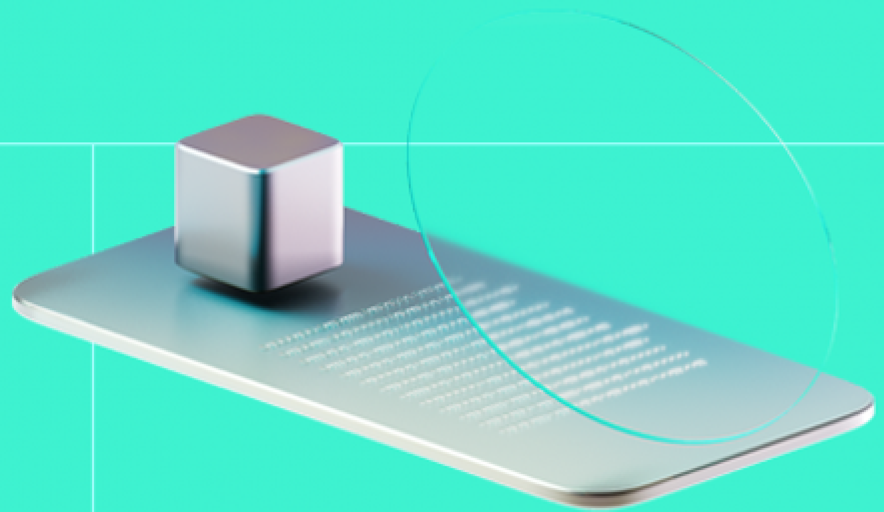
Smart Contract Code Review And Security Analysis Report



Customer: Bitlayer

Date: 22/04/2024

We express our gratitude to the Bitlayer team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Bitlayer leads with its innovative BitVM technology, offering a secure and computationally complete solution for enhancing Bitcoin's layer 2 capabilities.

Platform: EVM

Language: Solidity

Tags: Fungible Token; Permit Token; Centralization; Upgradable

Timeline: 11/04/2024 - 22/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/bitlayer-org/peg-tokens-contract
Commit	f0d2a54

Audit Summary

10/10	9/10	69.23%	5/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 8.1/10

The system users should acknowledge all the risks summed up in the risks section of the report

2	2	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Severity	Count
Critical	0
High	0
Medium	1
Low	1

Vulnerability	Status
F-2024-2131 - Insufficient Check for Blacklisted Users in TransferFrom Function	Fixed
F-2024-2154 - Unvalidated Chain ID in crossChainBurn Function	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Bitlayer
Audited By	Ivan Bondar
Approved By	Grzegorz Trawinski
Website	https://www.bitlayer.org/
Changelog	17/04/2024 - Preliminary Report; 22/04/2024 - Final Report

Table of Contents

System Overview **6**
- Privileged Roles 6

Executive Summary **8**
- Documentation Quality 8
- Code Quality 8
- Test Coverage 8
- Security Score 8
- Summary 8

Risks **10**

Findings **11**
- Vulnerability Details 11
- Observation Details 16
- Disclaimers 23

Appendix 1. Severity Definitions **24**

Appendix 2. Scope **25**

System Overview

BitLayer introduces PegToken, an innovative system operating on the EVM blockchains. This project, utilizing a UUPS upgradeable framework, is designed for dynamic token management and control.

The files in the scope:

- PegToken.sol:
  - This contract is an ERC20 token with extended functionalities. Key features include pausing/unpausing token transfers, blacklisting addresses, freezing/unfreezing tokens, and cross-chain burning.
  - Token minting is controlled, ensuring no pre-minting.
  - Allows cross-chain token burning, enabling wider ecosystem interoperability.
- TokenManager.sol:
  - Central control unit for creating and managing PegToken instances.
  - Implements a robust role-based access control system with distinct roles such as Admin, Operator, Freezer, and Pauser.
  - Facilitates token creation, upgrade, minting, and management of token state (pause/unpause, freeze/unfreeze).

Key Functionalities and Roles:

- Token Creation and Management: Operators can dynamically create new PegToken instances with specific attributes and upgrade existing tokens.
- Governance and Control: Admins have overarching control over the system, capable of pausing entire operations (stopTheWorld).
- Token Minting and Supply Management: Operators control minting of PegTokens, crucial for managing the token supply and ecosystem balance.
- Pause and Freeze Mechanisms: Pausers can temporarily halt token transactions, while Freezers can immobilize tokens in specific accounts, adding layers of control and security.
- Cross-Chain Functionality: CrossChainBurn feature for PegToken indicates a bridge between different blockchain networks, enhancing the token's utility and reach.

Privileged roles

- TokenManager.sol:
  - AdminRole:
    - Role: The AdminRole is a pivotal role with comprehensive control over the entire contract. Initially granted to the contract creator, this role has the highest level of authority.
    - Capabilities:
      - Authorize upgrades of the contract using _authorizeUpgrade.
      - Grant or revoke all other roles.
      - Implement the stopTheWorld feature, impacting the entire system.
  - Operator:
    - Role: Operators are responsible for the day-to-day management of the token ecosystem. They handle critical functions related to token operations.
    - Capabilities:

        - Create new token instances using createToken.
        - Upgrade token contracts with upgradeToken.
        - Set blacklist status for addresses using setBlackList.
        - Assign or revoke minter roles for each token using setMinter.
  - FreezeRole:
    - Role: Holders of the FreezeRole have control over the liquidity of tokens by managing their availability.
    - Capabilities:
      - Freeze or unfreeze tokens for specific addresses using freezeToken and unfreezeToken.
      - Recall tokens from one address to another using recall.
  - PauserRole:
    - Role: The PauserRole is designated to manage the transactional state of tokens, providing a control mechanism over their transferability.
    - Capabilities:
      - Pause or unpause token transactions on an individual token basis using pauseToken and unpauseToken.
- PegToken.sol:
  - Minter:
    - Role: Minters are addresses authorized to create new tokens in the system.
    - Capabilities:
      - Mint new tokens using mint, subject to not being blacklisted and the token not being paused or the system stopped.
  - Manager:
    - Role: The Manager is a central figure with overarching control over critical functionalities of the token.
    - Capabilities:
      - Upgrade the contract using _authorizeUpgrade.
      - Set or remove addresses from the blacklist using setBlacklist.
      - Assign or revoke minter status using setMinter.
      - Pause or unpause token transfers using pause and unpause.
      - Mint new tokens via mint.
      - Recall tokens from one account to another using recall.
      - Freeze or unfreeze tokens in an account using freeze and unfreeze.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **5** out of **10**.

- Functional requirements are partially missed.
  - Project overview is basic.
  - Roles in the system are adequately described.
  - Use cases are not explicitly described.
  - For each contract, key features are moderately detailed.
  - Interactions within the system are basically outlined.
- Technical description is not provided.
  - Run instructions are provided.
  - Technical specification is not provided.
  - NatSpec documentation is not included.

Code quality

The total Code Quality score is **9** out of **10**.

- The development environment is configured.
- Solidity Style Guide violations.

Test coverage

Code coverage of the project is **69.23%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is partially missed.
- Interactions by several users are tested.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **1** medium, and **1** low severity issues, leading to a security score of **9** out of **10**. After addressing findings during the initial review, the security score was subsequently raised to **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **8.1.** This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- Centralization Risks:
  - Single Points of Failure and Control: The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
  - Administrative Key Control Risks: The digital contract architecture relies on administrative keys for critical operations. Centralized control over these keys presents a significant security risk, as compromise or misuse can lead to unauthorized actions or loss of funds.
  - Single Entity Upgrade Authority: The token ecosystem grants a single entity the authority to implement upgrades or changes. This centralization of power risks unilateral decisions that may not align with the community or stakeholders' interests, undermining trust and security.
- Upgradeability Risks:
  - Flexibility and Risk in Contract Upgrades: The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
  - Absence of Upgrade Window Constraints: The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.
  - Compatibility and Stability Risks with Mixed Contract Imports: The project utilizes a combination of upgradable and non-upgradable contract imports from OpenZeppelin, which can lead to compatibility issues and affect the stability of the contract in an upgradeable context. This mixing may restrict the full advantages of the upgradeable pattern, potentially leading to inefficiencies and heightening security risks. Proper management and continuous review are essential to mitigate these risks and ensure consistent functionality across different contract versions.

Findings

Vulnerability Details

F-2024-2131 - Insufficient Check for Blacklisted Users in TransferFrom Function - Medium

Description:

The `transferFrom` function in the LiquidityBootstrapPool.sol contract presents a potential security vulnerability related to its handling of blacklisted addresses. While the function checks to ensure both the 'from' and 'to' addresses in a transfer are not blacklisted (`notBlacklisted(from)` and `notBlacklisted(to)`), it fails to verify if the `msg.sender` executing the `transferFrom` is blacklisted. This oversight allows a blacklisted user to circumvent the blacklist restrictions indirectly.

For example, if a blacklisted user has previously received an approval from another user to transfer tokens on their behalf, they can still execute `transferFrom` to move tokens to another address they control, despite being blacklisted. This scenario demonstrates how the lack of a `notBlacklisted(msg.sender)` check can be exploited, leading to potential unauthorized token transfers and undermining the effectiveness of the blacklist mechanism.

Affected Code:

```
function transferFrom(address from, address to, uint256 value)
public
override
notBlacklisted(from)
notBlacklisted(to)
whenNotPaused
worldNotStopped
returns (bool)
{
return super.transferFrom(from, to, value);
}
```

Assets:

- contracts/PegToken.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Fixed

Classification

Severity: Medium

Impact: 3/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Recommendations

Remediation: To mitigate this risk, implement a `notBlacklisted(msg.sender)` check within the `transferFrom` function. This additional verification step ensures that not only the sender and receiver of the tokens are compliant with the blacklist criteria, but also the entity executing the transaction is not blacklisted, thus providing a more robust security framework against unauthorized transfers.

Remediation (Revised commit: 35be917): The `transferFrom` function in the LiquidityBootstrapPool.sol contract now includes a `notBlacklisted(msg.sender)` check, ensuring that the person initiating the transfer is not blacklisted. This update prevents blacklisted users from circumventing restrictions by using approvals granted before their blacklisting.

Evidences

Reproduce:

PoC Steps:

- Setup and Token Allocation: Load the necessary contracts and accounts, then mint tokens for two users, where one of them will be blacklisted.
- Approval Setup: Have a legitimate user (user1) approve a future blacklisted user to transfer tokens on their behalf.
- Blacklisting: Mark the approved account as blacklisted in the contract.
- Circumvention Attempt: Use the blacklisted account to execute the `transferFrom` function, attempting to transfer tokens from the legitimate user to a new account.
- Validation: Check if the new account received the tokens, which would indicate the blacklisted user's success in bypassing the restriction.

PoC Code:

```
it("TransferFrom Function Vulnerable to Blacklisted User Circumventi
on", async function () {
// Load necessary contracts and accounts
const { tokenManager, operator, pegUSDC, blacklisted, user1, newBlac
klisted } = await loadFixture(deploySys);

// Define token transfer value
const mintValue = ethers.parseEther("200.00");

// Mint tokens for user1 and blacklisted accounts
```

```
await tokenManager.connect(operator).mint("USDC", user1.address, min
tValue);
await tokenManager.connect(operator).mint("USDC", blacklisted.addres
s, mintValue);

// User1 approves the blacklisted account to transfer their tokens
await pegUSDC.connect(user1).approve(blacklisted.address, mintValue)
;

// Mark the account as blacklisted
await tokenManager.connect(operator).setBlackList("USDC", blackliste
d.address, true);
expect(await pegUSDC.isBlacklist(blacklisted.address)).to.be.true;

// Attempt to transfer tokens from user1 to a new account via the bl
acklisted account
await pegUSDC.connect(blacklisted).transferFrom(user1.address, newBl
acklisted.address, mintValue);

// Verify that the new account received the tokens, indicating a cir
cumvention of blacklist restrictions
expect(await pegUSDC.balanceOf(newBlack
```

See more

Results:

```
PegToken
✔ TransferFrom Function Vulnerable to Blacklisted User Circumvention
(1174ms)


1 passing (1s)
```

Files: Blacklisted.test.ts

F-2024-2154 - Unvalidated Chain ID in crossChainBurn Function - Low

Description: In the `crossChainBurn` function of the LiquidityBootstrapPool.sol contract, there is a notable risk due to the absence of validation for the `toChainId` parameter. Users can input any chain ID, including non-existent or unsupported ones, leading to unintentional token burns.

This function allows users to burn tokens on the current chain for use on another chain. However, without proper checks on the `toChainId`, there is a risk that users might accidentally input an incorrect or unsupported chain ID. Such a mistake would result in the permanent loss of tokens, as they would be burned without the possibility of recovery or use on the intended chain.

Affected Code:

```
function crossChainBurn(uint256 value, address to, uint256 toChainId
)
public
notBlacklisted(msg.sender)
whenNotPaused
worldNotStopped
{
require(toChainId != block.chainid, "burn to same chain");

_burn(msg.sender, value);

emit TokenCrossChainBurned(msg.sender, to, toChainId, value);
}
```

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Recommendations

Remediation: To prevent this issue, implement a validation mechanism for `toChainId` to ensure that it corresponds to a supported and existing blockchain network. This could involve maintaining a list of valid chain IDs within the contract or integrating with an external service that confirms the validity

of a chain ID. Adding such a check minimizes the risk of accidental token loss due to user input errors.

Remediation (Revised commit: 645e4ba): The issue with the `crossChainBurn` function in the LiquidityBootstrapPool.sol contract was resolved by its removal. The decision to eliminate this function mitigates the risk of accidental token loss due to improper chain ID inputs by users. Going forward, cross-chain operations will be managed by a dedicated cross-chain bridge contract, enhancing security and ensuring that tokens are transferred or burned in a controlled and verified manner.

Observation Details

F-2024-2124 - Floating Pragma - Info

Description: A "floating pragma" in Solidity refers to the practice of using a pragma statement that does not specify a fixed compiler version but instead allows the contract to be compiled with any compatible compiler version. This issue arises when pragma statements like `pragma solidity ^0.8.23;` are used without a specific version number, allowing the contract to be compiled with the latest available compiler version. This can lead to various compatibility and stability issues.

Assets:

- contracts/TokenManager.sol [https://github.com/bitlayer-org/peg-tokens-contract]
- contracts/PegToken.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Accepted

Recommendations

Remediation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

Remediation (Revised commit: 35be917): The recommendation to lock the pragma version for Solidity contracts was not implemented. The project team has decided to continue using a floating pragma.

F-2024-2125 - Public functions not called by the contract should be declared external instead - Info

Description: In Solidity, function visibility is an important aspect that determines how and where a function can be called from. Two commonly used visibilities are `public` and `external`. A `public` function can be called both from other functions inside the same contract and from outside transactions, while an `external` function can only be called from outside the contract. A potential pitfall in smart contract development is the misuse of the `public` keyword for functions that are only meant to be accessed externally. When a function is not used internally within a contract and is only intended for external calls, it should be labeled as `external` rather than `public`.

Affected Function:

```
187: function crossChainBurn(uint256 value, address to, uint256 toCh
ainId)
```

Assets:

- contracts/PegToken.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Fixed

Recommendations

Remediation: Declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`.

Remediation (Revised commit: 35be917): The `crossChainBurn` function was removed from the contract, fixing the issue with its visibility setting.

F-2024-2127 - Unused Import - Info

Description: In the TokenManager.sol contract, there is an import statement for Create2 from OpenZeppelin's contracts package, but the Create2 functionality is not utilized anywhere in the contract. The specific import statement is:

```
import "@openzeppelin/contracts/utils/Create2.sol";.
```

The presence of an unused import statement, like Create2, contributes to unnecessary clutter in the code, potentially leading to confusion or misunderstanding about the contract's functionality.

Assets:

- contracts/TokenManager.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Fixed

Recommendations

Remediation: Remove the unused import statement for Create2 to clean up the contract, making it more straightforward and reducing potential confusion for readers.

Remediation (Revised commit: 35be917): The unused import of `Create2` was removed from the TokenManager.sol contract.

F-2024-2128 - Inconsistent Use of Upgradable and Non-Upgradable OpenZeppelin Contracts - Info

Description:

The TokenManager.sol contract is importing a mix of upgradable (UUPSUpgradeable) and non-upgradable (AccessControlEnumerable) OpenZeppelin contracts. This mixture can lead to compatibility issues in an upgradable contract context.

Affected Imports:

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradea
ble.sol";
import "@openzeppelin/contracts/access/extensions/AccessControlEnume
rable.sol";
```

The contract might not fully benefit from the upgradeable pattern, leading to inefficiencies and potential security risks.

Assets:

- contracts/TokenManager.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Accepted

Recommendations

Remediation:

- Replace non-upgradeable OpenZeppelin contract imports with their upgradeable counterparts. Specifically, substitute `AccessControlEnumerable` and `UUPSUpgradeable` with their respective versions from the `@openzeppelin/contracts-upgradeable` package.
- Update the `initialize` function to include initializers for the upgradable contracts, such as `__AccessControl_init()` and `__UUPSUpgradeable_init()`, to properly initialize their state.
- Ensure that these initializers are called in the correct order in the `initialize` function to set up the contract state correctly for upgradeable deployments.

Remediation (Revised commit: 35be917): The recommendation to switch all contract imports in the TokenManager.sol to upgradeable versions from OpenZeppelin was not implemented. The project team has decided to retain the current mix of upgradeable and non-upgradeable imports.

F-2024-2129 - Shadowing of State Variables in initialize Function - Info

Description: The initialize function in the PegToken contract shadows the `name` and `symbol` parameters which are also state variables defined in the inherited ERC20Upgradeable contract. Shadowing occurs when a local variable (function parameter in this case) has the same name as a state variable, potentially leading to confusion or errors.

While this is not a functional issue in itself, it is considered a bad practice as it can cause readability and maintenance problems.

Assets:

- contracts/PegToken.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Fixed

Recommendations

Remediation:

- Rename the parameters in the `initialize` function to avoid shadowing. Common practices include using an underscore prefix (e.g., `_name`, `_symbol`) or a different naming convention.
- Ensure that the new parameter names clearly convey their purpose and do not conflict with any existing variables or functions in the contract.

Remediation (Revised commit: 35be917): The initialize function in the PegToken contract was updated to use distinct parameter names (`name_`, `symbol_`, `decimals_`), effectively eliminating variable shadowing and improving code clarity and maintainability.

F-2024-2130 - Hardcoded EIP712 Domain Name in PegToken Initialization - Info

Description:

In **PegToken.sol,** the `initialize` function is designed to set up the newly created token instance with specific characteristics like `name`, `symbol`, and `decimals`.

The `__ERC20Permit_init` function is invoked with a hardcoded string `"PegToken"`:

```
__ERC20Permit_init("PegToken");
```

This approach does not align the `EIP712` domain name with the token's actual `name`. The `EIP712` signature protection, part of the permit functionality, includes the contract address and the domain name in its calculation. Here, every token generated by the factory will use "`PegToken`" as their domain name, irrespective of their individual `name` and `symbol`.

All tokens created share the same domain name for `EIP712` signatures, leading to potential confusion, as the domain name does not reflect the actual token identity. While the contract address in the `EIP712` domain separator ensures uniqueness, the uniform domain name might not be the most transparent or intuitive approach.

Users interacting with these tokens might face confusion, especially when dealing with multiple tokens, since the `EIP712` domain name does not match the actual token name.

Assets:

- contracts/PegToken.sol [https://github.com/bitlayer-org/peg-tokens-contract]

Status: Fixed

Recommendations

Remediation:

Update the `initialize` method to dynamically set the `EIP712` domain name to match the actual token name:

```
__ERC20Permit_init(name);
```

Ensuring that the `EIP712` domain name reflects the actual token name enhances clarity and alignment with token identity.

Remediation (Revised commit: 35be917): The `initialize` function in the PegToken contract now dynamically sets the EIP712 domain name to

match the actual token name during initialization, enhancing clarity and ensuring the token identity aligns with its EIP712 signature protection.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/bitlayer-org/peg-tokens-contract
Commit	f0d2a54db1648ec76b55c1dcd227a9af16de4c8b
Whitepaper	
Requirements	AUDIT.md
Technical Requirements	README.md

Contracts in Scope

./contracts/PegToken.sol

./contracts/TokenManager.sol