Security Audit

Size Credit (DeFi)

June 2025

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

CAUTION

Executive Summary

The Size Credit 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

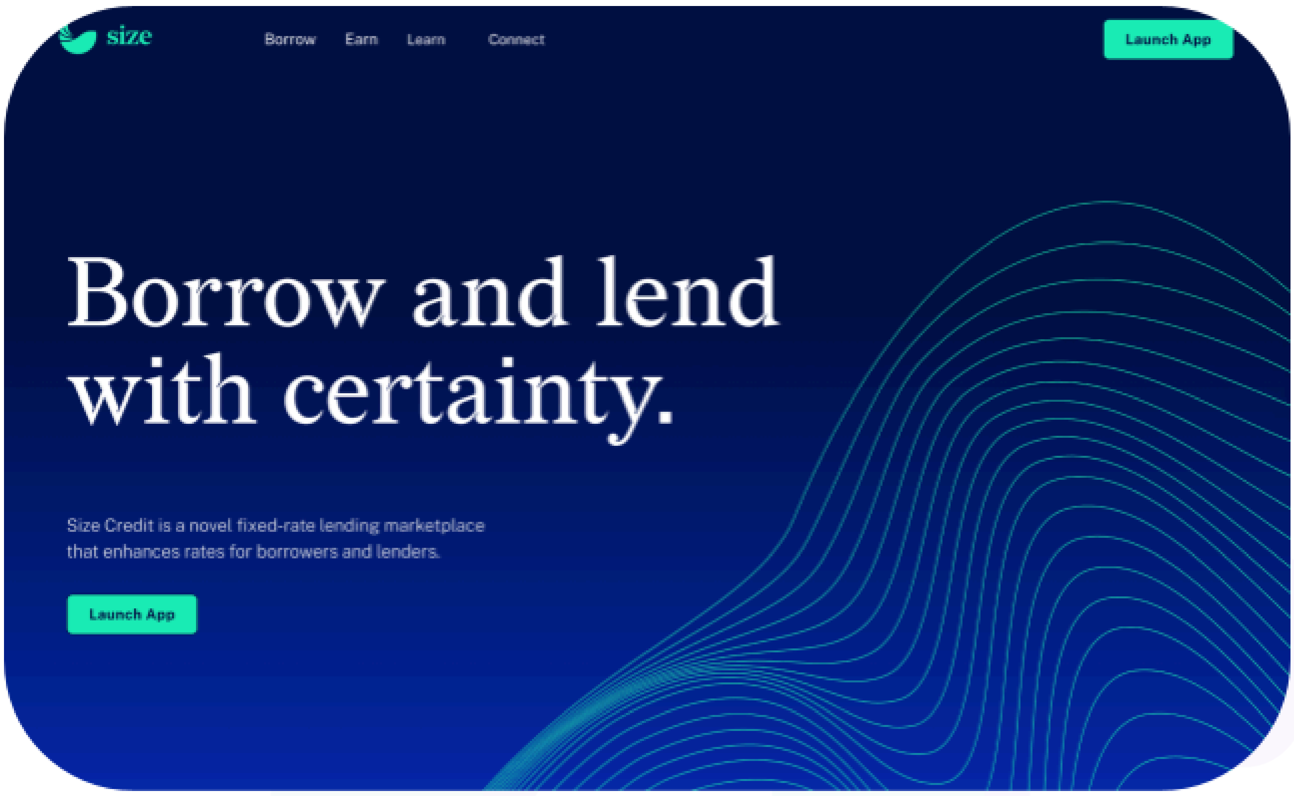
Size Credit is a decentralized finance (DeFi) protocol that facilitates fixed-rate lending and borrowing across customizable maturities, while simultaneously enabling users to earn variable yields. The platform offers flexibility through early position exits and opportunities to speculate on interest rate fluctuations. In essence, Size Credit integrates the benefits of both fixed-rate lending and variable yield strategies, providing users with enhanced financial versatility.

Project Name: Size Credit
Project Type: DeFi
Compiler Version: 0.8.23
Website: https://www.size.credit/
Logo:

Visualised Context:

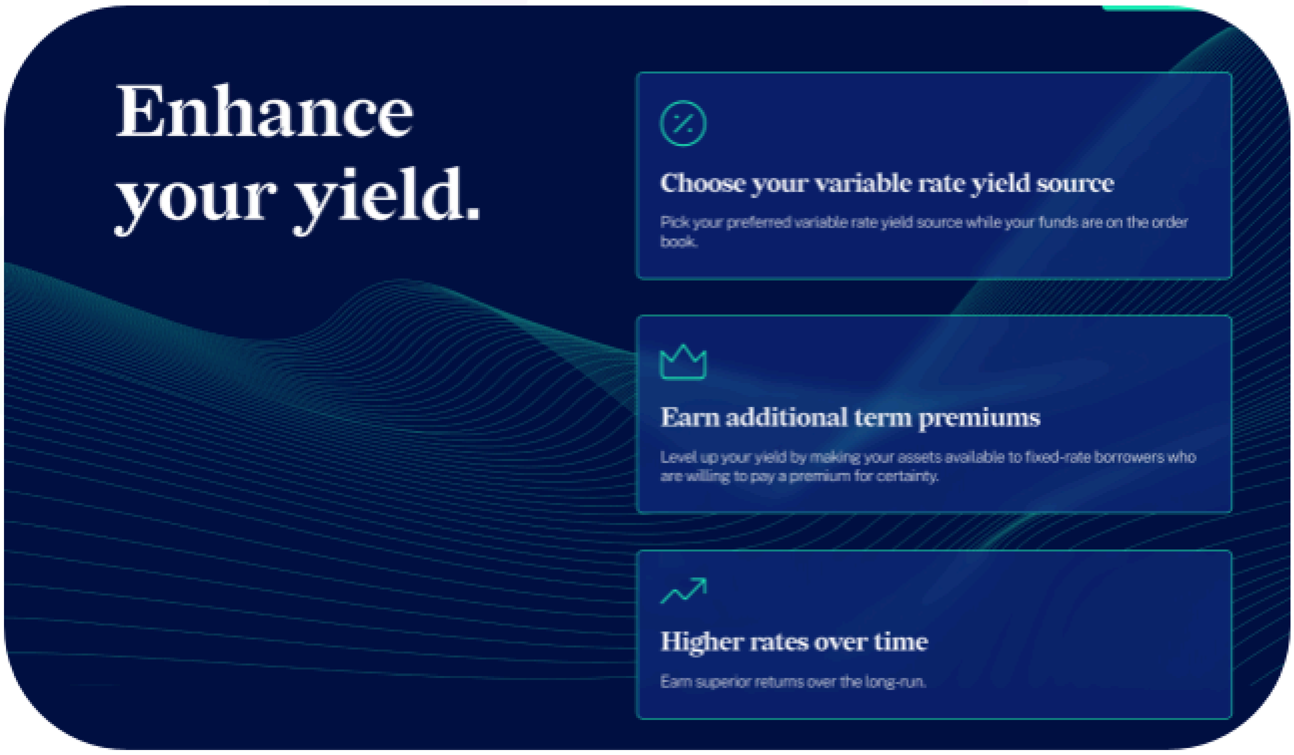
Project Name	Size Credit
Launch Date	TBA
Compiler Version	v0.8.23
Language	SOLIDITY
Network	ETHEREUM, BASE
Token Ticker	ERC20

Project Visuals:



Borrow and lend with certainty.

Size Credit is a novel fixed-rate lending marketplace that enhances rates for borrowers and lenders.



Enhance your yield.

Choose your variable rate yield source

Pick your preferred variable rate yield source while your funds are on the order book.

Earn additional term premiums

Level up your yield by making your assets available to fixed-rate borrowers who are willing to pay a premium for certainty.

Higher rates over time

Earn superior returns over the long-run.

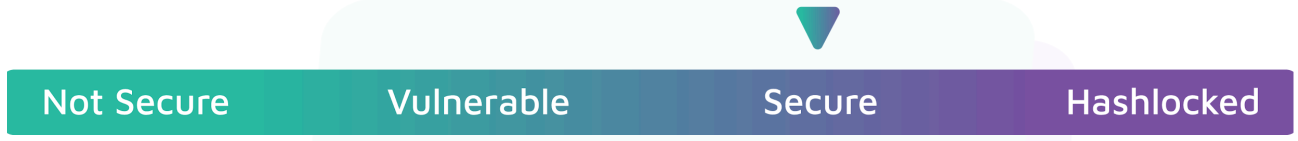
Audit Scope

We at Hashlock audited the solidity code within the Size Credit 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	**Size Credit Smart Contracts**
Platform	**EVM / Solidity**
Audit Date	**June, 2025**
Contract 1	NonTransferrableRebasingTokenVault.sol
Contract 1 MD5 Hash	4ceef75fa62563a718ad38dcee4d41f4
Contract 2	AaveAdapter.sol
Contract 2 MD5 Hash	f7c35523cb8ee906af473a40ad26ee1d
Contract 3	ERC4626Adapter.sol
Contract 3 MD5 Hash	823cff29412617a08f75ab28dd4d7e0c
Audited GitHub Commit Hash	6911b4564c01a77cb30657b6593af3fcf3640778
Fix Review GitHub Commit Hash	02c4ae4c1317070c17aa3bf3538b463a2feba0e2

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 and acknowledged.

Hashlock found:

2 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
NonTransferrableRebasingTokenVault.sol - Manage non-transferable rebasing ERC-20 tokens representing deposits in yield-generating vaults - Deposit underlying tokens into whitelisted Aave pools and ERC4626 vaults on behalf of users - Allow users to switch between different vault strategies while maintaining their positions - Handle cross-vault transfers and emergency exits from compromised vaults - Provide adapter-based architecture for extensible vault type support	**Contract achieves this functionality.**
AaveAdapter.sol - Interface between NonTransferrableRebasingTokenVault and Aave lending protocol - Handle deposits and withdrawals of underlying tokens to/from Aave lending pools - Manage scaled token accounting and convert between Aave's scaled balances and actual token amounts - Validate vault configurations and check liquidity availability before operations - Provide real-time balance calculations using Aave's liquidity index for accurate rebasing of token values	**Contract achieves this functionality.**
ERC4626Adapter.sol - Interface between NonTransferrableRebasingTokenVault and ERC4626	**Contract achieves this functionality.**

standard vaults - Handle deposits and withdrawals of underlying tokens to/from ERC4626 compliant yield vaults - Convert between ERC4626 vault shares and underlying asset amounts using standard conversion methods - Execute internal transfers between users within the same ERC4626 vault without token movement - Validate vault compatibility by checking the underlying asset matches the expected token - Monitor vault liquidity and withdrawal limits before executing operations	

Code Quality

This audit scope involves the smart contracts of the Size Credit 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 Size Credit 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] ERC4626Adapter#withdraw - Unlimited share token allowance window in withdraw flow

Description

`ERC4626Adapter.withdraw` grants itself a `uint256.max` allowance on the vault's `ERC4626` share token immediately before invoking the vault's `withdraw` method, then resets it back to zero afterward.

Vulnerability Details

```
function withdraw(address vault, address from, address to, uint256 amount)
        external
        onlyOwner
        returns (uint256 assets)
    {
        uint256 userBalanceBefore = balanceOf(vault, from);
        if (userBalanceBefore < amount) {
                   revert  IERC20Errors.ERC20InsufficientBalance(from,  userBalanceBefore,
amount);
        }
        uint256 sharesBefore = IERC4626(vault).balanceOf(address(tokenVault));
        uint256 assetsBefore = underlyingToken.balanceOf(address(this));
        uint256 userSharesBefore = tokenVault.sharesOf(from);
 >>       tokenVault.requestApprove(vault, type(uint256).max); //@audit if the vault goes
malicious, they consume the approval
        // slither-disable-next-line unused-return
        IERC4626(vault).withdraw(amount, address(this), address(tokenVault));
        tokenVault.requestApprove(vault, 0);
        uint256 shares = sharesBefore - IERC4626(vault).balanceOf(address(tokenVault));
        assets = underlyingToken.balanceOf(address(this)) - assetsBefore;
        if (userSharesBefore < shares) {
```

```
                    revert  IERC20Errors.ERC20InsufficientBalance(from,  userBalanceBefore,
amount);
        }
        underlyingToken.safeTransfer(to, assets);
        tokenVault.setSharesOf(from, userSharesBefore - shares);
    }
```

When the market initiates a withdrawal, the adapter first has the vault grant it an unlimited allowance on the vault's `ERC4626` share token. Then the adapter calls the vault's `withdraw` method to burn the requested shares by pulling them from the vault using that allowance.

While this `withdraw` call is underway, a malicious or compromised `ERC4626` vault implementation could exploit the temporary unlimited allowance to transfer all of the vault's share tokens to an attacker. Finally, once the `withdraw` call returns, the adapter revokes its allowance by setting it back to zero, but by then, the shares may already have been stolen. The same issue exists in `ERC4626Adapter#fullWithdraw`.

Additionally, because the adapter never checks how many shares will be burned (e.g., via a `previewWithdraw` or `convertToShares` call), a vault whose fees spike can end up burning more shares than the user expected and returning fewer underlying tokens. When you combine that slippage risk with the unlimited allowance window, it amplifies both the user's loss and the vault's ability to steal everything.

Impact

A whitelisted vault with malicious code (or one that is later upgraded to malicious logic or a fee-spiked one) can exploit the temporary allowance window to steal the entire share balance of the vault contract in a single transaction, resulting in total loss of user funds held there.

Recommendation

Approve exactly the number of shares needed rather than `uint256.max`, then reset to zero.

Status

Resolved

[M-02] ERC4626Adapter#fullWithdraw - Share redemption can block withdrawals

Description

The adapter's `fullWithdraw` function tells the vault to burn every share a user holds in one go without first checking how many shares the vault will permit. `ERC4626` vaults can enforce a redemption cap via maxRedeem or apply exit fees. If the adapter attempts to redeem more than the vault allows, the redeem call will revert, and the user's funds will remain locked until they manually retry in smaller batches.

Vulnerability Details

When the market calls `fullWithdraw` on the `NonTransferrableRebasingTokenVault`, the vault identifies its `ERC4626Adapter` and invokes its `fullWithdraw` method. The adapter then asks the vault to grant it an unlimited share allowance, calls redeem to burn the user's entire share balance, and revokes its allowance once redeem returns.

Redeem will revert if the requested shares exceed `maxRedeem` or violate exit fee or even slippage rules, preventing any shares from being cleared.

```
    function fullWithdraw(address vault, address from, address to) external onlyOwner
returns (uint256 assets) {
        uint256 assetsBefore = underlyingToken.balanceOf(address(this));
        uint256 userSharesBefore = tokenVault.sharesOf(from);
        tokenVault.requestApprove(vault, type(uint256).max);
        // slither-disable-next-line unused-return
            IERC4626(vault).redeem(userSharesBefore, address(this), address(tokenVault));
//@audit no maxRedeem check here
        tokenVault.requestApprove(vault, 0);
        assets = underlyingToken.balanceOf(address(this)) - assetsBefore;
        underlyingToken.safeTransfer(to, assets);
        tokenVault.setSharesOf(from, 0);
    }
```

Impact

Users whose share balance exceeds the vault’s `maxRedeem` limit may be unable to withdraw at all, resulting in a temporary lock of their funds.

Recommendation

Before redeeming, the adapter should query `IERC4626(vault).maxRedeem(address(tokenVault))` to discover the per-call share limit. It should then redeem up to that limit, verify the tokens received, and repeat until all shares are processed. Only once every share has been successfully redeemed should the adapter clear the user’s share balance.

Status

Acknowledged

Low

[L-01] NonTransferrableRebasingTokenVault#setVault - `forfeitOldShares` flag burns all shares before any migration attempt

Description

The `` `setVault` `` function allows a market to switch a user's vault, and if `` `forfeitOldShares` `` is true, it immediately zeros out the user's share balance. This happens without ever calling the adapter's `` `fullWithdraw` `` or `` `deposit` `` methods to move the underlying assets.

A user who unknowingly sets `forfeitOldShares` loses their entire share balance and can no longer recover their underlying assets, even though a valid migration path would have succeeded.

Recommendation

Change `setVault` so that in a `try/catch` block, it first invokes `adapterOld.fullWithdraw` to pull assets from the old vault and then `adapterNew.deposit` to credit the new vault address. Only if that migration call reverts and `forfeitOldShares` is true should the code clear the user's share balance. This ensures that forfeiting shares is only a deliberate fallback after a failed migration and never an unexpected loss.

Status

Acknowledged

[L-02] ERC4626Adapter#validate - Misconfiguration Risk Due to Discrepancy in underlyingToken Validation Between ERC4626Adapter and NonTransferrableRebasingTokenVault

Description

The `ERC4626Adapter` contract's (also AaveAdapter) constructor accepts an `_underlyingToken` immutable address, which is stored in its state. The `NonTransferrableRebasingTokenVault` (NTRTV) contract, which manages all operations, also has its own `underlyingToken` defined.

When an administrator whitelists a new ERC-4626 vault using `setVaultAdapter`, the NTRTV calls the `validate` function on the specified adapter. The current implementation of `ERC4626Adapter.validate` only checks if the vault's asset matches the *adapter's* underlyingToken, not the *NTRTV's* underlyingToken.

This discrepancy makes it possible for an administrator to deploy an `ERC4626Adapter` configured with `TokenA` and link it to an `NTRTV` that is designed to manage `TokenB`. The `validate()` check will pass if the vault's asset is also `TokenA`, successfully whitelisting an incompatible vault.

Impact

This enables a privileged administrator to misconfigure the system, causing all deposit and withdrawal operations for the whitelisted vault to revert, leading to a persistent Denial of Service for that feature.

Recommendation

To ensure system integrity and prevent misconfiguration, the adapter's `underlyingToken` should be sourced directly from the `NonTransferrableRebasingTokenVault` it serves. This creates an immutable and guaranteed link.

```
// In ERC4626Adapter.sol

contract ERC4626Adapter is Ownable, IAdapter {
    // ...
    NonTransferrableRebasingTokenVault public immutable tokenVault;
    IERC20Metadata public immutable underlyingToken; // Remains immutable
    // ...


            constructor(NonTransferrableRebasingTokenVault   _tokenVault)   //   Remove
_underlyingToken from parameters
        Ownable(address(_tokenVault))
    {
        if (address(_tokenVault) == address(0)) {
            revert Errors.NULL_ADDRESS();
        }
        tokenVault = _tokenVault;
        // Set the underlyingToken from the authoritative source: the NTRTV.
        underlyingToken = _tokenVault.underlyingToken();
    }


    // ...
}
```

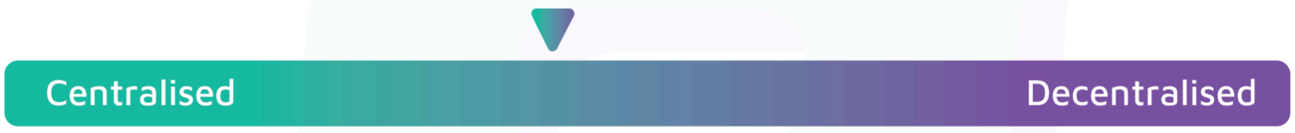
Status

Resolved

Centralisation

The Size Credit 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 Size Credit 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