hashlock.

Security Audit

MortgageFi (DeFi)

August 2024

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 Behaviours 9

Code Quality 10

Audit Resources 10

Dependencies 10

Severity Definitions 11

Audit Findings 12

Centralisation 22

Conclusion 23

Our Methodology 24

Disclaimers 26

About Hashlock 27

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT 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 THE USE OF THE CLIENT.

Executive Summary

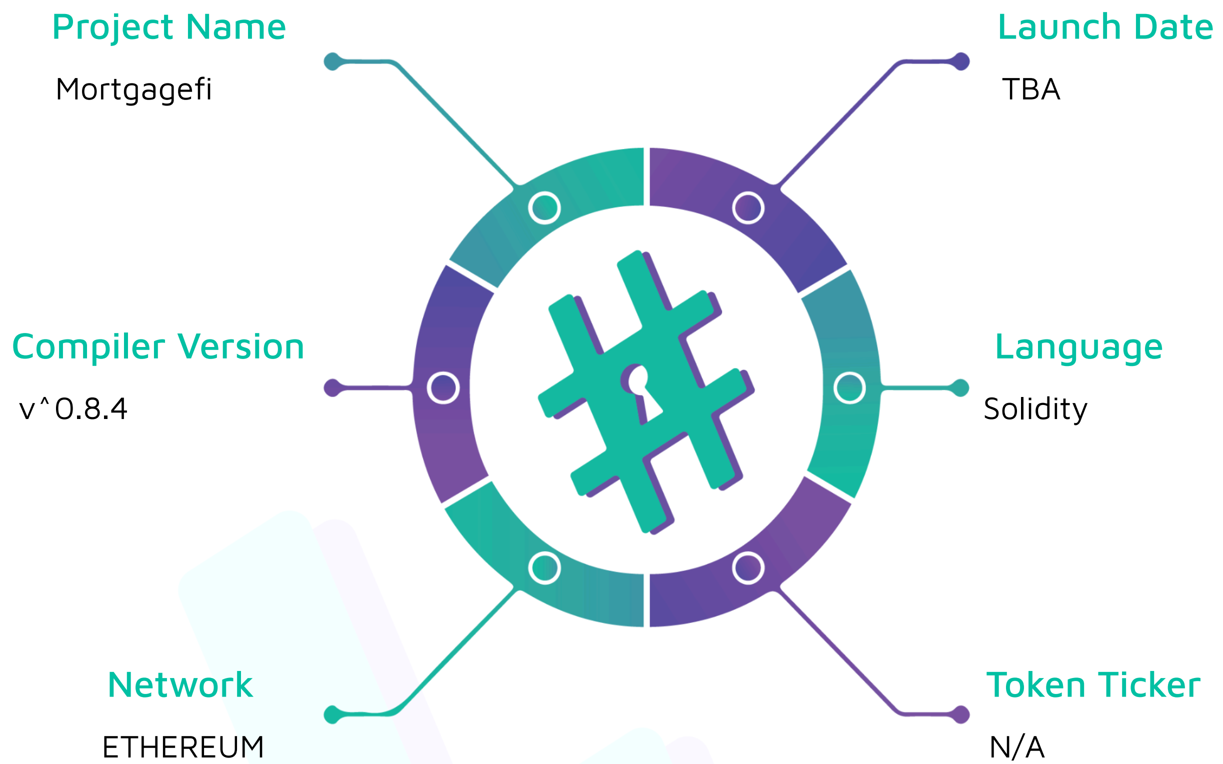
The Mortgagefi team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

Mortgagefi is a DeFi dApp built on Ethereum Network that allows you to take long-term collateralized mortgages.

Project Name: Mortgagefi
Compiler Version: ^0.8.4
Website: https://teamcat.fi/
Logo:

Visualised Context:



Project Visuals:

Audit scope

We at Hashlock audited the solidity code within the Mortgagefi 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	**Mortgagefi Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**August, 2024**
Contract 1	mortgagefipoolwethusdc.sol
Contract 1 MD5 Hash	ad38892cb87d96815657f0a6adb42efd
Contract 2	mortgagecontracts.sol
Contract 2 MD5 Hash	f4fb06f77c2291196a134db5e32a7c0b
Contract 3	mortgagefeetickets.sol
Contract 3 MD5 Hash	202b6019673b86d11eafc20d67d031f6
Contract 4	mortgageconversionvault.sol
Contract 4 MD5 Hash	e70175959bef5439da6f544c6d290f4b

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.

Not Secure	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.

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

Hashlock found:

4 High-severity vulnerabilities

2 Medium-severity vulnerabilities

5 Low-severity vulnerabilities

0 Gas Optimisations

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 Behaviours

Claimed Behaviour	Actual Behaviour
mortgagefipoolwethusdc.sol - Main contract and the entry point of the system - Users can deposit stable coins - Users can create a new mortgage and repay existing one	**Contract achieves this functionality.**
mortgagecontracts.sol - An NFT contract used as a receipt of an existing mortgage. It should be kept by the holder to be able to properly interact with the existing mortgage.	**Contract achieves this functionality.**
mortgagefeetickets.sol - An NFT contract used as a receipt of a fee ticket that can be redeemed for fees for a limited amount of time. Minted to protocol users.	
mortgageconversionvault.sol - An ERC4626-style contract where funds are deposited from the pool contract.	

Code Quality

This audit scope involves the smart contracts for the Mortgagefi project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices 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 Mortgagefi projects 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 help us understand the overall architecture of the protocol.

Dependencies

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

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

Audit Findings

High

[H-01] Mortgagefeetickets.sol - It is possible to use tickets of other users

Description

In mortgagefeetickets.sol, in function `use` there is no check if the caller is the NFT owner. This means that users can use any NFT ID and specify it to the function to take advantage of other users' bonus.

Additionally, the owner check is not present in any function that calls `use` ().

Vulnerability Details

Below code shows the `use`() function that does not have an NFT owner check.

```
    function use(uint256 _nftID, uint256 _size) public returns (uint256){
          require(hasRole(MINTER_ROLE, _msgSender()), "ERC721PresetMinterPauserAutoId:
must have minter role to mint");
        require(block.timestamp >= experation[_nftID], "Ticket not yet valid");
        if(_size > size[_nftID]){
            _size = size[_nftID];
        }
        size[_nftID] -= _size;
        return _size;
    }
```

Impact

Users may lose their discount because other users may use their tickets.

Recommendation

Implement a check w.g. require(ownerOf[_nftID] == original_sender, “Not ticket owner!”); where original_sender is passed from the calling contract (since the function is not called directly)

Status

Resolved

Medium

[M-01] Mortgagefipoolwethusdc - Usage of mint instead of safeMint

Description

In order to mint a NFT to the user, function mint is used. The problem with this function is that it does not check if the recipient supports NFT transfers. This way, the NFT may be not delivered via revert or stuck there forever.

Vulnerability Details

The `createContract` function contains following code in line 193:

```
uint256 _nftID = INFT(contracts).mint(msg.sender);
```

Impact

NFT might be lost if the caller is e.g. using a smart contract.

Recommendation

Use safeMint from openZeppelin. On the other hand, safeMint performs a callback on target contract therefore might be subject to a reentrancy attack, so it is important to use it at the end of the function (CEI pattern) or use nonReentrant modifier if unsure.

Status

Acknowledged

[M-02] Multiple contracts - Usage of transferFrom is insecure

Description

Despite importing safeTransferLib, multiple contracts use transfer/transferFrom function.

This has severe security implications since transferFrom does not check for return value.

Some ERC20 tokens do not revert on failed transfer, but instead return false. If such a token is deposited to the protocol, it may be accounted for but not transferred.

Vulnerability Details

The transfer/transferFrom is commonly used throughout the whole project to transfer tokens.

For example, BAT does not revert on failure but returns false, and USDT does not return any value.

Impact

If tokens that are not reverting on failure are used, then the protocol might suffer losses due to accounting amounts that in fact are not in the protocol.

Recommendation

Use safeTransferLib's safeTransfer/safeTransferFrom to move assets.

Status

Resolved

Low

[L-01] Mortgagefipoolwethusdc - Mortgage NFTs being burnable increases the risk of user insolvency due to a mistake

Description

The mortgage is symbolised by an NFT that is minted upon contract creation. The issue is that it is burnable. While it is rather unlikely that a user will burn it, this option should be restricted, since burning an NFT will cause the user to lose the collateral and be unable to repay the debt.

Recommendation

Override burn function with one that contains just revert.

Status

Acknowledged

[L-02] Multiple contracts - Improper usage of AccessControl

Description

The project uses `AccessControl` library from OpenZeppelin but each time it is used in all contracts except mortgageconversionvault.sol, all the roles are initialised to the msg.sender. This is against the idea of AccessControl which allows separate roles ensuring better decentralisation and removing a single point of failure.

Recommendation

Either consider assigning multiple addresses to separate roles, or use a simple Ownable pattern and use variables for caller contract callers, e.g.

address public contract1 = 0xaddr;

modifier onlyContract1() { require msg.sender == contract1;}

Status

Acknowledged

[L-03] All contracts - Redundant code / libraries

Description

The project uses multiple libraries which are not used. This unnecessarily increases the complexity of the codebase and increases gas consumption due to its large size.

For example:

- mortgageconversionvault.sol contract are Multicall and ciabv2erc20. The latter is not attached to the repository, additionally, the contract does not seem to take advantage of multicall features.
- Safetransferlib is attached but never used.
- Mortgagefeetickets.sol is ERC721 Pausable, Burnable, Accesscontrolenumerable but these libraries are not really used. Moreover, there is no whenNotPaused modifier implemented that takes advantage of the pause. So the pause is redundant at this point
- Other contracts are unnecessarily multicall

Recommendation

Remove unused libraries/imports, and leave only those that are used by the protocol.

Status

Acknowledged

[L-04] Mortgagefeetickets.sol - Incorrect error message

Description

In mortgagefeetickets.sol line 101, the error message says “Ticket not yet valid”. However since it checks for expiration, it should be rather “Ticket expired”.

Recommendation

Change error message to “ticket expired”.

Status

Acknowledged

[L-05] Global issue - Missing events

Description

The project does not use any events aside of Deposit/Withdraw in mortgageconversionvault.sol and _debug in mortgagefipoolwethusdc.sol.

Events play a crucial role in blockchain ensuring transparency of key state changes.

Recommendation

Add events and emit them for key state changes: creation of mortgages, repayments, transfers, usage of tickets and also change of key state variables like parameters, roles, etc.

Status

Acknowledged

[L-06] Mortgageconversionvault.sol - Minting shares 1:1 to deposit amount can be used to arbitrage against the protocol

Description

The conversion vault is a ERC4626-style contract with some changes. One of the changes is that on deposit, the user always gets 1:1 shares to assets. On the other hand, when withdrawing, the standard vault formula is used which is

withdrawAmount = (shares*sharesSupply)/totalAssetsInVault

The problem with this approach is that a vault when generating yield, increases the denominator (totalAssets) which causes a share price increase over time.

In short words - if initially there are no extra assets, just user deposits, let's assume then there are 1000 shares for 1000 tokens deposited.

But if a strategy generates yield, or another contract transfers funds to the vault, e.g. 1000 tokens are sent extra, now there are 2000 tokens in the vault for the same 1000 shares, then each share is worth 2 tokens instead of initially one.

In such a scenario, if in that moment a user starts minting shares in a ratio of 1:1 to deposited tokens, he can at the same time redeem the shares for a larger amount of tokens, emptying the vault from yield.

Vulnerability Details

Function mint is shown below to mint 1:1 shares to assets. Further, a withdrawal is shown to calculate the withdrawn amount based on total assets and not on the initially provided rate.

```
    function mint(uint256 shares, address receiver) public  returns (uint256 assets) {

        doUpdate();

        // Need to transfer before minting or ERC777s could reenter.

                  entryCoin.transferFrom(msg.sender,  address(this),  shares);//@audit
transferFrom

        assets = shares;

        _mint(receiver, shares);

        emit Deposit(msg.sender, receiver, assets, shares);

        afterDeposit(assets, shares, receiver);

    }


   [...]

    function withdraw(

        uint256 assets,
```

```
        address receiver,
        address owner,
        uint256 _nftID
    ) public returns (uint256 shares) {
        shares = previewWithdraw(assets);
[...]
    function previewWithdraw(uint256 assets) public view returns (uint256) {
            uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
non-zero.
        return supply == 0 ? assets : assets.mulDivUp(supply, totalAssets());
    }
```

Impact

Users can arbitrage against the protocol's vault.

Recommendation

Use original vault formulas from ERC4626 from OpenZeppelin, version post-4.9 which is patched against thai kind of behavior

Status

Acknowledged

Centralisation

The Mortgagefi 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.

Centralised | Decentralised

Conclusion

After Hashlocks analysis, the Mortgagefi 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 behavior 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 still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to 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 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.