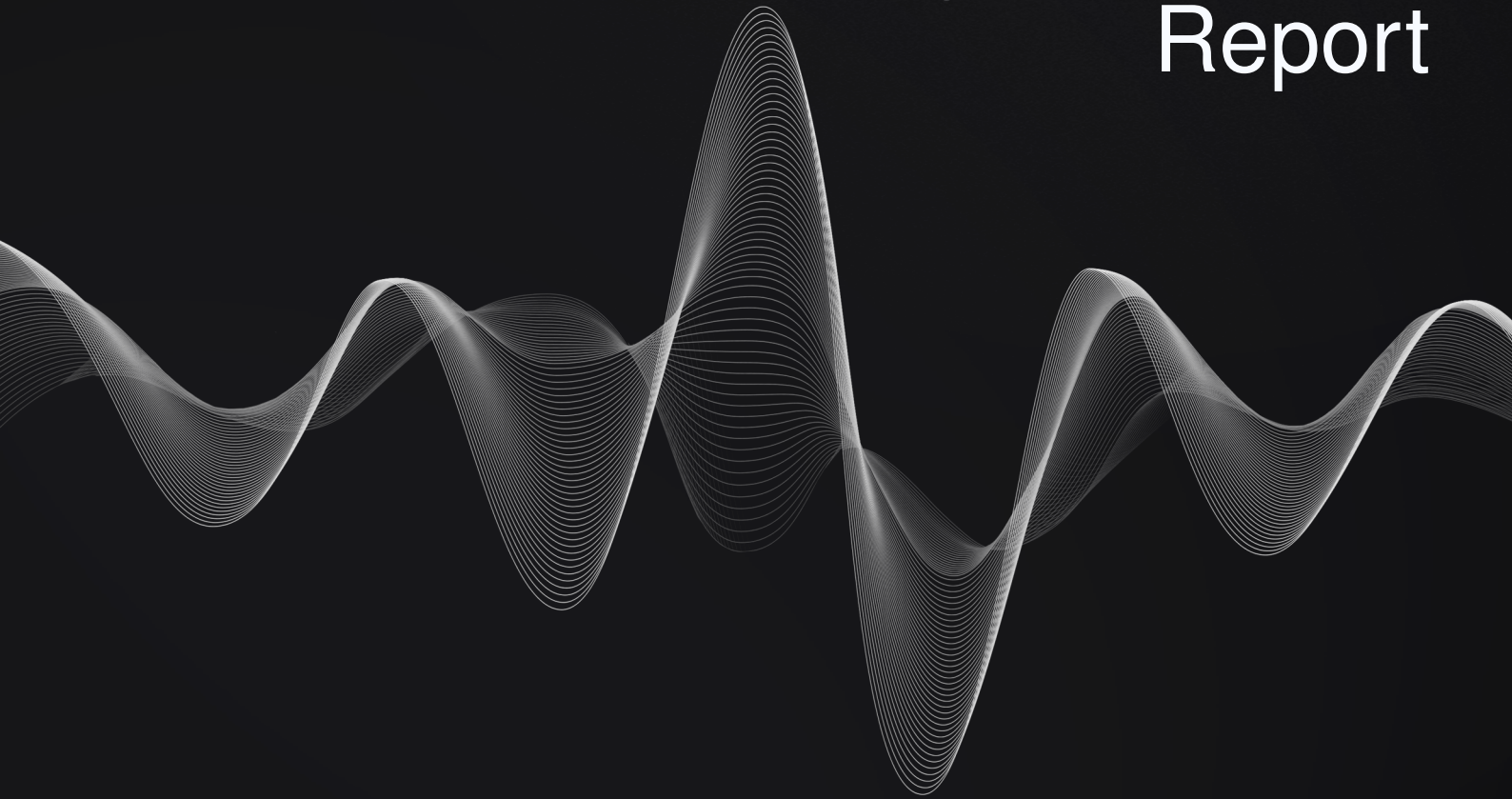


Primex Finance

Primex Protocol Flash Loan Integration Audit Report










Document Control

PUBLIC

FINAL(v2.1)

Audit_Report_PRMX-FLI_FINAL_21

Apr 19, 2024		v0.1	João Simões: Initial draft
Apr 25, 2024		v0.2	João Simões: Added findings
Apr 26, 2024		v1.0	Charles Dray: Approved
Jul 22, 2024		v1.1	João Simões: Reviewed findings
Oct 11, 2024		v1.2	João Simões: Reviewed contract updates
Oct 14, 2024		v2.0	Charles Dray: Finalized
Oct 14, 2024		v2.1	Charles Dray: Published

Points of Contact

Oleksandr
Marukhnenko
Charles Dray

Primex Finance
Resonance

alex@primex.finance
charles@resonance.security

Testing Team

João Simões
Ilan Abitbol
Michał Bazyli
Michał Bajor

Resonance
Resonance
Resonance
Resonance

joao@resonance.security
ilan@resonance.security
michal@resonance.security
michal.bajor@resonance.security

Copyright and Disclaimer

© 2024 Resonance Security, Inc. All rights reserved.

The information in this report is considered confidential and proprietary by Resonance and is licensed to the recipient solely under the terms of the project statement of work. Reproduction or distribution, in whole or in part, is strictly prohibited without the express written permission of Resonance.

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

Contents

1 Document Control	2
Copyright and Disclaimer	2
2 Executive Summary	4
System Overview	4
Repository Coverage and Quality.....	4
3 Target	6
4 Methodology	7
Severity Rating.....	8
Repository Coverage and Quality Rating.....	9
5 Findings	10
Possible To Use Outdated Variables During Flash Loans.....	11
Possible Unrecoverable Funds On updatePullOracle()	12
Possible Drain Of Funds On updatePullOracle()	13
Missing Validation Of Input Parameters On updatePullOracle().....	14
External Arbitrary Call On _swapWithArbitraryDex().....	15
Function Does Not Account For Gas Calculations On All Chains.....	16
Possible Duplicate Assets In Flashloan	17
A Proof of Concepts	18

Executive Summary

Primex Finance contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between April 17, 2024 and April 26, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 10 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Primex Finance with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.



System Overview

Primex Finance is a decentralized exchange platform powered by smart contract where users can lend and swap tokens, perform trading operations, and earn rewards by utilizing the protocol.

The system is composed of several components: the bucket infrastructure where all the lending and borrowing happens; the trading managers that implement logic to swap, open and close positions, with or without conditions; and the protocol management where Primex admin control emergency pauses, propositions and executions.

As a general workflow, a lender is able to stake collateral on a bucket and receive yield-bearing tokens. The collateral can be used by traders to borrow and open margin positions. The keepers handle the conditions that occur over time, such as, opening limit orders when price as reached the limit price, liquidating users with underwater positions, closing trades with set stop-losses, etc.

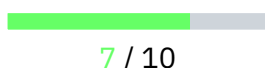
By using the different components of the protocol, all actors receive rewards that incentivise further usage. One of such rewards is the PMX token that offers utility and governance capabilities on the protocol.

Additional features were implemented such as, the integration of Enso and arbitrary contract swappers, the integration of the Optimistic chain and Supra oracle, and flash loan capabilities.

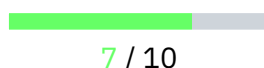


Repository Coverage and Quality

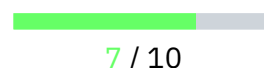
Code



Tests



Documentation



Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is good.**
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is undetermined. Overall, **tests coverage and quality is good.**
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is good.**

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository (main): [primex-finance/primex-protocol/src/contracts](#)
- Hash: bc23cffd9255276f9aabb9140b8216a69c128d20
- Repository: [primex-finance/primex_contracts/src/contracts](#)
- Hash: 6727c98e78a5a9e385974e8f65bd4dd7ceedd5c9

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial-related attack vectors

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions



Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info



Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
-|.. **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- ...||| **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	Possible To Use Outdated Variables During Flash Loans		Resolved	
RES-02	Possible Unrecoverable Funds On updatePullOracle()		Resolved	
RES-03	Possible Drain Of Funds On updatePullOracle()		Resolved	
RES-04	Missing Validation Of Input Parameters On updatePullOracle()		Resolved	
RES-05	External Arbitrary Call On _swapWithArbitraryDex()		Acknowledged	
RES-06	Function Does Not Account For Gas Calculations On All Chains		Resolved	
RES-07	Possible Duplicate Assets In Flashloan		Resolved	



Possible To Use Outdated Variables During Flash Loans

Critical RES-PRMX-FLI01

Code Workflow

Resolved

Code Section

- [contracts/FlashLoanManager/FlashLoanManager.sol#L78-L114](#)

Description

The function `flashLoan()` does not update important variables right after the transfer of the assets. Essentially, there are three steps to the function `flashLoan()`:

1. Calculate fees, available liquidities and transfer the assets to the borrower.
2. Give control to the borrower to execute the intended operation.
3. Handle the repayment of the flash loan.

During the first step, the transfer of the assets to the receiver is performed but the calls to the functions `_updateRates()` and `_updateIndexes()` are not performed. These are only performed later after the `executeOperation()` function call on the third step during `_handleFlashLoanRepayment()`. This effectively means that malicious borrowers may execute operations within the flash loan transaction, such as `deposit()` and `withdraw()`, that will operate under outdated rates and indexes variables. Taking into additional consideration that flash loans can amount to big numbers and different buckets, the execution of such functions may yield unfavorable results to the protocol.

The same attack vector is true for the variable `availableLiquidity`.

Recommendation

It is recommended to perform the update of relevant variables to occur earlier in the code workflow, before the function call to `executeOperation()`, e.g. during the function call `performFlashLoanTransfer()`.

Status

The issue has been fixed in `cc63163f8739b414d58a151b12985e68f6ecb9c6`.



Possible Unrecoverable Funds On `updatePullOracle()`

High

RES-PRMX-FLI02

Business Logic

Resolved

Code Section

- [contracts/PriceOracle/PriceOracle.sol#L156-L157](#)

Description

The function `updatePullOracle()` is payable and therefore, it can receive native Ether from callers of this function. When using `UpdatePullOracle.Pyth` as an oracle type, the calculated `updateFee` is required to be less than or equal to the amount of native Ether received via `msg.value`. When the caller of the function sends a higher amount of Ether than the one required by `updateFee`, the smart contract `PriceOracle` will retain the excess funds and it will be impossible to retrieve them, making the contract act as a sink for unrecoverable funds.

Recommendation

It is recommended to implement logic to refund callers of this function of excess Ether that is sent and not needed to pay the fees.

Status

The issue has been fixed in `c05e14c24848d83b9f7be32cd16d48cedb8fa25f`.



Possible Drain Of Funds On `updatePullOracle()`

High

RES-PRMX-FLI03

Data Validation

Resolved

Code Section

- `contracts/PriceOracle/PriceOracle.sol#L156`

Description

The function `updatePullOracle()` does not properly validate the amount of Ether sent to the smart contract and allows for funds to be drained to an external contract. This can occur when the validation on the variable `msg.value` is being done after funds have been transferred. While it seems counterintuitive, the variable `msg.value` does not change throughout the transaction, even after successful outbound transfers of native funds.

The following describes an example of an attack scenario for this specific case:

1. User calls `updatePullOracle()` while sending 10 Wei. The variable `msg.value` will contain the value 10.
2. For the Pyth `if` branch, the variable `updateFee` is calculated via `pyth.getUpdateFee()`.
3. Supposing a value of 9 for the variable `updateFee`, the `require` statement will pass successfully.
4. 9 Wei is sent to the external contract `pyth` via `updatePriceFeeds()`. 1 Wei is left from the funds deposited by the user.
5. Supposing input parameters that will result in the execution of multiple iterations of the `for` loop, and supposing a calculation result of 9 for the second `updateFee`, the next iteration of the `require` statement will still pass successfully (`9 <= 10`) since the value of `msg.value` does not change, even though funds have been transferred away on the first iteration. This means that if the contract `PriceOracle` contains funds, these will be used along with the 1 Wei left from the user.
6. Next iterations will drain the funds of the `PriceOracle` smart contract completely.

Recommendation

It is recommended to properly validate the amount of native Ether sent to the smart contract, taking into consideration that the variable `msg.value` is static throughout the transaction.

Status

The issue has been fixed in `c05e14c24848d83b9f7be32cd16d48cedb8fa25f`.



Missing Validation Of Input Parameters On updatePullOracle()

Medium RES-PRMX-FLI04

Data Validation

Resolved

Code Section

- [contracts/PriceOracle/PriceOracle.sol#L152-L163](#)

Description

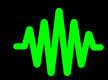
The function `updatePullOracle()` does not validate the input arrays `_data` and `_oracleTypes`. It is possible for this function to be called with mismatching input array lengths and allowing transactions to occur on only a subset of the input elements.

Recommendation

It is recommended to validate the input arrays for their expected length in prevention of possible unwanted scenarios that could be harmful or costly to the users of the smart contracts.

Status

The issue has been fixed in `cc63163f8739b414d58a151b12985e68f6ecb9c6`.



External Arbitrary Call On `_swapWithArbitraryDex()`

Low

RES-PRMX-FLI05

Business Logic

Acknowledged

Code Section

- [contracts/DexAdapter.sol#L795-L814](#)

Description

The function `_swapWithArbitraryDex()`, although private, can be called indirectly by users with malicious inputs that are not validated properly. One of the inputs is `_params.dexRouter` which is used as the target address of an external function call, making it possible for a malicious user to craft arbitrary transactions with input parameters that could damage the entire protocol.

There are multiple attack vectors associated with having this functionality enabled, mostly relating to the fact that it is possible for an attacker to impersonate the vulnerable smart contract. More specifically, it may be possible to drain the contract of all its funds, native or tokenized, as well as perform unauthorized actions acting as the `DexAdapter`. By taking into consideration the permissions and authorized actions that can be taken as `DexAdapter`, the impact may be more or less severe.

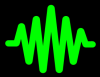
As an example, the provided proof of concept can be used to drain the contract `DexAdapter` of all its `NATIVE_CURRENCY`. Furthermore, by using a combination of calls, delegate calls and reentrancy, a malicious smart contract may abuse the `doApprove()` function to gain access to tokens owned by `DexAdapter`.

Recommendation

It is recommended to revise the necessity of having the functionality of swapping with arbitrary smart contracts.

Status

The issue was acknowledged by Primex's team. The development team stated "We allow interaction with arbitrary contracts to enable users to interact with any router they wish to use for swapping their assets. The DexAdapter will not have any permanent balance or special roles in the protocol, thus not posing an additional risk."



Function Does Not Account For Gas Calculations On All Chains

Low

RES-PRMX-FLI06

Business Logic

Resolved

Code Section

- [contracts/lens/PrimexLens.sol#L637-L641](#)

Description

The function `getEstimatedMinProtocolFee()` does not account for gas calculations for the newly introduced Optimistic chain integration. The variable `l1CostWei` will always yield a value of 0. While this may be intended, it does not conform with code implemented across other smart contracts on the protocol.

Recommendation

It is recommended to implement logic to calculate the proper amount for the variable `l1CostWei` for Optimistic chain transactions.

Status

The issue has been fixed in 9755e99e17cf8855f217f1ae6d709e7f33fcdea9.



Possible Duplicate Assets In Flashloan

Info

RES-PRMX-FLI07

Gas Optimization

Resolved

Code Section

- [contracts/FlashLoanManager/FlashLoanManager.sol#L78-L114](#)

Description

The `FlashLoanManager` contract contains logic related to flashloans. Most notably - the `flashLoan` function used to initiate the loan. Among other arguments, this function takes two arrays - `buckets` and `amounts` which specify assets to be used in the flashloan. It was observed that the `flashLoan` function does not verify the uniqueness of entries in the `buckets` array. Hence, it is possible to use the same asset multiple times during the same flashloan. As a consequence, in this scenario the contract will need to perform additional cross-contract calls and use more memory for the execution which would be inefficient.

Recommendation

It is recommended to implement a mechanism that will assure the uniqueness of entries contained in the `buckets` array.

Status

The issue has been fixed in `22edcae80fb92dd09d9948d10900a7d687a3ce30`.

Proof of Concepts

RES-05 External Arbitrary Call On _swapWithArbitraryDex()

```
// SPDX-License-Identifier: BUSL-1.1
```

```
pragma solidity 0.8.20;
```

```
contract DexAdapterExploit {  
    struct SwapParams {  
        bytes encodedPath;  
        address tokenIn;  
        address tokenOut;  
        uint256 amountIn;  
        uint256 amountOutMin;  
        address to;  
        uint256 deadline;  
        address dexRouter;  
    }  
  
    event Test();  
  
    address constant NATIVE_CURRENCY =  
    ↪ address(uint160(bytes20(keccak256("NATIVE_CURRENCY"))));  
  
    function callSwapExactTokensForTokens(address _dexAdapter) external {  
        SwapParams memory data;  
        data.encodedPath = abi.encode(address(this),  
    ↪ abi.encodeWithSignature("callback()"));  
        data.tokenIn = NATIVE_CURRENCY;  
        data.tokenOut = address(0);  
        data.amountIn = 1;  
        data.amountOutMin = 0;  
        data.to = address(1);  
        data.deadline = 0;  
        data.dexRouter = address(this);  
        (bool success, bytes memory returndata) = _dexAdapter.call(  
    ↪ abi.encodeWithSignature("swapExactTokensForTokens((bytes,address,address,uint256,uint256,a  
    ↪ data)  
        );  
        require(success, "failed");  
    }  
  
    function callback() payable external {  
        emit Test();  
    }  
}
```