Security Audit Report

27/03/2025

ThinkToken

Content

Introduction

Futureverse has the objective to be an open and scalable metaverse infrastructure with world-class content and a highly engaged community, they focus on the user experience and on developing custom protocols to deliver that experience.

ThinkToken is an ERC20 capped token that is to be minted to the TokenPeg contract. It is developed to help with the bridging solution between Ethereum and The Root Network (TRN).

As solicited by **Futureverse** and as part of the vulnerability review and management process, Red4Sec has been requested to perform a security code audit in order to evaluate the security of the **ThinkToken** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of **ThinkToken**. The performed analysis shows that the smart contract does not contain any critical vulnerabilities.

Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered either "endorsement" or "disapproval" of the guarantee of the correct business model of the analyzed project, nor as guarantee on the operation or viability of the implemented financial product.

Red4Sec makes full effort and applies every resource available for each audit, however it does not warrant the function, nor the safety of the project and it cannot be deemed a sufficient assessment of the code's utility and safety, bug-free status, or any other declarations of the project. Additionally, Red4Sec makes no security assessments or judgments about the underlying business strategy, or the individuals involved in the project.

Blockchain technology and cryptographic assets come with their own new risks and challenges, where the ecosystem, platform, its programming language, and other software related to said technology can have vulnerabilities that could lead to exploits. As a result, the audit cannot guarantee the explicit security of the audited projects.

The audit reports can be used to improve the code quality of smart contracts, to help limit the vectors of attack and to lower the high level of risks associated with utilizing new and continually changing technologies such as cryptographic tokens and blockchain, but they are unable to detect any future security concerns with the related technologies.

Scope

Red4Sec Cybersecurity has made a thorough audit of the **ThinkToken** security level against attacks, identifying possible errors in the design, configuration, or programming; therefore, guaranteeing the availability, integrity and confidentiality of the project and the possible assets treated and stored.

The scope of this evaluation includes the following items provided by **Futureverse**:

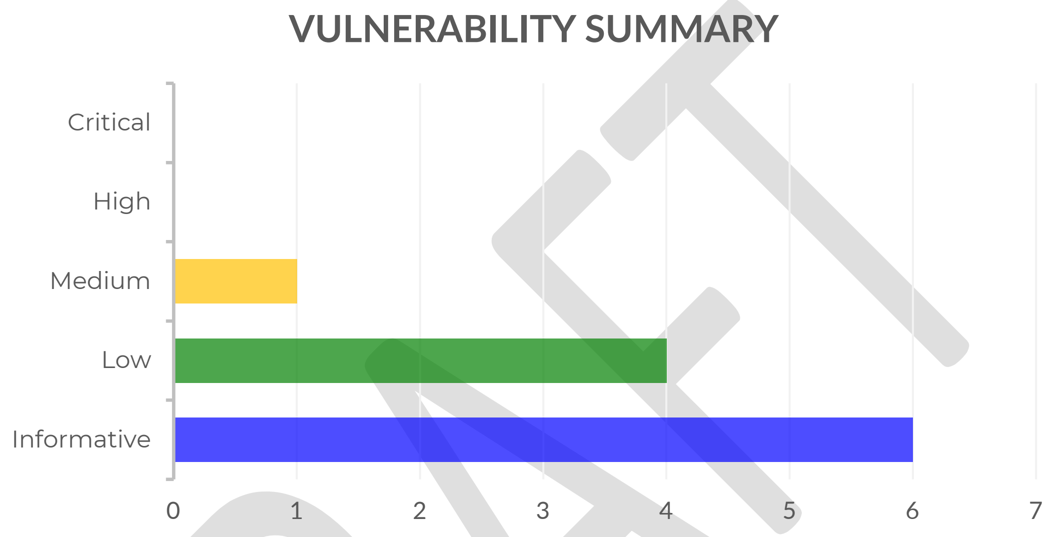
- https://github.com/futureversecom/ThinkToken
    - Initial commit: `101248bb248cf90088e0d28d77e678cf1a88a484`
    - March commit: `0846e5a5266570e122c8efe4758739210d5063e6`
        - `contracts/Token.sol`
        - `contracts/Roles.sol`
        - `contracts/ERC20Peg.sol`

Executive Summary

The security audit against **ThinkToken** has been conducted between the following dates: **17/02/2025** and **27/03/2025**.

Once the analysis of the technical aspects has been completed, the performed analysis shows that the audited source code contains non-critical issues that should be mitigated as soon as possible.

During the analysis, a total of **11 vulnerabilities** were detected. These vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.



Conclusions

Having conducted a comprehensive audit of the **ThinkToken**, the conclusion is that the overall security status of the project is fundamentally secure. However, Red4Sec has identified certain areas that could benefit from potential improvements, although these do not pose any immediate risks to the project's security posture.

The most notable issue found during the assessment is classified as a medium-risk Project Information Leak, primarily concerning data exposure. While this does not pose an immediate threat, addressing it would improve the overall security posture of the project.

Among the identified potential improvements, four are categorized as low-risk vulnerabilities: Lack of Input Validation, Contract Management Risks, Unrestricted Token Burning and Inverted Contract Detection Logic. These issues mainly relate to data validation, transparency in governance, access control and business logic. While these are not critical bugs, they could, over time, lead to inefficiencies or complications.

One notable observation is that the contracts grant the owner the ability to modify certain values at will. This creates a scenario where the owner could gain an unfair advantage in specific situations, possibly leading to unfair practices. Therefore, it is recommended to implement a more decentralized decision-making process or to have checks and balances in place.

Additional areas for improvement, including Unsecured Ownership Transfer, Optimized Error Reporting, and GAS Optimization, are considered informative. These are mostly related to design weaknesses, auditing and logging, and codebase quality, respectively. The first two primarily aim to increase transparency and user-friendliness, while GAS Optimization seeks to improve the contract's execution efficiency.

While the proper functioning of the project depends on external entities that were not included in the initial scope of the current audit, this should be considered when assessing the overall risk profile of the **ThinkToken** project.

In summary, the code quality, organization, and implemented security measures within the **ThinkToken** project are optimal and adhere to standard coding practices. Despite the minor potential improvements highlighted, the project is considered secure.

Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered upon the security audit.

List of vulnerabilities

Below, we have gathered a complete list of the vulnerabilities detected by Red4Sec, presented, and summarized in a way that can be used for risk management and mitigation.

Table of vulnerabilities

ID	Vulnerability	Risk	State
FTT-01	Lack of Inputs Validation	**Low**	**Assumed**
FTT-02	Contract Management Risks	**Low**	**Assumed**
FTT-03	Unsecured Ownership Transfer	**Informative**	**Assumed**
FTT-04	Optimize Error Reporting	**Informative**	**Partially Fixed**
FTT-05	Outdated Compiler	**Informative**	**Assumed**
FTT-06	GAS Optimization	**Informative**	**Partially Fixed**
FTT-07	Project Information Leak NEW	**Medium**	**Open**
FTT-08	Unrestricted Token Burning NEW	**Low**	**Open**
FTT-09	Inverted Contract Detection Logic NEW	**Low**	**Open**
FTT-10	Emit Events on State Changes NEW	**Informative**	**Open**
FTT-11	Code Quality Improvements NEW	**Informative**	**Open**

Vulnerability details

In this section, we provide the details of each of the detected vulnerabilities indicating the following aspects:

- Category
- Active
- Risk
- Description
- Recommendations

Lack of Inputs Validation

Identifier	Category	Risk	State
FTT-01	Data Validation	**Low**	**Assumed**

The `deposit` method in `ERC20Peg.sol` lacks fundamental validations for the `_destination` and `_amount` input arguments. Specifically, it does not check for null destination addresses (`address(0)`) or zero amounts. Without these validations, the contract could allow invalid or redundant transactions, leading to unnecessary gas consumption.

The lack of input validation in a method interacting with the bridge contract could present an operational risk, potentially leading to unexpected behavior or failed transactions on the target chain. Zero-address or zero-amount transfers not only represent a waste of resources in terms of gas but can also complicate cross-chain transaction reconciliation and system accounting.

In a context where the contract handles the transfer of assets between Ethereum and The Root Network, implementing these validations is crucial to maintaining system integrity and preventing inconsistencies between chains.

Additionally, it would be advisable to also include a zero-address check for the `_palletAddress` argument in the `setPalletAddress`.

Recommendations

- Implement validations for the destination address (`_destination`) ensuring it is not the zero-address using `require(_destination != address(0), "invalid destination")`.
- Add minimum amount check using `require(_amount > 0, "amount must be greater than 0")`.

Source Code References

- contracts/ERC20Peg.sol#L58-L91
- contracts/ERC20Peg.sol#L169

Fixes Review

The **Think Token** team has reviewed this issue and marked it as acknowledged, based on the clarification provided at the following reference:

- https://github.com/futureversecom/ThinkToken/tree/main/audit#audit-report-response

Contract Management Risks

Identifier	Category	Risk	State
FTT-02	Governance	**Low**	**Assumed**

A code analysis of the smart contracts, specifically `Token.sol` and `ERC20Peg.sol`, has identified architectural decisions and patterns that could impact the system's security and reliability. The current implementation presents certain risks that should be reviewed and addressed for overall improvement.

Unrestricted Token Acceptance Risk

The `deposit` method currently accepts any ERC20 token address without enforcing a whitelist or validating allowed tokens. This lack of restriction introduces significant risks, particularly with tokens that exhibit non-standard or malicious behavior.

For instance:

- **Deflationary tokens** that apply transfer fees could create mismatches between deposited and withdrawable amounts.
- **Tokens with variable balances** could manipulate holdings after deposit, breaking 1:1 peg parity.
- **Malicious tokens** could implement reentrancy functions or manipulate balances, compromising system integrity.

In the context of a cross-chain bridge between Ethereum and The Root Network, these risks could lead to financial losses, permanently locked funds, or inconsistencies in cross-chain accounting.

Centralized Owner Risk

The contract implements administrative functions that grant a high level of control to the contract owner.

The `setBridgeAddress` method is redundant considering that the bridge address is already set in the constructor, and its existence allows for subsequent modifications that could compromise the integrity of the system.

Additionally, `adminEmergencyWithdraw` grants the Owner the ability to withdraw any token from the contract without restrictions, which could result in the loss of user funds if the owner's account is compromised or if there is a malicious administrator.

(Only commit: ae8c5c5b4da93bfefa4d89326444ec3200958eb9)

In addition, the `setPeg` function in `Token.sol` is identified to present similar administrative control vulnerabilities. The function allows modifying the `peg` contract address previously set in `init`. The ability to modify the `peg` address without additional restrictions could allow a compromised administrator to redirect the flow of tokens to a malicious contract, potentially compromising user funds.

If this method needs to be maintained, several security improvements should be considered: restricting `setPeg` control exclusively to the `MULTISIG_ROLE`, implementing an event (e.g., `PegAddressChanged`) to log address modifications for transparency, and introducing a timelock mechanism to delay the change's effectiveness. This delay would provide users and system monitors with the opportunity to detect and respond to any potentially malicious modifications.

These administrative functions introduce a significant centralization risk in the bridge system's architecture.

Source Code References

- contracts/ERC20Peg.sol#L58
- contracts/ERC20Peg.sol#L164
- contracts/ERC20Peg.sol#L174
- contracts/Token.sol#L79

Fixes Review

The **Think Token** team has reviewed this issue and marked it as acknowledged, based on the clarification provided at the following reference:

- https://github.com/futureversecom/ThinkToken/tree/main/audit#audit-report-response

Unsecured Ownership Transfer

Identifier	Category	Risk	State
FTT-03	Design Weaknesses	**Informative**	**Assumed**

The `ERC20Peg.sol` contract implements OpenZeppelin's `Ownable` pattern for ownership management, using a direct transfer mechanism that could result in permanent loss of control of the contract if the new owner's address is specified incorrectly.

The current implementation of the ownership transfer mechanism poses a significant risk to the operational security of the bridge, as a mistake in specifying the new owner's address during the transfer would result in permanent loss of control of the contract.

A more secure approach would be adopting OpenZeppelin's `Ownable2Step` pattern, which introduces a two-step transfer process. This process requires explicit acceptance by the new owner.

Recommendations

- Replace the current implementation of `Ownable` with OpenZeppelin's `Ownable2Step`.
- Update technical documentation to reflect the new ownership transfer process and establish operational procedures for this process.

Source Code References

- contracts/ERC20Peg.sol#L15

Fixes Review

The **Think Token** team has reviewed this issue and marked it as acknowledged, based on the clarification provided at the following reference:

- https://github.com/futureversecom/ThinkToken/tree/main/audit#audit-report-response

Optimize Error Reporting

Identifier	Category	Risk	State
FTT-04	Auditing and Logging	**Informative**	**Partially Fixed**

The project has adequate error handling and clear error messages, however, the system for reporting errors can be optimized by using the improvements introduced in the latest solidity versions for a more efficient way of notifying users about a failure in the operation of the project and lower GAS consumption.

Reduce Require Messages Length

Ethereum Virtual Machine operates under a 32-byte word memory model where an additional gas cost is paid by any operation that expands the memory that is in use.

Therefore, exceeding error messages of this length means increasing the number of slots necessary to process the require, reducing the error messages to 32 bytes or less would lead to saving gas.

Source Code References

- contracts/Token.sol#L79
- ERC20Peg.sol#L68-L71
- contracts/ERC20Peg.sol#L74-L77
- contracts/ERC20Peg.sol#L103-L106

Use Custom Errors instead of Require

Custom errors are more gas efficient than revert strings in terms of deployment and runtime cost when the revert condition is met. The `require` statement is more expensive than the use of custom errors because it requires placing the error message on the stack, whether or not the transaction is reverted and regardless of the result of the condition.

Solidity **0.8.4** introduced custom errors, a more efficient way to notify users of an operation failure, The new custom errors are defined through the `error` statement, and they can also receive arguments. As indicated in the following illustrative example:

```
error Unauthorized();
error InsufficientPrice(uint256 available, uint256 required);
```

Afterwards, it is enough to call when needed using `if` conditionals:

```
if (msg.sender != owner) revert Unauthorized();
if (amount > balance[msg.sender]) {
            revert InsufficientPrice({
                available: balance[msg.sender],
                required: amount
            });
```

The downside is that the use of custom errors with the `require` function is not introduced until Solidity `0.8.26`.

Source Code References

- contracts/Token.sol#L69-L70
- contracts/Token.sol#L79
- contracts/ERC20Peg.sol#L63
- contracts/ERC20Peg.sol#L68
- contracts/ERC20Peg.sol#L74
- contracts/ERC20Peg.sol#L98
- contracts/ERC20Peg.sol#L103
- contracts/ERC20Peg.sol#L123
- contracts/ERC20Peg.sol#L127
- contracts/ERC20Peg.sol#L150

References

- https://blog.soliditylang.org/2021/04/21/custom-errors
- https://soliditylang.org/blog/2024/05/21/solidity-0.8.26-release-announcement/

Fixes Review

This issue has been partially addressed in the following commit:

- https://github.com/futureversecom/ThinkToken/commit/40b37c318b4646f8d06893ca789b18cec12f5f8d

Outdated Compiler

Identifier	Category	Risk	State
FTT-05	Outdated Software	**Informative**	**Assumed**

The `ERC20Peg.sol` contract has been identified as using an outdated compiler version (`0.8.17`) which contains known issues when newer versions are available with these bugs fixed. Using older compiler versions exposes the contract to known vulnerabilities and bugs that have been fixed in later versions.

Using outdated compiler versions poses a significant risk to the security of the contract, as new versions not only incorporate gas optimizations and new functionalities but also include critical security patches.

Recommendations

- Update the compiler version to the latest stable version (`0.8.20`) by modifying the pragma statement in `ERC20Peg.sol`.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md
- https://docs.soliditylang.org/en/latest/bugs.html

Source Code References

- contracts/ERC20Peg.sol#L2

Fixes Review

The **Think Token** team has reviewed this issue and marked it as acknowledged, based on the clarification provided at the following reference:

- https://github.com/futureversecom/ThinkToken/tree/main/audit#audit-report-response

GAS Optimization

Identifier	Category	Risk	State
FTT-06	Codebase Quality	**Informative**	**Partially Fixed**

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On EVM blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage increases, so will the value of GAS optimization.

Unnecessarily Indexed Event Arguments

The `Endowed` event could benefit from removing the indexed keyword since amount is a *uint256* and indexing large values is gas inefficient and provides limited filtering benefit.

The same applies to the `DepositActiveStatus` and `WithdrawalActiveStatus` events, which index a *bool* which is completely inefficient.

Source Code References

- contracts/ERC20Peg.sol#L34
- contracts/ERC20Peg.sol#L30-L31

Dead Code

Unused code elements have been identified in `Token.sol`, specifically the `fees` state variable and the `nonReentrant` modifier imported from `ReentrancyGuard.sol`. The presence of dead code unnecessarily increases the contract size and contributes to higher GAS costs during deployment.

Unused code in smart contracts represents a quality and efficiency issue that can have both technical and economic implications.

Source Code References

- contracts/Token.sol#L6
- contracts/Token.sol#L23

Fixes Review

The **Think Token** team has reviewed this issue and marked it as acknowledged, based on the clarification provided at the following reference:

- https://github.com/futureversecom/ThinkToken/tree/main/audit#audit-report-response

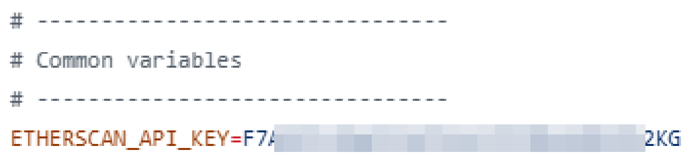
Additionally, the unused `fees` variable in `Token.sol` was already removed in the following commit:

- https://github.com/futureversecom/ThinkToken/commit/bb277e3033500181d05400ac19c47d19dade123b

Project Information Leak

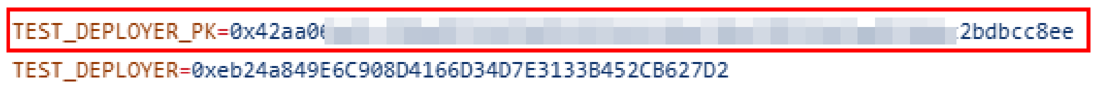
Identifier	Category	Risk	State
FTT-07	Data Exposure	**Medium**	**Open**

During the review, it was detected that the `.env.example` file contains sensitive information that was committed to the GitHub repository. This includes an Etherscan API key and several private keys, one of which corresponds to the `TEST_DEPLOYER` address (`0xeb24a849E6C908D4166D34D7E3133B452CB627D2`), which currently holds a balance on the Ethereum mainnet.



```
# --------------------------------
# Common variables
# --------------------------------
ETHERSCAN_API_KEY=F7A[redacted]2KG
```

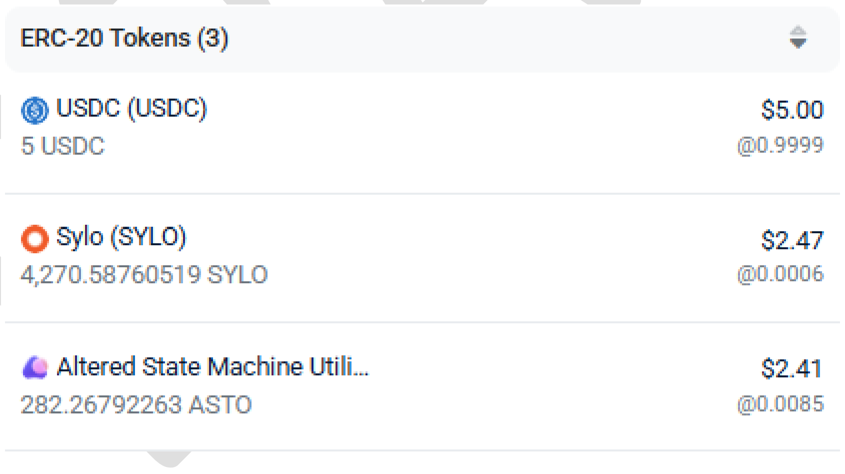
EtherScan API Key



```
TEST_DEPLOYER_PK=0x42aa0[redacted]2bdbcc8ee
TEST_DEPLOYER=0xeb24a849E6C908D4166D34D7E3133B452CB627D2
```

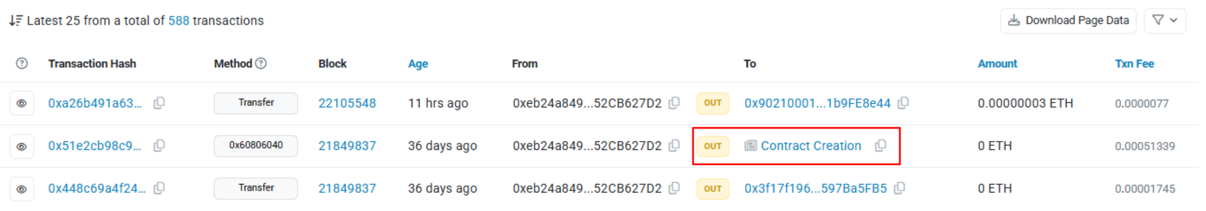
0xeb24a849E6C908D4166D34D7E3133B452CB627D2 Private Key

As observed, the address `0xeb24a849E6C908D4166D34D7E3133B452CB627D2` holds a balance on the Ethereum mainnet and is also the owner of a deployed contract.



ERC-20 Tokens (3)

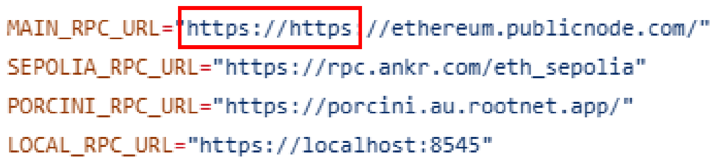
USDC (USDC) 5 USDC	$5.00 @0.9999
Sylo (SYLO) 4,270.58760519 SYLO	$2.47 @0.0006
Altered State Machine Utili... 282.26792263 ASTO	$2.41 @0.0085



Latest 25 from a total of 588 transactions

Transaction Hash	Method	Block	Age	From		To	Amount	Txn Fee
0xa26b491a63...	Transfer	22105548	11 hrs ago	0xeb24a849...52CB627D2	OUT	0x90210001...1b9FE8e44	0.00000003 ETH	0.0000077
0x51e2cb98c9...	0x60806040	21849837	36 days ago	0xeb24a849...52CB627D2	OUT	Contract Creation	0 ETH	0.00051339
0x448c69a4f24...	Transfer	21849837	36 days ago	0xeb24a849...52CB627D2	OUT	0x3f17f196...597Ba5FB5	0 ETH	0.00001745

Additionally, the constant `MAIN_RPC_URL` is incorrectly set, as it contains a duplicated protocol prefix (`https://https://ethereum.publicnode.com/`), which would prevent proper RPC communication.



```
MAIN_RPC_URL="https://https://ethereum.publicnode.com/"
SEPOLIA_RPC_URL="https://rpc.ankr.com/eth_sepolia"
PORCINI_RPC_URL="https://porcini.au.rootnet.app/"
LOCAL_RPC_URL="https://localhost:8545"
```

Wrong MAIN_RPC_URL established value

Recommendations

- Immediately rotate all exposed private keys and transfer any funds to new secure addresses.
- Modify the `.env.example` file to contain only variable structures without actual values, using placeholders such as *"YOUR_PRIVATE_KEY_HERE"*.

References

- https://etherscan.io/address/0xeb24a849E6C908D4166D34D7E3133B452CB627D2

Source Code References

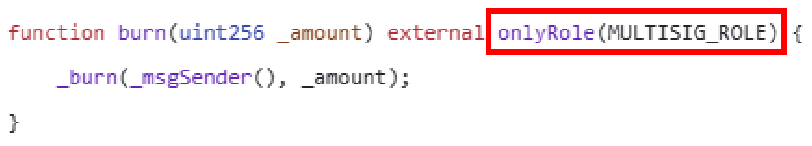
- .env.example

Unrestricted Token Burning

Identifier	Category	Risk	State
FTT-08	Access Controls	**Low**	**Open**

A change has been made to the implementation of the token burning mechanism that, while possibly intentional, warrants mention for review. The previous implementation restricted the `burn` function exclusively to accounts with the `MULTISIG_ROLE` role, while the current implementation, by inheriting directly from OpenZeppelin's `ERC20Burnable`, allows any user to burn their own tokens without restrictions.

This modification to access controls represents a significant change to the token's security model that could have significant implications for the system's economics and governance. In the original implementation, the ability to burn tokens was strictly controlled by entities with the `MULTISIG_ROLE` role, providing a centralized oversight mechanism for this critical operation. With the current implementation, any user can burn their own tokens at will, potentially impacting the total circulating supply of the token without oversight.



```
function burn(uint256 _amount) external onlyRole(MULTISIG_ROLE) {
    _burn(_msgSender(), _amount);
}
```

Previous role restriction

Recommendations

- Overwrite the `burn` function inherited from `ERC20Burnable` to reintroduce the access restriction to the `MULTISIG_ROLE` role if the system design requires centralized control over this operation.
- Alternatively, if you decide to maintain burn capability for all users, ensure that the bridge system can properly handle these events and maintain cross-chain consistency, and clearly document this change in the permissions model.

Source Code References

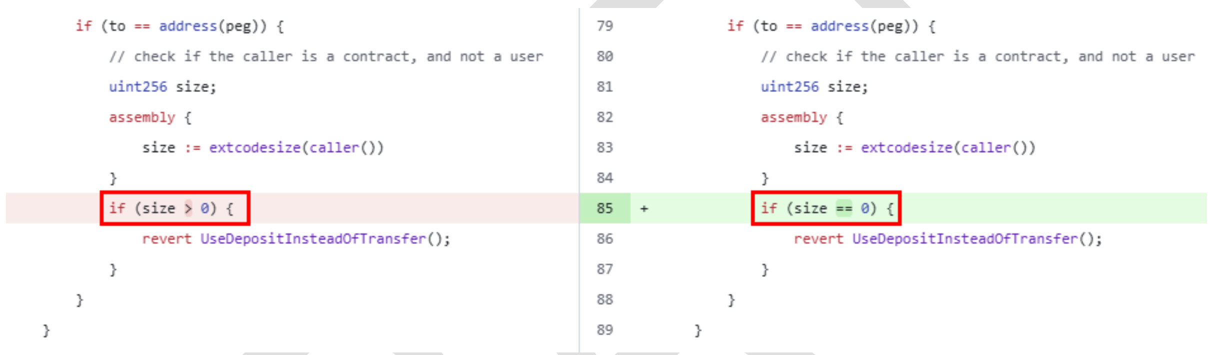
- contracts/Token.sol#L99

Inverted Contract Detection Logic

Identifier	Category	Risk	State
FTT-09	Business Logic	**Low**	**Open**

An important change has been identified in the validation logic of the `_beforeTokenTransfer()` function of the `Token.sol` contract. The condition that previously prevented contracts from transferring tokens directly to the **peg** contract has been reversed, now specifically blocking user wallets (EOA) while allowing contracts to perform these transfers.

The modification in validation logic represents a fundamental change in the token's behavior that could have significant implications for the security and usability of the system. In the previous implementation, contracts were forced to use the `deposit()` function instead of `transfer()` when interacting with the **peg** contract, ensuring proper processing of deposits through the bridge.



```
        if (to == address(peg)) {
            // check if the caller is a contract, and not a user
            uint256 size;
            assembly {
                size := extcodesize(caller())
            }
            if (size > 0) {
                revert UseDepositInsteadOfTransfer();
            }
        }
    }
```

```
        if (to == address(peg)) {
            // check if the caller is a contract, and not a user
            uint256 size;
            assembly {
                size := extcodesize(caller())
            }
+           if (size == 0) {
                revert UseDepositInsteadOfTransfer();
            }
        }
    }
```

With the logic reversed, it is now user accounts (EOAs) that receive the `UseDepositInsteadOfTransfer` error, while contracts can transfer tokens directly to the peg without restrictions.

Furthermore, this type of validation based on contract code detection (`extcodesize`) presents increasing risks with the adoption of smart wallets based on the ERC-4337 standard. As more users migrate to smart wallets, which are technically contracts, enforcing restrictions based on the EOA-contract distinction could introduce significant usability challenges in the future.

Recommendations

- Carefully review the purpose of this validation and consider restoring the original logic if the goal is to force the use of `deposit` for all bridge interactions.
- Consider an alternative approach that does not rely on the distinction between EOAs and contracts, in anticipation of the growing adoption of smart wallets.

Source Code References

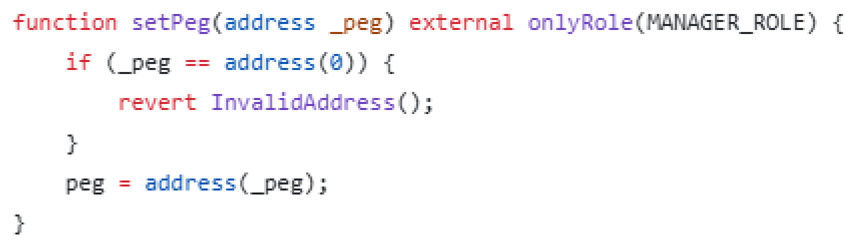
- contracts/Token.sol#L98-L100

Emit Events on State Changes

Identifier	Category	Risk	State
FTT-10	Auditing and Logging	**Informative**	**Open**

It is a good practice to emit events when there are significant changes in the states of the contract that can affect the result of its execution by the users.

The changes in the `setPeg` should emit events so that the potential actors monitoring the blockchain; such as DApps, automated processes and users, can be notified of these significant state changes.



```
function setPeg(address _peg) external onlyRole(MANAGER_ROLE) {
    if (_peg == address(0)) {
        revert InvalidAddress();
    }
    peg = address(_peg);
}
```

Recommendations

- Consider issuing events to notify of changes in the contract values that may affect the relationship of the users with the contract.

Source Code References

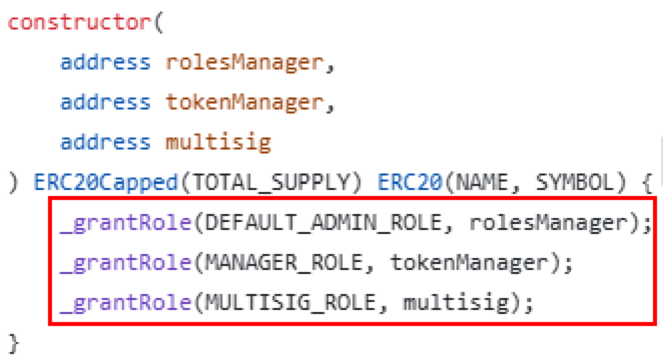
- contracts/Token.sol#L108

Code Quality Improvements

Identifier	Category	Risk	State
FTT-11	Codebase Quality	**Informative**	**Open**

Several opportunities for improving code quality have been identified in the `Token.sol` contract. While these do not pose security vulnerabilities, they impact the contract’s efficiency and maintainability. Specifically, the unused import of the `SafeERC20` library and inconsistencies in validation between the constructor and the `init` method have been noted.

Unused import of `SafeERC20` unnecessarily increases the contract's bytecode size, resulting in higher gas costs during deployment without providing any functionality.



```
constructor(
    address rolesManager,
    address tokenManager,
    address multisig
) ERC20Capped(TOTAL_SUPPLY) ERC20(NAME, SYMBOL) {
    _grantRole(DEFAULT_ADMIN_ROLE, rolesManager);
    _grantRole(MANAGER_ROLE, tokenManager);
    _grantRole(MULTISIG_ROLE, multisig);
}
```

Additionally, the inconsistency in validations between the `constructor` and the `init` method creates an inconsistent pattern of input validation. While `init` explicitly ensures that the peg address is not zero, the constructor does not apply similar checks for the `rolesManager`, `tokenManager`, and `multisig` addresses.

Recommendations

- Remove the unused `SafeERC20` import to reduce bytecode size and improve code readability.
- Implement consistent validations in the constructor for `rolesManager`, `tokenManager`, and `multisig` addresses, following the same pattern used in the `init` method.

Source Code References

- contracts/Token.sol#L30
- contracts/Token.sol#L44-L46

Annexes

Methodology

A code audit is a thorough examination of the source code of a project with the objective of identifying errors, discovering security breaches, or contraventions of programming standards. It is an essential component to the defense in programming, which seeks to minimize errors prior to the deployment of the product.

Red4Sec adopts a set of cybersecurity tools and best security practices to audit the source code of the smart contract by conducting a search for vulnerabilities and flaws.

The audit team performs an analysis on the functionality of the code, a manual audit, and automated verifications, considering the following crucial features of the code:

- The implementation conforms to protocol standards and adheres to best coding practices.
- The code is secure against common and uncommon vectors of attack.
- The logic of the contract complies with the specifications and intentions of the client.
- The business logic and the interactions with similar industry protocols do not contain errors or lead to dangerous situations to the integrity of the system.

In order to standardize the evaluation, the audit is executed by industry experts, in accordance with the following procedures:

Manual Analysis

- Manual review of the code, line-by-line, to discover errors or unexpected conditions.
- Assess the overall structure, complexity, and quality of the project.
- Search for issues based on the SWC Registry and known attacks.
- Review known vulnerabilities in the third-party libraries used.
- Analysis of the business logic and algorithms of the protocol to identify potential risk exposures.
- Manual testing to verify the operation, optimization, and stability of the code.

Automatic Analysis

- Scan the source code with static and dynamic security tools to search for known vulnerabilities.
- Manual verification of all the issues found by the tools and analyzes their impact.
- Perform unit tests and verify the coverage.

Vulnerabilities Severity

Red4Sec determines the severity of vulnerabilities found in risk levels according to the impact level defined by CVSSv3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST), classifying the risk of vulnerabilities on the following scale:

Severity	Description
Critical	Vulnerabilities that possess the highest impact over the systems, services, and/or sensitive information. The existence of these vulnerabilities is dangerous and should be fixed as soon as possible.
High	Vulnerabilities that could compromise severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.
Medium	Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.
Low	These vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.
Informative	It covers various characteristics, information or behaviors that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

RED4SEC

Invest in Security, invest in your future