Security Audit

Levva (DeFi)

October 2024 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 Levva team partnered with Hashlock to conduct a security audit of their Vault.sol and Adapter 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

Levva is a decentralized finance (DeFi) platform leveraging Web3 technologies to offer trustless financial services like lending, borrowing, staking, and yield farming. By utilizing smart contracts on a decentralized network, it ensures secure, permissionless transactions and user control through non-custodial Web3 wallets like MetaMask. The platform likely integrates with other DeFi protocols, enhancing its composability and optimizing yields for users. Governance tokens may empower the community to shape the platform's future, while strong security measures, such as audits, help mitigate risks. Levva.fi embodies decentralization, promoting financial autonomy and community-driven growth.

Project Name: Levva
Compiler Version: 0.8.26
Website: https://levva.fi/
Logo:

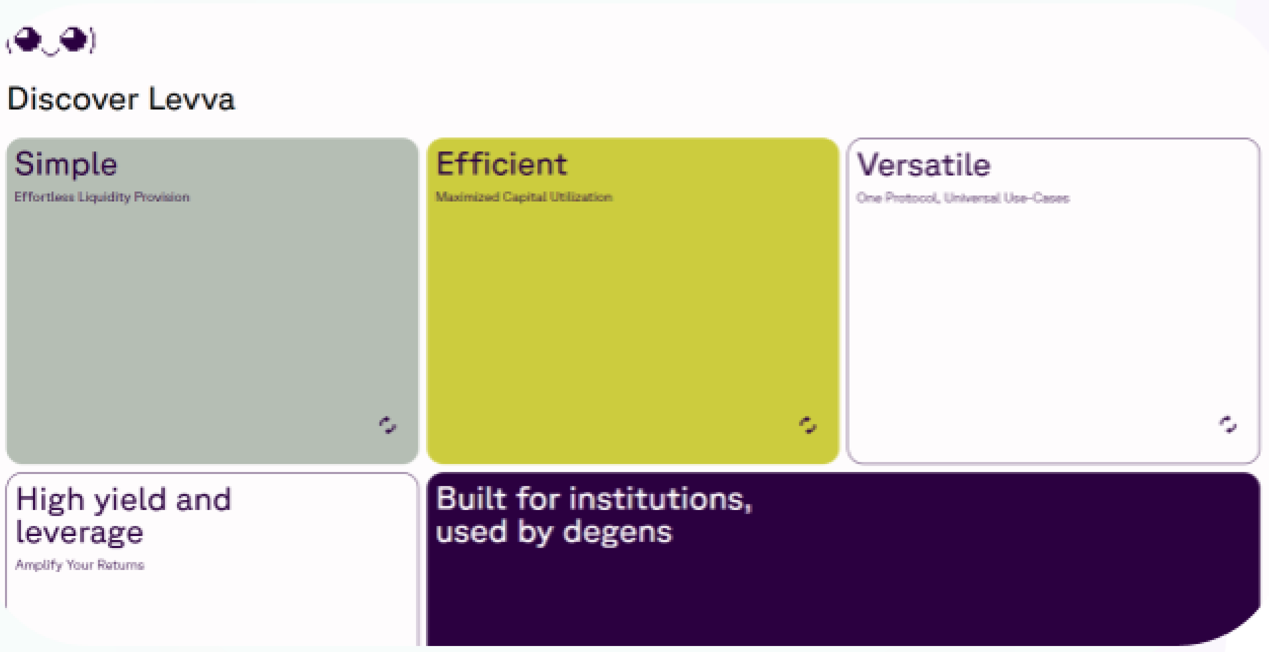
Visualised Context:

Project Name	Levva
Launch Date	TBA
Compiler Version	0.8.26
Language	Solidity
Network	ETHEREUM, Arbitrum One
Token Ticke	N/A

Project Visuals:

The universal leverage layer

Levva is the easiest way to access leverage and yield in DeFi. Our permissionless protocol unites liquidity providers and borrowers, ensuring efficient liquidity use. Maximize your gains and minimize fragmentation.



Discover Levva

Simple
Effortless Liquidity Provision

Efficient
Maximized Capital Utilization

Versatile
One Protocol, Universal Use-Cases

High yield and leverage
Amplify Your Returns

Built for institutions, used by degens

Audit scope

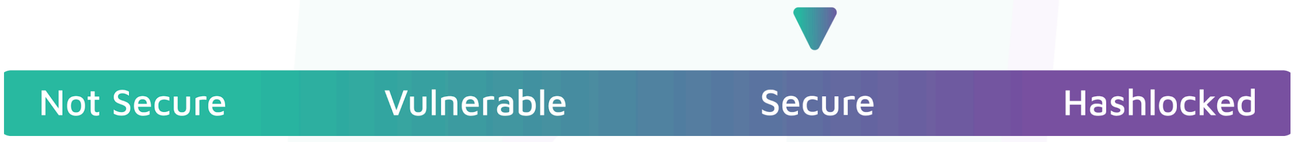
We at Hashlock audited the solidity code within the Levva project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Levva Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**October, 2024**
Contract 1	AbstractVault.sol
Contract MD5 Hash	4fcfa0b9d5025d80a4c1bffdc3a063e5
Contract 2	AccessControl.sol
Contract MD5 Hash	fc803985802da7b5e26a34d94dfb4788
Contract 3	LendingAdaptersStorage.sol
Contract MD5 Hash	d00ac65c72b9a9556656a359f3fe7520
Contract 4	WithdrawRequestQueue.sol
Contract MD5 Hash	ce71e28c09a5a65158373e6ea67f9a8f
Contract 5	ILendingAdapter.sol
Contract MD5 Hash	6daacc5d5cb1332ad35e30bdf481a90a
Contract 6	IVault.sol
Contract MD5 Hash	1d741cc58aa0dda0153dbd409c77ebe3
Contract 7	Errors.sol
Contract MD5 Hash	316e02b2f22fcbec8bc768cce9fabb06
Contract 8	ProtocolType.sol

Contract MD5 Hash	1c0697cf64c1145a7fe9c0568e0f0149
Contract 9	AaveAdapter.sol
Contract MD5 Hash	68b9add11320f29d4801bb2bf146a1bf
Contract 10	AaveAdapterConfigStorage.sol
Contract MD5 Hash	efa1d7f2153050a2cbe3df46a9a8bacc
Contract 11	EtherfiAdapter.sol
Contract MD5 Hash	3eb1bb2134d3c1b0e1610b148927cde2
Contract 12	EtherfiAdapterConfigStorage.sol
Contract MD5 Hash	4ec3c152bd2380b065c5c14c8f68c896
Contract 13	MarginlyAdapter.sol
Contract MD5 Hash	41c439764ffd95df1e8466bcd67c0279
Contract 14	MarginlyAdapterConfigStorage.sol
Contract MD5 Hash	ad8122e3987351d690f61266118a2cc6
Contract 15	ConfigManager.sol
Contract MD5 Hash	f996932ad749c6406977c9aa1bfbfb91
Contract 16	VaultAccessible.sol
Contract MD5 Hash	55d5862fff707d59d43de23634ef3833
Contract 17	Vault.sol
Contract MD5 Hash	83ef8e14814c8a4acb6430253d6f178b
GitHub Commit Hash	6e83dc5ca70618c085522f0511c22359ea51183c

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 Upgradeable Open Zeppelin contracts and Interacts with external lending pools. We initially identified some significant vulnerabilities that have since been addressed.



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 High severity vulnerabilities

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
Vault.sol - Allows users to: - Deposit tokens to the vault - Withdraw token from the vault - Queue withdraw when vault do not have sufficient token - Allows protocol admins to: - finalize Withdraw Request - Supply tokens to lending pools through delegate call - Manage config and adaptors - Add vault manager	**Contract achieves this functionality.**
Protocols/*.sol - AaveAdapter contract used to: - Supply and withdraw to the aave pool - EtherfiAdapter contract used to: - Stake and unstake - Claim withdraw - MarginlyAdapter contract used to: - Deposit and withdraw to the marginly pool	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Levva 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 Levva 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

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] AbstractVault#deposit - A donation attack on the vault can cause depositors to lose funds

Description

The `AbstractVault` contract has a vulnerability where an attacker can donate assets directly to the vault, increasing `_getFreeAmount()` while `totalSupply()` remains zero. When a user tries to deposit assets afterward, the `_convertToShares` function calculates the number of shares to mint based on these values. Due to the imbalance (assets with zero shares), the calculation results in zero shares being minted for the depositor, causing them to lose their assets without receiving any shares in return.

Vulnerability Details

The `AbstractVault` contract is vulnerable to a donation attack that can cause the first depositor to lose their funds. The root of the vulnerability lies in the `_convertToShares` and `_getFreeAmount()` function, which is used to calculate the number of shares to mint when a user deposits assets. The function computes shares based on the current total supply of shares and the total assets managed by the vault. `_getFreeAmount()` function uses `balanceOf` to track balance but this can be manipulated by sending assets directly

```
function _convertToShares(uint256 assets, Math.Rounding rounding) internal view override
returns (uint256 shares) {

       shares = assets.mulDiv(

       totalSupply() + 10 ** _decimalsOffset(),

       _getTotalLent() + _getFreeAmount() + 1,rounding

       );
}
```

The issue arises when the vault has a non-zero balance of assets (due to direct transfers or "donations") but a total share supply of zero. In this scenario, the calculation within `_convertToShares` results in zero shares being minted for the depositor, even though they have provided assets to the vault.

Proof of Concept

An example that simulates precision loss is shown below.

```
Calculation in _convertToShares:

- assets: 100 (user deposit)

- totalSupply(): 0

- _getTotalLent() + _getFreeAmount() + 1: 1000 (donated assets) + 0 (no lent assets) + 1
= 1001

- shares calculation:

solidity

shares = 100 * (0 + 10 ** decimals) / 1001;



- Since totalSupply() is zero, shares rounds down to zero.
```

Impact

Depositors receive zero shares for their deposits, losing their assets without any compensation.

Recommendation

The team should deposit themselves as soon as the new vault is deployed and revert when depositors receive 0 shares.

Status

Resolved

Medium

[M-01] AbstractVault#mint - Missing max asset slippage protection in `mint()` function

Description

The `AbstractVault` contract's `mint` function lacks slippage protection, which can be exploited by an attacker to manipulate the amount of assets that depositors are required to provide when minting shares. Specifically, an attacker can increase the vault's asset balance by transferring assets directly to the vault without increasing the total supply of shares. This manipulation affects the calculations within the `_convertToAssets` function, leading depositors to deposit more assets than intended for the same number of shares by frontrunning.

Vulnerability Details

In the `AbstractVault` contract, the `mint` function allows users to specify the number of shares they want to mint. It calculates the required amount of assets by calling `previewMint`, which internally uses `_convertToAssets` to determine how many assets are needed for the desired number of shares.

Here's the relevant code snippet:

```
function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view override
returns (uint256 assets) {

assets = shares.mulDiv(_getTotalLent() + _getFreeAmount() + 1,totalSupply() + 10 **
_decimalsOffset(),rounding);
}
```

- `_getTotalLent() + _getFreeAmount()`: Represents the total assets managed by the vault.

- `totalSupply()`: Represents the total number of shares issued.

The problem arises because `_getFreeAmount()` can be manipulated by an attacker sending assets directly to the vault. Since these assets increase the vault's balance without increasing `totalSupply()`, the ratio of assets to shares increases. When a user calls `mint`, the required assets for minting the desired shares become higher than expected.

Proof of Concept

1. Victim Minting Shares: A user attempts to mint a specific number of shares.

2. Calculation in `_convertToAssets`:

- Before Attack:

- `_getTotalLent() + _getFreeAmount()`: 10,000 assets

- `totalSupply()`: 1,000 shares

- Assets per share: 10 assets/share

- Required assets for 100 shares: 1,000 assets

- After Attack:

- `_getTotalLent() + _getFreeAmount()`: 15,000 assets (10,000 initial + 5,000 from attacker)

- `totalSupply()`: 1,000 shares (unchanged)

- Assets per share: 15 assets/share

- Required assets for 100 shares: 1,500 assets

3. Result: The user now has to provide 1,500 assets to mint the same 100 shares, overpaying by 500 assets due to the attacker's manipulation.

Impact

Depositors may end up depositing more assets than intended for the shares they receive.

Recommendation

Add slippage protection

Status

Resolved

[M-02] Vault#requestWithdraw -DOS in `requestWithdraw` function via manipulation with deposit and withdraw

Description

The `requestWithdraw` function in the `Vault` contract can be exploited to cause legitimate users' withdrawal requests to revert, resulting in a DoS. An attacker can manipulate the vault's asset balance (`_getFreeAmount()`) by strategically depositing assets before a user's `requestWithdraw` call (`frontrunning`). This increases `_getFreeAmount()`, causing the condition `assets <= _getFreeAmount()` to evaluate to `true` and the function to revert with `Errors.NoNeedToRequestWithdraw()`. The attacker can then withdraw their assets (`backrunning`), returning the vault to its original state. This manipulation prevents users from queuing their withdrawals, effectively locking their funds and disrupting the normal operation of the vault.

Vulnerability Details

- The function checks if the amount of assets corresponding to the shares to be withdrawn (`assets`) is less than or equal to the vault's free asset balance (`_getFreeAmount()`).

- If the condition `assets <= _getFreeAmount()` is `true`, the function reverts with `Errors.NoNeedToRequestWithdraw()`.

- The vault's free asset balance can be manipulated by an attacker within the same block to make the condition `assets <= _getFreeAmount()` always true.

```
function requestWithdraw(uint256 shares) external returns (uint128 requestId) {

      uint256 assets = previewRedeem(shares);
     // Attacker can manipulate _getFreeAmount() to cause this check to pass or fail

      if (assets <= _getFreeAmount()) {

      revert Errors.NoNeedToRequestWithdraw();

      }

       _transfer(msg.sender, address(this), shares);

      requestId = _enqueueWithdraw(msg.sender, shares);

      emit WithdrawRequested(requestId, msg.sender, shares);

      return requestId;

}
```

Impact

Legitimate users are prevented from queuing withdrawals.

Recommendation

I could not think of any quick fix without major refactoring

Status

Resolved

Low

[L-01] EtherfiAdapterConfigStorage#dequeueUnstakeRequest - State is not cleared

Description

`QueueItem` struct is not cleared when `dequeueUnstakeRequest` is called.

Recommendation

Clear State carefully.

Status

Resolved

[L-02] AaveAdapter - Pool change can cause temporary Dos

Description

The pool address from their `PoolAddressesProvider` should not be saved in the state var because it is subject to change.

It is recommended to query the `PoolAddressesProvider` every time the pool address is needed to ensure the address used is current.

Recommendation

Insert a function call to `PoolAddressesProvider.getPool()` within every function that interacts with the Aave pool: https://docs.aave.com/developers/core-contracts/pooladdressesprovider.

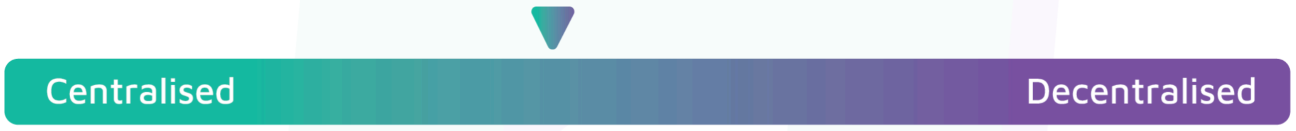
Status

Resolved

Centralisation

The Levva 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 Hashlocks analysis, the Levva 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

hashlock.

hashlock.

Hashlock Pty Ltd