hashlock.

Security Audit

Cabal Labs (DeFi)

May 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.

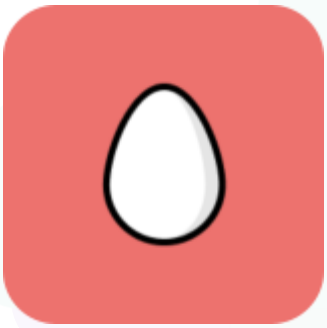
Executive Summary

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

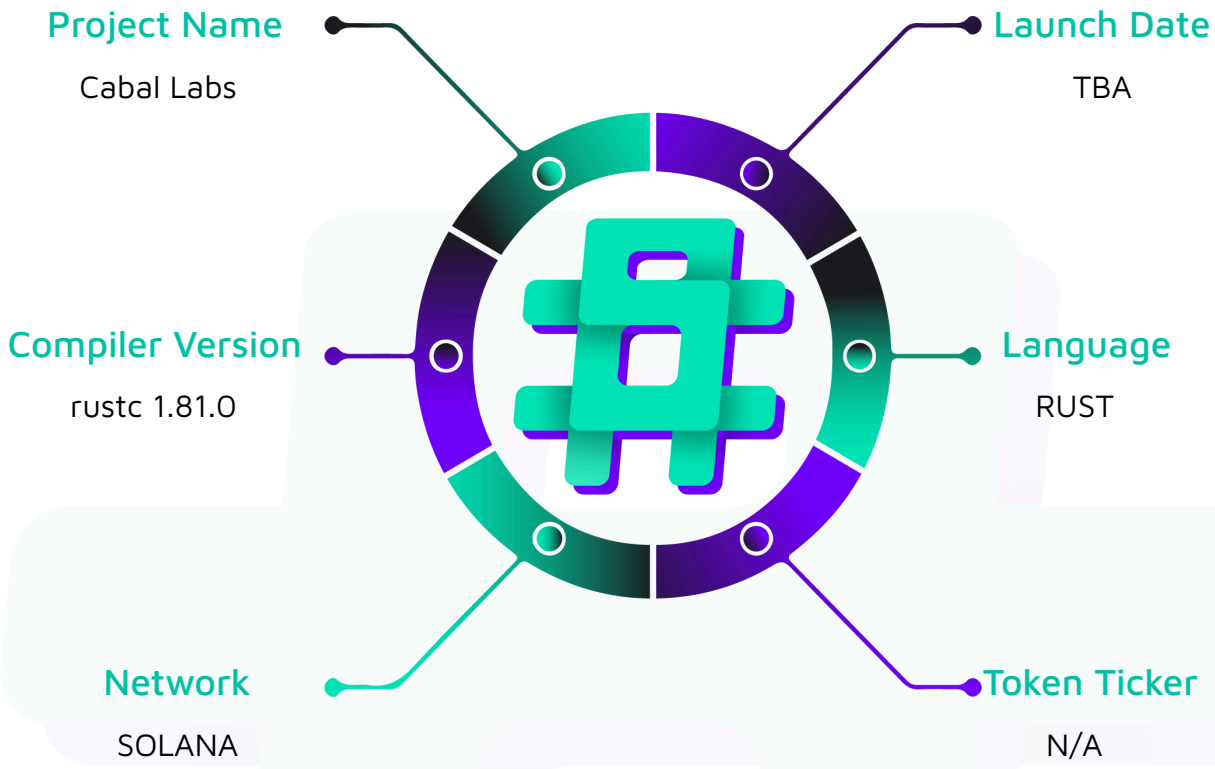
Project Context

Cabal Labs is a DeFi dApp built on Solana Network that allows users to create new tokens and have a fair sale. These tokens can be swapped in and out for Solana native currency. Lastly, liquidity pools will be launched to Raydium.

Project Name: Cabal Labs
Project Type: DeFi, dApp
Compiler Version: rustc 1.81.0
Logo:



Visualised Context:



Audit scope

We at Hashlock audited the Rust code within the Cabal Labs 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	Cabal Labs Smart Contracts
Platform	Solana / Rust
Audit Date	May, 2025
Github URL	https://github.com/cabalfun/cabal-solana-meme-contract-raydium
Component 1	buy.rs
Component 2	create_amm_liquidity.rs
Component 3	initialize.rs
Component 4	lock_amm_liquidity.rs
Component 5	sell.rs
Component 6	create_config.rs
Component 7	update_config.rs
Audited GitHub Commit Hash	5fc17a37bf82edbb8beb6f56b8cb5ea3dd0e7caa
Fix Review GitHub Commit Hash	b7d6b7fd847420056435b8c96fd829ec1a73bcc3

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.

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:

1 Medium severity vulnerabilities

3 Low severity vulnerabilities

1 Gas Optimisations

4 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
buy.rs - Allows users to purchase mint tokens.	**Contract achieves this functionality.**
create_amm_liquidity.rs - Allows users to create AMM liquidity.	**Contract achieves this functionality.**
initialize.rs - Allows users to initialize a mint token and native token pool.	**Contract achieves this functionality.**
lock_amm_liquidity.rs - Allows users to lock AMM liquidity after it is fully funded.	**Contract achieves this functionality.**
sell.rs - Allows users to sell mint tokens.	**Contract achieves this functionality.**
create_config.rs - Allows the admin to create Cabal configuration.	**Contract achieves this functionality.**
update_config.rs - Allows the admin to update existing Cabal configuration.	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Cabal Labs 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 Cabal Labs 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

Medium

[M-01] programs/cabal_meme_v1/src/states/cabal.rs#require_amm_lfg - Incorrect flag validation for require_amm_lfg

Description

The `require_amm_lfg` function checks for the incorrect flag. Ideally, it should validate the `self.amm_lfg` flag is set to true. However, the current implementation incorrectly checks for `self.lfg`.

Impact

If the authority disables the `self.amm_lfg` flag to prevent users from calling the `CreateAmmLiquidity` and `LockAmmLiquidity` instructions, this restriction will be ineffective.

Recommendation

Consider updating the `require_amm_lfg` function to check for `self.amm_lfg` flag.

Status

Resolved

Low

[L-01] programs/cabal_meme_v1/src/states/cabal.rs#update - Two step ownership transfer is not implemented

Description

Suppose the authority provided `ConfigOption::AuthorityAddress`, the `Cabal` authority will be updated via the `UpdateConfig` instruction. This is problematic because if the ownership is set to an incorrect address, the ownership will be lost.

Impact

The `Cabal` authority will be invalid, and no one can govern the protocol. The authority will be renounced unintendedly.

Recommendation

Consider implementing a two-step ownership transfer within two transactions: the first transaction occurs by nominating an admin, and the second transaction requires the nominated admin to accept the ownership transfer.

This mechanism ensures that the receiver is intended to receive the ownership transfer and prevents the human error of transferring the ownership to an incorrect address.

Status

Acknowledged

[L-02] programs/cabal_meme_v1/src/states/cabal.rs#update - Cabal misconfiguration may cause the protocol to fail to work properly

Description

The `UpdateConfig` instruction allows the authority to update the configuration for the `Cabal` account. However, several validations are not implemented to ensure the protocol works correctly:

- The `tx_min_cap` and `tx_max_cap` are not validated to ensure that `tx_min_cap` is less than `tx_max_cap`.
- The `tx_min_fee_bps` and `tx_max_fee_bps` are not validated to ensure that `tx_min_fee_bps` is less than `tx_max_fee_bps`.
- The `pool_min_cap` and `pool_max_cap` are not validated to ensure that `pool_min_cap` is less than `pool_max_cap`.
- The `tx_token_creator_fee_bps` is not validated to ensure its value is not zero and does not exceed `BPS_DENOMINATOR`.

Impact

The `Cabal` configuration will be invalid, and the protocol may not work as expected. For example, users cannot buy tokens because a zero `token_creator_fee` is configured.

Recommendation

Consider implementing validations as mentioned above.

Status

Resolved

[L-03] programs/cabal_meme_v1/src/instructions/initialize.rs#Initialize - token_mint PDA account is not validated fully

Description

In the following instances, PDAs for the `Mint` accounts are not validated:

- Line 14 of `programs/cabal_meme_v1/src/instructions/buy.rs`
- Line 19 of `programs/cabal_meme_v1/src/instructions/create_amm_liquidity.rs`
- Line 18 of `programs/cabal_meme_v1/src/instructions/lock_amm_liquidity.rs`
- Line 14 of `programs/cabal_meme_v1/src/instructions/sell.rs`

Impact

Invalid PDA accounts could be incorrectly supplied, potentially bypassing authentications. While the current implementation prevents this from happening (as other accounts require the correct PDA), not validating PDAs opens up possibilities for exploitation if the codebase were to change in the future.

Recommendation

Consider validating the `Mint` account similar to the validation in the `Initialize` instruction by checking `mint::decimals = 6`.

Status

Acknowledged

Gas

[G-01] programs/cabal_meme_v1/src/instructions/*.rs - PDA bump can be stored to decrease gas consumption

Description

Across the contract, multiple instances exist where the PDAs of `Cabal` and `Pool` are required. Since their bumps are not stored, Anchor must derive the PDA by iterating over all bumps, which is gas-consuming.

Recommendation

The iteration can be skipped by storing the `Cabal` and `Pool` bumps when initialized and accessing them directly in the instructions.

For example, in the `CreateConfig` instruction, the bump computed for `ctx.accounts.cabal_state` can be stored inside the `Cabal` struct, and used directly as `Cabal.bump`.

A similar approach can be implemented in the `Initialize` instruction to record the `ctx.accounts.pool_state` and `ctx.accounts.pool_liquidity` bump values in the `Pool` struct and access them directly. This also removes the need to call `Pubkey::find_program_address` in several instances, decreasing overall gas efficiency.

Status

Acknowledged

QA

[Q-01] programs/cabal_meme_v1/src/instructions/sell.rs#Sell - Unneeded init_if_needed macro in investor_token_account

Description

In line 32, the `investor_token_account` account is implemented with the `init_if_needed` macro. This macro is unnecessary as the users will only call the `Sell` instruction to swap mint tokens for lamports, which means they must already have an associated token account.

Recommendation

Consider removing the `init_if_needed` macro and replacing it with the `mut` macro.

Status

Resolved

[Q-02] programs/cabal_meme_v1/src/instructions/admin/update_config.rs#UpdateConfig - Unneeded mut macro and system_program account

Description

The `UpdateConfig` instruction is called to update the `ctx.accounts.cabal_state` account. However, the `mut` macro for the `authority` account is unneeded, and the `system_program` is not required for this instruction.

Recommendation

Consider removing the `mut` macro for `authority` and removing the `system_program` account requirement.

Status

Resolved

[Q-03] programs/cabal_meme_v1/src/states/pool.rs#sell - Misleading min_native_token_amount_in parameter in Sell instruction

Description

The `min_native_token_amount_in` parameter is misleading. It intuitively hints to users that this will be the minimum amount of native tokens to provide, which is incorrect because it is actually the minimum amount of native tokens users must receive during the `Sell` instruction transaction.

Recommendation

Consider renaming the parameter to `min_native_token_amount_out`.

Status

Resolved

[Q-04] programs/cabal_meme_v1/src/states/pool.rs#create_amm_liquidity, lock_amm_liquidity - Unneeded and unused cabal_state parameter

Description

The `cabal_state` parameter in the `create_amm_liquidity` and `lock_amm_liquidity` functions is unnecessary as they are unused.

Recommendation

Consider removing the `cabal_state` parameter in the `create_amm_liquidity` and `lock_amm_liquidity` functions.

Status

Resolved

Centralisation

The Cabal Labs 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 Cabal Labs 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