#Hashlock.

Exactly Protocol

SMART CONTRACT

Security Audit

Performed on Contracts:

InterestRateModel.sol

MD5 Hash:d424a0758a877e95a9f21a3685c680e086dc2321

Market.sol

MD5 Hash:d424a0758a877e95a9f21a3685c680e086dc2321

Github Commit Hash:d424a0758a877e95a9f21a3685c680e086dc2321

Platform

OP

hashlock.com.au

MARCH 2024

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

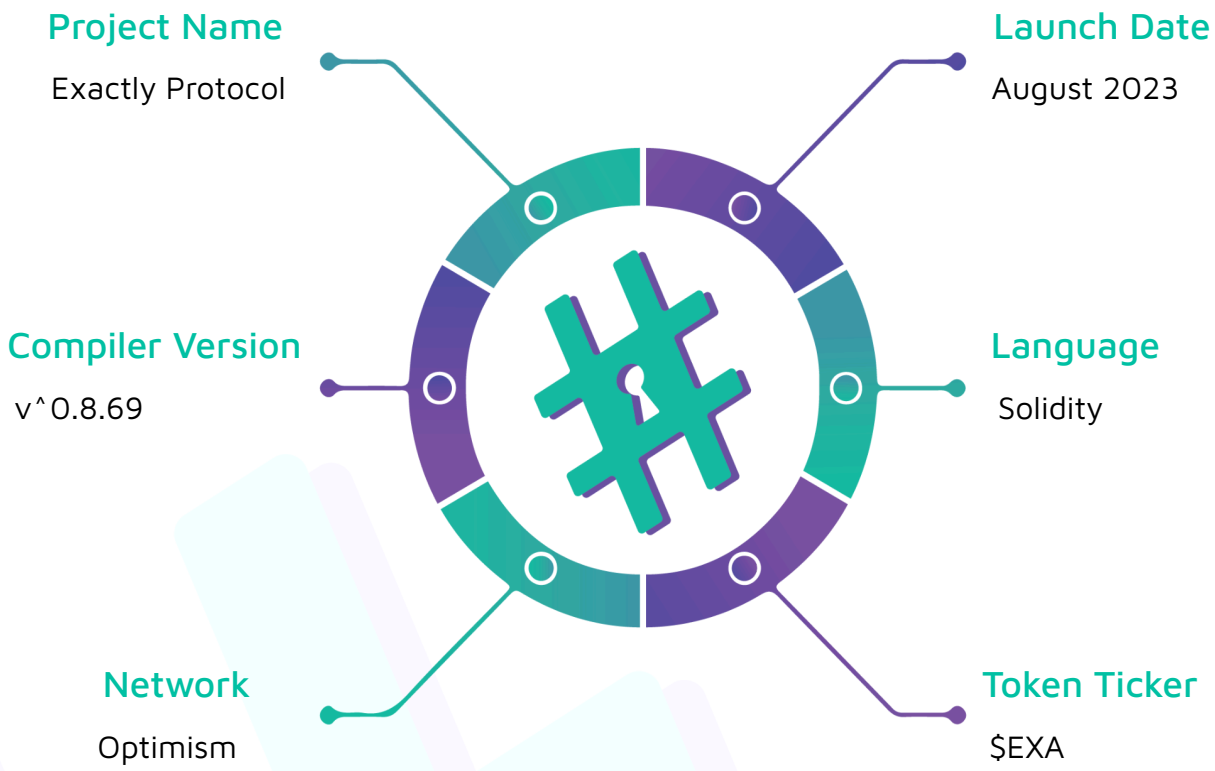
The Exactly team partnered with Hashlock to conduct a security audit of their InterestRate.sol and Market.sol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

Exactly is a decentralized and open-source DeFi protocol that allows users to easily exchange the value of their crypto assets through deposits and borrows with variable and fixed interest rates.

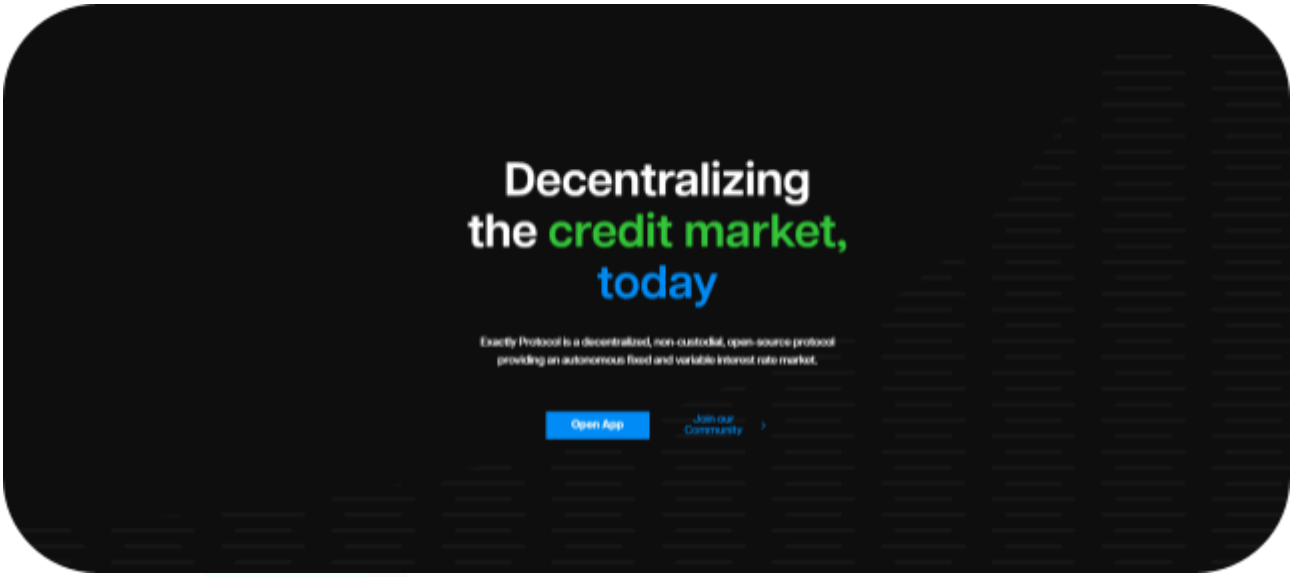
Project Name: Exactly Protocol
Compiler Version: ^0.8.42
Website: https://exact.ly/
Logo:

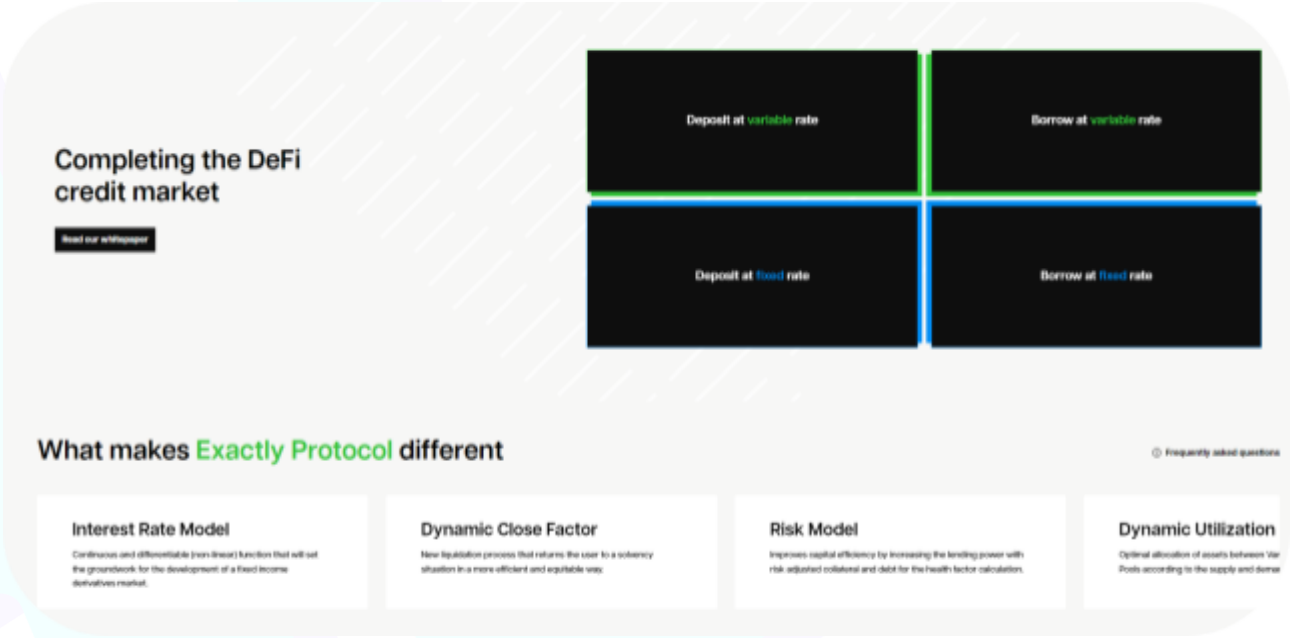
Visualised Context:



Iconography:

Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Exactly project, the scope of works 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	Template Protocol Smart Contracts
Platform	Ethereum / Solidity
Audit Date	August, 2023
Contract 1	InterestRateModel.sol
Contract 1 MD5 Hash	d424a0758a877e95a9f21a3685c680e086dc2321
Contract 2	Market.sol
Contract 2 MD5 Hash	d424a0758a877e95a9f21a3685c680e086dc2321
GitHub Commit Hash	d424a0758a877e95a9f21a3685c680e086dc2321

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 through Hashlocks partners.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. 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:

0 High severity vulnerabilities

0 Medium severity vulnerabilities

3 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.*

Standardised Checks

Main Category	Subcategory	Result
General Code Checks	Solidity/compiler version stated	Passed
	Consistent pragma version across each contract	Passed
	Outdated Solidity Version	Reviewed
	Overflow/underflow	Passed
	Correct checks, effects, interaction order	Reviewed
	Lack of check on input parameters	Reviewed
	Function input parameters check bypass	Passed
	Correct Access control	Reviewed
	Built in emergency features	Reviewed
	Correct event logs	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Reviewed
	Features claimed	Passed
	delegatecall() vulnerabilities	Passed
	Other programming issues	Reviewed
Code Specification	Correctly declared function visibility	Passed
	Correctly declared variable storage location	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Reviewed
Gas Optimization	"Out of Gas" Issue	Reviewed

	High consumption 'for/while' loop	Reviewed
	High consumption 'storage' storage	Reviewed
	Assert() misuse	Passed
Tokenomics Risk	The maximum limit for mintage not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Initial Audit Result: SECURE

Revised Audit Result: N/A

Intended Smart Contract Behaviour

Claimed Behaviour	Actual Behaviour
InterestRateModel.sol - Continuous and differentiable (non-linear) function that will set the groundwork for the development of a fixed income derivatives market. their collateral. calculate and return both fixed and variable rates. Contains parameters as state variables that are used to get the different points in the utilization curve for an asset.	**Contract achieves this functionality.**
Market.sol - Main contract of the protocol. It exposes all user-oriented actions for fixed and variable borrows, deposits, repayments, and withdrawals.	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the Exactly 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 is required.

The code is very well commented and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Exactly 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 are helpful in understanding the overall architecture of the protocol.

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

Gas

[G-01] InterestRateModel.sol#constructor - Use assembly to check for address(0)

Description

There are some checks for address(0) that can be optimised using assembly.

Recommendation

Add a new function for this type of checks

```
function assembly_notZero(address toCheck) public pure returns(bool success) {
    assembly {
        if iszero(toCheck) {
            let ptr := mload(0x40)
        mstore(ptr, 0xd92e233d00000000000000000000000000000000000000000000000000000000)
// selector for `ZeroAddress()`
            revert(ptr, 0x4)
        }
    }
    return true;
}
```

Status

Acknowledged. It's a style choice, as Exactly Protocol prefers to keep the check against address(0) for easier readability, considering that the gas-saving is not significant.

[G-02] Market.sol - Don't initialize variables with default value

Description

Gas can be saved by not initialising variables to its default value.

Recommendation

Remove the initialization.

```
623:     uint256 totalBadDebt;


928:       uint256 backupEarnings;
```

Status

Acknowledged. It's a style choice as Exactly Protocol prefers being explicit about these initial values. For more context, see here.

[G-03] InterestRateModel.sol and Market.sol - Use != 0 instead of > 0 for unsigned integer comparison

Description

Using != 0 instead of > 0 saves gas.

Recommendation

Change the below comparisons from > 0 to != 0 when the variables are uint256.

```
File: InterestRateModel.sol


50:       p.minRate > 0 &&


51:         p.naturalRate > 0 &&
```

```
53:         p.naturalUtilization > 0 &&

55:         p.growthSpeed > 0 &&

56:         p.sigmoidSpeed > 0 &&

57:         p.spreadFactor > 0 &&

58:         p.maturitySpeed > 0 &&

59:         p.maxRate > 0 &&
```

```
File: Market.sol

297:       if (backupDebtAddition > 0) {

339:                 if (newUnassignedEarnings > 0) pool.unassignedEarnings += newUnassignedEarnings;

587:     if (maxAssets > 0 && account.floatingBorrowShares > 0) {

587:     if (maxAssets > 0 && account.floatingBorrowShares > 0) {

589:       if (borrowShares > 0) {
```

```
646:                if   (account.floatingBorrowShares   >   0   &&   (accumulator   =
previewRepay(accumulator)) > 0) {


646:                if   (account.floatingBorrowShares   >   0   &&   (accumulator   =
previewRepay(accumulator)) > 0) {


650:     if (totalBadDebt > 0) {


816:     if (shares > 0) debt += previewRefund(shares);


833:     if (treasuryFeeRate > 0) {


843:     if (fee > 0) {
```

Status

Acknowledged. It's a style choice as Exactly Protocol prefers being more readable on the range checks.

Hacker's Notes

Prior to this last commit, Exactly Protocol has pushed two new commits after based on the analysis of the audit. One is a gas optimisation which can be found here, and the other is a natspec fix which can be found here. Both these optimisations came as a result of the audit, in collaboration between Hashlock and Exactly Protocol. It has been a pleasure working with Exactly Protocol and their very robust codebase, and Hashlock is happy to have contributed to further optimisations.

Centralisation

The Exactly 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 Exactly project seems to have a sound and well tested code base, however our findings need to be resolved in order to achieve security. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented 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 are 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 whitebox 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, 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 contracts 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 are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree 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 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.