hashlock.

Security Audit

Emergence.art (NFT)

October 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

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

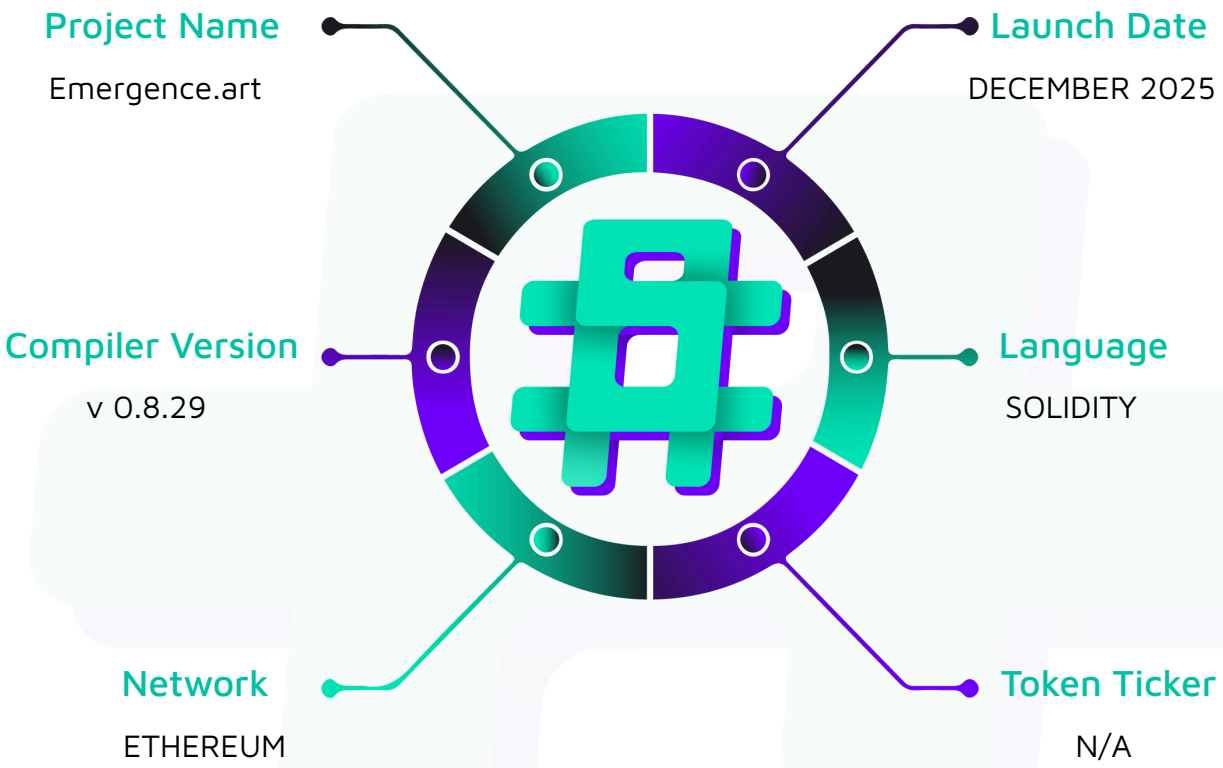
The Emergence.art team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

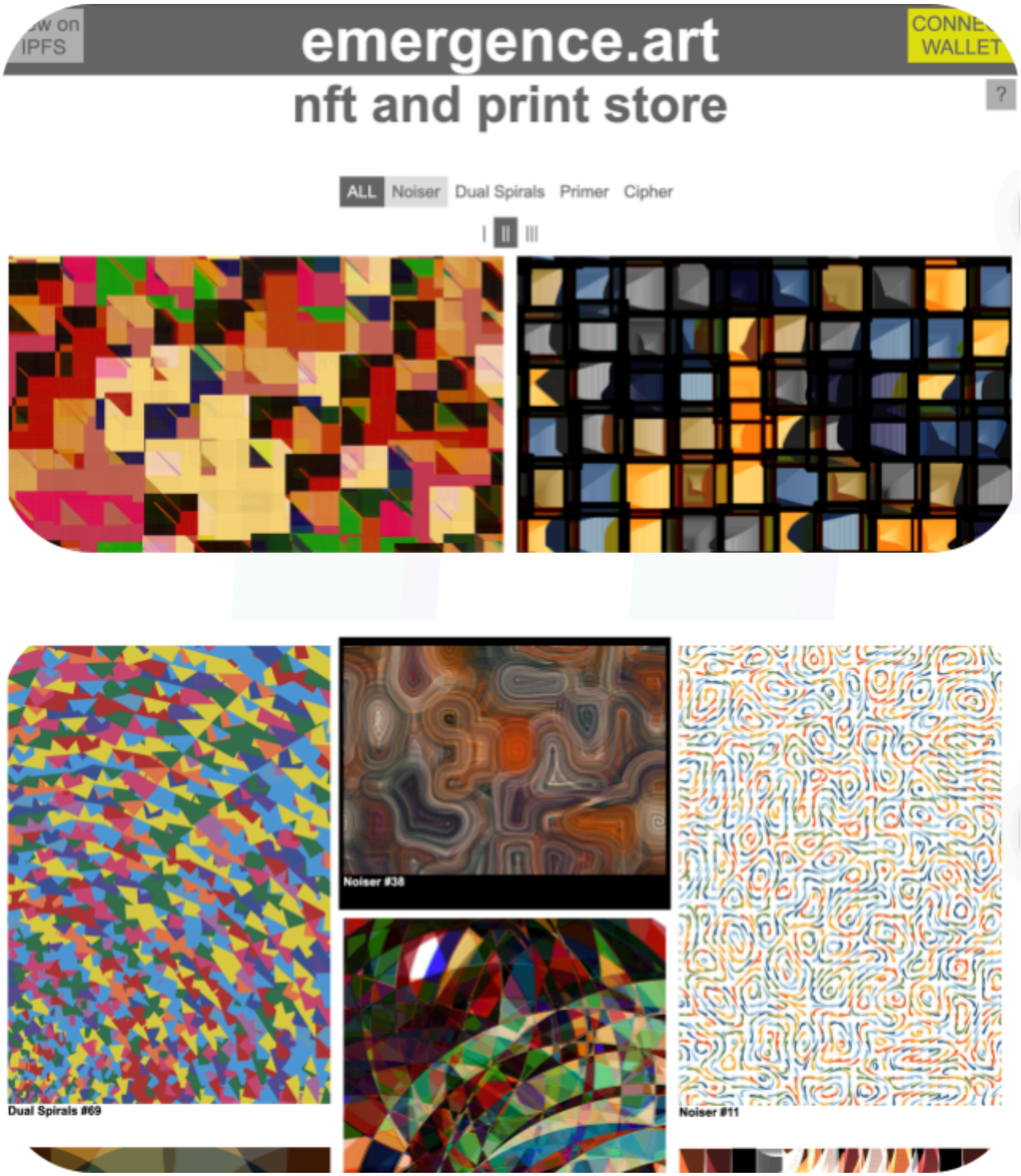
The Emergence project is a free, open-source system for selling NFTs and physical items on the blockchain—directly and independently.
It extends a standard ERC721 smart contract to manage item issuance, pricing, sales, and verification, while also supporting NFT owner commissions on item sales.
Each physical item features a tamperproof NFC tag with advanced encryption, linked to its NFT. When tapped with an NFC-enabled smartphone, a verification page is opened, and tag registration can be checked with the smart contract.
An Android app is included for secure tag setup, with optional key diversification for simplified key management. Each tag tap generates a unique signed URL containing the Tag ID, NFT Token ID, and an encrypted payload, verifiable offline with the provided tool.
A lightweight static web2/web3 site (HTML, CSS, JS) handles NFT and item configuration, listing, sales, and verification.
Originally built for digital art and prints, Emergence can secure any valuable physical item where authenticity matters.

Project Name: Emergence.art
Project Type: NFT
Compiler Version: 0.8.29
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the Emergence.art 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	**Emergence.art Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**October, 2025**
Contract 1	Emergence.sol
Audited GitHub Commit Hash	a41ac836bec13bc6faf7f80341541c955adfe261
Fix Review GitHub Commit Hash	c39b7a7176f18fd777285ad90c39bb383510c48f

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

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 list of audited assets is presented in the Audit Scope section, and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

2 High-severity vulnerabilities

3 Medium-severity vulnerabilities

2 Low-severity vulnerabilities

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
Emergence.sol - ERC-721 NFT contract that represents unique tokens, each associated with a collection of purchasable digital items. - The contract owner can mint new NFTs, define item pricing, maximum item limits, and commission rates. - External users can purchase items under a token, optionally paying a commission to the current NFT owner. - Metadata for each NFT is hosted via IPFS and resolved through tokenURI().	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Emergence.art 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 recommended to optimize security measures.

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 Emergence.art project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

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

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

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.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] Emergence - Owner Cannot Withdraw Accumulated ETH

Description

The Emergence contract receives ETH payments from NFT and item sales via payable functions, but provides no mechanism for the contract owner to withdraw the accumulated ETH balance. This leads to permanent locking of funds.

Vulnerability Details

Each `mint()` and buy `call()` is payable, and users send ETH to purchase NFTs or items. However, the contract lacks a withdrawal function or any other owner-accessible mechanism to transfer the received funds out of the contract. As a result, the ETH balance accumulates indefinitely.

The root cause is the absence of a withdrawal routine and access control logic that permits only the contract owner to recover the funds. This condition leads to trapped ETH, preventing revenue retrieval and operational fund management.

Impact

All ETH accumulated from NFT and item sales remains stuck in the contract, causing total loss of funds unless a migration or self-destruct upgrade path is added.

Recommendation

Introduce a secure `withdrawal` function callable only by the contract owner to allow ETH retrieval.

Status

Resolved

[H-02] Emergence - Signatures Can Be Replayed Across Deployments Or Chains

Description

The contract authorizes mints and purchases using off-chain signatures validated through `isSignedByOwner()`. These signatures are generated over parameter hashes that do not include contextual identifiers such as the contract address, chain ID, or nonce. Consequently, identical signatures can be reused across multiple deployments or chains where the same owner address exists.

Vulnerability Details

In both `mint()` and `buy()`, the contract uses `isSignedByOwner()` to verify ownership authorization. The function constructs the signed hash using only operational parameters like `msg.value`, `tokenId`, and `itemNumber`, omitting domain separation fields (address(this) and block.chainid) and nonce tracking.

Without these components, a valid signature from one deployment (e.g., on Chain X) can be reused on another deployment (Chain Y) or contract instance, since owner() resolves to the same signer.

Additionally, since no nonce is stored or invalidated, a single signature can authorize repeated unauthorized actions (e.g., unlimited mints or buys) within the same deployment.

Impact

An attacker can reuse valid signatures from other deployments or prior transactions to mint NFTs or perform purchases without fresh authorization, causing unauthorized asset creation and DoS for other users.

Recommendation

Implement domain separation by including `address(this)` and `block.chainid` in the signed data to ensure each signature is used only once.

Prefer EIP-712 structured signatures for strong replay protection.

Status

Resolved

Medium

[M-01] Emergence#ownerMint - Owner Can Mint NFT With Incorrect Payment Amount

Description

The `ownerMint()` function allows the contract owner to mint a new NFT by providing various pricing and configuration parameters. The function is marked as payable, but it does not validate that the sent ETH (`msg.value`) matches the `baseItemPrice` or any other pricing rule.

Vulnerability Details

The `ownerMint()` function sets `data[tokenId].basePrice` based on the provided `baseItemPrice`, but no check ensures that msg.value equals or corresponds to this value.

```
   function ownerMint(address to, uint256 tokenId, uint maxItems, uint256 baseItemPrice, uint8 itemPricingScheme, bool itemSequential, uint8 commissionRate) payable public onlyOwner {

        require(_ownerOf(tokenId) == address(0), "NFT already exists");


        data[tokenId].maxItems = maxItems;

>>>>    data[tokenId].basePrice = baseItemPrice;

        data[tokenId].pricingScheme = itemPricingScheme;

        data[tokenId].sequential = itemSequential;

        data[tokenId].commissionRate = commissionRate;



        _safeMint(to, tokenId);

    }
```

As a result, the transaction may include arbitrary or unintended ETH amounts — either overpaying (locking excess ETH in the contract) or underpaying (creating an NFT for less than its intended value).

Impact

The contract owner (or potentially any future extended role) may unintentionally send excess ETH or mint NFTs without paying the intended base price. This results in accounting inconsistencies and potentially locked funds.

Recommendation

Validate that the ETH sent in the transaction matches the defined baseItemPrice, or remove the payable modifier if no payment is required for ownerMint().

Status

Resolved

[M-02] Emergence#buy - Contract Does Not Validate Token Ownership Before Paying Commission

Description

The `buy()` function pays commission to the owner of an NFT whenever `commissionRate > 0`. However, it does not check whether the token is owned by the contract itself. The inline comment `// don't do if token is owned by contract` indicates that such validation was intended but never implemented.

Vulnerability Details

In the current implementation, the function calculates the commission as `msg.value * data[tokenId].commissionRate / 100` and directly transfers this amount to `_ownerOf(tokenId)`.

The intended behavior, as suggested by the code comment, is that only external NFT owners (not the contract) should receive commission payments.

```
    function buy(...) payable public {

        //.. code

>>>>    // don't do if token is owned by contract

>>     if (data[tokenId].commissionRate != 0) {

            uint256 commission = msg.value * data[tokenId].commissionRate / 100;


            payable(_ownerOf(tokenId)).transfer(commission);

            //.. code

      }

    }
```

Impact

ETH is unnecessarily transferred to the contract's own balance, increasing gas consumption and potentially disrupting internal fund flow and withdrawal reconciliation.

Recommendation

Add a check to ensure that commission is only paid when the NFT owner is not the contract itself, and `commissionRate` is greater than `zero`.

Status

Resolved

[M-03] Emergence#buy() - Commission Transfer Uses transfer Instead Of call

Description

The `buy()` function distributes a commission to the NFT owner using Solidity's `.transfer()` method. This approach forwards a fixed `2300` gas stipend, which can cause reverts if the recipient is a contract requiring more gas to execute its fallback or receive function.

Vulnerability Details

In the current implementation, commission is paid via `payable(_ownerOf(tokenId)).transfer(commission);`.

```
function buy(...) payable public {

        //.. code

       if (data[tokenId].commissionRate != 0) {

            uint256 commission = msg.value * data[tokenId].commissionRate / 100;




>>>>        payable(_ownerOf(tokenId)).transfer(commission);

            // (bool sent,) = _ownerOf(tokenId).call{value: commission}("");

            // require(sent, "Failed to send commission to NFT owner");

       }

    }
```

Since `.transfer()` forwards only `2300` gas, any recipient contract with a non-trivial `receive()` or `fallback()` function will fail to accept the commission, reverting the entire purchase transaction. This behavior is unsafe and deprecated in modern Solidity patterns, as EIP-1884 and later changes increased gas costs for certain opcodes, making `.transfer()` unreliable.

Impact

If the NFT owner is a smart contract, the entire `buy()` call may revert due to insufficient gas forwarded by `.transfer()`. This prevents valid sales and disrupts marketplace operations.

Recommendation

Use the low-level `.call{value: ...}("")` method with success verification instead of `.transfer()` to safely forward ETH and handle failures.

Status

Resolved

Low

[L-01] Emergence - Use Ownable2Step

Vulnerability Details

The "`Ownable2Step`" pattern is an improvement over the traditional "`Ownable`" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "`Ownable`" pattern, where ownership can be transferred directly to a specified address, the "`Ownable2Step`" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Recommendation

It is recommended to use either `Ownable2Step` or `Ownable2StepUpgradeable` depending on the smart contract.

Status

Resolved

[L-02] Emergence - Floating and Outdated Pragma

Vulnerability Details

Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.

The contract allowed floating or unlocked pragma to be used, i.e., `^0.8.28`. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Recommendation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the `0.8.29` pragma version

Reference: https://scs.owasp.org/SCWE/SCSVS-CODE/SCWE-060/

Status

Resolved

Centralisation

The Emergence.art project values security and utility over decentralisation.

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

Conclusion

After Hashlock's analysis, the Emergence.art project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved. 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