Zellic

Mantle

Smart Contract Security Assessment

July 11, 2023

Prepared for:

Mantle Network

Prepared by:

Nipun Gupta and Sina Pilehchiha

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Mantle Network from June 7th to June 9th, 2023. During this engagement, Zellic reviewed Mantle's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could a malicious entity gain control over the minting process and create tokens beyond the limit? Is the minting-cap mechanism fair, and does it operate as expected?
- Is the token-burning mechanism implemented correctly and securely, and could it be exploited to burn tokens belonging to other users maliciously?
- Are the functions that are restricted to the contract owner (such as minting tokens and setting mint cap) properly protected, and can these functions be accessed or exploited by nonowners?
- Can the ownership of the contract be transferred maliciously, or are there any potential security vulnerabilities around the ownership?
- Are there any aspects of the ERC-20 token standard that are not correctly implemented or could be exploited? Specifically, are transfer, balance, allowance, and approval functions implemented and secured properly?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Any other components that have not been explicitly listed in the scope of this engagement

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

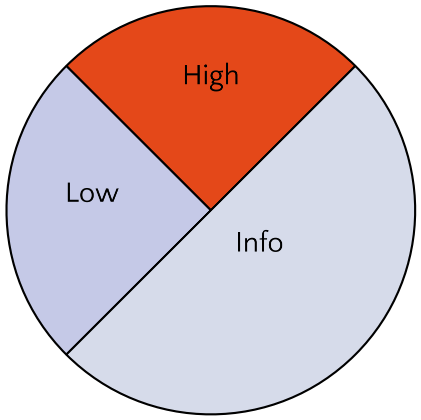
1.3 Results

During our assessment on the scoped Mantle contracts, we discovered four findings. No critical issues were found. One was of high impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Mantle Network's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	0
Low	1
Informational	2



2 Introduction

2.1 About Mantle

Mantle is a suite of Ethereum scaling solutions including an optimistic rollup and ZK rollup built using an iterative modular chain approach and supported by BitDAO's native token $BIT.

It is designed to bolster support for hyper-scaled-throughput decentralized applications (dApps) –– from decentralized derivatives exchanges (DEXs), to gaming, to operations of decentralized autonomous organizations (DAOs).

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Mantle Contracts

Repositories	https://github.com/mantlenetworkio/mantle
	https://github.com/mantlenetworkio/mantle-token-contracts
Versions	mantle: `d627d242fe19f50f344f1ff4b27532d1757303a6`
	mantle-token-contracts: `b2016dfb932d85b8b33a9294e8280aa04ca46975`
Programs	• L1MantleToken.sol
	• L1StandardBridge.sol
	• L2StandardBridge.sol
	• MantleTokenMigrator.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-days. The assessment was conducted over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Nipun Gupta, Engineer
nipun@zellic.io

Sina Pilehchiha, Engineer
sina@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 7, 2023	Start of primary review period
June 9, 2023	End of primary review period

3 Detailed Findings

3.1 The `initialize` function is not using the `initializer` modifier

- **Target**: L1StandardBridge
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The `initialize` function in L1StandardBridge is not using the `initializer` modifier but instead uses `messenger` to verify if the function has already been initialized or not. If this contract is accidently initialized with `messenger` set to `address(0)`, an attacker can reinitialize the contract and thus steal tokens from the contract using the withdrawal functions.

```
function initialize(address _l1messenger, address _l2TokenBridge,
    address _l1MantleAddress) public {
    require(messenger == address(0), "Contract has already been
    initialized.");
    messenger = _l1messenger;
    l2TokenBridge = _l2TokenBridge;
    l1MantleAddress = _l1MantleAddress;
}
```

Impact

If there are any tokens in the contract and the `messenger` is set to `address(0)`, an attacker can steal those tokens from the contract.

Recommendations

Use the `initializer` modifier, or in the `initialize` function, revert the transaction if any parameter is `address(0)`.

Remediation

This issue has been acknowledged by Mantle Network, and a fix was implemented in commit a53dd956.

3.2 Protocol does not account for fee-on-transfer tokens

- **Target**: L1StandardBridge
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `_initiateERC20Deposit` function does not account for tokens that charge fees on transfer.

There is an expectation that the `_amount` of tokens deposited to the project contract when calling `depositERC20To` or `depositERC20` will be equal to the amount of tokens deposited, and hence the mapping `deposits` is updated by adding the same `_amount`. However, there are ERC-20s that do, or may in the future, charge fees on transfer that will violate this expectation and affect the contract's accounting in the `deposits` mapping.

Below is the function `_initiateERC20Deposit` from the L1StandardBridge contract (some part of the function is replaced by `// [removed code]` to only show the relevant code):

```
function _initiateERC20Deposit(
    address _l1Token,
    address _l2Token,
    address _from,
    address _to,
    uint256 _amount,
    uint32 _l2Gas,
    bytes calldata _data
) internal {
    // When a deposit is initiated on L1, the L1 Bridge transfers the
    funds to itself for future
    // withdrawals. The use of safeTransferFrom enables support of "broken
    tokens" which do not
    // return a boolean value.
    // slither-disable-next-line reentrancy-events, reentrancy-benign
    IERC20(_l1Token).safeTransferFrom(_from, address(this), _amount);

    // [removed code]

    // slither-disable-next-line reentrancy-benign
    deposits[_l1Token][_l2Token] = deposits[_l1Token][_l2Token] + _amount;
    // slither-disable-next-line reentrancy-events
    emit ERC20DepositInitiated(_l1Token, _l2Token, _from, _to, _amount,
    _data);
}
```

Impact

The deposits mapping will overestimate the amount of fee-on-transfer tokens in the contract.

Recommendations

Consider implementing a require check that compares the contract's balance before and after a token transfer to ensure that the expected amount of tokens are transferred.

```
    function _initiateERC20Deposit(
        address _l1Token,
        address _l2Token,
        address _from,
        address _to,
        uint256 _amount,
        uint32 _l2Gas,
        bytes calldata _data
    ) internal {
        // When a deposit is initiated on L1, the L1 Bridge transfers the
    funds to itself for future
        // withdrawals. The use of safeTransferFrom enables support of
    "broken tokens" which do not
        // return a boolean value.
        // slither-disable-next-line reentrancy-events, reentrancy-benign
        uint256 expectedTransferBalance =
        IERC20(_l1Token).balanceOf(address(this)) + _amount;
        IERC20(_l1Token).safeTransferFrom(_from, address(this), _amount);
        uint256 postTransferBalance =
        IERC20(_l1Token).balanceOf(address(this));
        require(expectedTransferBalance ==
        postTransferBalance, "Fee on transfer tokens not supported");
        // [removed code]

        // slither-disable-next-line reentrancy-benign
        deposits[_l1Token][_l2Token] = deposits[_l1Token][_l2Token]
    + _amount;

        // slither-disable-next-line reentrancy-events
        emit ERC20DepositInitiated(_l1Token, _l2Token, _from, _to,
    _amount, _data);
    }
```

Remediation

This issue has been acknowledged by Mantle Network, and a fix was implemented in commit 305b5cab.

3.3 Possible rounding issues in L1MantleToken

- **Target**: L1MantleToken
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `mint` function in L1MantleToken calculated the `maximumMintAmount` using the following formula:

```
uint256 maximumMintAmount = (totalSupply() * mintCapNumerator)
    / MINT_CAP_DENOMINATOR;
```

Below is the `mint` function:

```
function mint(address _recipient, uint256 _amount) public onlyOwner {
    uint256 maximumMintAmount = (totalSupply() * mintCapNumerator)
    / MINT_CAP_DENOMINATOR;
    if (_amount > maximumMintAmount) {
        revert MantleToken_MintAmountTooLarge(_amount,
    maximumMintAmount);
    }
    if (block.timestamp < nextMint)
    revert MantleToken_NextMintTimestampNotElapsed(block.timestamp,
    nextMint);

    nextMint = block.timestamp + MIN_MINT_INTERVAL;
    _mint(_recipient, _amount);
}
```

If the `totalSupply` and `mintCapNumerator` are small enough, they might round down to zero when divided by `MINT_CAP_DENOMINATOR`. This would revert the transaction because of the if condition following the calculation, and an admin would not be able to mint the tokens. It is advised to use the `mintCapNumerator` and `_initialSupply` at a value large enough so the above calculations do not round down the `maximumMintAmount` to zero.

Impact

The `mint` function would revert.

Recommendations

Set the `_initialSupply` in initialize and `mintCapNumerator` using `setMintCapNumerator` to values large enough so the division does not round down the `maximumMintAmount` to zero.

Remediation

Mantle Network rejected this finding and provided the response below:

> In our practical use case, it is unlikely to encounter situations where `totalSupply` and `mintCapNumerator` are too small. The situation mentioned in the report does not exist.

3.4 Possible rounding issues in MantleTokenMigrator

- **Target**: MantleTokenMigrator
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The swap calculation in `MantleTokenMigrator` is done using `_tokenSwapCalculation` (as shown below). This function uses `TOKEN_CONVERSION_NUMERATOR` and `TOKEN_CONVERSION_DENOMINATOR` variables declared in the constructor.

```
function _tokenSwapCalculation(uint256 _amount)
    internal view returns (uint256) {
    return (_amount * TOKEN_CONVERSION_NUMERATOR)
    / TOKEN_CONVERSION_DENOMINATOR;
}
```

If the `TOKEN_CONVERSION_NUMERATOR` and `TOKEN_CONVERSION_DENOMINATOR` values differ in large order of magnitude, the `_tokenSwapCalculation` would return a lesser value. This might transfer less tokens to a user than they expect in the `_migrateTokens` functions as shown below:

```
function _migrateTokens(uint256 _amount) internal {
    if (_amount == 0) revert MantleTokenMigrator_ZeroSwap();

    uint256 amountToSwap = _tokenSwapCalculation(_amount);

    // transfer user's BIT tokens to this contract
    ERC20(BIT_TOKEN_ADDRESS).safeTransferFrom(msg.sender, address(this),
    _amount);

    // transfer MNT tokens to user, if there are insufficient tokens, in
    the contract this will revert
    ERC20(MNT_TOKEN_ADDRESS).safeTransfer(msg.sender, amountToSwap);

    emit TokensMigrated(msg.sender, _amount, amountToSwap);
}
```

Impact

A user might receive less tokens than they expect.

Recommendations

The values `TOKEN_CONVERSION_NUMERATOR` and `TOKEN_CONVERSION_DENOMINATOR` should be carefully set so the above calculations do not affect the tokens transferred to a user.

Remediation

This issue has been acknowledged by Mantle Network, and a fix was implemented in commit eced8f7f.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Possible centralization issue

The `sweepTokens` function gives the owner the ability to transfer any amount of tokens (except BIT and MNT) to any external address.

```
function sweepTokens(address _tokenAddress, address _recipient,
    uint256 _amount) public onlyOwner {
    // we can only sweep tokens that are not BIT or MNT to an arbitrary
    addres
    if ((_tokenAddress == address(BIT_TOKEN_ADDRESS)) || (_tokenAddress
    == address(MNT_TOKEN_ADDRESS))) {
        revert MantleTokenMigrator_SweepNotAllowed(_tokenAddress);
    }
    ERC20(_tokenAddress).safeTransfer(_recipient, _amount);

    emit TokensSwept(_tokenAddress, _recipient, _amount);
}
```

Although this is done to rescue tokens that are accidently sent to this contract, there is a possible centralization risk.

4.2 Two-step ownership transfer for critical roles

In the MantleTokenMigrator contract, a one-step ownership transfer is in place. This could be a security risk in the case a new owner is accidentally set to the wrong address. In such a scenario, the owner would never be able to be recovered and could render the `onlyOwner` functions in the contract dysfunctional.

A two-step ownership transfer is recommended for critical admin roles such as `owner`. A two-step ownership transfer entails the new admin having to claim their role first before the ownership transfer is complete. This mitigates the scenario in which a wrong address is supplied.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: L1MantleToken.sol

Function: `initialize(uint256 _initialSupply, address _owner)`

Initializes the contract.

Inputs

- `_initialSupply`
  - **Control**: Full.
  - **Constraints**: Needs to be something other than `0`.
  - **Impact**: Discarded.
- `_owner`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: Discarded.

Branches and code coverage (including function calls)

Intended branches

- Successfully initializes if all conditions are met.
  ☑ Test coverage

Negative behavior

- Reverts if `_initialSupply` is equal to `0` or the owner is not `_owner`.
  ☑ Negative test

Function: `mint(address _recipient, uint256 _amount)`

Mints new tokens at the amount `_amount` and assigns them to address `_recipient`.

Inputs

- `_recipient`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: Discarded.
- `_amount`
  - **Control**: Full.
  - **Constraints**: Needs to be less than `maximumMintAmount`.
  - **Impact**: Allows `_amount` number of tokens to be minted.

Branches and code coverage (including function calls)

Intended branches

- Successfully mints if all conditions are met.
  - ☑ Test coverage

Negative behavior

- Reverts if `_amount` is greater than `maximumMintAmount`.
  - ☑ Negative test
- Reverts if `block.timestamp` is less than `nextMint`.
  - ☑ Negative test
- Reverts if the caller is now the owner.
  - ☑ Negative test

Function call analysis

- `mint → totalSupply()`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setMintCapNumerator(uint256 _mintCapNumerator)`

Sets the numerator for the mint cap.

Inputs

- `_mintCapNumerator`
  - **Control**: Full.
  - **Constraints**: Needs to be less than `MINT_CAP_MAX_NUMERATOR`.
  - **Impact**: New mint cap numerator to be set.

Branches and code coverage (including function calls)

Intended branches

- Successfully sets the new mint cap numerator if all conditions are met.
  ☑ Test coverage

Negative behavior

- Reverts if the newly set `_mintCapNumerator` is greater than `MINT_CAP_MAX_NUMERATOR`.
  ☑ Negative test
- Reverts if the caller is now the owner.
  ☑ Negative test

5.2 Module: L1StandardBridge.sol

Function: `depositERC20To(address _l1Token, address _l2Token, address _to, uint256 _amount, uint32 _l2Gas, byte[] _data)`

Initiates deposit to address `_l2Token`.

Inputs

- `_l1Token`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Address of the L1 ERC-20 being deposited.
- `_l2Token`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Address of the L1 respective L2 ERC-20.
- `_to`
  - **Control**: User.
  - **Constraints**: Discarded.

    - **Impact**: Account to give the deposit to on L2.
- `_amount`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Amount of the ERC-20 to deposit.
- `_l2Gas`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Gas limit required to complete the deposit on L2.
- `_data`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Optional data to forward to L2.

Branches and code coverage (including function calls)

Intended branches

- Successfully initiates deposit if all conditions are met.
    ☑ Test coverage

Negative behavior

- Checked.
    ☑ Negative test

Function: `depositERC20(address _l1Token, address _l2Token, uint256 _amount, uint32 _l2Gas, byte[] _data)`

Initiates deposit.

Inputs

- `_l1Token`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Address of the L1 ERC-20 being deposited.
- `_l2Token`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Address of the L1 respective L2 ERC-20.
- `_amount`

    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Amount of the ERC-20 to deposit.
- `_l2Gas`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Gas limit required to complete the deposit on L2.
- `_data`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Optional data to forward to L2.

Branches and code coverage (including function calls)

Intended branches

- Successfully deposits ETH if all the conditions are met.
  - [x] Test coverage

Negative behavior

- Reverts if the caller does not have enough `_amount`.
  - [x] Negative test

Function: `depositETHTo(address _to, uint32 _l2Gas, byte[] _data)`

Deposits ETH to address `_to`.

Inputs

- `_to`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Account to give the deposit to on L2.
- `_l2Gas`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Gas limit required to complete the deposit on L2.
- `_data`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Optional data to forward to L2.

Branches and code coverage (including function calls)

Intended branches

- Successfully deposits ETH if all the conditions are met.
  - [x] Test coverage

Function: `depositETH(uint32 _l2Gas, byte[] _data)`

Deposits ETH.

Inputs

- `_l2Gas`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Gas limit required to complete the deposit on L2.
- `_data`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Optional data to forward to L2.

Branches and code coverage (including function calls)

Intended branches

- Successfully deposits ETH if all conditions are met.
  - [x] Test coverage

Negative behavior

- Reverts if the caller is not EOA.
  - [x] Negative test

Function: `finalizeERC20Withdrawal(address _l1Token, address _l2Token, address _from, address _to, uint256 _amount, byte[] _data)`

Finalizes withdrawal of an ERC-20 token from L2 to L1.

Inputs

- `_l1Token`
  - **Control**: User.
  - **Constraints**: Discarded.

  - **Impact**: Determines which token to be withdrawn.
- `_l2Token`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Address of the L1 respective L2 ERC-20.
- `_from`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The address from which the tokens are transferred.
- `_to`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The destination address for the transferred tokens.
- `_amount`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Amount of the ERC-20 to withdraw.
- `_data`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Optional data.

Branches and code coverage (including function calls)

Intended branches

- Successfully withdraws if all conditions are met.
  ☑ Test coverage

Negative behavior

- Reverts if called from a non-`crossDomainMessenger` L1 account.
  ☑ Negative test
- Reverts if called from the right `crossDomainMessenger` but wrong `xDomainMessageSender`.
  ☑ Negative test

Function call analysis

- `finalizeERC20Withdrawal` → `IERC20(_l1Token).safeTransfer(_to, _amount)`
  - **What is controllable?** `_l1Token_`, `_to`, and `_amount`.

    - **If return value controllable, how is it used and how can it go wrong?** Discarded.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `finalizeETHWithdrawal(address _from, address _to, uint256 _amount, byte[] _data)`

Finalizes withdrawal of ETH from L2 to L1.

Inputs

- `_from`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: The address from which the tokens are transferred.
- `_to`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: The destination address for the transferred tokens.
- `_amount`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Amount of the ERC-20 to withdraw.
- `_data`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Optional data.

Branches and code coverage (including function calls)

Intended branches

- Successfully withdraws if all conditions are met.
    - [x] Test coverage

Negative behavior

- Reverts if called from a non-`crossDomainMessenger` L1 account.
    - [x] Negative test
- Reverts if called from the right `crossDomainMessenger` but wrong `xDomainMessageSender`.

☑ Negative test

Function: `finalizeMantleWithdrawal(address _from, address _to, uint256 _amount, byte[] _data)`

Finalizes withdrawal of Mantle tokens from L2 to L1.

Inputs

- `_from`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The address from which withdrawal is conducted.
- `_to`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The address to which withdrawal is conducted.
- `_amount`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The amount of tokens to be withdrawn.
- `_data`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Data associated with the transaction.

Branches and code coverage (including function calls)

Intended branches

- Successfully finalizes withdrawal if all conditions are met.
  ☑ Test coverage

Function call analysis

- `finalizeMantleWithdrawal` → `finalizeERC20Withdrawal(l1MantleAddress, Lib_PredeployAddresses.BVM_MANTLE, _from, _to, _amount, _data)`
  - **What is controllable?** `_from_`, `_to`, `_amount`, and `_data`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Discarded.

Function: `initialize(address _l1messenger, address _l2TokenBridge, address _l1MantleAddress)`

Initializes contract.

Inputs

- `_l1messenger`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: L1 messenger address being used for cross-chain communications.
- `_l2TokenBridge`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: L2 standard bridge address.
- `_l1MantleAddress`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Initialize L1 mantle address.

Branches and code coverage (including function calls)

Intended branches

- Successfully initializes if all other conditions are met, such as the addresses being valid.
  - [x] Test coverage

Negative behavior

- Reverts if it is already called once.
  - [x] Negative test

Function: `receive()`

Receive function.

Branches and code coverage (including function calls)

Intended branches

- Successfully deposits an amount of ETH to the caller's balance on L2.
  - ☑ Test coverage

5.3 Module: L2StandardBridge.sol

Function: `finalizeDeposit(address _l1Token, address _l2Token, address _from, address _to, uint256 _amount, byte[] _data)`

Finalizes deposit of tokens from L1 to L2.

Inputs

- `_l1Token`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The L1 token.
- `_l2Token`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The L2 token.
- `_from`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The address from which deposit takes place.
- `_to`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: The address to which deposit takes place.
- `_amount`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Amount of tokens.
- `_data`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Optional data.

Branches and code coverage (including function calls)

Intended branches

- Successfully finalizes deposit if all conditions are met.
  - ☑ Test coverage

Negative behavior

- Reverts on calls from a non-`crossDomainMessenger` L2 account.
  - ☑ Negative test

Function call analysis

- `finalizeDeposit → ERC165Checker.supportsInterface(_l2Token, 0x1d1d8b63)`
  - **What is controllable?** `_l2Token`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `finalizeDeposit → IL2StandardERC20(_l2Token).l1Token()`
  - **What is controllable?** `_l2Token`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `finalizeDeposit → IL2StandardERC20(_l2Token).mint(_to, _amount)`
  - **What is controllable?** `_l2Token`, `_to`, and `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `finalizeDeposit → sendCrossDomainMessage(l1TokenBridge, 0, message)`
  - **What is controllable?** `l1TokenBridge`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `withdrawTo(address _l2Token, address _to, uint256 _amount, uint32 _l1Gas, byte[] _data)`

Initiates a withdrawal of `_amount` of `_l2Token` tokens to address `_to`.

Inputs

- `_l2Token`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Determines the token that the user wants to withdraw from L2 to L1.
- `_to`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Destination address.
- `_amount`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Amount of tokens to withdraw.
- `_l1Gas`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Amount of gas for execution of the message.
- `_data`
  - **Control**: User.
  - **Constraints**: Discarded.
  - **Impact**: Optional data.

Branches and code coverage (including function calls)

Intended branches

- Successfully withdraws if all conditions are met.
  ☑ Test coverage

Function: `withdraw(address _l2Token, uint256 _amount, uint32 _l1Gas, byte[] _data)`

Initiates withdrawal.

Inputs

- `_l2Token`
  - **Control**: User.
  - **Constraints**: Discarded.

    - **Impact**: Determines the token that the user wants to withdraw from L2 to L1.
- `_amount`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Amount of tokens to withdraw.
- `_l1Gas`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Amount of gas for execution of the message.
- `_data`
    - **Control**: User.
    - **Constraints**: Discarded.
    - **Impact**: Optional data.

Branches and code coverage (including function calls)

Intended branches

- Successfuly withdraws if all conditions are met.
    ☑ Test coverage

5.4 Module: MantleTokenMigrator.sol

Function: `defundContract(address _tokenAddress, uint256 _amount)`

Defunds the contract.

Inputs

- `_tokenAddress`
    - **Control**: Full.
    - **Constraints**: Needs to be either `BIT_TOKEN_ADDRESS` or `MNT_TOKEN_ADDRESS`.
    - **Impact**: The valid address of the token to be defunded.
- `_amount`
    - **Control**: Full.
    - **Constraints**: Needs to be less than or equal to the full amount available in the contract.
    - **Impact**: The amount to be defunded.

Branches and code coverage (including function calls)

Intended branches

- Successfully defunds the contract if all conditions are met.
  ☑ Test coverage

Negative behavior

- Reverts if the contract is not sufficiently funded as `_amount`.
  ☑ Negative test
- Reverts if a nonowner address attempts to fund the contract.
  ☑ Negative test
- Reverts if the contract attempts to defund invalid tokens.
  ☑ Negative test

Function call analysis

- `defundContract` → `ERC20(_tokenAddress).safeTransfer(treasury, _amount)`
  - **What is controllable?** `_tokenAddress` and `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `fallback()`

Handles nonvalid calldata.

Branches and code coverage (including function calls)

Intended branches

- Successfully reverts the transaction.
  ☑ Test coverage

Negative behavior

- Reverts when `msg.data` is not equal to `0`.
  ☑ Negative test

Function: `haltContract()`

Halts the contract.

Branches and code coverage (including function calls)

Intended branches

- Successfully halts the contract if called by the owner.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `migrateAllBIT()`

Swaps all of the caller's BIT tokens for MNT tokens.

Branches and code coverage (including function calls)

Intended branches

- Successfully migrates all of caller's BIT tokens if the contract is not halted and the other conditions are met.
  - ☑ Test coverage

Negative behavior

- Reverts if the contract is not sufficiently funded.
  - ☑ Negative test
- Reverts if the caller does not have enough BIT balance.
  - ☑ Negative test
- Reverts if the contract is halted.
  - ☐ Negative test

Function call analysis

- `migrateAllBIT` → `ERC20(BIT_TOKEN_ADDRESS).balanceOf(msg.sender);`
  - **What is controllable?** The BIT balance held by the caller.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `migrateBIT(uint256 _amount)`

Swaps caller's BIT tokens for MNT tokens.

Inputs

- `_amount`
  - **Control**: Full.
  - **Constraints**: Caller must be at least sufficiently funded with respect to `_amount`.
  - **Impact**: Amount of tokens to be migrated.

Branches and code coverage (including function calls)

Intended branches

- Successfuly swaps if all conditions are met.
  ☑ Test coverage

Negative behavior

- Reverts when the contract is not funded.
  ☑ Negative test
- Reverts when the caller is not funded.
  ☑ Negative test
- Reverts when the contract is halted.
  ☐ Negative test

Function: `receive()`

Receive function.

Branches and code coverage (including function calls)

Intended branches

- Successfully reverts if ETH is sent to the contract with a call.
  ☑ Test coverage

Negative behavior

- Reverts when Ether is sent to the contract.
  ☑ Negative test

Function: `setTreasury(address _treasury)`

Sets the treasury address.

Inputs

- `_treasury`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: Sets the treasury address.

Branches and code coverage (including function calls)

Intended branches

- Successfully sets the treasury address by the owner.
  ☑ Test coverage

Negative behavior

- Reverts if called by a nonowner.
  ☑ Negative test

Function: `sweepTokens(address _tokenAddress, address _recipient, uint256 _amount)`

Sweeps an amount of tokens to a `_recipient` address.

Inputs

- `_tokenAddress`
  - **Control**: Full.
  - **Constraints**: Needs to be either `BIT_TOKEN_ADDRESS` or `MNT_TOKEN_ADDRESS`.
  - **Impact**: Address of tokens to be sweeped.
- `_recipient`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: Discarded.
- `_amount`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: Amount of tokens to be sweeped.

Branches and code coverage (including function calls)

Intended branches

- Successfully sweeps tokens if called by the owner.

☑ Test coverage

Negative behavior

- Reverts if not called by the owner.
  ☑ Negative test

Function call analysis

- `sweepTokens` → `ERC20(_tokenAddress).safeTransfer(_recipient, _amount)`
  - **What is controllable?** `_tokenAddress`, `_recipient`, and `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `transferOwnership(address _newOwner)`

Transfers ownership of the contract to `_newOwner`.

Inputs

- `_newOwner`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: The address of the new owner.

Branches and code coverage (including function calls)

Intended branches

- Successfully transfers ownership if called by the owner.
  ☑ Test coverage

Negative behavior

- Reverts if not called by the owner.
  ☑ Negative test

Function: `unhaltContract()`

Unhalts the contract.

Branches and code coverage (including function calls)

Intended branches

- Successfully unhalts the contract if called by the owner.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

6 Audit Results

At the time of our audit, the audited code was not deployed to mainnet EVM.

During our assessment on the scoped Mantle contracts, we discovered four findings. No critical issues were found. One was of high impact, one was of low impact, and the remaining findings were informational in nature. Mantle Network acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.