Zellic

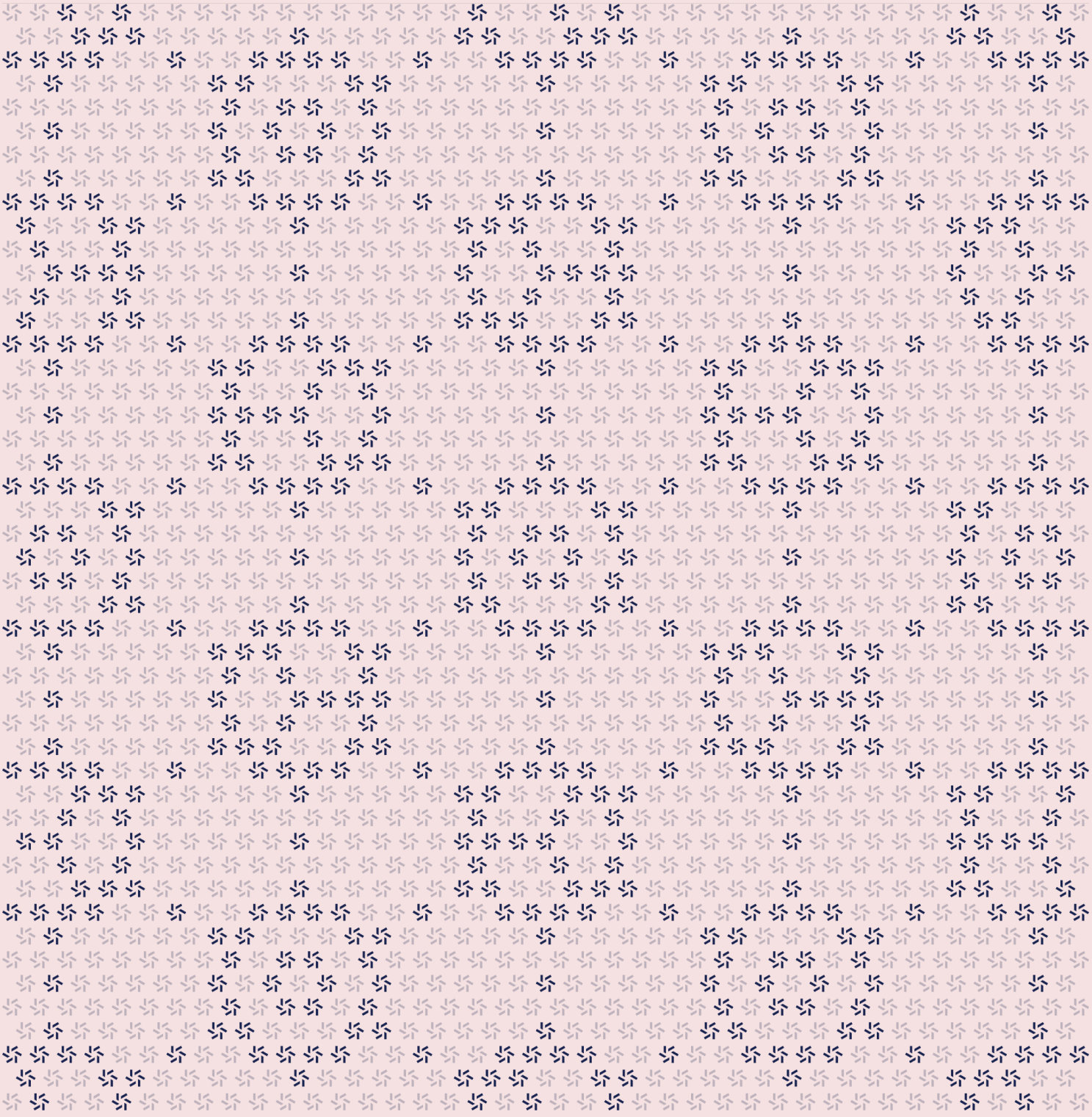
Prepared for
David Lin
Eclipse Laboratories

Prepared by
Jasraj Bedi
Katerina Belotskaia
Aaron Esau
Zellic

March 13, 2024

Eclipse

Smart Contract Security Assessment



Contents

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 Laboratories from March 6th to March 13th, 2024. During this engagement, Zellic reviewed Eclipse'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:

- Have the reentrancy guards, cross-contract interactions, and ETH transfers between contracts been properly implemented?
- Do only authorized callers have the capability to withdraw ETH?
- Is the checks-effects-interactions (CEI) pattern correctly implemented? Are there any missing/invalid checks, incorrect or missing state transitions?
- Is ETH securely stored in the Treasury contract and in the specified amount?

1.3. Non-goals and Limitations

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

- The relayer component responsible for sending and receiving messages.
- 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 contracts, we discovered six findings. Two 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 Eclipse Laboratories's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Eclipse

Eclipse Laboratories contributed the following description of Eclipse:

> The contracts included in this audit implement our canonical bridge which allows users to bridge ETH on ETH mainnet.

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 Contracts

Repositories	https://github.com/Eclipse-Laboratories-Inc/syzygy ↗ https://github.com/Eclipse-Laboratories-Inc/eclipse-programs ↗
Versions	syzygy: `2718982b807249a2c12d406d7e892f5c997d2dab` eclipse-programs: `e2de17da6cf7fd5f36bc2f33fc9fc1682f760ef0`
Programs	• Mailbox • EtherBridge • Router • Registry • Treasury • crates/solana_programs/programs/canonical_bridge/src/*.rs
Types	Solidity, Rust
Platforms	EVM-compatible, Eclipse

2.4. Project Overview

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

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:

Jasraj Bedi
Co-Founder
jazzy@zellic.io ↗

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Aaron Esau
Engineer
aaron@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 6, 2024	Start of primary review period
March 13, 2024	End of primary review period

3. Detailed Findings

3.1. The deposited funds are locked in wrong contract

Target	EtherBridge		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

It is assumed that the `deposit` function of the EtherBridge contract should lock all deposited ETH without fee in the Treasury contract and then route a deposit message out. But instead of sending the `amount` deposited ETH, the function sends a `message.amount`, which is equal to the `amount / WEI_PER_GWEI` where `WEI_PER_GWEI` is `1_000_000_000`.

```
function deposit(bytes32 recipient, uint256 amount, uint256 fee)
    // [...]
{
    uint256 lamportAmount = amount / WEI_PER_GWEI;
    uint256 feeAmount = fee / WEI_PER_GWEI;
    IEtherBridge.DepositMessage memory message = IEtherBridge.DepositMessage({
        sender: msg.sender,
        recipient: recipient,
        amount: lamportAmount,
        fee: feeAmount
    });
    // [...]
    // Lock ether by sending it to treasury
! (bool success,) = payable(address(treasury)).call{value:
    message.amount}("");
    // [...]
}
```

Impact

Most of the ETH funds will be locked in the EtherBridge contract without the possibility of withdrawal.

Recommendations

We recommend changing the `message.amount` to `amount`.

Remediation

This issue has been acknowledged by Eclipse Laboratories, and a fix was implemented in commit a7bd9820 ↗.

3.2. Lack of relayer validation on withdrawal

Target	CanonicalBridge		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The withdraw instruction on the bridge allows users to withdraw their SOL from the Eclipse side to ETH on the Ethereum side. The way this instruction works is the caller provides the following accounts:

- `withdrawer`, which signs the transaction and is the account providing the tokens to withdraw
- `withdrawal_receipt`, which is a program-derived address (PDA) to store the withdrawal information
- `relayer`, which is the relayer to send the token to
- `rent`, which is the rent program
- `system_program`, which is the system program

This instruction works by transferring the amount of tokens sent in the instruction from the `withdrawer` account to the `relayer` account. This information is recorded and saved in the `withdrawal_receipt` account. Lastly, a message is emitted from the Eclipse side so the relayer program can initiate the transfer on the Ethereum side.

A critical check that must be included here is that the `relayer` is the expected account. Otherwise, attackers could send tokens back and forth to accounts they can control while completely draining the relayer on the Ethereum side of the exchange.

It appears that, unlike the deposit instruction, no such check is present on the withdraw instruction.

Impact

An attacker could completely drain the relayer of ETH for almost zero cost.

Recommendations

Add an anchor constraint to enforce that the relayer is the expected account address.

Remediation

This issue has been acknowledged by Eclipse Laboratories.

3.3. The `message.amount` is not converted from Lamport to Wei

Target	EtherBridge		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The withdrawal message sent from the Eclipse allows users to withdraw their ETH on the Ethereum side. This message includes the `recipient` address and the `amount` to be withdrawn. However, there is an issue: the amount specified in the message is expressed in Lamports, not in Wei.

```
function withdraw(WithdrawMessage calldata message)
    external
    override
    onlyRouter
    correctWithdrawValue(message.amount)
    nonReentrant
{
    require(address(treasury).balance >= message.amount, "Insufficient funds
    in treasury for withdrawal");
    treasury.withdraw(payable(message.recipient), message.amount);
    [...]
}
```

Impact

The customer documentation specifies that the deposited amount on Ethereum should equate 1 Gwei to 1 Lamport on the Eclipse and vice versa. Therefore, as result of a lack of conversion from Lamports to Wei, the user will receive only 1/1,000,000,000 of the intended funds in Wei during the withdraw operation.

Recommendations

Multiply the `message.amount` value by `WEI_PER_GWEI = 1_000_000_000`.

Remediation

This issue has been acknowledged by Eclipse Laboratories, and a fix was implemented in commit a7bd9820 ↗.

3.4. Improve test suite

Target	Multiple contracts		
Category	Code Maturity	**Severity**	Low
Likelihood	N/A	**Impact**	Informational

Description

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, not just surface-level functions. It is important to test cross-chain function calls and transfers is recommended to ensure the desired functionality.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Impact

Because correctness is so critically important when developing smart contracts, we recommend that all projects strive for 100% code coverage. Testing should be an essential part of the software development lifecycle. No matter how simple a function may be, untested code is always prone to

bugs.

Recommendations

Several of the findings in this audit would have been discovered with simple test cases. Therefore, we strongly recommend to implement test cases for all the modules and especially for cross-chain functionality.

Remediation

This issue has been acknowledged by Eclipse Laboratories.

3.5. Lack of input validation

Target	Mailbox, EtherBridge		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The following functions lack input validation.

- The `initialize` function lacks a check that the `_relayer` is not zero address.
- The `mailOutgoingMessage` function lacks a check that `to_` is not zero address.
- The `EtherBridge.withdraw` function lacks a check that `message.recipient` is not zero address.

Impact

If important input parameters are not checked, it can result in functionality issues and unnecessary gas usage and can even be the root cause of critical problems. It is crucial to properly validate input parameters to ensure the correct execution of a function and to prevent unintended consequences.

Recommendations

Consider adding require statements and necessary checks to the above functions.

Remediation

This issue has been acknowledged by Eclipse Laboratories, and a fix was implemented in commit a7bd9820 ↗.

3.6. The `receiveMessage` function does not follow CEI pattern

Target	Mailbox		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `receiveMessage` does not follow the CEI pattern ↗:

```
/// @dev Receives and processes messages from other chains, ensuring they
    conform to the expected rollup chainId.
function receiveMessage(
    bytes memory eclipseChainId_,
    bytes memory sender_,
    bytes memory message_,
    bytes memory data_
)
    external
    payable
    override
    onlyRelayer
    validEclipseChainId(eclipseChainId_)
    validEncodedIdentifierLength(sender_)
    nonReentrant
    returns (bool)
{
    // [...]

    // Route the message
    router.routeIncomingMessage(sender_, message_);

    // Update the nonce
    receiveNonces[secParam.nonce] = true;

    // [...]
}
```

Impact

The function is not vulnerable to reentrancy because of the presence of the `nonReentrant` modifier. However, Eclipse Laboratories requested that we document any functions that do not follow the CEI pattern.

Recommendations

Move the effects above the function's interactions.

```
/// @dev Receives and processes messages from other chains, ensuring they
    conform to the expected rollup chainId.
function receiveMessage(
    bytes memory eclipseChainId_,
    bytes memory sender_,
    bytes memory message_,
    bytes memory data_
)
    // [...]
{
    // [...]
    EclipseSecurityParam memory secParam = abi.decode(data_,
    (EclipseSecurityParam));
    require(receiveNonces[secParam.nonce] == false, "Message with this nonce
    already received.");

    bytes32 hash = keccak256(abi.encodePacked(eclipseChainId_, sender_,
    message_, secParam.nonce));
    require(ECDSA.recover(hash, secParam.authoritySignature) == relayer,
    "Invalid relayer signature");

    // Update the nonce
    receiveNonces[secParam.nonce] = true;

    // Route the message
    router.routeIncomingMessage(sender_, message_);

    // Update the nonce
    receiveNonces[secParam.nonce] = true;

    // Emit the message received event
    emit MessageReceived(sender_, eclipseChainId_, message_);

    return true;
}
```

Remediation

This issue has been acknowledged by Eclipse Laboratories, and a fix was implemented in commit a7bd9820 ↗.

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. Redundant checks

Note the following checks, which are redundant or can be simplified:

1. The `deposit` function has `validDepositAmount` and `correctDepositValue` modifiers, which are not used anywhere else and both check that `amount <= MAX_DEPOSIT_AMOUNT`. This check can be removed from one of these modifiers.

```
modifier validDepositAmount(uint256 amount) {
    require(amount >= MIN_DEPOSIT_AMOUNT, "Deposit amount must be
    greater than equal to the minimum wei amount");
    require(amount <= MAX_DEPOSIT_AMOUNT, "Deposit amount must be
    less than equal to the max deposite amount");
    _;
}

modifier correctDepositValue(uint256 amount, uint256 fee) {
    require(msg.value == amount + fee, "Deposit amount and fee must
    match sent value");
    require(amount <= MAX_DEPOSIT_AMOUNT, "Amount cannot be more than
    the max deposit amount");
    _;
}
```

2. The `validSender` modifier is only used in the `deposit` function, which passes the `msg.sender` address to it. Therefore, this modifier only compares the `msg.sender` address with the `msg.sender` address and can be deleted.

```
modifier validSender(address sender) {
    require(msg.sender == sender, "The sender of the deposit does not
    match");
    _;
}
```

3. The `Mailbox:sendMessage()` function is only called from the `mailOutgoingMessage` function and can be changed to the internal function. Also, this function uses the `validEthereumChainId` modifier, which compares the input `ethereumChainId_` with global variable `ethereumChainId`, but the `mailOutgoingMessage` function passes this global `ethereumChainId` to the `sendMessage` function. Therefore, this modifier compares the `ethereumChainId` with the same `ethereumChainId` and can be deleted.

4. The `routeIncomingMessage()` function gets the contract address from the `registry.getAddress()` function and checks that the received address is not zero. But the `registry.getAddress()` function already has verification that the returned address is not zero and reverts in this case.

```
function routeIncomingMessage(bytes calldata encodedIdentifier,
    bytes calldata message)
    // [...]
{
    // [...]
    address actualAddress = registry.getAddress(identifier);
    require(actualAddress != address(0), "This identifier does not
    have a registered address");
    // [...]
}
```

```
function getAddress(string memory identifier)
    external view override returns (address) {
    address addr = identifierAddress[identifier];
    require(addr != address(0), "Identifier not registered");
    return addr;
}
```

4.2. Unused values in `EthereumSecurityParams`

Note that the `receipt_proof` and `proof` values in `EthereumSecurityParams` are currently unused in all bridge code. Consider removing it in the meantime, until functionality that uses it is implemented in the future.

4.3. Unnecessary data in the `addressContractType` mapping

In the `Registry:unregisterContract()` function, the value of `addressContractType` for `contractAddress` corresponding to the input `identifier` can be deleted for a gas reimbursement, since `addressContractType` cannot be used after deleting `addressIdentifier`.

4.4. Overprovisioned account size leads to excessive fees

In the `canonical_bridge` program on the Eclipse side, two accounts using PDAs are used: `DepositReceipt` and `WithdrawalReceipt`. It was noted that the `WithdrawalReceipt` account is overprovisioned on account size, requesting 1,024 bytes when it appears that less than 80 bytes would be used in the typical case. This is primarily a concern as the `withdrawer` has to pay the fee to create the rent-exempt account. Given the account is significantly overprovisioned, this incurs a greater-than-necessary fee per withdraw transaction.

It is noted that a comment was left on the `WithdrawalReceipt` account, noting a need to optimize the size.

4.5. Missing withdrawal amount validation

The withdraw instruction on the bridge allows users to withdraw their SOL from the Eclipse side to ETH on the Ethereum side. Once the transfer from the user account to relayer is complete, a message is emitted containing the amount of Lamports sent and the instruction terminates. On the Ethereum side, if the amount transmitted is greater than 1 ETH, a check rejects the transaction.

It was noted that the Eclipse program did not contain a check to ensure that the user was trying to transmit more than the maximally allowed 1 ETH which, depending on the behavior of the relayer, may result in lost or incomplete transfers.

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: EtherBridge.sol

Function: `deposit(bytes32 recipient, uint256 amount, uint256 fee)`

Allows to send a deposit message, provided Ether will be locked in the Treasury contract.

Inputs

- `recipient`
  - **Validation**: Should not be zero.
  - **Impact**: The recipient of the deposit message in the destination chain.
- `amount`
  - **Validation**: `amount + fee` should be equal to the `msg.value`.
  - **Impact**: The funds will be locked in the treasury before withdraw.
- `fee`
  - **Validation**: `amount + fee` should be equal to the `msg.value`.
  - **Impact**: Part of the fee will be provided for the `router.routeOutgoingMessage` execution and part will be locked in the current contract.

Branches and code coverage (including function calls)

Intended branches

- The expected Ether amount was locked in the Treasury contract.
  - ☐ Test coverage
- The deposit message has been sent properly.
  - ☐ Test coverage

Negative behavior

- `amount` is less than `MIN_DEPOSIT_AMOUNT`.
  - ☑ Test coverage
- Insufficient fee.
  - ☑ Test coverage
- `msg.value` less than `amount + fee`.
  - ☑ Test coverage

- `recipient` is zero.
  - ☑ Test coverage

Function call analysis

- `router.routeOutgoingMessage{gas: MIN_DEPOSIT_GAS_ONLY_ROUTING}(encodedMessage)`
  - **External/Internal?** External.
  - **Argument control?** `encodedMessage`.
  - **Impact**: Send deposit message to the router for further sending to the destination chain.
- `payable(address(treasury)).call{value: message.amount}("")`
  - **External/Internal?** External.
  - **Argument control?** `message.amount`.
  - **Impact**: Transfer deposit funds to the treasury to lock them until withdrawal.

Function: `withdraw(WithdrawMessage calldata message)`

Initiates the transfer of the specified amount of Ether from the Treasury contract to the recipient.

Inputs

- `message`
  - **Validation**: `amount` should be less than or equal to the `MAX_WITHDRAW_AMOUNT`.
  - **Impact**: Contains `recipient` address and `amount` Ether to transfer.

Branches and code coverage (including function calls)

Intended branches

- The expected Ether amount was sent to the recipient.
  - ☐ Test coverage

Negative behavior

- Caller is not an owner.
  - ☐ Test coverage
- `message.recipient` is zero.
  - ☐ Test coverage

Function call analysis

- `treasury.withdraw(payable(message.recipient), message.amount);`
  - **External/Internal?** External.

- **Argument control?** `message.recipient` and `message.amount`.
- **Impact**: Send the `message.amount` of Ether to the `message.recipient` address — the function will not revert if `message.recipient` is zero address.

5.2. Module: Mailbox.sol

Function: `receiveMessage(byte[] eclipseChainId_, byte[] sender_, byte[] message_, byte[] data_)`

Receives and processes messages from other chains, ensuring they conform to the expected rollup `eclipseChainId_`.

Inputs

- `eclipseChainId_`
  - **Control**: Full.
  - **Constraints**: Must be equal to the `eclipseChainId` of the contract.
  - **Impact**: The sending chain ID.
- `sender_`
  - **Control**: Full.
  - **Constraints**: Length must be less than or equal to 32 bytes. Must be registered in the registry. An external call with `message_` to the address must not revert.
  - **Impact**: The sender of the message.
- `message_`
  - **Control**: Full.
  - **Constraints**: Must be well-formed for the external call to the `sender_` address.
  - **Impact**: Arbitrary message to be passed to the router, used to make an external call.
- `data_`
  - **Control**: Full.
  - **Constraints**: Must decode as an `EclipseSecurityParam`. Inside the decoded struct, the `nonce` must not have been used before. The `authoritySignature` must be a valid signature from the relayer of `eclipseChainId_`, `sender_`, `message_`, and `nonce`.
  - **Impact**: Security parameters for the message.

Branches and code coverage

Intended branches

- Successfully routes the message.
  - ☑ Test coverage

Negative behavior

- Caller is not the relayer (“Only the relayer can call this method”).
  - ☑ Negative test
- `eclipseChainId_` is not equal to the `eclipseChainId` of the contract (“Unsupported Eclipse chain id”).
  - ☑ Negative test
- `sender_` length is not less than or equal to 32 bytes (“The encoded identifier must be less than equal to 32 bytes”).
  - ☐ Negative test
- Function in the Mailbox contract is already entered (see the `nonReentrant` modifier on `receiveMessage`).
  - ☐ Negative test
- Function in the Router contract is already entered (see the `nonReentrant` modifier on `routeIncomingMessage`).
  - ☐ Negative test
- Message with this nonce is already received.
  - ☐ Negative test
- Invalid relayer signature.
  - ☐ Negative test
- `sender_` is not registered in the registry (“This identifier does not have a registered address”).
  - ☐ Negative test
- Address in registry is not registered as a `IRegistry.ContractType.MODULE` (“This address is not a module”).
  - ☐ Negative test
- External call to `sender_` address reverts (“External call failed”).
  - ☐ Negative test

5.3. Module: Router.sol

Function: `routeIncomingMessage(bytes calldata encodedIdentifier, bytes calldata message)`

Forwards the message to the appropriate destination contract based on the provided `identifier`.

Inputs

- `encodedIdentifier`
  - **Validation**: The `registry` should contain a nonzero destination contract address for the provided `identifier`.
  - **Impact**: Contains `identifier` to the appropriate destination contract address, which will be called for delivery message.
- `message`

- **Validation**: N/A.
- **Impact**: The message for delivery, which contains the signature of the function that will be called and necessary arguments.

Branches and code coverage (including function calls)

Intended branches

- The function from the `message` was called correctly.
  - ☐ Test coverage

Negative behavior

- Caller is not the Mailbox contract.
  - ☑ Test coverage
- Registry does not have a contract address for the provided identifier.
  - ☑ Test coverage

Function call analysis

- `registry.getAddress(identifier)`
  - **External/Internal?** External.
  - **Argument control?** `identifier`.
  - **Impact**: Returns contract address for the provided `identifier`. This function will revert if `identifierAddress[identifier]` is zero.
- `actualAddress.call(message)`
  - **External/Internal?** External.
  - **Argument control?** `message`.
  - **Impact**: Calls arbitrary function of the destination contract.

Function: `routeOutgoingMessage(byte[] message)`

Routes an outgoing message the mailbox.

Inputs

- `message`
  - **Control**: Full.
  - **Constraints**: Must not be empty.
  - **Impact**: The message to route.

Branches and code coverage

Intended branches

- Successfully sends message.
  - ☑ Test coverage

Negative behavior

- Sender is not a registered module contract.
  - ☑ Negative test
- Function in the Router contract is already entered (see the `nonReentrant` modifier on `routeOutgoingMessage`).
  - ☐ Negative test
- Function in the Mailbox contract is already entered (see the `nonReentrant` modifier on `mailOutgoingMessage`).
  - ☐ Negative test
- The message is empty (“Message must be provided”).
  - ☑ Negative test

Function call analysis

- `registry.getIdentifier(msg.sender)`
  - **External/Internal?** External.
  - **Argument control?** N/A.
  - **Impact**: Returns the `identifier` for the `msg.sender`. This function will revert if `addressIdentifier[addr]` is zero. The `identifier` specifies the recipient of the message.
- `mailbox.mailOutgoingMessage(destination, message);`
  - **External/Internal?** External.
  - **Argument control?** `message`.
  - **Impact**: Routes a message to the Mailbox contract to be sent to the destination chain.

5.4. Module: Treasury.sol

Function: `withdraw(address payable to, uint256 amount)`

Allows owner of the contract to transfer an arbitrary amount of Ether to a specific address.

Inputs

- `to`
  - **Validation**: N/A.

- **Impact**: The receiver of Ether.
- `amount`
  - **Validation**: N/A.
  - **Impact**: The amount to transfer.

Branches and code coverage (including function calls)

Intended branches

- The Ether was successfully transferred to the recipient.
  - ☐ Test coverage

Negative behavior

- Caller is not the owner.
  - ☐ Test coverage
- Amount is more than contract balance.
  - ☐ Test coverage

Function call analysis

- `to.transfer(amount);`
  - **External/Internal?** External.
  - **Argument control?** `to` and `amount`.
  - **Impact**: The function sends the specified amount of Ether to the recipient's address. The `transfer` function reverts the transaction if the transfer fails (e.g., in case the recipient spent more than 2,300 gas).

6. Assessment Results

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

During our assessment on the scoped Eclipse contracts, we discovered six findings. Two critical issues were found. One was of high impact, one was of low impact, and the remaining findings were informational in nature. Eclipse Laboratories 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 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.