Zellic

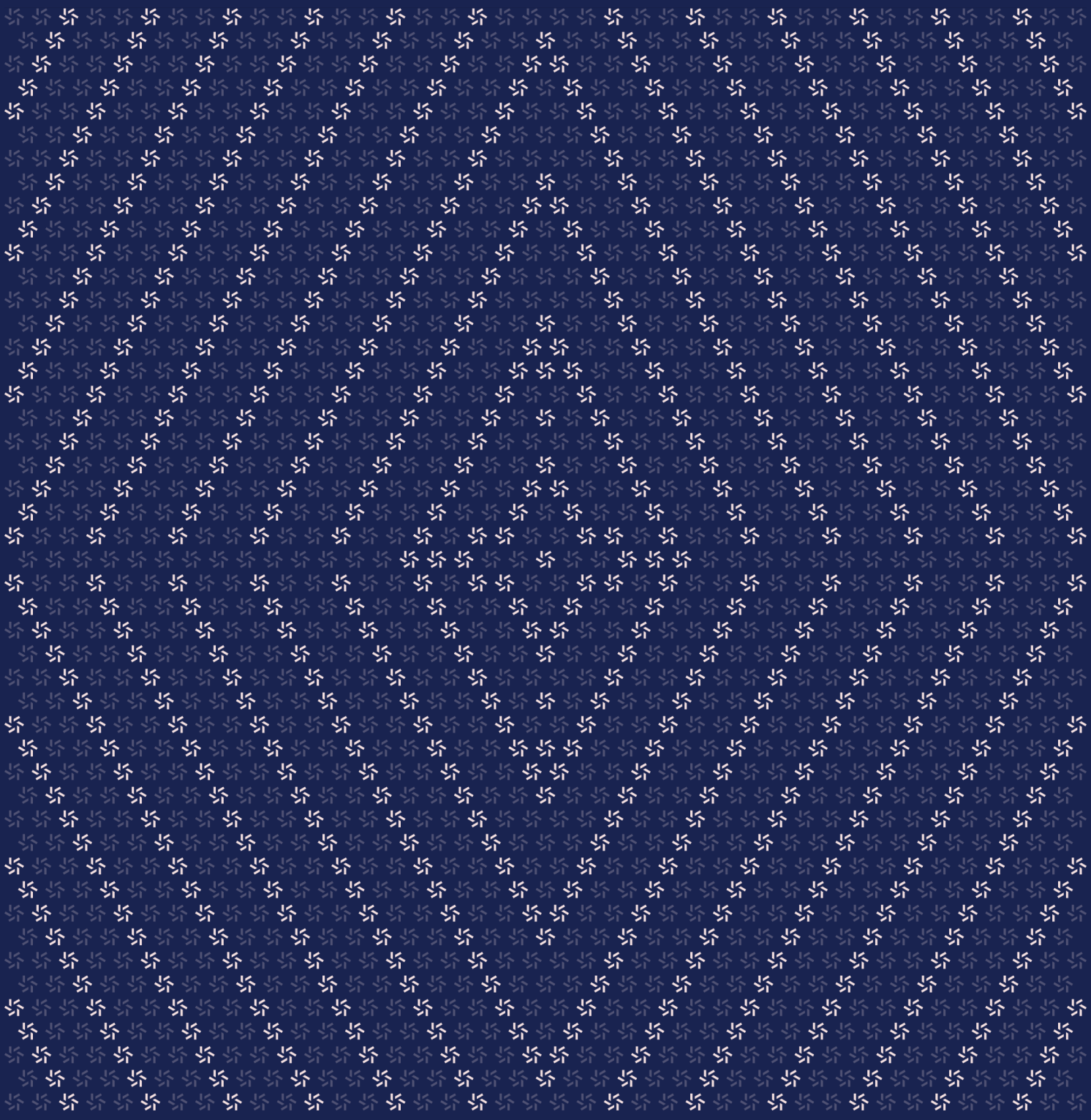
October 30, 2024

Prepared for
David Lin
Eclipse

Prepared by
Chongyu Lv
Doyeon Park
Sylvain Pelissier
Vlad Toie
Zellic

Eclipse Withdraw

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Eclipse Withdraw 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Relayer might become insolvent due to small withdraw authorizations 11

3.2. Collision risk on withdraw message ID 13

3.3. Missing role attribution during deployment 15

4. Discussion **17**

4.1. Centralization risks 18

4.2. Lack of documentation 19

4.3. Use a more recent Solidity version 19

4.4. Use calldata instead of memory for function parameters 20

5. Threat Model 20

5.1. Module: CanonicalBridge.sol 21

6. Assessment Results 24

6.1. Disclaimer 25

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Eclipse from October 22nd to October 25th, 2024. During this engagement, Zellic reviewed Eclipse Withdraw's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- How does Eclipse handle cross-chain messages?
- What are the attack vectors with regards to the cross-chain messaging system?
- Who are the actors involved in the approval, forwarding, and execution of cross-chain messages?

1.3. Non-goals and Limitations

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

- Eclipse SVM L2 bridge counterpart
- Front-end components
- Infrastructure relating to the project
- Key custody

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

1.4. Results

During our assessment on the scoped Eclipse Withdraw contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Eclipse in the Discussion section (4. ↗).

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Eclipse Withdraw

Eclipse contributed the following description of Eclipse Withdraw:

> **Treasury:** Acts as a secure vault for ether deposits, with controlled deposit and withdrawal mechanisms.
>
> **CanonicalBridge:** Handles the deposit and withdrawal of ether, interfacing directly with end users. It's immutable, providing a fixed and auditable interface. Can only be replaced via role grants in the Treasury.
>
> **Upgrader0to1:** Facilitates the upgrade process from v0 to v1 contracts, managing role assignments and contract interactions.

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 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 codebase 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, upgradability

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Eclipse Withdraw Contracts

Type	Solidity
Platform	EVM-compatible
Target	syzygy
Repository	https://github.com/Eclipse-Laboratories-Inc/syzygy ↗
Version	`ea3e53c1067b42f1c1cf1e4d1258f4c3a9a470b0`

Programs	`Treasury`
	`CanonicalBridge`
	`UpgraderOto1`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.1 person-weeks. The assessment was conducted by four consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Chongyu Lv
Engineer
chongyu@zellic.io

Doyeon Park
Engineer
doyeon@zellic.io

Sylvain Pelissier
Engineer
sylvain@zellic.io

Vlad Toie
Engineer
vlad@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 22, 2024	Start of primary review period
October 25, 2024	End of primary review period

3. Detailed Findings

3.1. Relayer might become insolvent due to small withdraw authorizations

Target	CanonicalBridge		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The Relayer is an off-chain entity that is responsible for submitting withdrawal requests to the bridge contract. The bridge contract has a function called `authorizeWithdraws` that allows the relayer to authorize multiple withdrawal requests at once. The `authorizeWithdraws` function is called by the relayer, and it iterates over the withdrawal messages and calls the `_authorizeWithdraw` function for each message.

```
function authorizeWithdraws(
    WithdrawMessage[] calldata messages
)
    external
    override
    whenNotPaused
    onlyRole(WITHDRAW_AUTHORITY_ROLE)
{
    for (uint256 i = 0; i < messages.length; i++) {
        _authorizeWithdraw(messages[i]);
    }
}

function _authorizeWithdraw(
    WithdrawMessage memory message
)
    private
    validWithdrawMessage(message)
{
    bytes32 messageHash = withdrawMessageHash(message);
    uint256 messageStartTime = block.timestamp + fraudWindowDuration;
    /// @dev This would occur if the relayer passed the same message twice.
    if (withdrawMessageStatus(messageHash) != WithdrawStatus.UNKNOWN) {
        revert CanonicalBridgeTransactionRejected(message.withdrawId, "Message
    already exists");
    }
    /// @dev This would only occur if the same message Id was used for two
    different messages.
```

```
    if (withdrawMsgIdProcessed[message.withdrawId] != 0) {
        revert CanonicalBridgeTransactionRejected(message.withdrawId, "Message
    Id already exists");
    }
    startTime[messageHash] = messageStartTime;
    withdrawMsgIdProcessed[message.withdrawId] = block.number;

    emit WithdrawAuthorized(
        msg.sender,
        message,
        messageHash,
        messageStartTime
    );
}
```

A malicious user could potentially request small amounts to withdraw multiple times on the other side of the bridge, severely impacting the gas usage of the Relayer when calling `authorizeWithdraws` on Ethereum.

Impact

Due to the potential high gas usage of the `authorizeWithdraws` function, the relayer might become insolvent, temporarily unable to process any more withdrawals and DOSing the system.

Recommendations

We recommend adequately compensating the relayer for the gas costs incurred when recovering the fees for the cross-chain operation. This must ensure that the relayer is not at risk of becoming insolvent due to the gas costs incurred by the `authorizeWithdraws` function.

Remediation

This issue has been acknowledged by Eclipse, and a fix was implemented in commit `46d7612b` ↗. Additionally, the Eclipse team has stated that

> The flow and timing is adjusted so the relayer receives the fee immediately upon authorizing this withdrawal. Thus, the relayer remains solvent provided it charges, on average, sufficient fees to offset gas expenditures.

3.2. Collision risk on withdraw message ID

Target	CanonicalBridge		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In order to authorize a withdrawal with the `_authorizeWithdraw` function, the bridge takes as input a withdraw message and checks its validity. The hash of the message is computed then; based on this hash, it is checked if this message was already submitted or not:

```
function _authorizeWithdraw(WithdrawMessage memory message)
    private
    validWithdrawMessage(message)
{
    bytes32 messageHash = withdrawMessageHash(message);
    uint256 messageStartTime = block.timestamp + fraudWindowDuration;
    /// @dev This would occur if the relayer passed the same message twice.
    if (withdrawMessageStatus(messageHash) != WithdrawStatus.UNKNOWN) {
        revert CanonicalBridgeTransactionRejected(message.withdrawId, "Message
    already exists");
    }
    /// @dev This would only occur if the same message Id was used for two
    different messages.
    if (withdrawMsgIdProcessed[message.withdrawId] != 0) {
        revert CanonicalBridgeTransactionRejected(message.withdrawId, "Message
    Id already exists");
    }
    startTime[messageHash] = messageStartTime;
    withdrawMsgIdProcessed[message.withdrawId] = block.number;

    emit WithdrawAuthorized(
        msg.sender,
        message,
        messageHash,
        messageStartTime
    );
}
```

The message is composed of the following fields that are hashed:

```
struct WithdrawMessage {
    bytes32 from;
    address destination;
    uint256 amountWei;
    uint64 withdrawId;
    address feeReceiver;
    uint256 feeWei;
}
```

If the message hash was not used before, the `startTime` mapping is updated with the computed message start time. However, the withdraw message identifier `message.withdrawId` is also checked to not have been already used. In this case, `message.withdrawId` is only a 64-bit value. According to the `withdraw_txn` implementation in crates/eclipse_withdrawal_kit/src/main.rs, this value is randomly generated. According to the birthday paradox, there is a nonnegligible probability to obtain a collision, so the withdrawal would revert. For example, after 610,000,000 IDs have been used, there would be a more than 1% chance of rejecting a withdraw.

Impact

As soon as a large number of `withdrawId`s would be used, the withdraw would start to revert because of collisions. An attacker may submit many withdraw messages at the beginning of the contract creation in order to make the collision probability increase for future users.

Recommendations

The `withdrawMsgIdProcessed` function should rely on `messageHash` instead of `withdrawId`, which would greatly lower this collision probability. Another solution would be to use a larger ID — for example, 256 bits. In addition, in `validWithdrawMessage`, there is a check that the value of the ID is not zero. This check is not necessary if the value is randomly generated and increases the collision probability.

Remediation

This issue has been acknowledged by Eclipse. Eclipse informed us that the ids are generated by the out-of-scope Eclipse programs and are guaranteed to be unique in production.

3.3. Missing role attribution during deployment

Target	DeployV1.s		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

In the CanonicalBridge contract, the `authorizeWithdraw` function is called to schedule withdrawal information for the withdrawal message sent from Layer 2 to Layer 1. Once the message status becomes pending, anyone can call the `claimWithdrawal` function to claim it. When claimed, the `withdrawEth` function in the Treasury contract is invoked, which sends the assets to the address specified in the `destination` field.

```
function claimWithdrawal(
    WithdrawMessage calldata message
)
    external
    override
    whenNotPaused
    validWithdrawMessage(message)
{
    bytes32 messageHash = withdrawMessageHash(message);
    if (withdrawMessageStatus(messageHash) != WithdrawStatus.PENDING)
    revert WithdrawUnauthorized();

    // Transfer funds to recipient.
    bool success = ITreasury(TREASURY).withdrawEth(
        message.destination,
        message.amountWei,
        message.feeReceiver,
        message.feeWei
    );
    /// @dev The following condition should never occur and the error should be
    unreachable code.
    if (!success) revert WithdrawFailed();
    startTime[messageHash] = NEVER;
    emit WithdrawClaimed(message.destination, message.from, messageHash,
    message);
}
```

At this point, the Treasury contract ensures that the caller is authorized to invoke the `withdrawEth` function by verifying that they hold the `WITHDRAW_AUTHORITY_ROLE`. Therefore, the CanonicalBridge contract, which calls the `withdrawEth` function, must necessarily hold the `WITHDRAW_AUTHORITY_ROLE`. This role should be assigned in the DeployV1.s.sol script. However, this part was omitted.

```
// Grant the trustedRelayer the withdraw authority role
canonicalBridgeV1.grantRole(
    canonicalBridgeV1.WITHDRAW_AUTHORITY_ROLE(),
    trustedRelayerAddress
);

// Grant the canonicalBridge the depositor role
_treasuryV1 = Treasury(treasuryProxyAddress);
_treasuryV1.grantRole(_treasuryV1.DEPOSITOR_ROLE(),
    address(canonicalBridgeV1));

// Log deployed configuration
console.log("");
```

Impact

The impact of this code mistake could delay the bridge's operation. Users who have sent messages through the CanonicalBridge will attempt to claim by calling the `claimWithdrawal` function after their status becomes pending. However, since the CanonicalBridge was not granted the `WITHDRAW_AUTHORITY_ROLE` in the Treasury, the transaction will continuously revert. After repeated reverts, the team will identify the issue and then confirm an unnecessary grant transaction to fix the problem. This code mistake will cause a delay until the system operates normally again.

Recommendations

We recommend adding the grant operation to the DeployV1.s.sol code. The modification we recommend is as follows.

```
// Grant the canonicalBridge the depositor role
_treasuryV1 = Treasury(treasuryProxyAddress);
_treasuryV1.grantRole(_treasuryV1.DEPOSITOR_ROLE(),
    address(canonicalBridgeV1));
_treasuryV1.grantRole(_treasuryV1.WITHDRAW_AUTHORITY_ROLE(),
    address(canonicalBridgeV1));
```

Remediation

This issue has been acknowledged by Eclipse. Eclipse informed us that the `DeployV1.s` script was used for isolated testing only but not in production.

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. Centralization risks

There are nine types of privileged accounts for the contracts:

- The admin
- The upgrader
- The pauser
- The starter
- The withdraw authority
- The withdraw canceller
- The fraud window setter
- The depositor
- The emergency

The starter, pauser, and upgrader roles can respectively start, pause, or upgrade the contract according to the Universal Upgradeable Proxy Standard (UUPS) scheme but have no other special powers.

The emergency role is able to withdraw any amount of the Treasury contract at any time with the `emergencyWithdraw` function.

Similarly, the withdraw-authority role is able to withdraw any amount to any address when the Treasury contract is not paused.

The fraud-window role is able to change the `fraudWindowDuration` time, which may lead to a griefing attack. Greatly increasing the fraud-window duration just before a call to `authorizeWithdraw` would prevent a specific withdrawal. The duration can be set back right after to prevent affecting the other withdrawals.

The withdraw-canceller role is able to delete any withdraw message.

The admin is able to grant or revoke any other roles.

The above introduces centralization risks that users should be aware of, as it grants a single point of control over the system.

We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

4.2. Lack of documentation

There are certain areas in the code where some mechanisms are not documented. For example, the upgrade mechanism implemented in the Upgrader0to1 contract would benefit from documentation about the rationales behind the design choices and the interactions with other parts of the system.

The fee calculation implemented in the CanoncialBridge contract should also be documented to have a clear view of the process between the contract and the relayer. In addition, in the CanoncialBridge contract, there are some leftover unfinished NatSpec comments:

```
/// @notice Explain to an end user what this does
/// @dev Explain to a developer any extra details
/// @dev Ensures that bytes32 data is initialized with data.
modifier bytes32Initialized(bytes32 _data) {
    if(_data == NULL_BYTES32) revert EmptyBytes32();
    _;
}
```

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified later on. In general, a lack of documentation impedes the auditors' and external developers' ability to read, understand, and extend the code. The problem is also carried over if the code is ever forked or reused.

We recommend adding more comments to the code — especially comments that tie operations in code to locations in the documentation and brief comments to reaffirm developers' understanding.

4.3. Use a more recent Solidity version

The contracts specify a fixed Solidity version 0.8.21 to be used, which does not allow compilation with a more recent version of Solidity. Allowing the compilation with a version from 0.8.21 up to the latest release, or enforcing the use of the latest version, would prevent introducing bugs from the previous versions and may add the latest optimizations to the contract. In addition, the contracts seem fully functional with the latest version.

4.4. Use calldata instead of memory for function parameters

If a reference-type function parameter is read-only, it is cheaper in gas to use calldata instead of memory. Calldata is an unmodifiable, nonpersistent area where function arguments are stored, and it behaves mostly like memory.

Consider changing the parameters of the `withdrawMessageHash` function, `_authorizeWithdraw` function, and `validWithdrawMessage` modifier of the CanonicalBridge contract from `memory` to `calldata` to save gas.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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: CanonicalBridge.sol

Function: `deposit()`

This sends a deposit to be locked in the Treasury contract by calling the `depositEth` function.

Inputs

- `recipient`
  - **Validation**: The recipient address is checked to be nonzero by the `bytes32Initialized` modifier.
  - **Impact**: The recipient of the deposit.
- `amountWei`
  - **Validation**: The amount is checked with the `validDepositAmount` modifier to equal the message value, to be greater than the minimal amount, and to be divisible by `WEI_PER_LAMPORT`.
  - **Impact**: The amount paid matches the amount sent by the message, and the amount is valid.

Branches and code coverage (including function calls)

Intended branches

- A call to `deposit` with the correct amount and value does not revert.
  - ☑ Test coverage
- The treasury balance is updated according to the amount sent after a deposit.
  - ☐ Test coverage

Negative behavior

- The `deposit` function reverts when the contract is paused.
  - ☑ Negative test
- The `deposit` function reverts when the amount is different than the message value.
  - ☑ Negative test
- The `deposit` function reverts when the amount is less than minimum.
  - ☑ Negative test
- The `deposit` function reverts when the amount is not divisible by `WEI_PER_LAMPORT`.

- ☑ Negative test
- The `deposit` function reverts when the recipient is not set.
  - ☑ Negative test
- The `deposit` function reverts when the treasury deposit fails.
  - ☑ Negative test

Function: `authorizeWithdraws()`

The function initiates the withdraw process based on information of the withdraw message passed as a parameter. If the message was not used before and is valid, the withdraw time is programmed to be after the `fraudWindowDuration`.

The same analysis applies to the function `authorizeWithdraw`, which takes only a single withdraw message as input.

Inputs

- `messages`
  - **Validation**: The messages are validated with the `validWithdrawMessage` modifier during the call to `_authorizeWithdraw`.
  - **Impact**: Prevents using zero addresses and the amount being zero.

Branches and code coverage (including function calls)

Intended branches

- The `authorizeWithdraws` function executes when the message is valid.
  - ☑ Test coverage

Negative behavior

- The `authorizeWithdraws` function reverts when the contract is paused.
  - ☑ Negative test
- The `authorizeWithdraws` function reverts when the caller does not have the `WITHDRAW_AUTHORITY_ROLE` role.
  - ☑ Negative test
- The `authorizeWithdraws` function reverts when the source or the destination is zero.
  - ☑ Negative test
- The `authorizeWithdraws` function reverts when the amount is zero.
  - ☑ Negative test
- The `authorizeWithdraws` function reverts when the withdrawing was already authorized.
  - ☑ Negative test

Function call analysis

- `authorizeWithdraws -> _authorizeWithdraw(messages[i])`
  - **External/Internal?** Internal.
  - **Argument control?** `messages`.
  - **Impact**: Verify message validity and set the withdrawing time.

Function: `claimWithdrawal()`

Based on the information stored in the `message`, the `withdrawEth` function is called from the Treasury contract to proceed with the withdrawal.

Inputs

- `messages`
  - **Validation**: The `withdrawMessageStatus` function checks if the `message`, hashed by the `withdrawMessageHash` function, is currently in a pending state.
  - **Impact**: Proceeds with the withdrawal based on the provided `message` information.

Branches and code coverage (including function calls)

Intended branches

- The `claimWithdrawal` function successfully logs the `WithdrawClaimed` event when the withdrawal is completed successfully.
  - ☑ Test coverage
- The `claimWithdrawal` function is successfully called on a message that failed to be deleted by the `deleteWithdrawMessage` function.
  - ☑ Test coverage
- The `claimWithdrawal` function is successfully called when the `WithdrawStatus` is in the `PENDING` state.
  - ☑ Test coverage
- The treasury balance is updated according to the amount sent after a claim.
  - ☐ Test coverage

Negative behavior

- The `claimWithdrawal` function reverts when the contract is paused.
  - ☑ Negative test
- The `claimWithdrawal` function reverts if `authorizeWithdraw` has not been called beforehand.
  - ☑ Negative test
- The `claimWithdrawal` function reverts if the withdrawal fails in the Treasury contract.

☑ Negative test

- The `claimWithdrawal` function reverts if called on a message that has been deleted by the `deleteWithdrawMessage` function.
  ☑ Negative test
- The `claimWithdrawal` function reverts if called when the `WithdrawStatus` is in the `UNKNOWN` state, even after the `fraudWindowDuration` has elapsed.
  ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Ethereum Mainnet.

During our assessment on the scoped Eclipse Withdraw contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and one was of low impact.

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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.