QuillAudits

Audit Report
May, 2023

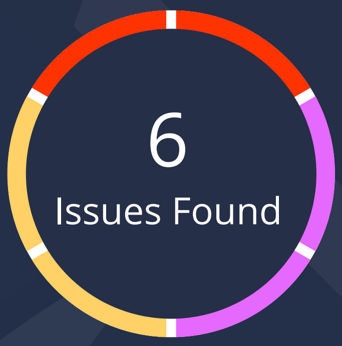
For

VAULTY

Table of Content

Executive Summary

Project Name	Vaulty
Overview	Vaulty allows people to freeze their assets, create trust funds and offers long term holders to rely less on emotional mastery.
Timeline	13th April, 2023 to 25th April, 2023.
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze Vaulty codebase for quality, security, and correctness.
Contracts in Scope	- MultiTokenTimeLockedVault.sol *https://github.com/ashwwwin/VaultyTimeVault* Commit Hash: 089167348f36ca5808c5ecafc3be9685859a3d8e
Fixed In	Branch Name: main Commit Hash: 692800dc83d06102b0b33b2d9ee4ffbb795a38af



🟥 High 🟨 Medium

🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	1	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	1	1	0	2

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Using block.timestamp
- ✓ Multiple Sends
- ✓ Using SHA3
- ✓ Using suicide
- ✓ Using throw
- ✓ Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - MultiTokenTimeLockedVault.sol

High Severity Issues

A.1 Zero fee deposits: *POC*

Description

The protocol generates revenue by collecting fees on deposits to vaults. The deposit fee can be set by the owner at any time, but if not set, depositFee will be 0. This should be a bug unless otherwise required to be a zero-fee vault at some times.

Remediation

This can be fixed by the owner setting the fees immediately the contract gets deployed and including a check in the setDepositFee function to assert the 'newFee' variable is greater than 0. In the deposit function, a require(msg.value > 0, ErrorMessage) would be a useful mitigation factor as well.

Vaulty Team Comment: Not a bug.

Status

Resolved

A.2 Unhandled custom ERC20 tokens

Description

Custom ERC20 tokens could be used to interact with the vault if whitelisted by the vault owner. These custom tokens could affect the internal accounting of the vault. A key example would be 'Fee on Transfer' tokens which can deduct a portion of tokens before or after transfers happen. The vault checks for how much was sent in the deposit() function call but not how much the contract holds and this can be problematic when fees are deducted.

```
uint256 amount = userDeposit.amount;
```

Remediation

The owner should vet the tokens to be whitelisted to ensure there is no accounting mismatch or refactor the code to withdraw the tokens present in the vault for that depositor instead of the amount stored in the Deposit struct.

Status

Acknowledged

Medium Severity Issues

A.3 Missing test cases

Description

The codebase lacks unit test coverage. It is advisable to have test coverage greater than 95% of the codebase to reduce unexpected functionality and help fuzz test as many invariants as possible.

Remediation

Include unit tests for the codebase.

Status

Acknowledged

A.4 Privileged account transfers

Description

If renouncing ownership is not done appropriately, the current contract owner can renounce ownership, lose owner privileges, and lose access to the tokens sent to the contract forever. When ownership is renounced, functions like withdrawFees() will be uncallable and any fees generated from deposits will be inaccessible.

Remediation

Functions such as renounceOwnership and transferOwnership can be overridden or set up for 2-step verification to prevent mistaken privilege transfer or renouncing.

References - Link 1 **|** *Link 2*

Status

Resolved

Low Severity Issues

No issues found

Informational Issues

A.5 Unlocked pragma

Description

The solidity pragma version in this codebase is unlocked.

Remediation

It is advised to use a specific Solidity version when deploying to production to reduce the surface of attacks with future releases which were not accounted for when the contracts originally went live.

Status

Resolved

A.6 Event emission

Description

Some of the contract functions that update the vault's state do not emit events when called. It is advisable for ease of tracking changes on the blockchain to emit events. The following functions could have events emitted when called:

- setDepositFee,
- addToWhitelist,
- removeFromWhitelist,
- setDepositFee,
- deposit,
- withdraw
- withdrawFees

Remediation

Emit events for these state changes for ease of tracking and logging.

Status

Resolved

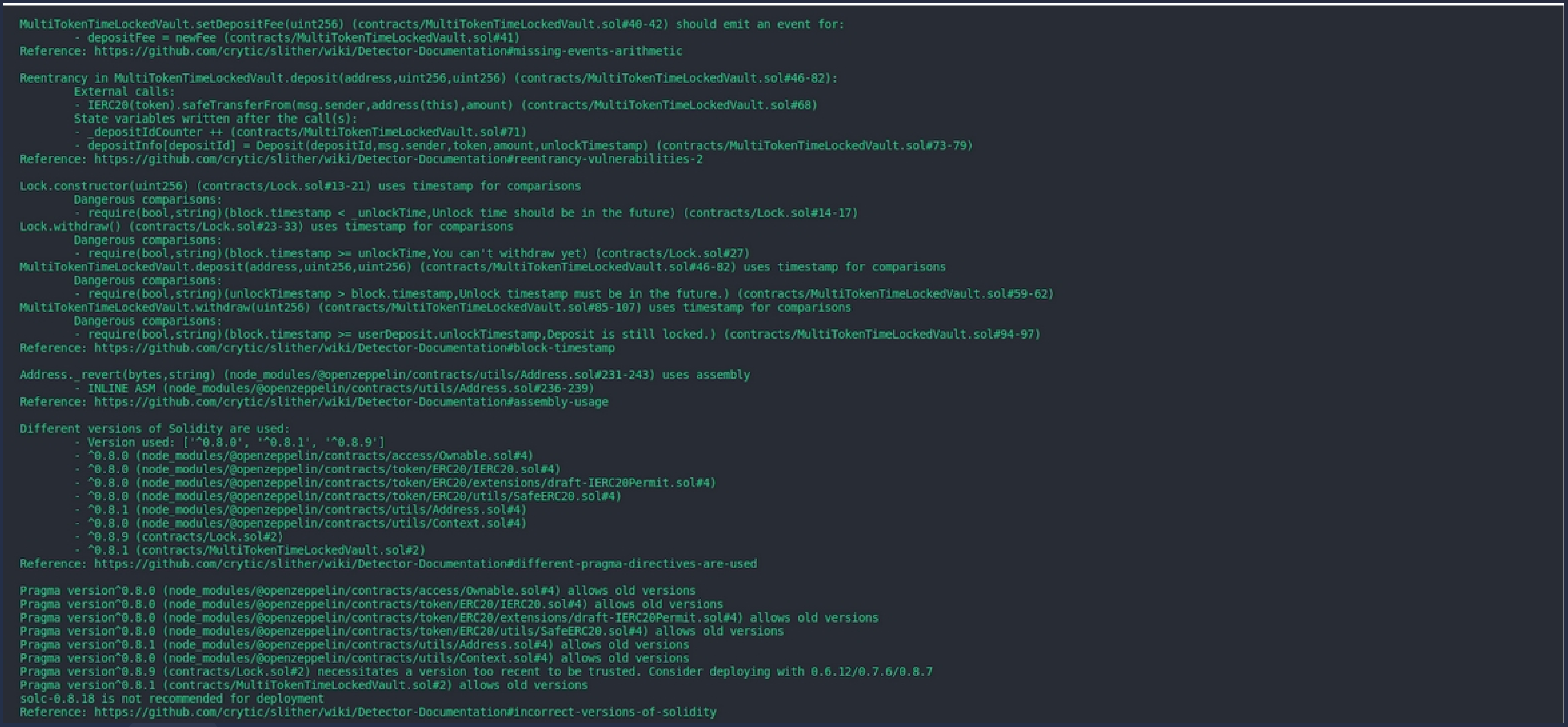
Functional Testing

Some of the tests performed are mentioned below

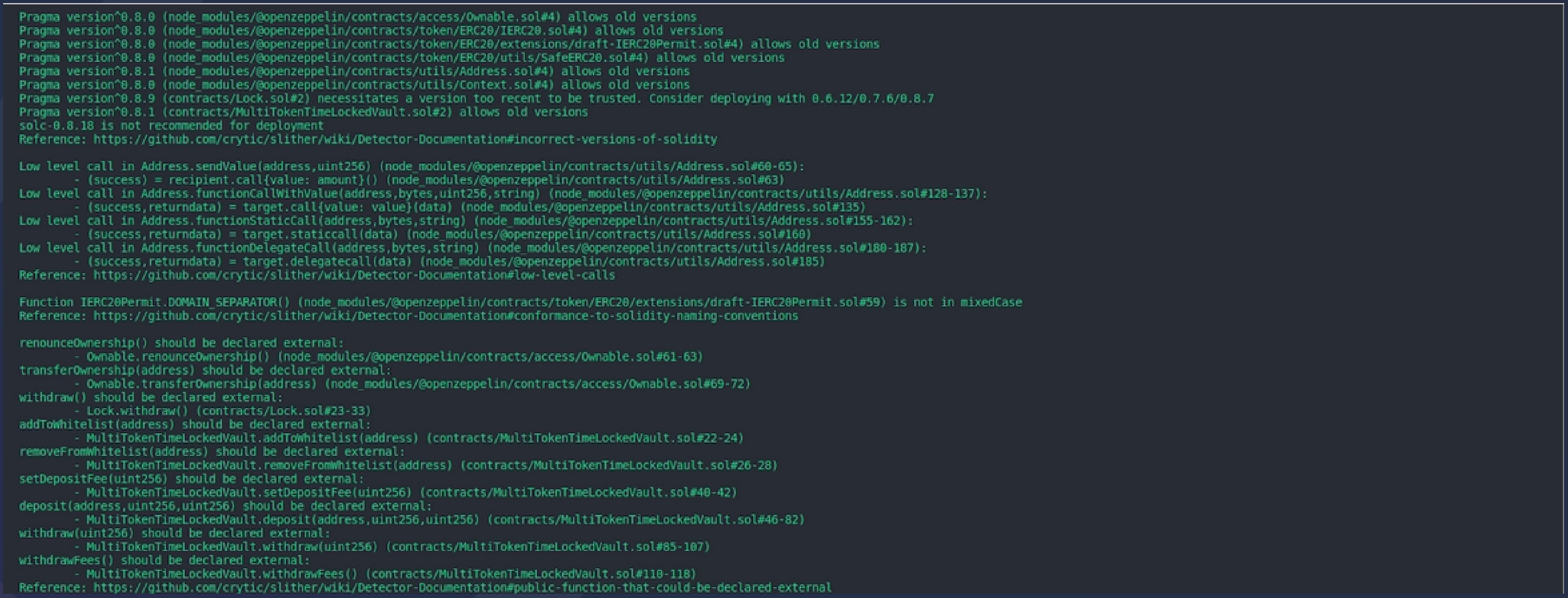
- ✓ Should deposit tokens to vault
- ✓ Should deposit tokens to vault at 0 fee
- ✓ Should withdraw tokens when unlockTimestamp is reached
- ✓ Should only deposit whitelisted tokens
- ✓ Should withdraw accumulated fees

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.



```
MultiTokenTimeLockedVault.setDepositFee(uint256) (contracts/MultiTokenTimeLockedVault.sol#40-42) should emit an event for:
        - depositFee = newFee (contracts/MultiTokenTimeLockedVault.sol#41)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic

Reentrancy in MultiTokenTimeLockedVault.deposit(address,uint256,uint256) (contracts/MultiTokenTimeLockedVault.sol#46-82):
        External calls:
        - IERC20(token).safeTransferFrom(msg.sender,address(this),amount) (contracts/MultiTokenTimeLockedVault.sol#68)
        State variables written after the call(s):
        - _depositIdCounter ++ (contracts/MultiTokenTimeLockedVault.sol#71)
        - depositInfo[depositId] = Deposit(depositId,msg.sender,token,amount,unlockTimestamp) (contracts/MultiTokenTimeLockedVault.sol#73-79)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2

Lock.constructor(uint256) (contracts/Lock.sol#13-21) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp < _unlockTime,Unlock time should be in the future) (contracts/Lock.sol#14-17)
Lock.withdraw() (contracts/Lock.sol#23-33) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= unlockTime,You can't withdraw yet) (contracts/Lock.sol#27)
MultiTokenTimeLockedVault.deposit(address,uint256,uint256) (contracts/MultiTokenTimeLockedVault.sol#46-82) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(unlockTimestamp > block.timestamp,Unlock timestamp must be in the future.) (contracts/MultiTokenTimeLockedVault.sol#59-62)
MultiTokenTimeLockedVault.withdraw(uint256) (contracts/MultiTokenTimeLockedVault.sol#85-107) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= userDeposit.unlockTimestamp,Deposit is still locked.) (contracts/MultiTokenTimeLockedVault.sol#94-97)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address._revert(bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#231-243) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#236-239)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Different versions of Solidity are used:
        - Version used: ['^0.8.0', '^0.8.1', '^0.8.9']
        - ^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#4)
        - ^0.8.1 (node_modules/@openzeppelin/contracts/utils/Address.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4)
        - ^0.8.9 (contracts/Lock.sol#2)
        - ^0.8.1 (contracts/MultiTokenTimeLockedVault.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#4) allows old versions
Pragma version^0.8.1 (node_modules/@openzeppelin/contracts/utils/Address.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
Pragma version^0.8.9 (contracts/Lock.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7
Pragma version^0.8.1 (contracts/MultiTokenTimeLockedVault.sol#2) allows old versions
solc-0.8.18 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```



```
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#4) allows old versions
Pragma version^0.8.1 (node_modules/@openzeppelin/contracts/utils/Address.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
Pragma version^0.8.9 (contracts/Lock.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7
Pragma version^0.8.1 (contracts/MultiTokenTimeLockedVault.sol#2) allows old versions
solc-0.8.18 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in Address.sendValue(address,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#60-65):
        - (success) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#63)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#128-137):
        - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)
Low level call in Address.functionStaticCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#155-162):
        - (success,returndata) = target.staticcall(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#160)
Low level call in Address.functionDelegateCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#180-187):
        - (success,returndata) = target.delegatecall(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#185)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Function IERC20Permit.DOMAIN_SEPARATOR() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#59) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (node_modules/@openzeppelin/contracts/access/Ownable.sol#61-63)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (node_modules/@openzeppelin/contracts/access/Ownable.sol#69-72)
withdraw() should be declared external:
        - Lock.withdraw() (contracts/Lock.sol#23-33)
addToWhitelist(address) should be declared external:
        - MultiTokenTimeLockedVault.addToWhitelist(address) (contracts/MultiTokenTimeLockedVault.sol#22-24)
removeFromWhitelist(address) should be declared external:
        - MultiTokenTimeLockedVault.removeFromWhitelist(address) (contracts/MultiTokenTimeLockedVault.sol#26-28)
setDepositFee(uint256) should be declared external:
        - MultiTokenTimeLockedVault.setDepositFee(uint256) (contracts/MultiTokenTimeLockedVault.sol#40-42)
deposit(address,uint256,uint256) should be declared external:
        - MultiTokenTimeLockedVault.deposit(address,uint256,uint256) (contracts/MultiTokenTimeLockedVault.sol#46-82)
withdraw(uint256) should be declared external:
        - MultiTokenTimeLockedVault.withdraw(uint256) (contracts/MultiTokenTimeLockedVault.sol#85-107)
withdrawFees() should be declared external:
        - MultiTokenTimeLockedVault.withdrawFees() (contracts/MultiTokenTimeLockedVault.sol#110-118)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

Closing Summary

In this report, we have considered the security of the Vaulty codebase. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and Informational severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Vaulty Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Vaulty Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

700+
Audits Completed

$16B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report May, 2023

For

VAULTY

QuillAudits

Canada, India, Singapore, United Kingdom

www.quillaudits.com

audits@quillhash.com