hashlock.

Security Audit

Vrine (Blockchain)

October, 2025 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
Status Definitions	12
Audit Findings	13
Centralisation	23
Conclusion	24
Our Methodology	25
Disclaimers	27
About Hashlock	28

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

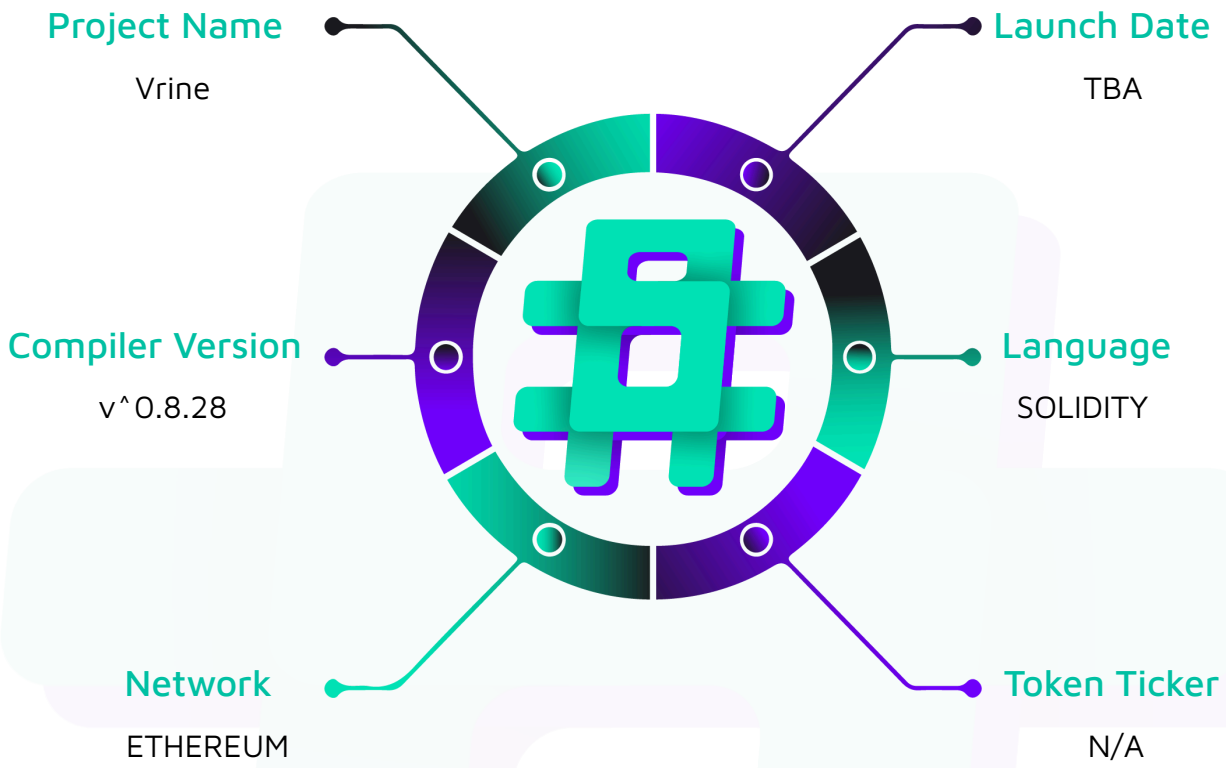
The Vrine team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

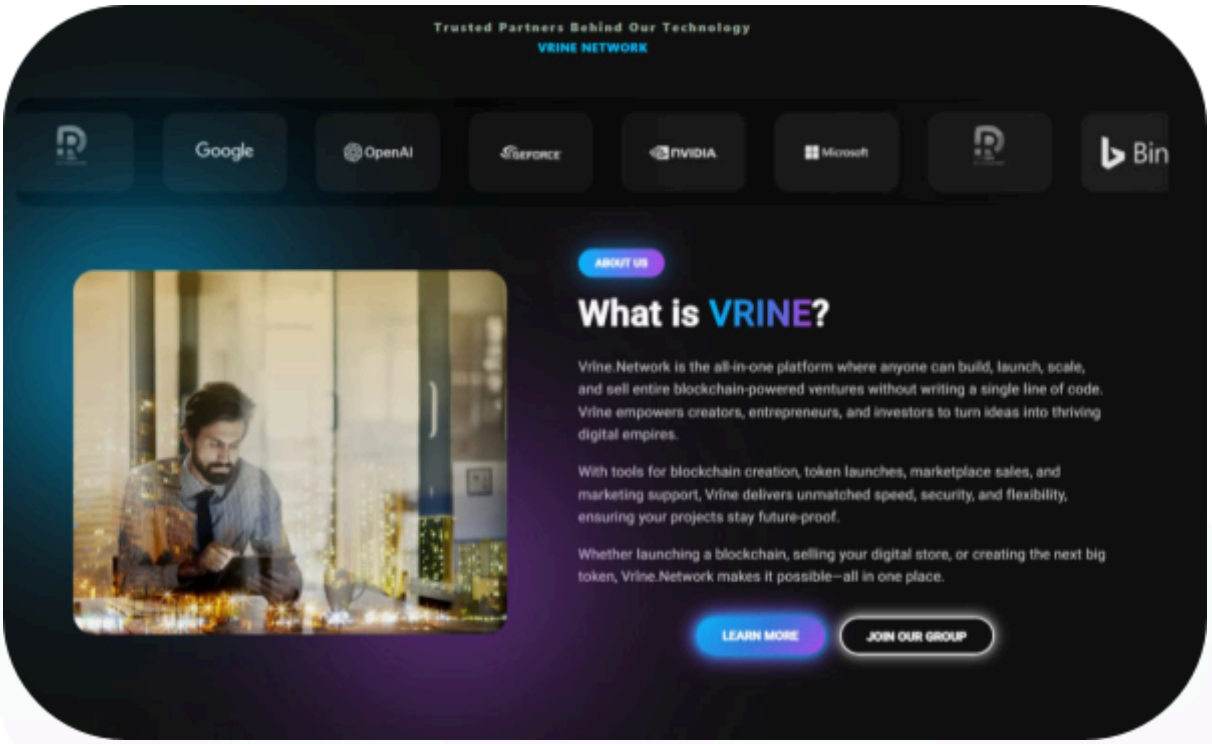
Vrine Network is an innovative no-code platform that allows users to create, launch, scale, and monetize blockchain-based projects without writing a single line of code. Its mission is to democratize access to blockchain technology, enabling entrepreneurs, businesses, and enthusiasts to develop decentralized solutions in an accessible and efficient way. The platform provides an intuitive interface, integrated tools, and support for features such as smart contracts and tokens, making it easier to implement innovative ideas within the Web3 ecosystem.

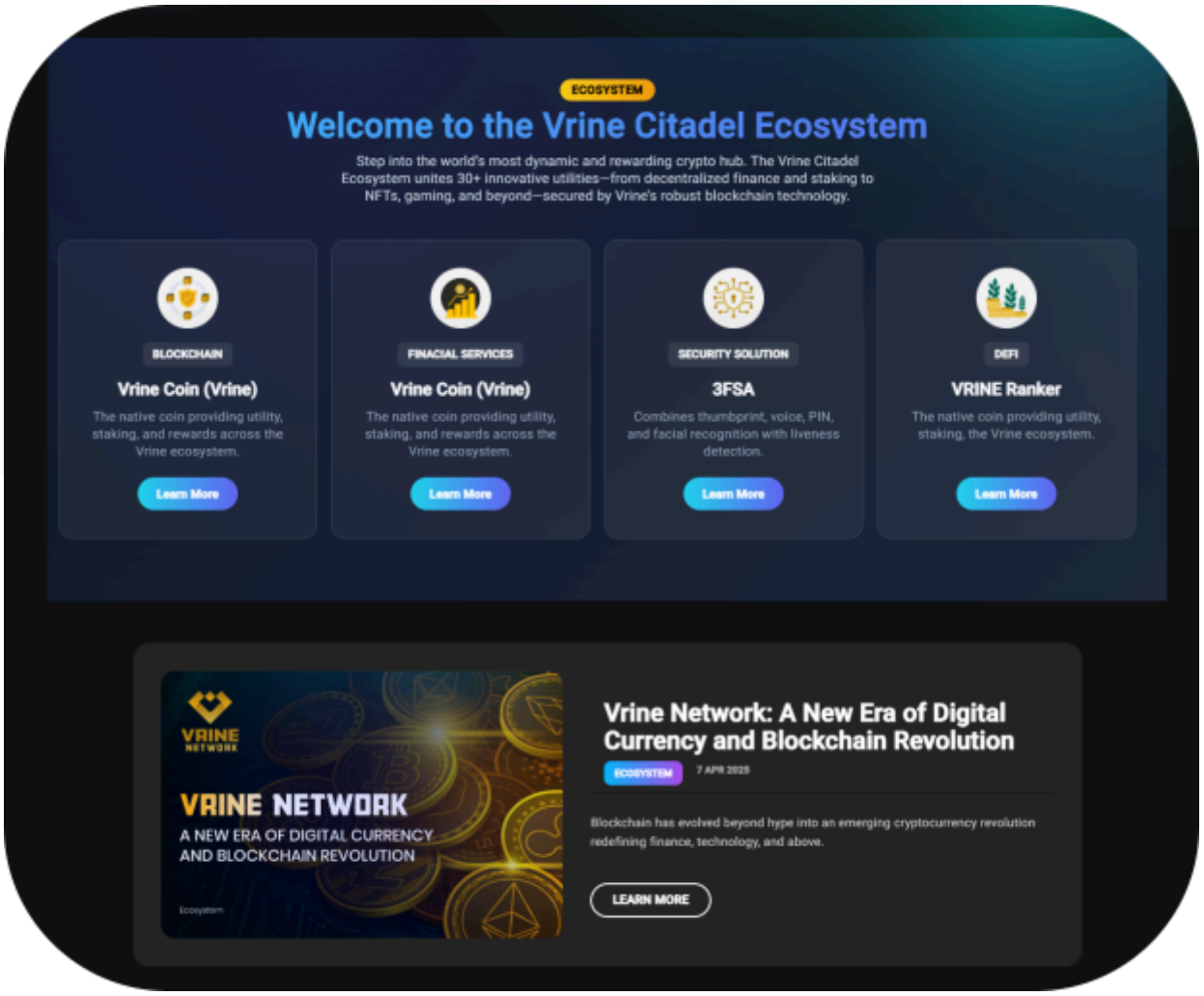
Project Name: Vrine
Project Type: Blockchain
Compiler Version: ^0.8.28
Website: https://vrine.network/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the Solidity code within the Vrine 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	Vrine Protocol Smart Contracts
Platform	Ethereum / Solidity
Audit Date	October, 2025
Contract 1	https://testnet.bscscan.com/address/0xb7baff6647f019d699244236401da561122411 0c#code

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:

1 High severity vulnerabilities

2 Medium severity vulnerabilities

4 Low severity vulnerabilities

1 QA

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
Vrine_Presale.sol - This contract facilitates the token presale for the Vrine ecosystem. - It enables users to purchase Vrine tokens using BNB at a dynamically fetched USD rate through a Chainlink oracle. - The contract supports minimum purchase limits, referral-based rewards, hardcap enforcement, and buyer claim tracking. - Administrative controls include managing oracle configurations, presale parameters, blacklist management, and fund withdrawals. - Safety features such as reentrancy protection, oracle sanity checks, slippage validation, and timelocked price updates ensure secure operation.	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Vrine 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 were 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 Vrine project smart contract code in the form of testnet 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] Vrine_Presale#BuyWithBNB() - Missing slippage/deadline while buying Vrine token

Description

A malicious acor controlling or manipulating the Chainlink `BNB/USD` feed can alter the price reported to `BuyWithBNB()` at execution time, causing buyer transactions to be settled at a distorted rate. Because `BuyWithBNB()` computes token amounts at execution using `getLatestPriceETH()` and `TokenPricePerUsdt` and does not accept buyer protections (no `minTokensOut`, no `deadline`), an attacker who can influence the oracle can drastically reduce the tokens a buyer receives or cause transactions to revert.

Vulnerability Details

The core issue is that the `BuyWithBNB()` function relies on `getLatestPriceETH()` and `TokenPricePerUsdt` without any validation from the buyer. The buyer has no control to ensure they receive at least a minimum number of tokens, nor can they set a deadline to prevent execution in stale blocks. This creates a race condition where privileged parties and a malicious user (attacker) with the ability to influence the on-chain price feed can exploit the lack of buyer-side protections and the reliance on a single live oracle value.

The attacker can monitor the mempool for large `BuyWithBNB()` transactions, manipulate the oracle immediately (or publish a malicious round) and ensure their manipulated price is the one consumed when the buyer's transaction is mined.

Although admins are assumed trusted, they have immediate power to call `setPresalePricePerUsdt/setPresalePricePerUsdtAdmin` and can intentionally or via compromise, change `TokenPricePerUsdt` right before pending buys. This enables the same `front-run` / `value-extraction` outcome even without direct oracle manipulation.

```
function BuyWithBNB(...) public payable nonReentrant {

        //..

        require(presaleStatus == true, "Presale : Presale is not started");

        require(

            isBlacklist[msg.sender] == false,

            "Presale : you are blacklisted"

        );

>>      uint256 ETHToUsd = (msg.value * (getLatestPriceETH())) / (1 ether);

        require(ETHToUsd / 1e12 >= minBuy, "Can't buy less than Min Amount");

        //..

        emit buy(msg.sender, usdamt, tokensToTransfer, block.timestamp);

    }
```

Impact

Buyers can receive far fewer tokens than expected or fail to complete purchases while still transferring `BNB`, enabling attackers to extract value from buyers and the sale.

Recommendation

Introduce buyer-controlled safeguards (`minTokensOut` and `deadline`) in `BuyWithBNB`, and enforce timelocks or delayed updates on `setPresalePricePerUsdt` and `setPresalePricePerUsdtAdmin` to prevent immediate manipulation of buyer outcomes.

Status

Resolved

Medium

[M-01] Vrine_Presale#latestRoundData() - Missing chainlink min/max price validation

Description

Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed, among other details for the latest round. Chainlink aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the `minPrice` instead of the actual price of the asset.

```
function getLatestPriceETH() public view returns (uint256) {

        (, int256 price, , , ) = priceFeedBNB.latestRoundData();

        return uint256(price * 1e10); }
```

Impact

Users can interact with the system at mispriced valuations, resulting in misallocation of assets. For example, if the market price of an asset drops far below the circuit breaker's `minPrice`, the system would continue accepting deposits at the higher capped value, allowing users to gain more credit than they should.

Recommendation

Implement checks to compare the returned price against expected `minPrice` and `maxPrice` bounds, reverting if the answer lies outside the range.

```
+ require(price > minPrice && price < maxPrice, "oracle price out of bounds");
```

Status

Resolved

[M-02] Vrine_Presale#BuyWithBNB() - Referral rewards can exceed presale hardcap

Description

The `BuyWithBNB()` function enforces a hardcap by checking that `TokenSold + ETHToToken(msg.value) <= maxTokeninPresale`. This ensures that direct buyer allocations do not exceed the configured maximum. However, referral rewards are calculated separately based on a percentage of `msg.value` and credited to the referrer's balance without re-validating against the `hardcap`.

The core issue is that referral rewards (`RtokensToTransfer`) are minted from the same token supply as regular buyer allocations but are not included in the `hardcap` check. The `TokenSold` variable is only updated with the buyer's `tokensToTransfer`, leaving referral credits (`RClaimable[_ref]` and `Treferralrewards`) unchecked.

```
function BuyWithBNB(address _ref) public payable nonReentrant {

        //..

        if (_ref != address(0)) { //@audit self referral is possible.

            if (referalPercent != 0) {

                referETHamt = (msg.value * referalPercent) / 100;

                uint256 RtokensToTransfer = ETHToToken(referETHamt);

>>              RClaimable[_ref] += RtokensToTransfer;

                Treferralrewards += RtokensToTransfer;

            }

        }

        //..

    }
```

Impact

Total tokens owed (`TokenSold + Treferralrewards`) can exceed `maxTokeninPresale`, creating insolvency for the sale and making it impossible to honor all claims..

Recommendation

Update the referral reward logic to include referral allocations in the hardcap validation. Either increment `TokenSold` with referral tokens, or enforce a combined check of buyer tokens plus referral tokens against `maxTokeninPresale`.

Status

Resolved

Low

[L-01] Vrine_Presale#changeAdmin() - Wrong event emitted on admin change

Description

The `changeAdmin()` function is designed to allow the contract owner to update the admin address. The function correctly validates that the new admin address is not the zero address and updates the state variable.

However, it emits the event `updateRecipient(newAdmin)` instead of an `updateAdmin` event, which creates inconsistency between the function's purpose and the emitted log.

```
function changeAdmin(address payable newAdmin) external onlyOwner {

        require(newAdmin != address(0), "Invalid admin address");

        admin = newAdmin;

>>      emit updateRecipient(newAdmin);

}
```

Recommendation

Emit the correct `updateAdmin` event in `changeAdmin`.

Status

Resolved

[L-02] Vrine_Presale#setreferalPercent() - Missing _refpercent validation in setreferalPercent()

Description

`ReferalPercent` is owner-controlled with no upper bound. Since referral rewards are calculated as `ETHToToken(msg.value * referralPercent / 100)` and not counted in the hard cap, large values (`≥100`) dramatically inflate `RClaimable` and `Treferral` rewards beyond intended supply and likely beyond any escrowed token balance.

```
function setreferalPercent(uint256 _refpercent) external onlyOwner {

        referalPercent = _refpercent; //@audit can be more than 100%

}
```

Recommendation

Enforce sane bounds on `referralPercent` (`e.g., 0–30`) and include referral rewards in cap checks.

Status

Resolved

[L-03] Vrine_Presale#BuyWithBNB() - Using transfer instead of call when sending BNB to feeReceiver

Description

When sending native `BNB` through `BuyWithBNB()` function, we use `transfer()` instead of native low-level call. This is not the best practice, when transferring native tokens as transfer only use `2300` gas, which is only enough for event emitting and not for fallback logic.

Keeping into consideration that to be a smart contract wallet, there can be a fallback logic there. For example `SafeWallet` can consume `> 2300` or the first time accessing it. and if any smart wallet implements a fallback logic the tx will fail.

```
function BuyWithBNB(...) public payable nonReentrant {

        //..

        // Update claimable tokens for the buyer

>>      payable(feeReceiver).transfer(msg.value);

        uint256 tokensToTransfer = ETHToToken(msg.value);

        //..

}
```

Recommendation

We should use `call()` instead of `transfer()` for transferring native token.

Status

Resolved

[L-04] Vrine_Presale#BuyWithBNB() - Buyer Can Self Refer To Gain Extra Rewards

Description

The `BuyWithBNB()` function allows buyers to specify a referral address `_ref` which, if valid, receives referral rewards in tokens. There is no restriction preventing the buyer from entering their own address as `_ref`.

```
function BuyWithBNB(...) public payable nonReentrant {

        //..

>>       if (_ref != address(0)) {

            if (referalPercent != 0) {

                referETHamt = (msg.value * referalPercent) / 100;

                uint256 RtokensToTransfer = ETHToToken(referETHamt);

                RClaimable[_ref] += RtokensToTransfer;

                Treferralrewards += RtokensToTransfer;

            }

        }

        //..

}
```

Recommendation

Add a check to prevent self-referrals by ensuring `_ref != msg.sender`.

Status

Resolved

QA

[Q-01] Vrine_Presale - Redundant Functions

Description

The contract implements two functions to set the presale price per USDT: `setPresalePricePerUsdt` (restricted to `onlyOwner`) and `setPresalePricePerUsdtAdmin` (restricted to `onlyAdmin`).

Both functions perform the same logic, updating `TokenPricePerUsdt` after requiring the input to be greater than zero, and emitting the same Price event. The duplication increases maintenance overhead and creates unnecessary code redundancy.

Recommendation

Remove one of the redundant functions and enforce a single, consistent authority (either owner or admin) for managing presale price updates.

Status

Resolved

Centralisation

The Vrine 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 Vrine 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