hashlock.

Security Audit

PinLink (Token)

October 2024 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	10
Audit Resources	10
Dependencies	10
Severity Definitions	11
Audit Findings	12
Centralisation	22
Conclusion	23
Our Methodology	24
Disclaimers	26
About Hashlock	27

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

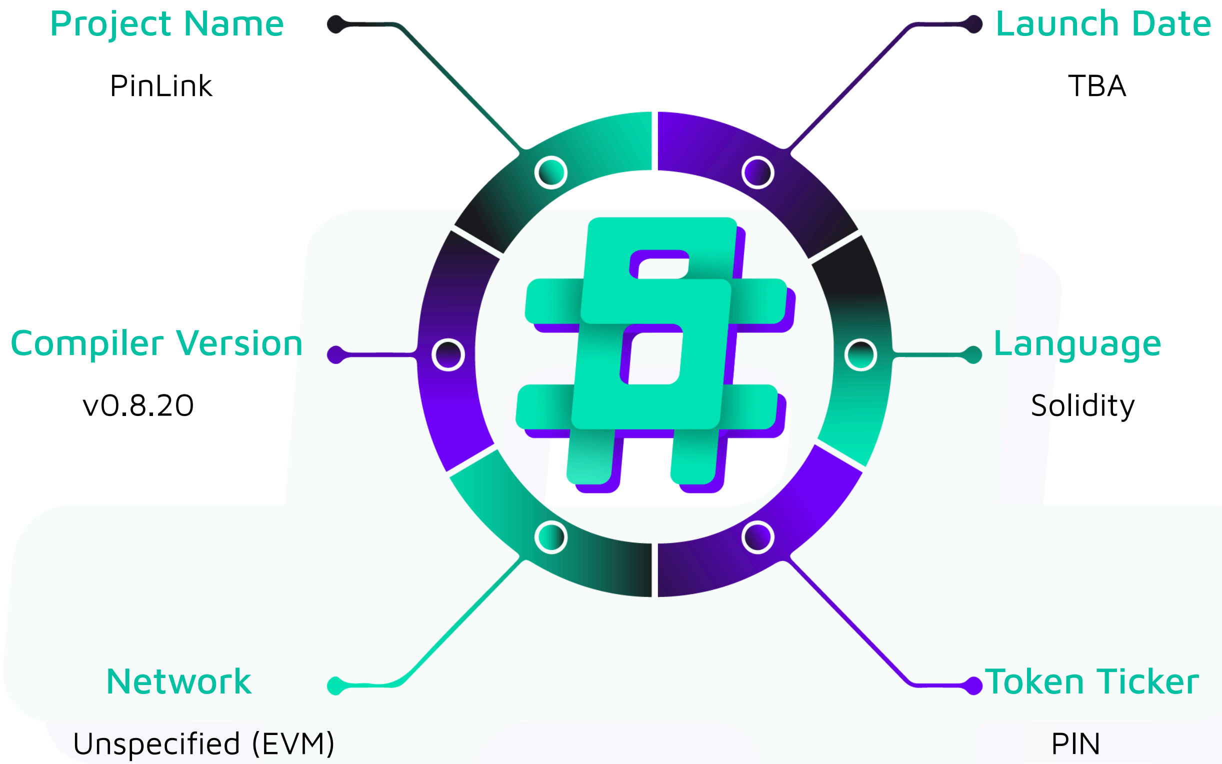
The PinLink team partnered with Hashlock to conduct a security audit of their FractionalToken.sol, Marketplace.sol and PinToken.sol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

PinLink implements a token ecosystem with a marketplace for renting and trading ERC1155 tokens, alongside support for fractional ownership. It includes contracts for managing token issuance, marketplace operations, and fractionalizing assets, enabling flexible token usage and trading mechanisms as well as their implementation of an ERC-20 token - PinLink Token.

Project Name: PinLink
Compiler Version: 0.8.20
Website: https://pinlink.ai/
Logo:

Visualised Context:



Project Visuals:

Audit scope

We at Hashlock audited the solidity code within the PinLink project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**PinLink Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**October, 2024**
Contract 1	FractionalToken.sol
Contract 1 MD5 Hash	03737b79865c50998398f345cf295dc1
Contract 2	MarketPlace.sol
Contract 2 MD5 Hash	7879872bf89f5223bebaa9365bc0eb42
Contract 3	PinToken.sol
Contract 3 MD5 Hash	be6f9a0dda6380b5e835468ee4e8b167
Contract 4	IMarketplace.sol
Contract 4 MD5 Hash	c33884f50ab35ebd2e0d9f46bfeb84b5
GitHub Commit Hash	3c0b0b31adc097e78e6d8c2cb3eeecdddb081e95

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with partially incorrect ordering. The protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.

Not Secure | Vulnerable | Secure | Hashlocked

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The general security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

4 Low severity vulnerabilities

3 Gas Optimisations

2 QA findings

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
FractionalToken.sol - Allows users to: - Mint new ERC1155 Tokens, if they are granted the MINTER_ROLE - Transfer fractional tokens - Query the contract for data such as tokensOfOwner()	**Contract achieves this functionality.**
Marketplace.sol - Allows Users to: - Rent tokens by calling takeOnRent() - Rent fractions of ERC1155 tokens - List tokens on the marketplace - Calculate rewards for renters	**Contract achieves this functionality.**
PinToken.sol ERC-20 Token containing usual functions Additional implementation of tax functionality allowing to impose and modify transfer, buy & sell taxes.	**Contract achieves this functionality.**
IMarketplace.sol Interface for interacting with Marketplace.sol through the getRentalAmount() function	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the PinLink project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the PinLink smart contract code in the form of Github access.

As mentioned above, code parts are poorly commented and no additional documentation was provided increasing the time needed to understand the project's purpose and its intended functionality. The development environment is configured and it is fairly easy to quickly build the contract suite.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies

Audit Findings

Low

[L-01] FractionalToken, Marketplace, PinToken - Missing Zero Checks

Description

Multiple functions across all contracts within the scope lack input validation mechanisms.

`FractionalToken.constructor()`

`FractionalToken.setMarketplace()`

`PinToken.constructor()`

```
constructor(address staking_, address treasury_, address initialOwner)
ERC20("PinLink", "PIN") Ownable(initialOwner) {

 staking = staking_;

treasury = treasury_;

_mint(initialOwner, (TOTAL_SUPPLY * 70) / 100); //responsible for adding
liquidity to dex

  _mint(staking, (TOTAL_SUPPLY * 20) / 100);

   _mint(treasury, (TOTAL_SUPPLY * 10) / 100);

     transactionTax = 200; // 2%

    buySellTax = 500; // 5%

     stakingTax = 7000; // 70%

   }
```

Recommendation

Add logic to ensure that critical variables cannot be initialized with the zero address.

Status

Resolved

[L-02] Marketplace - Missing inheritance

Description

The contract `Marketplace.sol` should inherit from `IMarketplace.sol`

Recommendation

Add an inheritance statement … `contract Marketplace is IMarketplace` …

Status

Resolved

[L-03] PinToken - Highly permissive contract logic.

Description

The `PinToken` contract contains highly permissive contract logic, leading to centralization risk. The owner of the contract has the privilege to impose and modify transfer tax, buy & sell tax as well as marking addresses as exempt from tax.

Recommendation

Consider a more permissionless and decentralized approach.

Status

Acknowledged

[L-04] PinToken#_update - Risk for rounding errors when calculating tax amount, potentially leading to incorrect tax amounts

Description

The `PinToken` contract's tax calculation mechanism may introduce rounding errors, especially when handling transactions involving small token amounts. Solidity's integer

division truncates decimal fractions, which can result in inaccuracies in tax computations. These rounding errors may cause the contract to collect less tax than intended, affecting the distribution of staking rewards and the treasury. This finding specifically affects the transfer of small token amounts.

Vulnerability Details

In the `_update` function of the `PinToken` contract, tax amounts are calculated using integer arithmetic:

```
function _update(address sender, address recipient, uint256 amount) internal override {

    if (taxExempt[sender] || taxExempt[recipient]) {

        super._update(sender, recipient, amount);

    } else {

        uint256 feeAmount = (amount * transactionTax) / TAX_DIVISOR;

        uint256 amountAfterFee = amount - feeAmount;

        uint256 toStaking = (feeAmount * stakingTax) / TAX_DIVISOR;

        uint256 toTreasury = feeAmount - toStaking;

        super._update(sender, recipient, amountAfterFee);

        if (toStaking > 0) {

            super._update(sender, staking, toStaking);

        }

        if (toTreasury > 0) {

            super._update(sender, treasury, toTreasury);

        }

    }

}
```

- Integer Division Truncation: Solidity performs integer division, discarding any fractional remainder. When calculating `feeAmount`, `toStaking`, and `toTreasury`, small transfer amounts may result in these values rounding down to zero.

Proof of Concept

An example of a test that simulates precision loss is shown below.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.20;
import "forge-std/Test.sol";
import "../../src/token/PinToken.sol";
contract PinTokenPrecisionLossTest is Test {
    PinToken pinToken;
    address owner = address(1);
    address staking = address(2);
    address treasury = address(3);
    address user1 = address(4);
    address user2 = address(5);

    function setUp() public {
        // Deploy PinToken with initial owner
        vm.prank(owner);
        pinToken = new PinToken(staking, treasury, owner);
        // Set owner as tax-exempt to avoid transaction tax when transferring to user1
        vm.prank(owner);
        pinToken.setTaxExempt(owner, true);
        // Transfer tokens from owner to user1 without tax
        vm.prank(owner);
        pinToken.transfer(user1, 1000 * 1e18);
        // Ensure user1 has tokens
        uint256 balance = pinToken.balanceOf(user1);
        assertEq(balance, 1000 * 1e18, "User1 did not receive tokens");
    }
    function testPrecisionLossWithSmallAmount() public {
        uint256 smallAmount = 1; // Transfer 1 wei (small amount)

        // Record initial balances
```

```
        uint256 user1InitialBalance = pinToken.balanceOf(user1);
        uint256 user2InitialBalance = pinToken.balanceOf(user2);
        uint256 stakingInitialBalance = pinToken.balanceOf(staking);
        uint256 treasuryInitialBalance = pinToken.balanceOf(treasury);
        // Transfer small amount from user1 to user2
        vm.prank(user1);
        pinToken.transfer(user2, smallAmount);
        // Calculate expected fee amount
        uint256 transactionTax = pinToken.transactionTax(); // 200 (2%)
        uint256 stakingTax = pinToken.stakingTax(); // 7000 (70%)
        uint256 TAX_DIVISOR = pinToken.TAX_DIVISOR(); // 10000
         uint256 feeAmount = (smallAmount * transactionTax) / TAX_DIVISOR; // Expected to
be 0
        uint256 amountAfterFee = smallAmount - feeAmount; // Expected to be 1
        // Calculate expected staking and treasury amounts
        uint256 toStaking = (feeAmount * stakingTax) / TAX_DIVISOR; // Expected to be 0
        uint256 toTreasury = feeAmount - toStaking; // Expected to be 0
        // Fetch balances after transfer
        uint256 user1FinalBalance = pinToken.balanceOf(user1);
        uint256 user2FinalBalance = pinToken.balanceOf(user2);
        uint256 stakingFinalBalance = pinToken.balanceOf(staking);
        uint256 treasuryFinalBalance = pinToken.balanceOf(treasury);
        // Expected final balances
        uint256 expectedUser1FinalBalance = user1InitialBalance - smallAmount;
        uint256 expectedUser2FinalBalance = user2InitialBalance + amountAfterFee;
        uint256 expectedStakingFinalBalance = stakingInitialBalance + toStaking;
        uint256 expectedTreasuryFinalBalance = treasuryInitialBalance + toTreasury;
        // Assertions to check balances
            assertEq(user1FinalBalance, expectedUser1FinalBalance, "User1 final balance
incorrect");
            assertEq(user2FinalBalance, expectedUser2FinalBalance, "User2 final balance
incorrect");
            assertEq(stakingFinalBalance, expectedStakingFinalBalance, "Staking balance
incorrect");
          assertEq(treasuryFinalBalance, expectedTreasuryFinalBalance, "Treasury balance
incorrect");
```

```
        // Log the calculated values for verification
        emit log_named_uint("Fee Amount", feeAmount); // Should be 0
        emit log_named_uint("Amount After Fee", amountAfterFee); // Should be 1
        emit log_named_uint("To Staking", toStaking); // Should be 0
        emit log_named_uint("To Treasury", toTreasury); // Should be 0
    }
}
```

Impact

Users can avoid taxation by sending small amounts or pay less tax than intended, potentially leading to smaller amounts of funds being available for staking rewards.

Recommendation

Set minimum tax amounts and use high-precision arithmetic libraries.

Status

Acknowledged

Gas

[G-01] FractionalToken, Marketplace, PinToken - Gas inefficient require statements

Description

Gas-intensive `require` statements are used across all contracts in order to validate inputs.

Recommendation

Introduce Custom Errors instead of using required statements to improve gas efficiency.

```
function setStaking(address staking_) external onlyOwner {
        if(staking_ == address(0)) {
            revert zeroAddress();
        else staking = staking_;
 emit SetStaking(staking_);
 }
```

Status

Acknowledged

[G-02] PinToken - Unused variable

Description

The variable `address public liquidity` is declared within the `PinToken.sol` contract but never initialized or used.

Recommendation

Remove unused variables to save gas.

Status

Resolved

[G-03] Marketplace - Functions not used internally can be marked as external

Description

In order to save Gas, `public` functions that are never called in the contract should be declared as `external`.

Recommendation

Decrease the function visibility whenever possible.

Specifically for:

`FractionalToken.mint()`

`FractionalToken.mint()`

`FractionalToken.setMarketplace()`

Status

Resolved

QA

[Q-01] FractionalToken - confusing function naming

Description

The `FractionalToken` smart contract contains two functions named `mint()`.

Recommendation

Use unique function naming to avoid confusion.

Status

Resolved

[Q-02] FractionalToken, Marketplace - Solidity Style Guide violation

Description

The code violates the official Solidity Style Guide especially by randomly ordering functions within the contracts.

Recommendation

It is recommended to review the latest Solidity Style Guide documentation and change the contracts in order to conform with it.

The Solidity Style Guide gives the following order of functions:

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Within a grouping, place the `view` and `pure` functions last.

Status

Resolved

Centralisation

The PinLink project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and add taxation functionality but depend highly on internal team responsibility.

Conclusion

After Hashlocks analysis, the PinLink project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:
In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:
Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au

hashlock.