hashlock.

Security Audit

Bucket Protocol (DeFi)

August 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

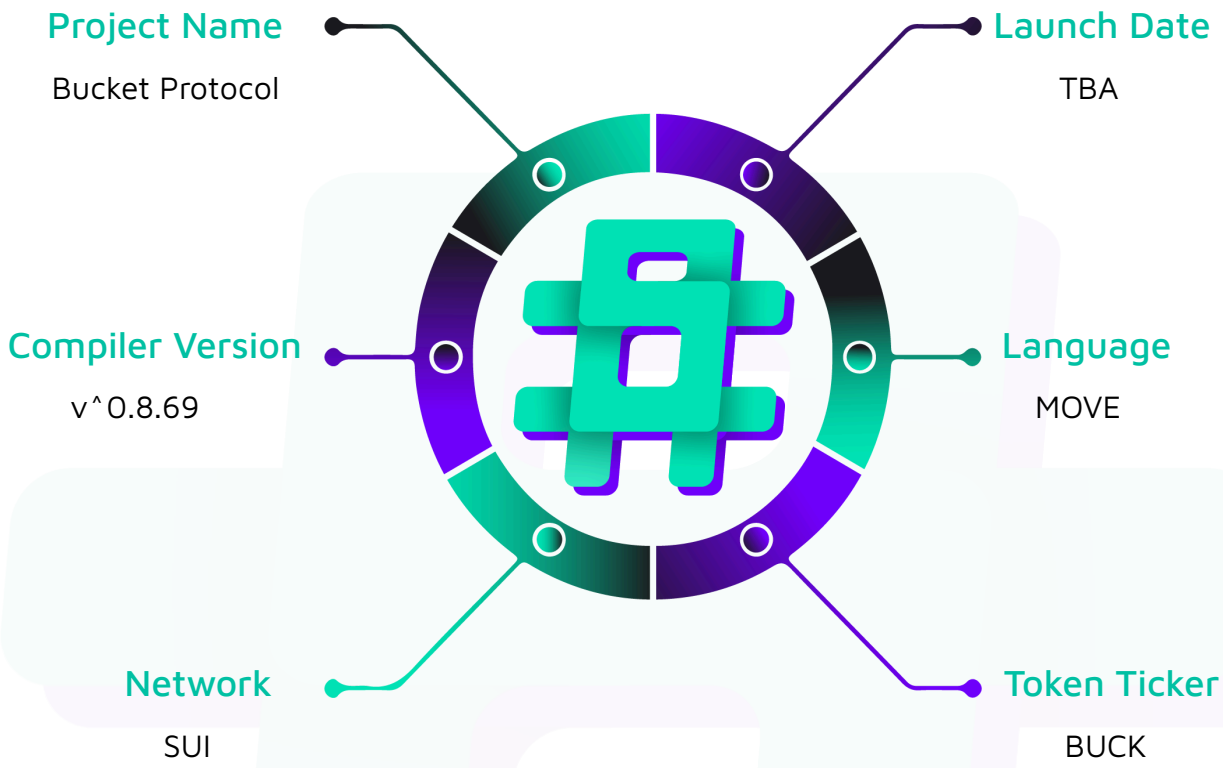
The Bucket Protocol team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

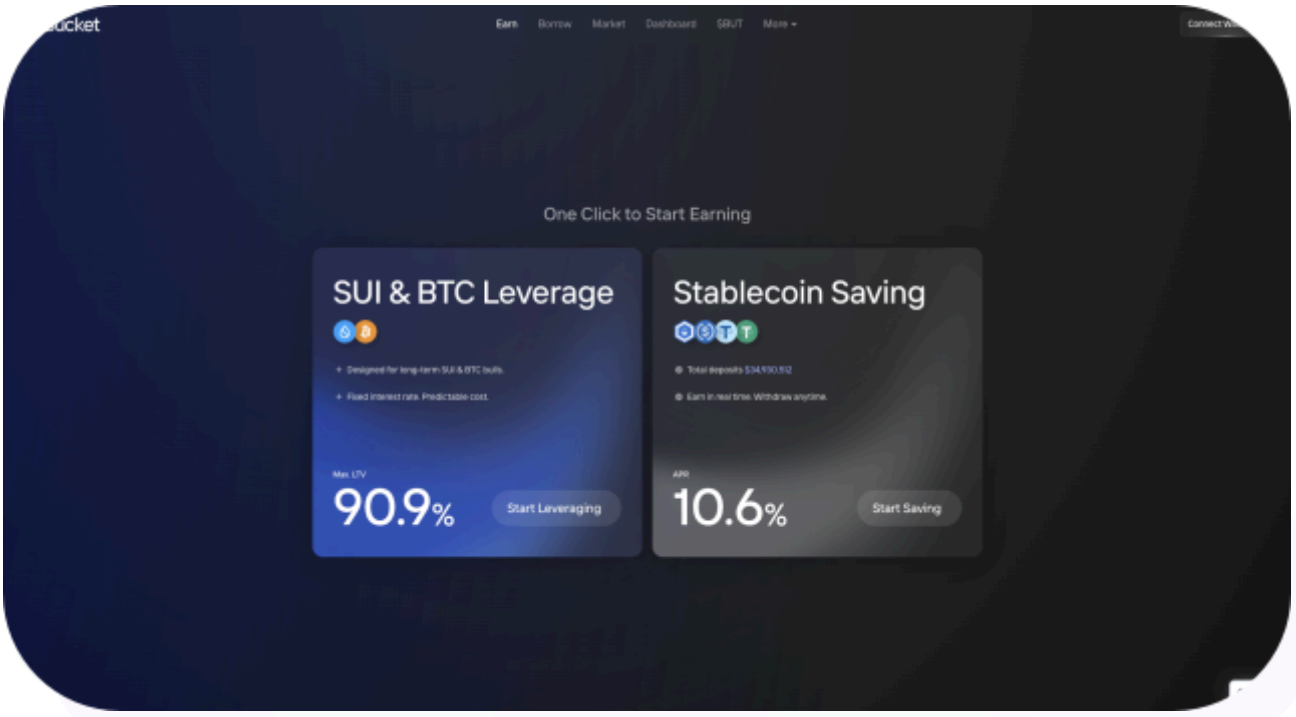
Bucket Protocol is a DeFi stablecoin platform built on the Sui blockchain that allows users to open Collateralized Debt Positions (CDPs), called "Bottles," by locking assets such as SUI, BTC, ETH, and LSD to borrow its USD-pegged stablecoin, $BUCK. The protocol is designed with low collateral requirements (around 110%) and features fast liquidation mechanisms to maintain system stability. Users can stake $BUCK to receive sBUCK, a yield-bearing token that accrues a fixed savings rate (BSR) funded by protocol revenue. Additionally, users can participate in Deposit-to-Farm and Liquidity Mining programs to earn "Drops," which are convertible into the upcoming governance token $BUT.

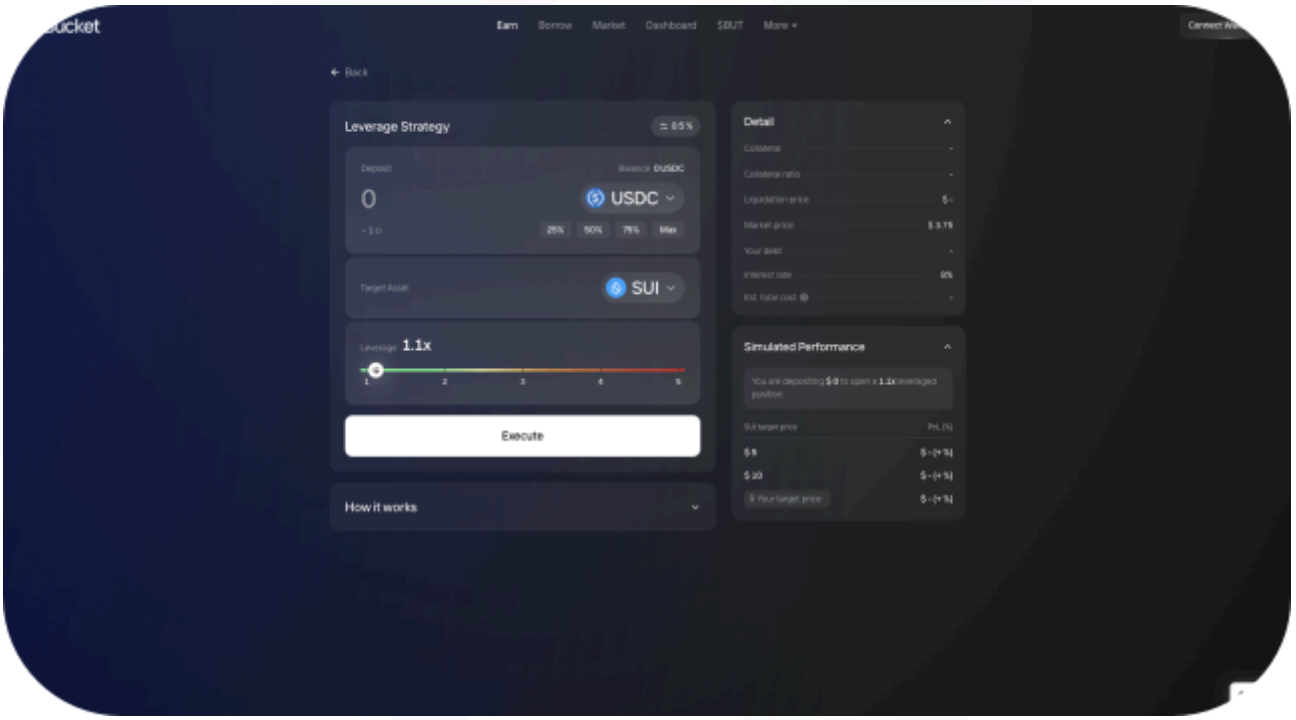
Project Name: Bucket Protocol
Project Type: DeFi, Stablecoin
Compiler Version: ^0.8.42
Website: https://www.bucketprotocol.io/earn
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the move code within the Bucket Protocol 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	**Bucket Protocol Smart Contracts**
Platform	**Sui/Move**
Audit Date	**August, 2025**
Contract 1	vault.move
Contract 2	aggregator.move
Contract 3	pool.move
Contract 4	usdb.move
Audited GitHub Commit Hash	32c4c41438345407fd4b8b2cd4c5c01651b171f5
Fix Review GitHub Commit Hash	75886a65030e81687b58c743e394668a16508d3f

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Hashlocked"**. 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 and acknowledged.

Hashlock found:

2 Medium severity vulnerabilities

6 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
vault.move - Allows users to: - Open CDPs by depositing collateral and borrowing USDB - Repay debts and withdraw collateral - Accrue interest on debt positions - Allows admins to: - Set interest rates, supply limits, and collateral ratios - Configure liquidation rules and security levels	**Contract achieves this functionality.**
aggregator.move - Periphery contract used to: - Aggregate prices from multiple oracle sources with weighted rules - Apply weight thresholds and remove outlier prices - Allows admins to: - Configure weights, thresholds, and tolerance parameters	**Contract achieves this functionality.**
pool.move - Allows users to: - Swap collateral assets for USDB - Benefit from partner-specific fee rates - Allows admins to: - Create pools and set fee configurations	**Contract achieves this functionality.**

- Configure price tolerance parameters	
usdb.move - Periphery contract used to: - Mint and burn USDB tokens with version validation - Collect fees and revenue from operations - Allows admins to: - Set supply limits and manage module access - Configure beneficiary addresses for fee collection	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Bucket Protocol 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 Bucket Protocol 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] Vault.move - Dust donor repayment can DoS full liquidations via strict overpayment check

Description

`vault::liquidate` aborts when `repayment.value()` exceeds the debtor's live `debt_amount`.

```
if (repayment.value() > debt_amount) {

    err_invalid_liquidation();

};
```

Vulnerability Details

Because debt can decrease between a liquidator composing a transaction and its execution, an attacker can front-run with a tiny donor repayment (e.g., 1 USDB). That makes the liquidator's intended full-close slightly over the new debt, and the call reverts. This stems from a strict `repayment > debt_amount` check without tolerating concurrent decreases in debt.

Impact

This creates a practical denial-of-service against full liquidations and fixed-amount partial liquidations at negligible cost.

Recommendation

Handle overpayments gracefully instead of reverting. Clamp the amount actually used to `min(repayment, current_debt)` and refund any excess to the liquidator.

Status

Acknowledged

[M-02] Vault.move - Mismatched security-level checks can block liquidations

Description

The protocol applies two different security-level checks along the liquidation path. The `liquidate` entry checks level 1 (as intended for safety-critical actions), but the resulting `UpdateRequest` carries `deposit = 0`. Inside `update_position`, the code interprets "no deposit" as a borrow/repay/withdraw flow and enforces level 2.

```
// check security by actions
if(request.deposit_amount() > 0) {
    // deposit actions will only be blocked when the security level equal to 1
    vault.check_secutiry_level(1);
}else{
    // borrow; repay; witdraw;
    vault.check_secutiry_level(2);
};
```

Vulnerability Details

If `vault.security_level == 2`, `update_position` reverts, even though `liquidate` already allowed the operation at level 1. This mismatch turns security level 2 into a blanket block on liquidations.

Impact

This creates a practical denial-of-service against full liquidations and fixed-amount partial liquidations at negligible cost.

Recommendation

When the vault is set to security level 2—often during incidents to restrict riskier actions—liquidations also fail, preventing unhealthy positions from being resolved. This creates a denial-of-service on liquidation during the exact periods when it is most needed.

Status

Resolved

Low

[L-01] Vault.move - Missing Event Emission in set_interest_rate Function

Description

The `set_interest_rate()` function updates the vault's interest rate but does not emit an event to track this critical parameter change, unlike other similar admin functions.

Vulnerability Details

The function modifies a critical vault parameter (interest rate) without emitting any event for transparency.

Other admin functions properly emit events:

- `set_supply_limit()` emits `SupplyLimitUpdated`
- `set_liquidation_rule()` emits `LiquidationRuleUpdated`

Proof of Concept

```
public fun set_interest_rate<T>(
    vault: &mut Vault<T>,
    treasury: &mut Treasury,
    _cap: &AdminCap,
    clock: &Clock,
    interest_rate_bps: u64,
    ctx: &mut TxContext,
) {
    version::assert_valid_package(treasury);
    vault.collect_interest(treasury, clock, ctx);
    vault.interest_rate = double::from_bps(interest_rate_bps); // No event emitted
}
```

Impact

Inconsistent event emission across admin functions

Recommendation

```
events::emit_interest_rate_updated<T>(
    object::id(vault),
    vault.interest_rate,
    double::from_bps(interest_rate_bps),
);
```

Status

Resolved

[L-02] Vault.move - Function Name Contains Typo

Description

The function `check_secutiry_level()` contains a typo in its name - "secutiry" should be "security".

Vulnerability Details

The internal function has a misspelled name that could lead to confusion during development

Proof of Concept

```
fun check_secutiry_level<T>(vault: &Vault<T>, level: u8) {
//     ^^^^^^^^^ TYPO
            if(vault.security_level   !=   0   &&   level   >=   vault.security_level)
err_against_security_level();
}
```

Impact

Potential confusion during development

Recommendation

Rename the function to correct the typo

```
fun check_security_level<T>(vault: &Vault<T>, level: u8) {
            if(vault.security_level   !=   0   &&   level   >=   vault.security_level)
err_against_security_level();
}
```

Status

Resolved

[L-03] Vault.move - Rounding down seized collateral favors the liquidated party

Description

In `liquidate`, the seized collateral is computed proportionally and then rounded down:

Vulnerability Details

This debtor-favored rounding systematically underseizes collateral. Example: `repay/debt = ⅓` with `coll = 100` yields `33` instead of `34`, letting the liquidated party retain value they shouldn't.

```
let mut withdraw_amount =

    double::from(repayment.value())

        .mul_u64(coll_amount)

        .div_u64(debt_amount)

        .floor();
```

Impact

Potential loss to liquidators.

Recommendation

Round up in favor of the liquidation process (use `ceil()`), then cap by available collateral (which is already done).

Status

Resolved

[L-04] limited_supply.move - set_limit missing check lets admin set limit below current supply

Description

`limited_supply::set_limit` assigns the new limit without validating it against the current `supply`. If an admin sets `limit < supply`, the module enters a weird state: any future `increase` will abort with `ESupplyExceedLimit`, effectively bricking new mints/borrows until enough burns/repayments reduce `supply` below the new limit.

Impact

Interest minting that depends on available headroom will stall.

Recommendation

Validate on update: require `limit >= self.supply()`. If a hard reduction is desired operationally, perform it only after bringing the `supply` down (via controlled burns/repayments), or gate the setter to reject values that are below the live `supply`.

Status

Acknowledged

[L-05] double/float.move - Missing overflow checks in fixed-point div/round (Double & Float)

Description

In both `double::div/float::div`, the implementation multiplies by `WAD` before dividing. For sufficiently large inputs, this pre-multiply can overflow (`u256/u128`) even when the mathematical result would fit, causing unexpected aborts.

```
public fun div(a: Double, b: Double): Double {
    if (b.value == 0) err_divided_by_zero();
    // BUG: a.value * WAD could overflow
    Double { value: (a.value * WAD) / b.value }
}
public fun div(a: Float, b: Float): Float {
    if (b.value == 0) err_divided_by_zero();
    // BUG: a.value * WAD could overflow
    Float { value: (a.value * WAD) / b.value }
}
```

Similarly, `double::round/float::round` add `WAD/2` before division; if the stored value is near the type max, the addition can overflow.

```
public fun round(v: Double): u64 {
    // BUG: add may overflow
    (((v.value + WAD / 2) / WAD) as u64)
}
public fun round(v: Float): u64 {
    // BUG: add may overflow
    (((v.value + WAD / 2) / WAD) as u64)
}
```

Impact

Unexpected overflows.

Recommendation

Add symmetric overflow guards to these code paths. For division, check `a.value <= max / WAD` (or restructure to divide first where safe). For rounding, check `v.value <= max - WAD/2`. Include boundary tests to assert the guards trigger cleanly rather than overflowing.

Status

Resolved

[L-06] Pool.move - Unused Sheet Element in Pool Struct

Description

The Pool struct contains a `sheet` field of type `Sheet<T, BucketV2PSM>` that is initialized during pool creation but never utilized throughout the contract's functionality.

Vulnerability Details

The `sheet` field is declared in the Pool struct and initialized `in` the `new()` function, but remains completely unused

Proof of Concept

```
public struct Pool<phantom T> has key, store {

    id: UID,

    decimal: u8,

    default_fee_config: FeeConfig,

    partner_fee_configs: VecMap<address, FeeConfig>,

    price_tolerance: Float,

    // states

    balance: Balance<T>,

    balance_amount: u64,

    usdb_supply: u64,

    sheet: Sheet<T, BucketV2PSM>, // @audit: unused field

}
```

Impact

Possible indication of incomplete implementation

Recommendation

Either implement the intended sheet functionality or remove the unused field

Status

Acknowledged

Centralisation

The Bucket Protocol 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 Bucket Protocol 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