CHAINLINK

Migration Lockbox

Security Assessment Report

Version: 2.1

June, 2025

Contents

Introduction

Sigma Prime was commercially engaged by Chainlink to preform a review of an ERC20 migration contract developed by Aphyla

The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Chainlink components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Chainlink components in scope.

Overview

The TokenMigrator contract facilitates the migration of tokens from a source ERC20 token to a target ERC20 token in a 1:1 exchange ratio. The contract implements ownership controls and pausability features, allowing the owner to manage token deposits, withdrawals, and contract operations.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Aphyla/token-migrator repository.

The scope of this time-boxed review was strictly limited to the following contracts at commit 28f2c74:

- TokenMigrator.sol
- TokenMigratorFactory.sol

The fixes of the identified issues were assessed at commit e7efe67 and fc7a2d7.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [**?**, **?**].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 2 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Informational: 1 issue.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Chainlink components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
MGL-01	Token Decimal Mismatch In Migration	Medium	Resolved
MGL-02	Fee-on-Transfer & Rebasing Token Compatibility	Informational	Resolved

MGL-01	Token Decimal Mismatch In Migration		
Asset	`src/TokenMigrator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `TokenMigrator` contract performs a 1:1 token migration without validating or accounting for differences in decimal places between the source and target tokens. This can lead to economic consequences if tokens with different decimals are used.

The current implementation in the `migrate()` function transfers the exact same numerical amount from source to target tokens:

```
function migrate(uint256 amount) external override whenNotPaused {
    s_totalMigrated += amount;
    _deposit(i_sourceToken, _msgSender(), amount);
    _withdraw(i_targetToken, _msgSender(), amount);
    emit TokenMigrated(_msgSender(), amount);
}
```

This approach assumes both tokens have identical decimal places, which may not always be the case in actual deployments.

Impact

Since any token can be set as the source and target token, when tokens with different decimal places are used, the migration can result in value discrepancies:

- **18-decimal to 6-decimal migration**: Migrating 1 token (1×10^{18} units) from an 18-decimal token would result in receiving 1×10^{18} units of a 6-decimal token, equivalent to 1×10^{12} actual tokens (1 trillion tokens)
- **6-decimal to 18-decimal migration**: Migrating 1 token (1×10^{6} units) from a 6-decimal token would result in receiving 1×10^{6} units of an 18-decimal token, equivalent to 1×10^{-12} actual tokens (essentially worthless)
- **Real-world example**: USDC (6 decimals) to a standard ERC20 token (18 decimals) migration would result in massive value loss for users

This could lead to:

- Significant financial losses for users
- Unintended token inflation or deflation

Recommendations

Decimal Validation: Add validation in the constructor to ensure both tokens have the same number of decimal places:

```
require(
    sourceToken.decimals() == targetToken.decimals(),
    "Token decimals must match"
);
```

Resolution

This issue has been resolved by implementing decimal validation in the contract deployment process. The development team added checks to ensure that the source and target tokens have matching decimal places.

The resolution was implemented at commit fc7a2d7.

MGL-02	Fee-on-Transfer & Rebasing Token Compatibility
Asset	`src/TokenMigrator.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `TokenMigrator` contract assumes standard ERC20 token behaviour and does not account for fee-on-transfer tokens. These tokens deduct a fee during transfers, meaning the recipient receives fewer tokens than the amount specified in the transfer call, which can break the intended 1:1 migration ratio.

Some tokens (e.g., STA, PAXG) deduct a fee during transfers, meaning the recipient receives fewer tokens than the amount specified in the transfer call. The current implementation does not account for this behaviour:

```
function _deposit(IERC20 token, address user, uint256 amount) internal {
    // ... validation checks ...
    token.safeTransferFrom(user, address(this), amount);
}

function _withdraw(IERC20 token, address user, uint256 amount) internal {
    // ... validation checks ...
    token.safeTransfer(user, amount);
}
```

When fee-on-transfer tokens are used as the source token:

- The contract receives fewer source tokens than expected during deposits (amount - fee)
- Users still receive the full `amount` of target tokens during withdrawal
- The `s_totalMigrated` counter becomes inaccurate, tracking the intended amount rather than actual received amount

Recommendations

Provide documentation specifying that the `TokenMigrator` system is designed for standard ERC20 tokens only. Explicitly document the limitations that arise when using fee-on-transfer tokens.

Resolution

The `TokenMigrator` documentation was updated to clarify token compatibility requirements.

The resolution was implemented at commit e7efe67.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Ran 8 tests for test/tests-local/TokenMigrator.t.sol:TokenMigratorTest
[PASS] test_CreateTokenMigratorViaFactory() (gas: 936685)
[PASS] test_InitialTokenBalances() (gas: 37414)
[PASS] test_OwnerCanDepositTargetTokens() (gas: 50046)
[PASS] test_OwnerCanWithdrawTokens() (gas: 140108)
[PASS] test_PauseUnpauseFunctionality() (gas: 133956)
[PASS] test_SimpleMigration() (gas: 131386)
[PASS] test_TokenMigratorDeployment() (gas: 25316)
[PASS] test_TokenMigratorFactoryDeployment() (gas: 5414)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 10.48ms (5.18ms CPU time)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

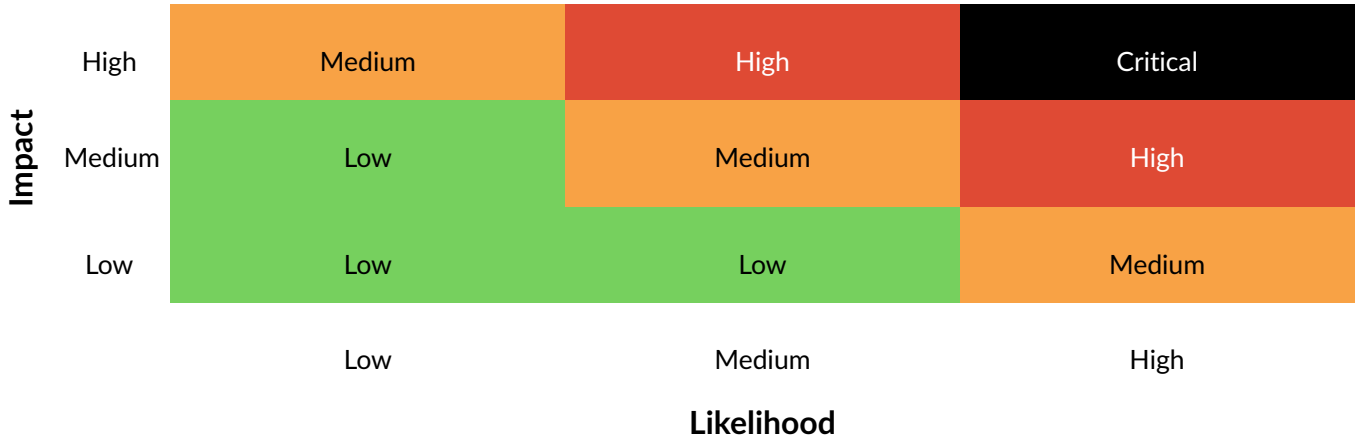

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.