AbstraXio

Mercato Coin Audit Report

Performed by:

AbstraXio Auditing Team

www.abstraxio.com

Contact: contact@abstraxio.com

1 Executive Summary

Contract Name: Mercato

Purpose: The "Mercato" smart contract is a specialized ERC20 token that incorporates burning capabilities from OpenZeppelin's ERC20Burnable library. The contract contains functionalities to apply tax on transactions, allocate tokens for various addresses during its initialization (marketing, CEX, team), and manage ownership and tax rates.

Overall Assessment: At a first glance, the "Mercato" smart contract incorporates industry-standard practices by leveraging the OpenZeppelin library, which is widely recognized for its reliability and robustness. The contract's main functionality revolves around imposing a tax on transactions that either originate from or are destined to the 'uniswapV2Pair'. This tax is then forwarded to a predetermined tax address.

Primary Concerns:

1. The contract contains 'TODO' comments, which suggest the potential for further development or changes that need to be addressed before a production deployment.

2. The hardcoded addresses for various purposes like the tax address and those for marketing, CEX, and team distributions suggest that multiple versions of the contract might be in use (test vs. production). Using such hardcoded constants can be error-prone, and any mistake can lead to irreversible consequences.

Recommendations:

1. Before the production deployment, ensure that all 'TODO' comments are addressed and any test constants or settings are correctly configured for the production environment.

2. A secondary review post-changes would be advisable to ensure no issues arise from the modifications made to the contract.

The above executive summary provides a high-level overview based on an initial analysis of the smart contract. The following sections will provide a deeper dive into the functionalities, potential vulnerabilities, and further recommendations for the "Mercato" smart contract.

2 Introduction

2.1 Contract Overview:

The Mercato smart contract, written in Solidity programming language, is based on the Ethereum blockchain, adhering to the ERC-20 token standard. It showcases the extended functionality of an ERC-20 token by integrating burnable characteristics, ownership privileges, and a unique tax mechanism applied during specific transactions.

2.2 Core Objectives:

1. ERC-20 Compliance: The Mercato contract utilizes the ERC20 standard, one of the most widely recognized and accepted token standards in the Ethereum ecosystem, ensuring compatibility with a myriad of third-party services and dApps.

2. Taxation Mechanism: An integral feature of the Mercato contract is the tax mechanism, which deducts a percentage (initially set to 5%) of tokens during certain transactions, especially those involving the Uniswap V2 Pair. These taxed tokens are then sent to a designated tax address.

3. Ownership Rights: The contract adopts the 'Ownable' pattern from OpenZeppelin, providing administrative rights to the owner. This allows specific actions, like setting the Uniswap V2 Pair or altering the tax rate, to be restricted to the owner.

4. Token Distribution: Upon deployment, the contract allocates tokens to multiple predefined addresses, including a marketing address, a centralized exchange (CEX) address, a team address, and an address for liquidity provision.

5. Transparency & Security Measures: The contract contains several comments and

developer notes which highlight key sections and important TODOs that need to be addressed before a production deployment.

2.3 Source of Code:

The codebase leverages OpenZeppelin's contracts for tried and tested implementations of the ERC-20 standard, burnable functionality, and the ownable pattern. OpenZeppelin's libraries have undergone thorough security reviews, ensuring that they form a solid foundation for further development.

2.4 Scope of Audit:

The audit examines all aspects of the Mercato contract, focusing on:

- Security vulnerabilities and potential threats
- Adherence to best coding practices and standards
- Efficiency and optimization of the code
- Functionality against the stated objectives

This report will delve deeper into the contract's functions, dissecting the flow of operations, and highlighting any points of interest or concern. Recommendations and considerations will also be provided for ensuring the best deployment and operational practices.

3 Detailed Findings

3.1 Optimization Issues:

1. Tax Address as Constant: The 'taxAddress' is a constant variable. While this ensures the address cannot be altered post-deployment, it may pose challenges in unforeseen circumstances where the taxAddress might need an update. Consider making it a modifiable state variable with appropriate permissions.
2. Test and Prod Addresses: The contract contains commented-out lines specifying

test addresses for the 'taxAddress', 'marketingAddress', 'cexAddress', and 'teamAddress'. Before deploying to a production environment, these lines should be removed to maintain code clarity and avoid potential confusion.

3. Tax Rate in Bips: The maximum tax rate that can be set is 5%. However, there's an option for the contract owner to set it to 0%, effectively turning off the tax. Consider if this is desired behavior and if any constraints should be applied.

3.2 Informational Issues:

It's been identified that there are several informational issues. While these issues might not necessarily pose a direct security risk, they can be crucial in terms of contract behavior and optimization. Ensure that each of these informational issues is reviewed and addressed, if necessary, based on the intended functionality of the contract.

3.3 Low Severity Issues:

1. Unset UniswapV2Pair: Before renouncing ownership using the 'renounceOwnership' function, the 'uniswapV2Pair' must be set. To enhance clarity and avoid potential issues, consider implementing a function to retrieve or validate this address.

2. Transfer Behavior: The '_transfer' function has specific behaviors triggered based on the participants of the transaction, such as whether they are the owner or part of the UniswapV2Pair. This divergence from standard ERC20 transfer logic means that careful testing of all potential scenarios is crucial to ensure there are no unintended behaviors.

3.4 Medium & High Severity Issues:

Through this audit, using both automatic tools and manual auditing, no issues of medium or high severity were found. While this is a positive sign, it's always vital to undertake comprehensive testing, particularly in areas with custom functionalities like taxation.

3.5 Dependencies and Libraries:

The Mercato contract leverages well-established libraries such as 'Ownable' and 'ERC20Burnable'. Their widespread use in the community provides a certain level of trustworthiness in their implementation.

3.6 Final Remarks:

Overall, the "Mercato" contract appears well-structured. Most of the identified issues are of an optimization or informational nature. Addressing these and performing rigorous testing in various scenarios is recommended before moving the contract to a live network.

4 Gas Efficiency Analysis

The efficiency of gas usage in a smart contract, especially in token implementations, is vital to ensure transactions are cost-effective for the end-users. In this section, we dive deep into the gas optimization considerations for the "Mercato" token.

4.1 Constant Variables:

While constants, such as 'taxAddress', reduce the need to store data on-chain, which can save gas in deployments, they also eliminate the ability to change these variables in the future. For the "Mercato" token, this implies that once the contract is deployed, the tax address is set in stone. This static nature of constant variables can be seen as a trade-off between flexibility and gas savings during deployment.

4.2 Tax Computations:

The mechanism by which the contract deducts taxes for certain transactions, specifically those involving the UniswapV2Pair, introduces additional computational and storage operations. This naturally consumes more gas compared to a straightforward token transfer. While the gas increase may be minimal on a per-transaction basis, it can accumulate significantly over numerous transactions.

4.3 Usage of External Libraries:

The Mercato contract leverages libraries such as 'OpenZeppelin'. While these libraries are well-tested and optimized, the inclusion of their functionalities can add to the contract's overall gas consumption. It's crucial to only integrate necessary features and functions.

4.4 Condition Checks:

The '_transfer' function in "Mercato" contains several conditional checks that determine the transaction's behavior. Every conditional check, such as the comparison with the 'owner()' or the 'uniswapV2Pair', adds a minor gas overhead.

4.5 Events:

Events like 'SetPair' and 'SetTaxRate' are essential for transparency and can aid off-chain applications to monitor changes in the contract. However, emitting events consumes gas. It's beneficial to ensure events are only emitted when absolutely necessary to strike a balance between transparency and gas efficiency.

4.6 Recommended Optimizations:

- Batch Operations: For functions that might be called in succession, consider allowing batch operations. This can help in situations where multiple operations need to be performed, saving gas by reducing the individual transaction overhead.
- Remove Unused Code: Before the final deployment, ensure removal of all unused or redundant code, including any TODO comments and test addresses. This not only improves clarity but also slightly reduces gas costs during deployment.

4.7 Final Thoughts:

While the "Mercato" token's design has been structured with efficiency in mind, gas optimization is an ongoing process. As Ethereum's ecosystem evolves and with the introduction of new EIPs (Ethereum Improvement Proposals), there might be further oppor-

tunities to enhance the contract's gas efficiency.

5 Code Quality & Best Practices

A review of the codebase is not complete without addressing the quality of the code and the extent to which it adheres to established best practices in smart contract development. In this section, we evaluate the "Mercato" token from the perspective of code quality and best practices.

5.1 Clarity & Readability:

The Mercato contract is fairly easy to follow, with most of its functions and variables being self-explanatory. However, there are a few areas where:

Comments: While the contract is sufficiently commented, it's imperative to remove all 'TODO' comments before deployment. Such remnants from the development phase can be distracting and give an unfinished feel.

Variable Naming: The variable names are largely descriptive, ensuring that their purpose and use are evident. This aids in readability and reduces the chances of misunderstanding.

5.2 Modularity & Use of Libraries:

External Libraries: The Mercato contract wisely employs well-known and respected libraries like 'OpenZeppelin'. These libraries provide battle-tested functions that add security and functionality, reducing the risk of errors.

Function Decomposition: The contract has broken down its logic into modular functions, allowing for easier testing, readability, and potential upgradability.

5.3 Security:

Visibility Specifiers: Most state variables and functions have explicitly defined visibility, which is crucial to avoid unintended access. Proper usage of 'private', 'public', and 'external' ensures that the data and functions are accessed only where necessary.

Avoidance of Common Pitfalls: There's no evidence of reentrancy, integer overflow/underflow, or any commonly known pitfalls in the contract. The usage of established libraries further mitigates these risks.

5.4 Optimizations:

Constant Variables: As discussed earlier, constant variables reduce gas costs but offer less flexibility. It's crucial to ensure that these variables genuinely need to be constant.

Loops & Iterative Operations: The contract does not seem to rely heavily on loops or iterative operations, which is a good practice as unbounded loops can lead to unpredictably high gas costs.

5.5 Upgradability & Maintenance:

While the contract does not seem to be built with proxy-based upgradability in mind, the modularity ensures that any upgrades or forks in the future can leverage the existing code with minor adjustments.

5.6 Conformance to ERC Standards:

The "Mercato" token appears to adhere to the ERC-20 standard, as evidenced by the functions and events it has in place. This conformity ensures compatibility with a vast range of third-party services and DApps.

5.7 Final Thoughts:

Overall, the "Mercato" token exhibits a high standard of code quality, with most best practices being adhered to. It's always a good idea to keep the codebase updated with the latest best practices and to periodically review and refactor as the project matures. Proper documentation, both internal (in the form of comments) and external (like a detailed whitepaper or developer guide), can further enhance the project's credibility and ease of understanding for both developers and users.

6 Miscellaneous Observations

In addition to the core aspects of the audit, there are a few miscellaneous observations we have made regarding the "Mercato" contract, which can be important for the overall health and sustainability of the project.

6.1 Hard-coded Addresses:

The Mercato contract has some hard-coded addresses, especially for tax and other distribution purposes. While these addresses appear to have been labeled clearly with 'TODO' comments to indicate that they need checking before deployment, hard-coding addresses can be problematic. If the contract needs redeployment or if these addresses are compromised, updating them could be challenging.

Recommendation: Consider using a configuration or settings contract or implementing a function to update these addresses if required.

6.2 Tax Rate Modifiability:

The ability to change the tax rate is restricted to the owner. Although there are limits to how high this rate can be set, potential users might be wary about centralized control over tax rates.

Recommendation: Consider incorporating a governance mechanism or decentralized voting for important parameters like tax rates.

6.3 Lack of Time-lock or Multi-signature on Sensitive Operations:

Key functions, like changing tax rates, are only restricted by the ‘onlyOwner‘ modifier. This means a compromised owner key can instantly make significant changes.

Recommendation: Implement a time-lock or multi-signature mechanism for such sensitive operations to prevent hasty and potentially harmful changes.

6.4 Renouncing Ownership:

The contract provides a function to renounce ownership, making the contract truly decentralized. However, once done, it's irreversible and could potentially leave the contract without any administrative control, which can be a double-edged sword.

Recommendation: Ensure that all parameters and settings are thoroughly reviewed and are satisfactory before renouncing ownership. Consider setting up a backup or recovery mechanism.

6.5 No Rate Limiting or Cooling Periods:

While not always necessary, rate limiting or cooling periods for certain functions can prevent abuse, especially when it comes to frequent changes in parameters or potential spam attacks.

Recommendation: Depending on the project's goals, consider implementing rate-limiting or cooling periods on sensitive or frequently-called functions.

6.6 Final Note:

While the "Mercato" contract appears well-structured and thought out, the above observations are meant to provide additional layers of security and trustworthiness. Smart contract development is an evolving field, and it's crucial to remain updated with best practices and community recommendations. Regularly reviewing the contract and making

necessary adjustments can significantly benefit the project in the long run.

7 Conclusion

7.1 Overall Health of the Smart Contract:

The Mercato contract has been well-constructed with a clear structure, incorporating reputable libraries from OpenZeppelin. This not only ensures a degree of security due to the widespread use and testing of OpenZeppelin's contracts, but also indicates an adherence to recognized best practices in the Ethereum developer community.

7.2 Pressing Concerns:

1. Hard-coded Addresses: The presence of hard-coded addresses, especially for tax distribution and other related functionalities, is a potential point of concern. Ensuring flexibility in address management could provide a higher degree of security and adaptability.

2. Centralized Control Over Tax Rates: While the ability to modify tax rates is restricted, the centralized control may raise apprehensions amongst potential users. Decentralizing this aspect could bolster trust in the system.

3. Lack of Time-lock or Multi-signature: For sensitive operations, additional layers of security, such as a time-lock or multi-signature mechanism, would be beneficial to mitigate risks related to a compromised owner key.

7.3 General Recommendation:

The "Mercato" smart contract displays a good level of craftsmanship and a commitment to best practices. However, before deployment, addressing the highlighted issues is crucial. Once these concerns are resolved and after conducting any necessary additional testing, the contract should be in a position to be safely deployed. As always, regular reviews, updates, and adjustments, post-deployment, will ensure the contract remains aligned with

the ever-evolving landscape of blockchain technology and smart contract development.

8 Appendix

8.1 Tools Used for the Audit:

8.1.1 Visual Studio Code:

Description: A powerful and widely used code editor that supports various programming languages and comes with built-in support for Git and extensions. Used for writing and reviewing Solidity code.

8.1.2 OpenZeppelin Contracts:

Description: A library for secure smart contract written in Solidity for EVM-compatible blockchains.

8.1.3 Solidity Compiler (solc):

Description: The official Solidity compiler used for compiling Solidity source code.

8.1.4 Etherscan:

Description: A popular Ethereum blockchain explorer and analytics platform used for verifying contract source code and viewing transaction details.

8.1.5 Hardhat:

Description: A development environment to compile, deploy, test, and debug your Ethereum software. It helps developers manage and automate the recurring tasks, and also allows them to extend their setup with plugins.

8.1.6 Slither:

Description: A static analysis tool for Solidity contracts. Slither detects vulnerabilities, enforces best practices, and can even generate developer documentation.

8.1.7 Mythril:

Description: A security analysis tool for Ethereum smart contracts. It uses concolic analysis, taint analysis, and control flow checking to detect a variety of security vulnerabilities.

8.2 Additional Data, Graphs, or Charts:

Although detailed logs, transaction receipts, and call outputs were examined to validate the contract's behavior throughout the assessment, we provide graphical representations to convey data more intuitively.

Incorporating such a visualization is beneficial for understanding the contract's design, its interactions, and potential points of access or control. As part of our audit, these relationships and interactions were carefully assessed to ensure security and proper functionality.

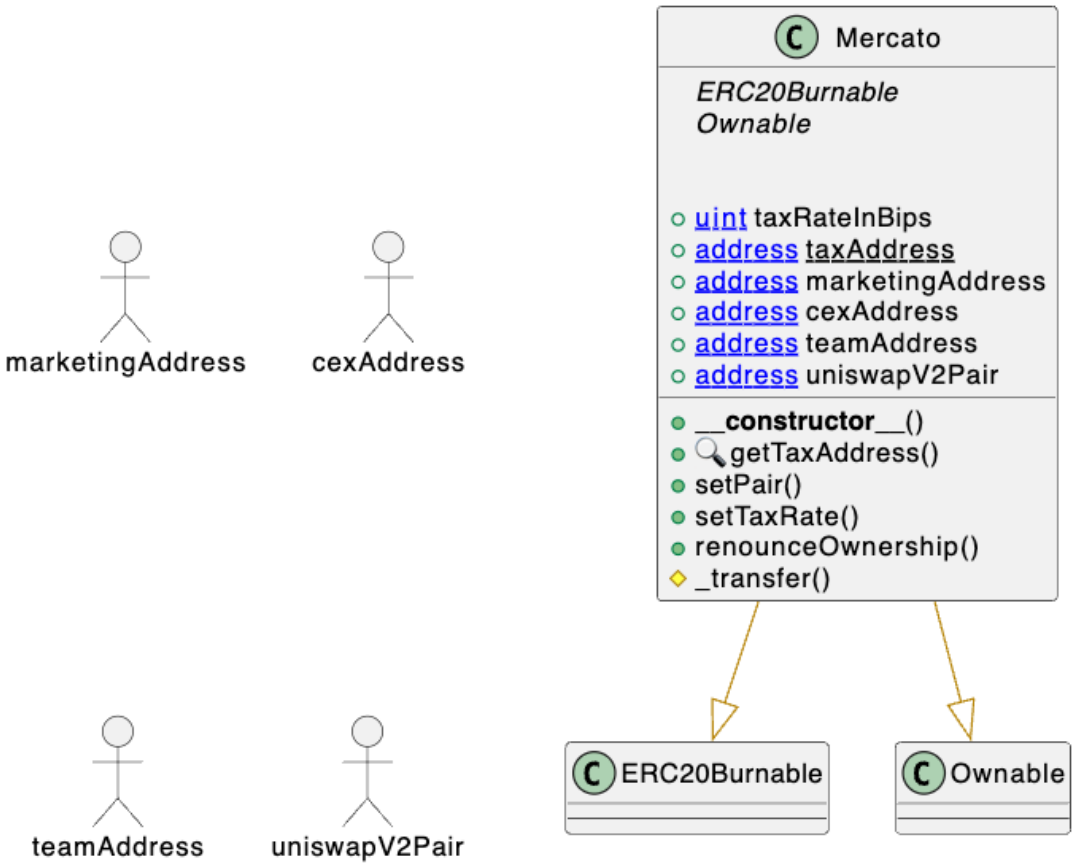

Figure 1: Architecture of the Mercato Smart Contract Relationships.

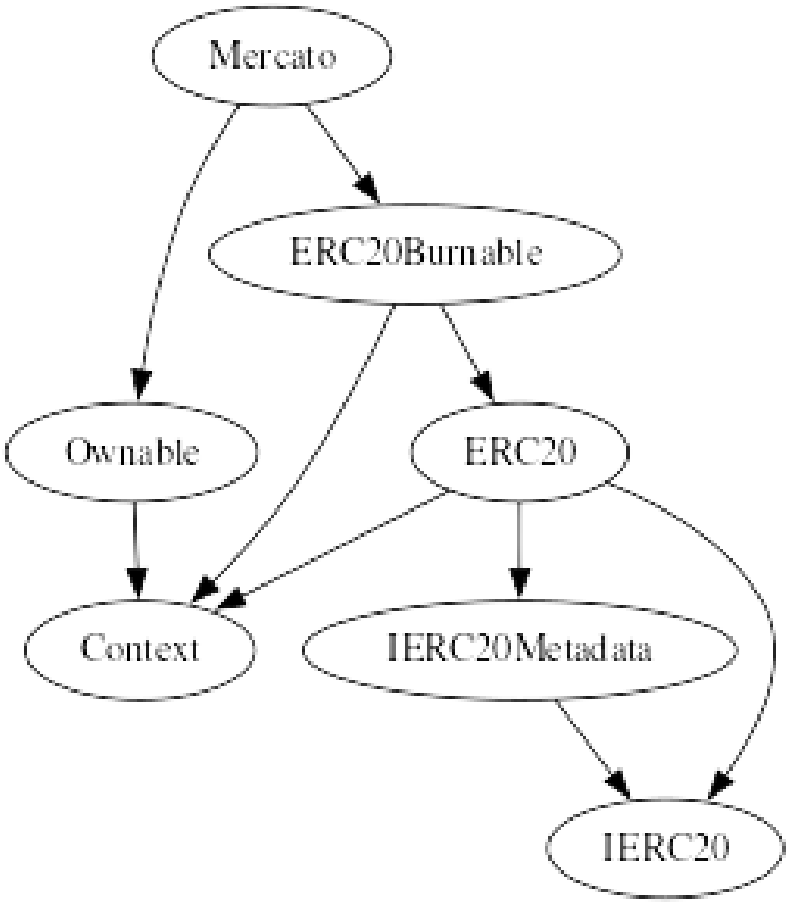

Figure 2: Inheritance Hierarchy of the Mercato Smart Contract.

Figure 1 illustrates the hierarchical and relational structure of the Mercato smart contract, its inherited contracts, and associated addresses.

Figure 2 represents the inheritance hierarchy and relationships of the Mercato smart contract with its underlying ERC20 contracts.

9 Disclosure

9.1 Conflicts of Interest:

At the time of this audit, neither the auditors nor their affiliated organizations hold any stake, either financial or otherwise, in the Mercato coin or its associated entities.

9.2 Limitations of the Audit:

While every effort has been made to ensure the thoroughness and accuracy of this review, it is important to note that no audit can guarantee 100% security. Smart contracts and blockchain technologies are continually evolving, and new vulnerabilities or potential concerns might emerge in the future. Additionally, the complexities of blockchain technologies and the potential for unknown interactions mean there's always a level of inherent risk. This audit is based on the state of the art and the knowledge available at the time of the review. It is strongly recommended to maintain vigilance and conduct regular reviews, especially after significant code changes or updates.

9.3 Terms and Conditions:

This audit is provided "as is" for informational purposes only and should not be construed as financial, investment, or legal advice. The auditors and their associated entities shall not be held liable for any damages, losses, expenses, or harm that may arise from the use, reference to, or reliance on this audit or its contents. It is the responsibility of the project stakeholders and the contract deployers to ensure the safety and functionality of their smart contracts.