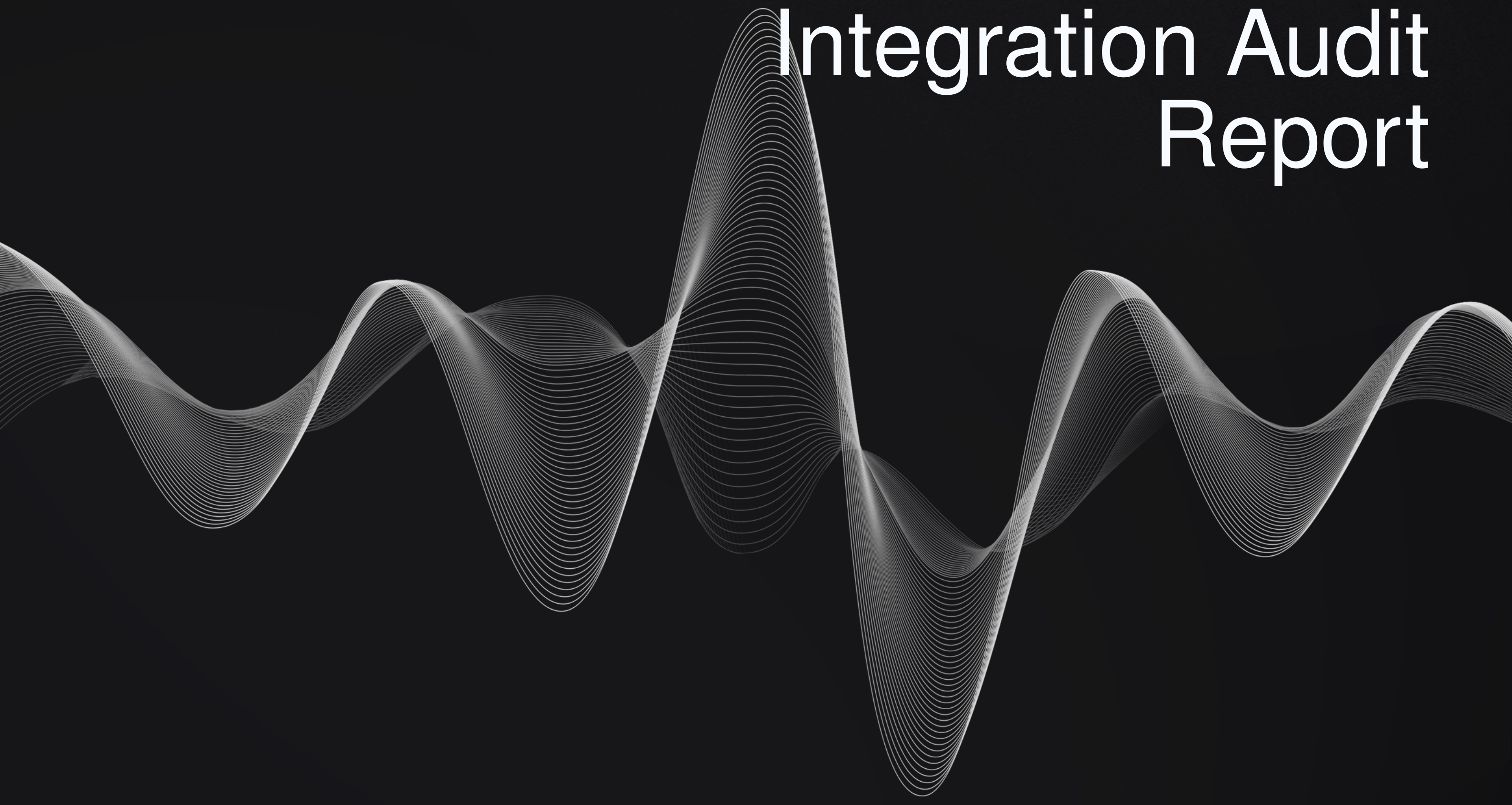
RESONANCE

bloq

Bloq

Metronome Synth Cross-Chain Integration Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.0)

Audit_Report_BLOQ-MSC_FINAL_20

Jul 31, 2023	○	v0.1	João Simões: Initial draft
Aug 16, 2023	○	v0.2	João Simões: Added findings
Aug 17, 2023	🟢	v1.0	Charles Dray: Approved
Sep 12, 2023	○	v1.1	João Simões: Reviewed findings
Oct 5, 2023	🟡	v2.0	Charles Dray: Published

Points of Contact	Michael McQuaid	Bloq	michael@bloq.com
	Charles Dray	Resonance	charles@resonance.security
	Luis Lubeck	Resonance	luis@resonance.security
Testing Team	João Simões	Resonance	joao.simoes@resonance.security
	Michał Bazyli	Resonance	michal.bazyli@resonance.security
	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Bloq contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between July 31, 2023 and August 17, 2023.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 3 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 14 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 Bloq 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

Metronome Synth is a decentralized finance (DeFi) multi-collateral and multi-synthetic protocol. Through the Metronome dApp, users are able to deposit crypto assets as collateral, and use that collateral to mint popular crypto synthetics. These synthetics allow users to perform slippage free trades (swaps) and engage in yield farming. This protocol operates on Ethereum and Optimism.

Metronome has included bridging/cross-chain capabilities through the usage of the LayerZero protocol where messages containing funds are sent cross-chain. The relevant swaps occur on LayerZero's mainnet and all liquidity is maintained in native tokens.

Repository Coverage and Quality

Code	Tests	Documentation
8 / 10	6 / 10	6 / 10

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 but does not use 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 average**.

- The documentation only includes the specification of the system and relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is average**.

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: `autonomoussoftware/metronome-synth/contracts`
- Hash: 55117e2ab9d809d6e93673ade203df53079a3604

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	🟧	Lack Of Validation Of collateralFactor Leads To Insufficient Funds When Paying Liquidation Fees	▇	Acknowledged
RES-02	🟧	Missing Modifier whenNotShutdown On Cross-Chain Calls	▁	Resolved
RES-03	🟧	Approve Function Allows Frontrunning	▃	Acknowledged
RES-04	🟧	Insufficient Validation Of collateralFactor	▁	Resolved
RES-05	🟩	Missing Zero Amount Validation Of amountIn_	▁	Resolved
RES-06	🟩	No Usage Of ERC165	▁	Acknowledged
RES-07	⬜	Unnecessary Modifier onlyIfSyntheticTokenIsActive On flashIssue()	▁	Resolved
RES-08	⬜	Hardcoded Zero Address As zroPaymentAddress	▃	Acknowledged
RES-09	⬜	Code Duplication Between Manageable And Pauseable Contracts	▃	Acknowledged

Lack Of Validation Of collateralFactor Leads To Insufficient Funds When Paying Liquidation Fees

Medium **RES-BLOQ-MSC01** Data Validation **Acknowledged**

Code Section

- contracts/Pool.sol#L548-L552
- contracts/Pool.sol#L419-L440

Description

The function `liquidate()` is meant to be used to liquidate unhealthy debt positions of other users. Whenever a position is unhealthy, this function makes a call to `quoteLiquidateOut()` to determine the amount of processing fees. As an example, we can suppose the processing fees are as follows.

```
amountToRepay_ / _toLiquidator = 100
_protocolFee = 10%
_liquidatorIncentive = 0%
```

After the function `quoteLiquidateOut()` is executed, it will return the following values:

```
_toLiquidator = 100
_fee = 10
_totalToSeize = 110
```

Since the account of the unhealthy position is paying the liquidation fees, the user of said account must possess a balance of deposit tokens greater than his actual debt. This factor is controlled by the `collateralFactor` variable. In order for the debt to be repaid in full, the `collateralFactor` needs to allow for a sufficient margin that should be greater than the fees paid during the liquidation process.

However, this fact is not guaranteed in the code. Not only can the `collateralFactor` variable range from 0% to 100%, but also the total percentage of liquidation fees is never checked if it is smaller than the current `collateralFactor`. This may ultimately lead to the impossibility of liquidating unhealthy positions.

Recommendation

It is recommended to implement code that guarantees the debtor as enough balance to pay for all the fees required during the liquidation process, i.e. guarantee that the set `collateralFactor` and `maxLiquidable` variables are sufficient to liquidate a maximum of 100% of the user's balance.

Status

The issue was acknowledged by Bloq's team. The development team stated "1) Liquidators are not allowed to repay all debt (check maxLiquidable that's at 50%). 2) It can be safely assumed that for most cases user will have enough collateral because CF won't be *close to 100% and this kind of price dropping is very rare. 3) Provided functions quoteLiquidateIn and quoteLiquidateMax help liquidator to calculate correct amount to repay in order to collect maximum collateral. It is already a governor method and unnecessary complexity would be introduced by adding validation around CF/liquidateFees relation.".*

Missing Modifier whenNotShutdown On Cross-Chain Calls

Medium **RES-BLOQ-MSC02** Access Control **Resolved**

Code Section

- `contracts/SmartFarmingManager.sol#L224`
- `contracts/SmartFarmingManager.sol#L338`
- `contracts/SmartFarmingManager.sol#L460`
- `contracts/SmartFarmingManager.sol#L503`

Description

The modifier `whenNotShutdown` is implemented to provide a way for the system's governor to stop all operations during emergencies. However, not all critical functions are protected with this modifier. The functions `layer2Leverage()`, `layer2FlashRepay()`, `retryLayer2LeverageCallback()`, and `retryLayer2FlashRepayCallback()` do not rely on the protection of the `whenNotShutdown` modifier and may be used my malicious actors on specific scenarios to damage the entire system.

Recommendation

It is recommended to guard critical functions with Pausable functionality such as the one implemented by the modifier `whenNotShutdown`.

Status

The issue has been fixed in 82e6411073c5497bbb5f5bbac350132356bfb520.

Approve Function Allows Frontrunning

Medium **RES-BLOQ-MSC03** Transaction Ordering **Acknowledged**

Code Section

- `contracts/DepositToken.sol#L170`
- `contracts/SyntheticToken.sol#L117`

Description

The ERC20 `approve()` function creates the potential for an approved spender to spend more than the intended amount. It is possible to perform a frontrunning attack, enabling an approved spender to call `transferFrom()` both before and after the call to `approve()` is processed. While there is no proper solution to this issue implemented into the standard, there are multiple workarounds that may or may not sacrifice backwards compatibility.

Recommendation

If backwards compatibility is not an issue, it is recommended to implement an `approve()` function that checks the actual current value against the perceived current value by the approver. The `approve()` function should only be successful if these two values match.

Implementing the functions `increaseAllowance()` and `decreaseAllowance()` is also a possible solution.

Status

The issue was acknowledged by Bloq's team. The development team stated "Our transferable ERC20 contracts (SyntheticToken and DepositToken) both implement increaseAllowance and decreaseAllowance. It's not desirable to change approve implementation to avoid breaking ERC20 standard.".

Insufficient Validation Of collateralFactor

Medium **RES-BLOQ-MSC04** Data Validation **Resolved**

Code Section

- `contracts/DepositToken.sol#L553`
- `contracts/DepositToken.sol#L371`
- `contracts/SmartFarmingManager.sol#L176`

Description

The functions `initialize()` and `updateCollateralFactor()` do not account for the possibility of a collateral factor being introduced as either `0` or `1e18`, corresponding to 0% or 100%. This effectively means that in some cases throughout the code, divisions will be made using this value that will yield "Division By Zero" errors, therefore reverting the transaction. Ultimately, this may cause a Denial of Service to legitimate users of the smart contract.

Recommendation

It is recommended to validate the variable `collateralFactor` whenever it is being initialized or updated against values that may cause on other functions of the system.

Status

The issue has been fixed in 72edd4670c2ca77c6fc3561908b4744549dee0ed.

Missing Zero Amount Validation Of amountIn_

Low

RES-BLOQ-MSC05 Data Validation **Resolved**

Code Section

- `contracts/SmartFarmingManager.sol#L364`

Description

The function `layer2Leverage()` does not perform validation on the input variable `amountIn_`, therefore making it possible for a user to input a zero amount to be transferred. Some tokens do not perform this check intrinsically and may waste gas and blockchain resources if not accounted for by developers.

Recommendation

It is recommended to perform the validation against zero amounts being passed in as function parameters during token transfers.

Status

The issue has been fixed in 82e6411073c5497bbb5f5bbac350132356bfb520.

No Usage Of ERC165

Low | **RES-BLOQ-MSC06** | Code Quality | **Acknowledged**

Code Section

- `contracts/DebtToken.sol#L122`
- `contracts/FeeProvider.sol#L62`
- `contracts/Pool.sol#L161`

Description

The ERC165 is a community-wide standard that implements a method to publish and query whether contracts support specific interfaces and their respective version. It is therefore useful when interacting between contracts and to perform more secure validations between functions that are expected to be implemented by specific contract addresses. This is possible through the usage of the function `supportsInterface()`. It should be noted that while it is not guaranteed that the implementation of the external contract is legitimate, it is possible to ensure that the respective interface being used in the external call is correct.

Throughout the code, several instances were found to be lacking the implementation and proper validation of address parameters that implement specific contracts. However, it should be noted that the specified list is not extensive.

Recommendation

It is recommended to implement the ERC165 standard when storing address variables of contracts that are being used for external calls.

Status

The issue was acknowledged by Bloq's team. The development team stated "To be evaluated later because 1) It introduces a change that impacts (almost) all contracts and the current branch has too much changes already 2) It will impact contract sizes.".

Unnecessary Modifier onlyIfSyntheticTokenIsActive On flashIssue()

Info **RES-BLOQ-MSC07** Code Quality **Resolved**

Code Section

- `contracts/DebtToken.sol#L258-L282`

Description

According to the code workflow implemented on the protocol, the modifier `onlyIfSyntheticTokenIsActive` being used on the function `flashIssue()` is not necessary since similar validations are already being made on the same smart contract interaction. This optimizes the code to save on gas during execution.

Recommendation

It is recommended to remove the usage of `onlyIfSyntheticTokenIsActive` on the function `flashIssue()` in order to optimize the execution of the code.

Status

The issue has been fixed in 82e6411073c5497bbb5f5bbac350132356bfb520.

Hardcoded Zero Address As zroPaymentAddress

Info **RES-BLOQ-MSC08** Code Quality **Acknowledged**

Code Section

- `contracts/ProxyOFT.sol#L144`

Description

The LayerZero protocol specification requires that whenever cross-chain estimates are being made or messages being sent, the input parameter `zroPaymentAddress` should not be hardcoded. This is due to the fact that different chains may have different implementations of the zero address, and, as such, this variable should be passed in as an argument by the user, and should not be hard-coded.

It should be noted that the impact for this finding is specific to the respective chains being included and implemented in the protocol.

Recommendation

It is recommended to not hardcode the variable `zroPaymentAddress` and pass it as a function input parameter.

Status

The issue was acknowledged by Bloq's team. The development team stated "In this specific case, it's a recommendation that it is safe not to follow. It should be related to non-EVM chains with different address(0) implementations that won't be integrated. It isn't worth adding it as function argument nor as storage parameter because of the complexity, code side, and gas consumption. In addition, in the worst-case scenario it is possible to upgrade the code and amend it if needed.".

Code Duplication Between Manageable And Pauseable Contracts

Info **RES-BLOQ-MSC09** Code Quality **Acknowledged**

Code Section

- `contracts/access/Manageable.sol`
- `contracts/utils/Pauseable.sol`

Description

The smart contracts Manageable and Pauseable implement the same functionality and could be adapted to remove the inherent duplication of code. Avoiding code duplication makes the code more readable for the smart contracts' users and saves gas during deployment.

Recommendation

It is recommended to rewrite duplicated code and adapt it the same business and workflow requirements of the protocol.

Status

The issue was acknowledged by Bloq's team. The development team stated "The only code duplication regarding these contracts are the whenNotPaused and whenNotShutdown modifiers, but their implementations are different.".

Proof of Concepts

No Proof-of-Concept was deemed relevant to describe findings in this engagement.