PAID.

SMART CONTRACT AUDIT

ZOKYO.

September 6th, 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

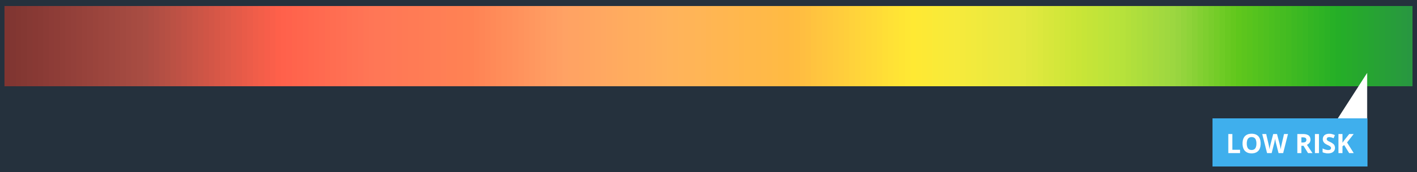
98

TECHNICAL SUMMARY

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

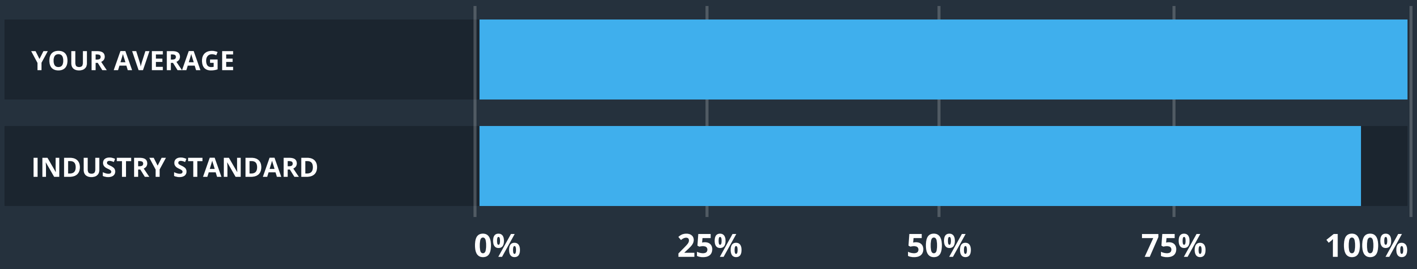
The scope of this audit was to analyze and document the PAID smart contract codebase for quality, security, and correctness.

Contract Status



There was 1 critical issue 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 PAID 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 PAID repository.

Repository (archive hash):
4fe3e5465d5e75ba492b0361eb31d40d2362ffce

Last commit (archive hash):
c55e5436806f8aab8382cee0dfaa510ce18fdcb1

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 PAID 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.

SUMMARY

There was one critical issue found during the audit, one issue with high severity, and a couple of issues with the low and informational severity levels. All the mentioned findings could have an effect only in case of specific conditions performed by the contract owner. It is worth mentioning that all of the issues were successfully resolved by the PAID team and bear no operational or security risks for the user and contract owner.

Hence, we can give a score of 98 to the contracts that were under an audit and state that the contracts are fully production-ready.

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

● 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

Wrong logic of pause of smart contract

CRITICAL | RESOLVED

Function isPausedUntilBlock() returns false if contract unpaused and true if paused. But requirement require(isPausedUntilBlock(), 'Contract is paused right now') will be triggered only then passed false value so the requirement will work the other way around.

Recommendation:
You need to change the line from

```
require(isPausedUntilBlock(), 'Contract is paused right now');
```

to

```
require(!isPausedUntilBlock(), 'Contract is paused right now');
```

Contract owner has too many rights

HIGH | RESOLVED

A contract owner can pause the contract at any time and users will not be able to transfer their tokens.

Recommendation:
Please modify this right with a more suitable and safer one. As a suggestion, you can set the time limit for the pause.

Re-audit:
Owner still has permission to pause transfers for as long as he wants. An average block is 13.4s so you can add require for function pausedUntilBlock() for parameter blocksDuration which will limit the time of maximum pause.

Additional checks are required at function addAllocations()

LOW | RESOLVED

There is no verification for the zero address and total amounts equal to zero.

Recommendation:
Add additional checks.

Missing requirement

LOW | RESOLVED

When transfer index of vesting type that is bigger than length of vestingTypes array returns error “invalid opcode”.

Recommendation:
Add additional check if index bigger than length of array.

Solidity version can be updated

INFORMATIONAL | RESOLVED

Since Solidity 0.7.4 ABI coder v2 is not considered experimental anymore and you can replace it with pragma abicoder v2.

Recommendation:
Update solidity version.

Gas optimization

INFORMATIONAL | RESOLVED

Functions getReleaseTime() and getMaxTotalSupply() can be replaced with constant variables.

Recommendation:
Replace those functions with constant variables.

Unnecessary requirement

INFORMATIONAL | RESOLVED

Function _mint() can be called only while initializing the contract and mint max token supply for the owner:

```
require(getMaxTotalSupply() >= totalSupply.add(amount), "Max total supply over");
```

Recommendation:
Remove the unnecessary requirement.

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security team

As part of our work assisting PAID in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Truffle testing framework.

Tests were based on the functionality of the code, as well as a review of the PAID contract requirements for details about issuance amounts and how the system handles these.

Code Coverage

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	UNCOVERED LINES
contracts\	100.00	97.06	100.00	100.00	
PaidTokenV4ETH.sol	100.00	97.06	100.00	100.00	
All files	**100.00**	**97.06**	**100.00**	**100.00**	

Test Results

```
Contract: PaidTokenV4ETH
  Methods
    initialize
      ✓ Check if zero address deployed the token (1943ms)
      ✓ Check is token deployed correctly (142ms)
    mulDiv
      ✓ Divide on zero (122ms)
      ✓ Overflow exception (101ms)
      ✓ Should calculate correctly (111ms)
    addAllocations
      ✓ Should be failed if address and total Amounts length are different (256ms)
      ✓ Should be failed if vesting type isn't valid (253ms)
      ✓ Should be failed if address of user is zero address (269ms)
      ✓ Should be failed if amount == 0 (248ms)
      ✓ Should be failed if caller isn't owner (253ms)
      ✓ Should work correctly (723ms)
    getTimestamp
      ✓ Should return block timestamp (86ms)
    getMonths
      ✓ Subtraction overflow (146ms)
      ✓ Should return correct value (468ms)
    isStarted
      ✓ Should return false before release date (232ms)
      ✓ Should return true after release date (523ms)
    getTransferableAmount
      ✓ Should change values over time (First vesting type) (964ms)
      ✓ Shouldn't change values over time (Fifth vesting type) (634ms)
    transferMany
      ✓ Should fail if caller isn't owner (205ms)
      ✓ Should fail if lengths of arrays are different (212ms)
      ✓ Should fail if the owner wants to transfer more than he has (211ms)
      ✓ Should fail if the recipient has zero address (196ms)
      ✓ Should fail if the contract is paused (374ms)
      ✓ Should work correctly (807ms)
    getRestAmount
      ✓ Should return correct values over time (1478ms)
    canTransfer
      ✓ Should be true if user's frozen wallet isn't scheduled (72ms)
      ✓ Should be true if contract hasn't start and user have some balance (656ms)
      ✓ Should be false if contract hasn't start and user have less balance than he
        want to transfer (513ms)
      ✓ Should be true if contract has started and user have some balance (899ms)
    withdraw
      ✓ Should fail if caller isn't owner (168ms)
      ✓ Shold fail if balance of contract is less than requested ammount (236ms)
      ✓ Should withdraw correctly (381ms)
    try to transfer token between users
      ✓ Should fail if contract is paused (1236ms)
      ✓ Shouldn't transfer paid Token between users before contract has started (1016ms)
```

```
        ✓ Should transfer paidToken between users correctly (2132ms)
      burn
        ✓ Should fail if caler isn't owner (193ms)
        ✓ Should work correctly (402ms)
      unpause
        ✓ Should fail if caller isn't owner (223ms)
        ✓ Should work correctly (170ms)


  39 passing (19s)
```

We are grateful to have been given the opportunity to work with the PAID 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 PAID team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

ZOKYO.