Hats Finance

SMART CONTRACT AUDIT

ZOKYO.

May 18th, 2021 | v. 1.0

PASS

Zokyo's Security Team has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

SCORE

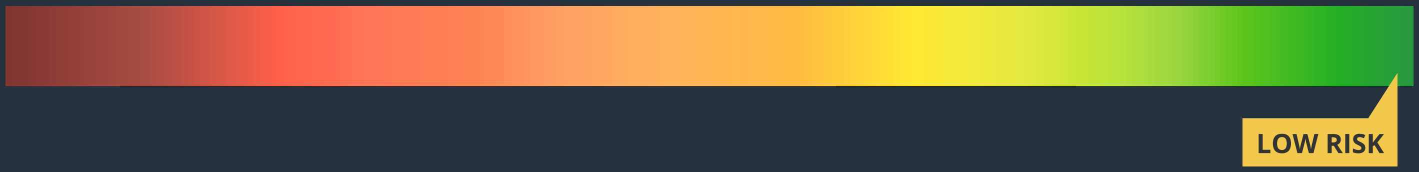
97

TECHNICAL SUMMARY

This document outlines the overall security of the Hats Finance smart contracts, evaluated by Zokyo's Blockchain Security team.

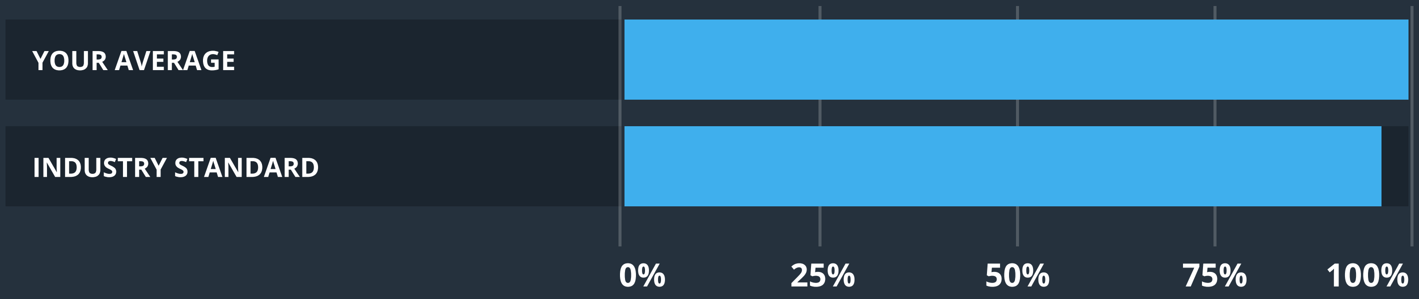
The scope of this audit was to analyze and document the Hats Finance smart contract codebase for quality, security, and correctness.

Contract Status



There were no critical issues found during the audit.

Testable Code



The testable code is 100%, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that’s able to withstand the Ethereum network’s fast-paced and rapidly changing environment, we at Zokyo recommend that the Hats Finance team put in place a bug bounty program to encourage further and active analysis of the smart contract.

TABLE OF CONTENTS

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Hats Finance repository.

Repository - https://github.com/hats-finance/hats-contracts
Branch: audit_1
Commit id - 997dd12d2bdb6d5f72942cee05b1e1f6c9ca0211

Last commit reviewed - 924fae9e93a7f79ddf1f245fad3778f05f7cb375 (branch audit_1_0)

Throughout the review process, care was taken to ensure that the token contract:

- Implements and adheres to existing Token standards appropriately and effectively;
- Documentation and code comments match logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices in efficient use of gas, without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the latest vulnerabilities;
- Whether the code meets best practices in code readability, etc.

Zokyo's Security Team has followed best practices and industry-standard techniques to verify the implementation of Hats Finance smart contracts. To do so, the code is reviewed line-by-line by our smart contract developers, documenting any issues as they are discovered. Part of this work includes writing a unit test suite using the Truffle testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

1 Due diligence in assessing the overall code quality of the codebase.

2 Cross-comparison with other, similar smart contracts by industry leaders.

3 Testing contract logic against common and uncommon attack vectors.

4 Thorough, manual review of the codebase, line-by-line.

EXECUTIVE SUMMARY

There were no critical issues found during the audit. All the mentioned findings may have an effect only in case of specific conditions performed by the contract owner.

Though, standard findings connected to low-risk reentrancy, calculation mistakes, gas savings and coding style were found. Also, the findings during the audit have a slight impact on contract performance or security, and can affect the further development and can influence some.

Nevertheless, all findings were successfully fixed by the Hats Finance team.

STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

● Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

● High

The issue affects the ability of the contract to compile or operate in a significant way.

● Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

● Low

The issue has minimal impact on the contract’s ability to operate.

● Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

MEDIUM | RESOLVED

Multiplication upon the result of division

HATVault.sol, line 482: calcClaimRewards()
There are several calculations based on the result of the division - claimRewardAmount is in the foundation of several operations. It is divided by REWARDS_LEVEL_DENOMINATOR and after other results are divided as well, so accuracy (set by the denominator) is performed twice. Such order of operations may lead to the accuracy loss. This issue was also signaled by several automatic tools.

Recommendation:
Verify, that calculations are performed without the accuracy losses and double division by the denominator (the second performed over the division result) is desired.

MEDIUM | RESOLVED

Storage variable is never used

HATVault.sol, line 46: vaultName
HATVault.sol, line 46: projectsRegistery

Unused variables should be removed.

Recommendation:
Remove unused variables.

LOW | RESOLVED

Low risk reentrancy

HATMaster.sol
deposit(): pool.lpToken.safeTransferFrom();
_withdraw(): pool.lpToken.safeTransfer()

HATVault.sol
approveClaim(): lpToken.safeTransfer()

Since there is no restriction on lpToken added in HATVault, there is a possibility of a vulnerability connected with custom safeTransfer() function - a very widely spreaded issue. Though, since the function connected to adding new pools is governed, we can assume that new pools (and their token contracts) are validated. Though, the reentrancy issue is in the standard list of issues and was flagged by several automatic tools, so it should be included in the report and iits fix is highly recommended. Nevertheless, the issue is marked as low (due to reasons above).

Recommendation:
Add ReentrancyGuard to all basic operations connected to the functions above OR provide fixes to make all storage changes before the call OR both.

LOW | RESOLVED

"cap" value should be set as constant.

HATToken.cap (HATToken.sol#38) should be constant

Recommendation:
set the value as constant.

LOW | RESOLVED

shadowed variables.

HATTokenLock.initialize):
_owner (tokenlock\HATTokenLock.sol#14) shadows:
        - Ownable._owner (tokenlock\Ownable.sol#19) (state variable)
HATVaults.constructor():
_governance (HATVaults.sol#127) shadows:
        - Governable._governance (Governable.sol#19) (state variable)
TokenLock._initialize():
_owner (tokenlock\TokenLock.sol#104) shadows:
        - Ownable._owner (tokenlock\Ownable.sol#19) (state variable)

Recommendation:
rename/override shadowed variable.

LOW | UNRESOLVED

Comparison with boolean variables

TokenLock.cancelLock() (tokenlock\TokenLock.sol#170):
        -require(bool,string)(isAccepted == false)
TokenLock.revoke() (tokenlock\TokenLock.sol#377):
      -require(bool,string)(isRevoked == false)

Consider removal of strict comparisons.

Recommendation:
remove strict comparison and use bool variables directly.

LOW | RESOLVED

Time lock period cannot be changed

There is no method (governance only) for lock period changing. It is set once in the constructor and cannot be changed.

Recommendation:
Confirm the functionality or add the governable method or set the value as constant.

LOW | RESOLVED

Extra Ownable contract

Ownable.sol is identical to standard openzeppelin one except the initialization. For better readability and for further development it is recommended to use the standard one (with appropriate changes in Tokenlock constructor) or rename the contract to e.g. OwnableInitializable. Especially because the standard contract is used in TokenLockFactory

Recommendation:
Use standard contract or rename the current one

INFORMATIONAL | RESOLVED

Missing interface inheritance

HATToken.sol
For certain ERC20 compatibility it is recommended to provide inheritance from standard IERC20 interface - it increases the readability of the code.

Recommendation:
Add inheritance from IERC20 for the token

LOW | RESOLVED

Methods should be set as external
HATMaster:
claimReward(uint256) should be declared external
getRewardPerBlock(uint256) should be declared external
pendingReward(uint256,address) should be declared external
getGlobalPoolUpdatesLength() should be declared external
getStakedAmount(uint256,address) should be declared external
setPendingGovernance(address) should be declared external
confirmGovernance() should be declared external
setPendingMinter(address,uint256) should be declared external

HATToken:
confirmMinter(address) should be declared external
burn(uint256) should be declared external
mint(address,uint256) should be declared external
increaseAllowance(address,uint256) should be declared external
decreaseAllowance(address,uint256) should be declared external
delegate(address) should be declared external
delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) should be declared external (HATToken.sol#316-325)
getPriorVotes(address,uint256) should be declared external

TokenLockFactory:
setMasterCopy(address) should be declared external:

In order to increase safety and decrease gas consumption, all public functions which are not called in the contract itself should be declared as external.

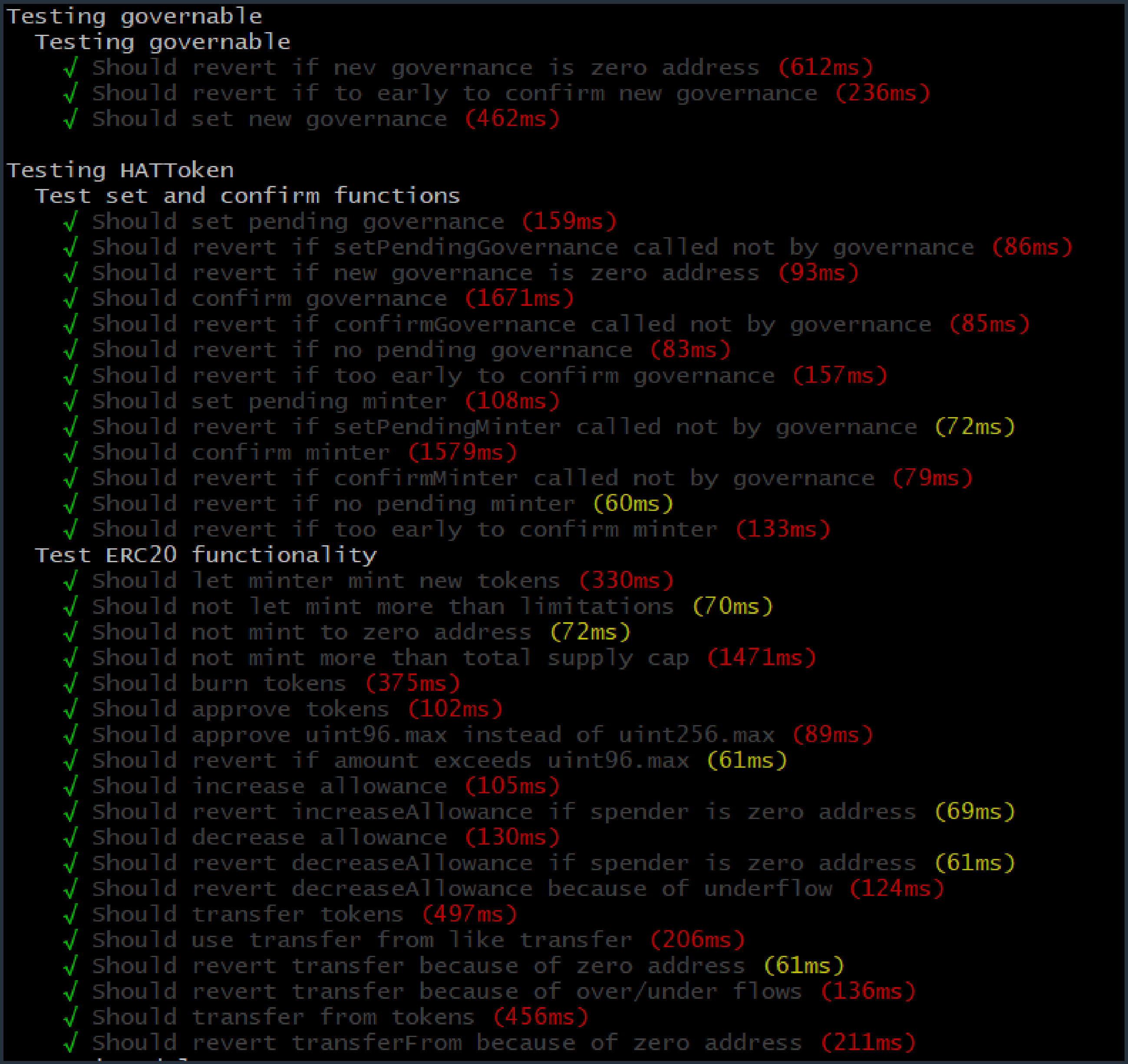
Recommendation:
declare methods as external.

	HATToken	HATMaster, HATVault	TokenLock
Re-entrancy	Pass	Pass	Pass
Access Management Hierarchy	Pass	Pass	Pass
Arithmetic Over/Under Flows	Pass	Pass	Pass
Unexpected Ether	Pass	Pass	Pass
Delegatecall	Pass	Pass	Pass
Default Public Visibility	Pass	Pass	Pass
Hidden Malicious Code	Pass	Pass	Pass
Entropy Illusion (Lack of Randomness)	Pass	Pass	Pass
External Contract Referencing	Pass	Pass	Pass
Short Address/ Parameter Attack	Pass	Pass	Pass
Unchecked CALL Return Values	Pass	Pass	Pass
Race Conditions / Front Running	Pass	Pass	Pass
General Denial Of Service (DOS)	Pass	Pass	Pass
Uninitialized Storage Pointers	Pass	Pass	Pass
Floating Points and Precision	Pass	Pass	Pass
Tx.Origin Authentication	Pass	Pass	Pass
Signatures Replay	Pass	Pass	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass	Pass

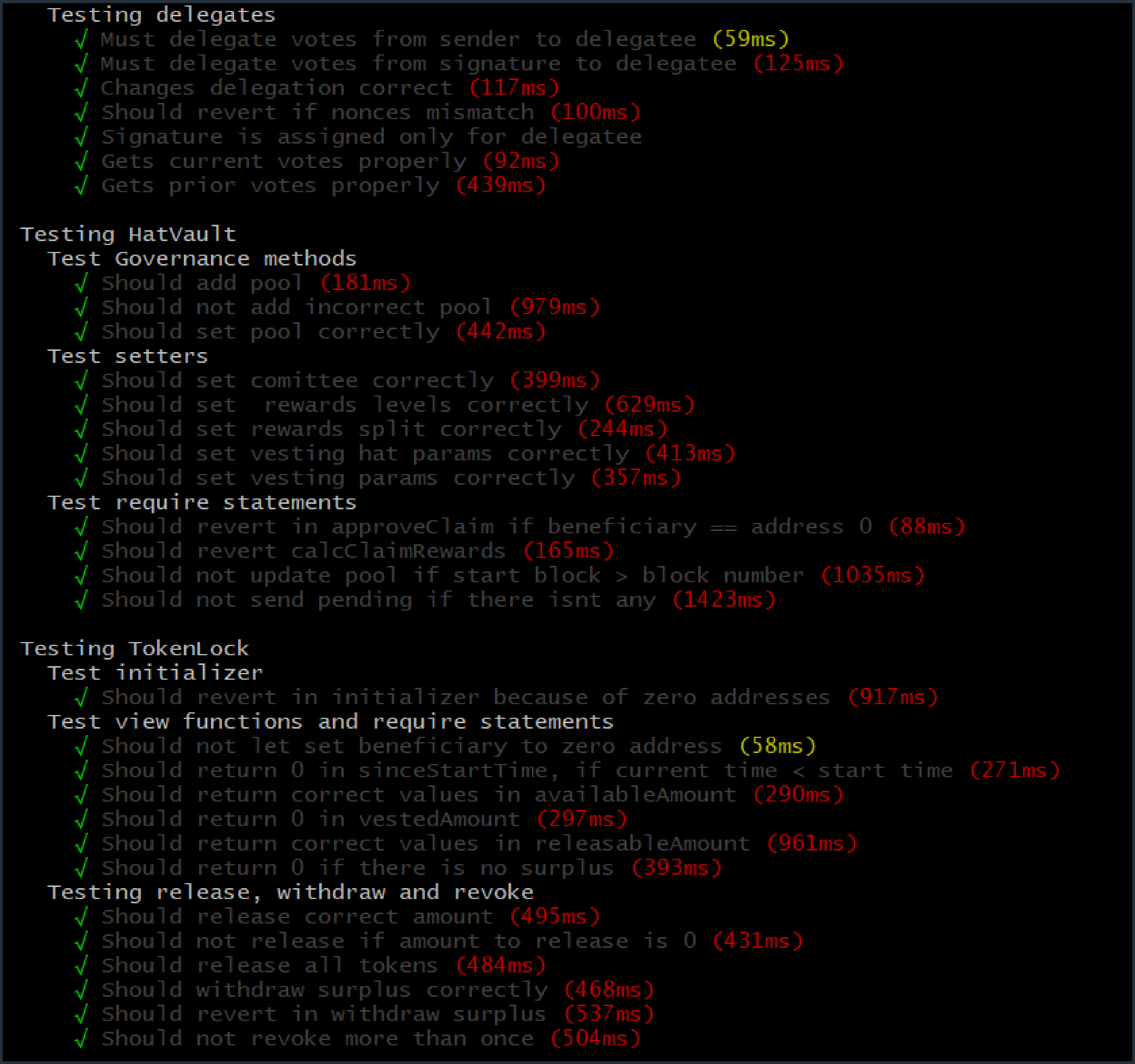
CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo team

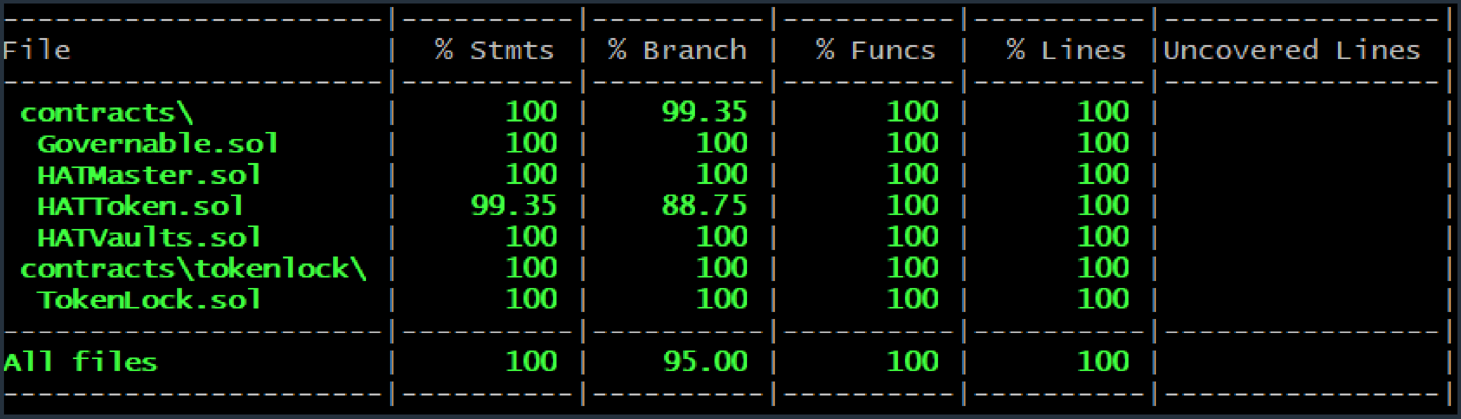
As part of our work assisting Hats Finance in verifying the correctness of their contract code, our team was responsible for writing integration tests using Truffle testing framework.
Tests were based on the functionality of the code, as well as review of the Hats Finance contract requirements for details about issuance amounts and how the system handles these.



```
Testing governable
  Testing governable
    √ Should revert if nev governance is zero address (612ms)
    √ Should revert if to early to confirm new governance (236ms)
    √ Should set new governance (462ms)

Testing HATToken
  Test set and confirm functions
    √ Should set pending governance (159ms)
    √ Should revert if setPendingGovernance called not by governance (86ms)
    √ Should revert if new governance is zero address (93ms)
    √ Should confirm governance (1671ms)
    √ Should revert if confirmGovernance called not by governance (85ms)
    √ Should revert if no pending governance (83ms)
    √ Should revert if too early to confirm governance (157ms)
    √ Should set pending minter (108ms)
    √ Should revert if setPendingMinter called not by governance (72ms)
    √ Should confirm minter (1579ms)
    √ Should revert if confirmMinter called not by governance (79ms)
    √ Should revert if no pending minter (60ms)
    √ Should revert if too early to confirm minter (133ms)
  Test ERC20 functionality
    √ Should let minter mint new tokens (330ms)
    √ Should not let mint more than limitations (70ms)
    √ Should not mint to zero address (72ms)
    √ Should not mint more than total supply cap (1471ms)
    √ Should burn tokens (375ms)
    √ Should approve tokens (102ms)
    √ Should approve uint96.max instead of uint256.max (89ms)
    √ Should revert if amount exceeds uint96.max (61ms)
    √ Should increase allowance (105ms)
    √ Should revert increaseAllowance if spender is zero address (69ms)
    √ Should decrease allowance (130ms)
    √ Should revert decreaseAllowance if spender is zero address (61ms)
    √ Should revert decreaseAllowance because of underflow (124ms)
    √ Should transfer tokens (497ms)
    √ Should use transfer from like transfer (206ms)
    √ Should revert transfer because of zero address (61ms)
    √ Should revert transfer because of over/under flows (136ms)
    √ Should transfer from tokens (456ms)
    √ Should revert transferFrom because of zero address (211ms)
```



```
    Testing delegates
      √ Must delegate votes from sender to delegatee (59ms)
      √ Must delegate votes from signature to delegatee (125ms)
      √ Changes delegation correct (117ms)
      √ Should revert if nonces mismatch (100ms)
      √ Signature is assigned only for delegatee
      √ Gets current votes properly (92ms)
      √ Gets prior votes properly (439ms)

  Testing HatVault
    Test Governance methods
      √ Should add pool (181ms)
      √ Should not add incorrect pool (979ms)
      √ Should set pool correctly (442ms)
    Test setters
      √ Should set comittee correctly (399ms)
      √ Should set  rewards levels correctly (629ms)
      √ Should set rewards split correctly (244ms)
      √ Should set vesting hat params correctly (413ms)
      √ Should set vesting params correctly (357ms)
    Test require statements
      √ Should revert in approveClaim if beneficiary == address 0 (88ms)
      √ Should revert calcClaimRewards (165ms)
      √ Should not update pool if start block > block number (1035ms)
      √ Should not send pending if there isnt any (1423ms)

  Testing TokenLock
    Test initializer
      √ Should revert in initializer because of zero addresses (917ms)
    Test view functions and require statements
      √ Should not let set beneficiary to zero address (58ms)
      √ Should return 0 in sinceStartTime, if current time < start time (271ms)
      √ Should return correct values in availableAmount (290ms)
      √ Should return 0 in vestedAmount (297ms)
      √ Should return correct values in releasableAmount (961ms)
      √ Should return 0 if there is no surplus (393ms)
    Testing release, withdraw and revoke
      √ Should release correct amount (495ms)
      √ Should not release if amount to release is 0 (431ms)
      √ Should release all tokens (484ms)
      √ Should withdraw surplus correctly (468ms)
      √ Should revert in withdraw surplus (537ms)
      √ Should not revoke more than once (504ms)
```



```
----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts\           |      100 |    99.35 |      100 |      100 |                |
  Governable.sol      |      100 |      100 |      100 |      100 |                |
  HATMaster.sol       |      100 |      100 |      100 |      100 |                |
  HATToken.sol        |    99.35 |    88.75 |      100 |      100 |                |
  HATVaults.sol       |      100 |      100 |      100 |      100 |                |
 contracts\tokenlock\ |      100 |      100 |      100 |      100 |                |
  TokenLock.sol       |      100 |      100 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
All files             |      100 |    95.00 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
```

We are grateful to have been given the opportunity to work with the Hats Finance team.

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.

Zokyo's Security Team recommends that the Hats Finance team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.