function can only be called by the owner.

2. Reliance on block.timestamp

Type	Severity	Location
Performance	Informational	MockAccessControl.sol L27

Description: The contract relies on the block.timestamp to create a time range in which users have access to the function.

Recommendation: Using **block.timestamp** in your contract is unreliable. This is because miners have the ability to change the block.timestamp if necessary. Miners can solve the block, and choose the block.timestamp that falls between `block.timestamp % 120 >= 0 && block.timestamp % 120 < 60` range, and thereby having access to the otherwise protected function. It is recommended to avoid using **block.timestamp** if it is being used to execute anything of high value.

3. Contract is poorly commented

Type	Severity	Location
Comments	Informational	The whole contract

Description: The contract has no comments explaining what each function does, which might make readability for other developers difficult.

Recommendation: Consider following the solidity guidelines on proper commenting of smart contracts using the natspec commenting format.

4. Solidity Compiler Version

Type	Severity	Location
Implementation	Minor	MockAccessControl.sol L1

Description: Slither detected the solidity compiler version of the contract to be an old version - 0.6.12. Older versions tend to have security-related bugs that would have been fixed in the newer versions.

Recommendation: When deploying contracts, you should use the latest released version of Solidity. Apart from exceptional cases, only the latest version receives security fixes. This reduces the chances of security holes and problems being generated in the bytecode.

CONTRACT 3: smockStakingReward.sol - A Staking Rewards Contract

Functions:

- getUserDepositedAmount(address _user)
- getUserPendingReward(address _user)
- getTotalPending()
- getLatestRewardPerShare()
- getLpSupplyAndLastBlock()
- setMaxFundAmount(uint256 _newMaxFundAmount)
- _getLastBlock()
- fund(uint256 _amount, uint256 _rewardPerBlock)
- updatePool()
- deposit(uint256 _amount)
- withdraw(uint256 _amount)
- harvest(address _to)
- _harvest(address _to,uint256 userAmount,uint256 userRewardDebt)
- _mockRewardsTransfer(address _to, uint256 _amount)

Events:

- event Funding(address indexed admin, uint256 amount, uint256 rewardPerBlock, uint256 endBlock)
- event Deposit(address indexed user, uint256 amount)
- event Withdraw(address indexed user, uint256 amount)
- event Harvest(address indexed user, uint256 amount)
- event MaxFundSet(uint256 newMaxFundAmount)

FINDINGS:

1. **Calculations of known values should be done**

Type	Severity	Location
Gas optimization	Informational	MockStakingRewards.sol L409

Description: Variable `MAX_FUND_LIMIT` is assigned to a mathematical calculation with an already known value.

Recommendation: Variable `MAX_FUND_LIMIT` should instead be assigned directly to the value of the multiplication, i.e (15 * 10 ** 24). This would help save some gas during deployment.

2. **Incorrect Public Visibility**

Type	Severity	Location
Gas Optimization	Informational	MockStakingRewards.sol L666

Description: The functions never called throughout the contract should be limited to the **external** visibility instead of **public**. This would improve the overall gas optimization of the contract.
The function `**harvest()**` is marked as **public** but never called throughout the contract.

Recommendation: The function `**harvest()**` should be marked as **external**

3. Contract Balance Check Before Transfer

Type	Severity	Location
Implementation	Medium	MockStakingRewards.sol L714

Description: The private function `_mockRewardsTransfer()` is to send rewards calculated from the user's amount deposited in the contract. The function does not check the balance of the `mockTokenContract` while the function is being called, thereby, if there's no `mockTokenContract` tokens left, the user stake yields no harvests, which defeats the whole point of the contract.

Recommendation: The function `_mockRewardsTransfer()` should first check the balance of the `mockTokenContract` tokens left in the contract, before a transfer is made, and if there are enough no tokens available, the function call should fail with an `insufficient token rewards` error.

4. **Users can make harvests on their funds as many times as possible, without any restraints**

Type	Severity	Location
Implementation	Major	MockStakingRewards.sol L691 MockStakingRewards.sol L666

Description: The public function `harvest()` can be called by a user as many times as possible and reward calculated on their total funds deposited in the contract is transferred to them without any checks or constraints. This could lead to users just syphoning mockTokens out of the contract uncontrollably.

Recommendation: There should be a check on the harvest and withdrawal functions that prevents users from being able to make an harvest everytime the functions are being called, or at least spaces out the intervals between when they can make a harvest on their funds.

Test Suite

- Hardhat
- Chai
- Remix
- Slither

General Recommendation

For smart contracts to meet the standard according to the solidity official docs, the Coding style and contract layout has to be taken into consideration, as this greatly affects the readability of the code. To read up more on the accepted contract layout style, click here. Also, smart contract commenting serves as a guide to your code, as it stands in as a type of yellow paper for your functions for developers reading your code. The recommended commenting style for solidity codes is the natspec commenting format. For more guidance to this commenting format, follow this link.

A passive way to save gas during deployment is to shorten the length of error messages on the `require` statements. A good way to do this, is to have a separate file containing all your error messages, and assigning each error message to a number. This file is not to be deployed as part of the contract, it is just there to serve as a guide for the developers, when assigning the number equivalent of these error messages in your contracts. Another way is to use the solidity custom errors. This is a convenient and gas-efficient way to explain to users why an operation failed. Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries). It is also possible to have these custom errors take in parameters. These custom errors decrease both deploy and runtime gas cost. Link to custom errors.

Lastly, **tx.origin** is a security vulnerability. Contracts using tx.origin as an authorization are prone to Phishing attacks. It breaks compatibility with other contracts including security contracts, and is almost never useful. Removing it would make Solidity more user-friendly. If there are exceptional cases where access to the transaction origin is needed, a library using in-line assembly can provide it.