hashlock.

Security Audit

Posse Studios (GameFi)

Table of Contents

Executive Summary	4
Project Context	4
Audit scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	11
Audit Resources	11
Dependencies	11
Severity Definitions	12
Status Definitions	13
Audit Findings	14
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 Posse Studios team partnered with Hashlock to conduct a security audit of their Yeehaw contract. 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

Posse Studios, the development arm of the Posse ecosystem, specializes in cutting-edge blockchain and gaming technologies, pioneering a bold new frontier in blockchain gaming and decentralized finance. At the heart of this ecosystem is Want3d, a Wild West-inspired GameFi metaverse that includes both an open-world desktop game and a mobile third-person shooter. In these games, players can immerse themselves in competition, skill development, and free play—blending fun with real earning potential. This immersive world features a unique multi-token system that strengthens economic stability by separating in-game and governance currencies. Captivating 3D NFTs offers cross-project functionality, enabling activities like bounty hunting and utility across various platforms. Posse Finance's SocialFi DEX and GameFi marketplace complete the ecosystem, seamlessly uniting digital art, gameplay, and decentralized finance.

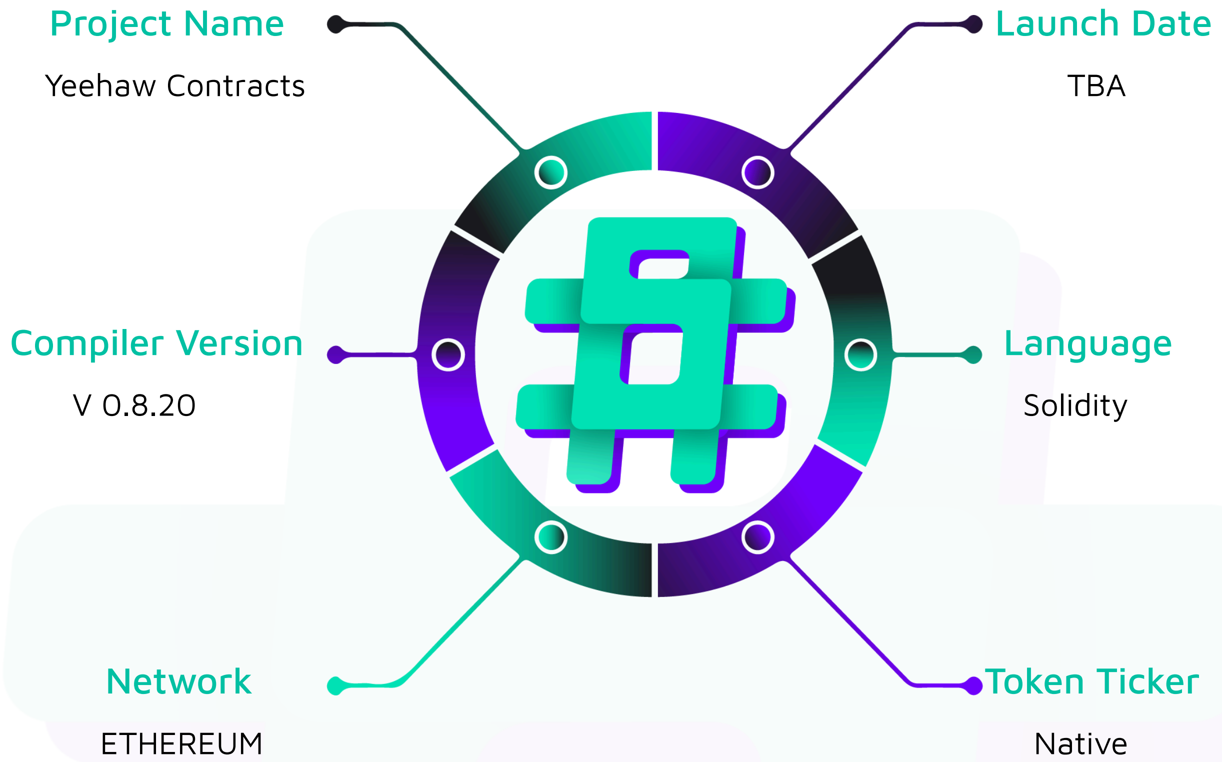
Project Name: Posse Studios (Yeehaw Contracts)

Compiler Version: 0.8.20

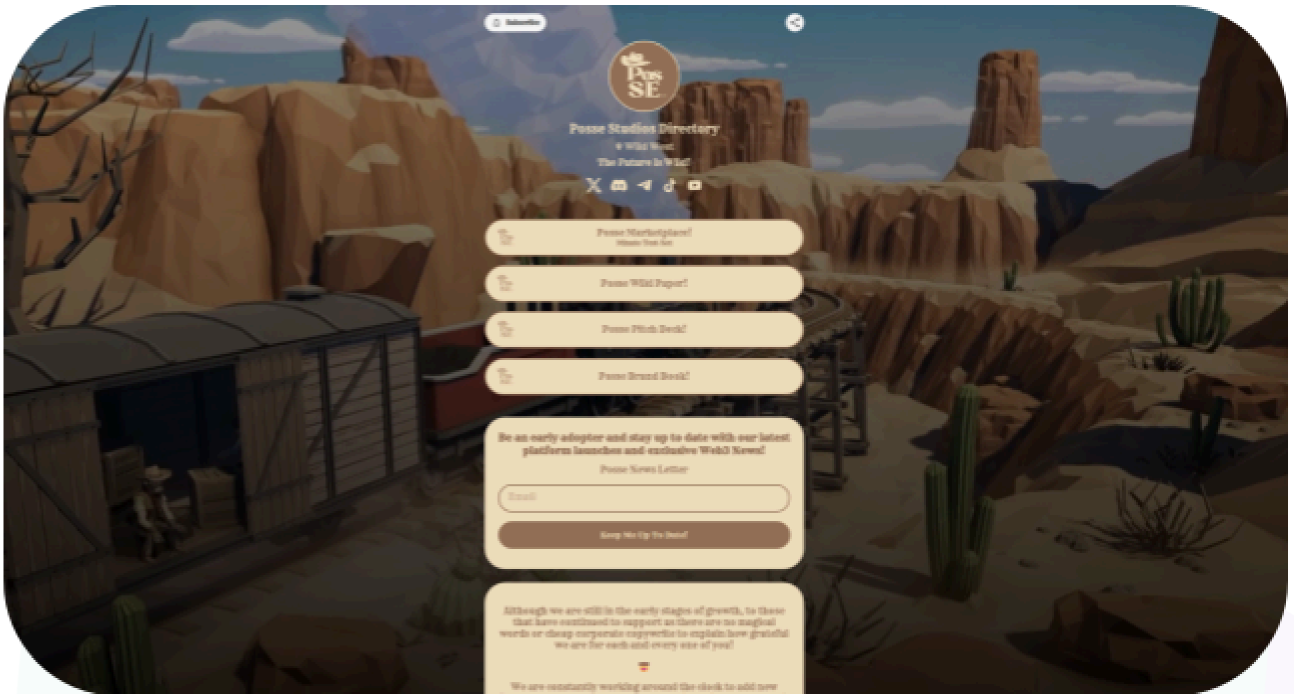
Website: https://possestudios.info/

Logo:

Visualised Context:



Project Visuals:



Audit scope

We at Hashlock audited the solidity code within the Yeehaw Contracts. 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	**Yeehaw Smart contracts**
Platform	**Ethereum / Solidity**
Audit Date	**January 2025**
Contract 1	ERC20FixedSupply.sol
MD5 Hash	0527ed4847f9032141db680cdb242789
Contract 2	BondingCurve.sol
MD5 Hash	f1ec3c053d68fe83a28c1ec3d2992017
Contract 3	PumpFactory.sol
MD5 Hash	a0ab3d493894005d06a966833672bb49
GitHub Commit Hash	32e2a669bf87e02e202fb68e1b9f6285a0665db3

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 and Uniswap contracts. We initially identified some significant vulnerabilities that have since been addressed.

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:

3 Low severity vulnerabilities

1 Gas Optimisation finding

3 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
PumpFactory.sol - Deploys new ERC20 tokens and their associated bonding curves - Sets and maintains protocol-wide parameters for all new tokens/curves - Stores virtual reserves for bonding curve calculations - Manages fee percentages for trading - Controls ETH allocations for liquidity, fees, and rewards - Tracks the relationship between tokens and their bonding curves - Handles access control for fee recipient management - Enables initial token purchases during creation	**Contract achieves this functionality.**
ERC20FixedSupply.sol - Periphery contract used to: - Act as the ERC20 token contract equivalent for the protocol	**Contract achieves this functionality.**
BondingCurve.sol - Manages the initial token distribution phase - Handles buy and sell functions with fee calculations - Tracks ETH and token reserves for price calculations - Controls trading restrictions before launch - Automates liquidity provision to Uniswap V2	**Contract achieves this functionality.**

upon completion - Burns initial LP tokens to lock liquidity - Distributes rewards and fees when curve completes - Manages the transition from bonding curve to open trading - Ensures one-way transition from initial distribution to launched state	

Code Quality

This audit scope involves the smart contracts of the Yeehaw Contracts, 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 mostly well commented on and closely follows best practice nat-spec styling. However, there are parts that nat-spec should be better enforced.

Audit Resources

We were given the Yeehaw smart contract code in the form of standard files.

As mentioned above, most 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

Low

[L-01] ERC20FixedSupply - Missing event emission for critical token launch state change

Description

The `ERC20FixedSupply` contract's `launch` function changes a critical state variable without emitting an event. This state change affects token transfer restrictions but lacks transparency, as it affects off-chain tracking without an on-chain record. Combined with the absence of other means of tracking and its occurrence for every launch, this can lead to confusion and potential issues.

Recommendation

A simple addition of a launch event will eliminate the bug.

Status

Resolved

[L-02] BondingCurve - Forced ETH via Selfdestruct can create balance inconsistencies

Description

The `BondingCurve` contract can receive ETH through `selfdestruct` despite having a reverting `receive` function. While current balance checks are safely implemented to only verify minimum amounts, this could affect future implementations or external contracts that might rely on the contract's ETH balance matching its tracked reserves.

```
receive() external payable {

    revert();
```

```
}

// Current balance checks are safe:

require(ethAmount <= address(this).balance, "Bonding curve does not have sufficient funds");

require(address(this).balance >= ethAmountToSendLP, "Insufficient ETH balance");
```

An attacker can send a minimal amount to the contract through self-destructing another contract that holds ETH. Then the forced ETH creates an inconsistency between actual balance and `ethReserve` tracking. While this doesn't affect current protocol operations, it could impact future upgrades/additions and external contracts interacting with the protocol that relies on specific balance tracking instead of `balance.(this)`

Recommendation

Consider adding specific documentation that makes it clear like:

```
/// @dev Note: Contract's ETH balance may exceed ethReserve due to forced ETH transfers (e.g., selfdestruct).

/// External integrations should rely on ethReserve rather than address(this).balance
```

Comments can also be added to warn about the behaviour. Finally, a more robust solution would be to implement a separate function to track or "get" the excess balance and use it for the protocol's benefit.

Status

Resolved

[L-03] PumpFactory- Zero address validation for critical infrastructure setters

Description

The `PumpFactory` contract's admin setter functions for critical infrastructure addresses lack zero address validation. This could allow an admin to accidentally/mistakenly set critical protocol addresses to the zero address, disrupting core protocol functionality.

```
function setFeeRecipient(address recipient) external onlyRole(ADMIN_ROLE) {

    if (feeRecipient == recipient) revert ValueUnchanged();

    feeRecipient = recipient;

    emit FeeRecipientUpdated(recipient);

}


function setRouter(address _router) external onlyRole(ADMIN_ROLE) {

    if (router == _router) revert ValueUnchanged();

    router = _router;

    emit RouterUpdated(_router);

}


function setPriceFeed(address feed) external onlyRole(ADMIN_ROLE) {

    if (priceFeed == feed) revert ValueUnchanged();

    priceFeed = feed;

    emit PriceFeedUpdated(feed);

}
```

Recommendation

Add zero address validation to the setter functions.

Status

Resolved

Gas

[G-01] ERC20FixedSupply and PumpFactory - Unnecessary parameter and calculations lead to gas inefficiency and less clear code structure

Description

The protocol's token implementation includes unnecessary parameters and on-chain calculations for a fixed total supply. The total supply is intended to always be 1 billion tokens, making the current implementation both gas-inefficient and needlessly complex.

current implementation:

```
// PumpFactory.sol

uint256 public tokenTotalSupply;  // Unnecessary variable for fixed 1 billion supply

constructor(uint256 _tokenTotalSupply, ...) {

    tokenTotalSupply = _tokenTotalSupply;

}

function setTokenTotalSupply(uint256 _newSupply) external onlyOwner {    // Unnecessary setter

    tokenTotalSupply = _newSupply;

    emit TokenTotalSupplyUpdated(_newSupply);

}

// ERC20FixedSupply.sol
```

```
constructor(string memory name, string memory symbol, uint256 initialSupply,
string memory tokenURI_) {

       _mint(msg.sender, initialSupply * 10 ** decimals());   // Unnecessary
on-chain calculation

}
```

This leads to unnecessary gas costs during each deployment, unneeded storage operations, redundant constructor parameters, and calculations done on-chain. There is also the aspect of the codebase being more complex than it should be, which can lead to confusion and human error.

Recommendation

Consider simplifying the implementation to use a constant and fixed supply from the get-go.

Status

Resolved

QA

[Q-01] PumpFactory#createToken - Unlimited token creation enables reward farming

Description

The `PumpFactory` contract allows unlimited token creation with no restrictions or costs. For each token, the creator is designated as `TOKEN_DEVELOPER` and receives an immutable `ETH_AMOUNT_FOR_DEV_REWARD` when the bonding curve completes. Once a bonding curve is deployed, its reward amount cannot be changed, even by the protocol owner.

Vulnerability Details

This enables a critical attack vector where an attacker can create a very large amount of tokens paying the minimal ETH investment required (marginally not at all, except if they want to boost the process, but still no actual incentive for them), then wait passively for their tokens to complete their respective bonding curves to receive the rewards. The rewards are guaranteed and cannot be changed by the protocol owner, because the value is locked and a reward change will affect only future tokens.

Proof of Concept

From `PumpFactory` and `BondingCurve`

In `PumpFactory`, there is no minimal cost to create tokens:

```
function createToken(string memory name, string memory symbol, string memory tokenURI)
        external
        payable
        returns (address)
    {
          ERC20FixedSupply token = new ERC20FixedSupply(name, symbol, tokenTotalSupply,
tokenURI);
        BondingCurve bondingCurve = new BondingCurve(
            address(msg.sender),
            address(token),
            virtualTokenReserve,
            virtualEthReserve,
            swapFeePercentage,
            ethAmountForLiquidity,
            ethAmountForLiquidityFee,
            ethAmountForDevReward,
            uniswapV2RouterAddress
```

```
        );
              require(token.transfer(address(bondingCurve), token.totalSupply()), "ERC20
transfer failed");
        getTokenBondingCurve[address(token)] = address(bondingCurve);
        IOwnable(address(token)).transferOwnership(address(bondingCurve));
        emit TokenCreated(address(token), address(bondingCurve));
        if (msg.value > 0) {
            bondingCurve.buy{value: msg.value}(msg.sender);
        }
        return address(token);
    }
```

Impact

The impact on the protocol is severe, as there can be token spam since there is a minimal cost to create tokens without any limit per address. This can lead to bloating and the creation of a large quantity of "empty" tokens awaiting investment. To add to that, the rewards are immutable, so any potential change will only affect the future creation of tokens, so there is no mitigating through altering rewards. Moreover, the rewards are paid from user-provided funds, the exploitation of which, along with the exploit in general, can lead to the protocol's quality degradation, the erosion of user trust, and network congestion.

Recommendation

Consider implementing a meaningful creation fee, or a token creation limit per address. Additionally, you might implement a requirement for `TOKEN_DEVELOPER` to stake a minimum amount of tokens of their own to be eligible for the reward. Alternatively, an additional cooldown for token creation per address can be added. Lastly, some more extreme measures would include an emergency stop/pause mechanism, a reputation system for creators, or the implementation of a whitelist for creators.

Note

The Hashlock team initially categorized this finding as a low vulnerability due to the reward farming and its possibilities to scale up in the future. After discussing with the development team from Posse Studios, the finding was moved to informational due to different interpretations of the outcomes of the finding and its risks to vulnerabilities.

Status

Resolved

[Q-02] Floating pragma used across the protocol

Description

Throughout the codebase, multiple instances of floating pragma directives were identified. Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

Recommendation

Consider using fixed pragma directives.

Status

Resolved

[Q-03] BondingCurve - Minor precision loss in fee calculation

Description

In `BondingCurve`, the `_calculateBuyFee` function performs division before multiplication, which causes rounding to occur earlier in the calculation resulting in rounding differences in fee calculations that slightly favor users.

```
function _calculateBuyFee(uint256 amount) internal view returns (uint256) {

    unchecked {

        return (amount / (BASIS_POINTS + SWAP_FEE_PERCENT)) * SWAP_FEE_PERCENT;
```

```
    }
}
```

Recommendation

For mathematical completeness, the fee calculation could be reordered to perform multiplication before division

```
function _calculateBuyFee(uint256 amount) internal view returns (uint256) {
    unchecked {
        return (amount * SWAP_FEE_PERCENT) / (BASIS_POINTS + SWAP_FEE_PERCENT);
    }
}
```

Status

Resolved

Centralisation

The Yeehaw contracts value 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 Yeehaw contracts seem 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.