Eclipse Canonical Bridge

Security Assessment

July 18th, 2024 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Eclipse engaged OtterSec to assess the `eclipse` and `sysgy` programs. This assessment was conducted between July 6th and July 11th, 2024. For more information on our auditing methodology, refer to chapter 07.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a discrepancy in calculating the minimum amount of lamports to deposit in the Ethereum Bridge, failing to account for the account storage overhead (OS-ECB-ADV-00). This oversight may result in an insufficient rent deposit amount. Additionally, the bump values for account initialization are directly supplied as function arguments, risking incorrect assignments and compromising program integrity (OS-ECB-ADV-01). Furthermore, the message-sending functionality incorrectly sets the destination chain ID as the Ethereum chain ID instead of the intended target (Eclipse) during event emission (OS-ECB-ADV-02).

We also made suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices (OS-ECB-SUG-02), and highlighted several instances where proper validation is not done, resulting in potential security issues (OS-ECB-SUG-01). Additionally, we advised against the utilization of Solana logs which are limited to 10 KB and truncate larger logs, potentially resulting in incomplete event data (OS-ECB-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Eclipse-Laboratories-Inc. This audit was performed against commit a8a06e0 and fd6b81b.

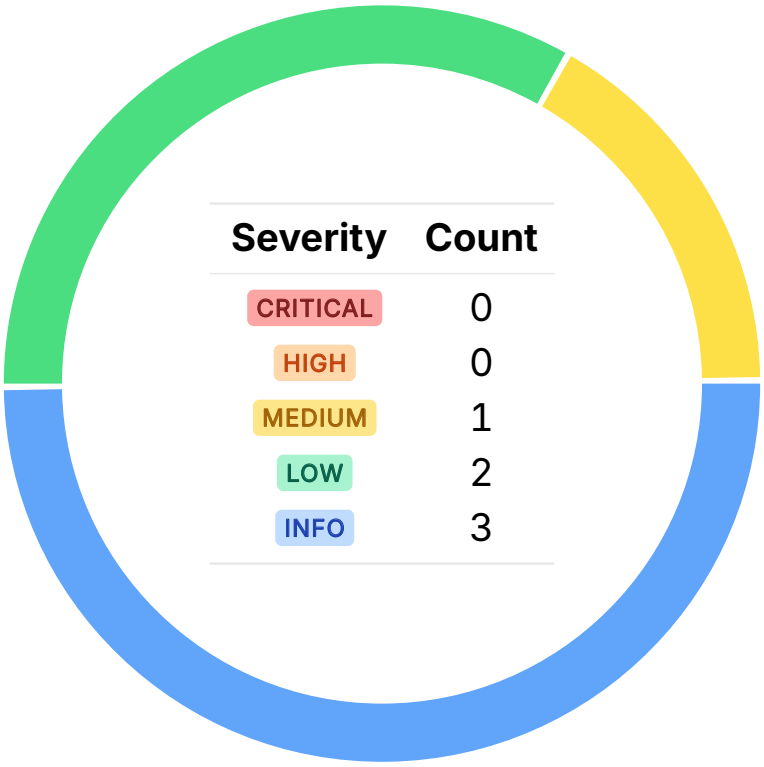
A brief description of the programs is as follows:

Name	Description
eclipse	The Canonical Bridge Solana program is designed to enable secure and efficient asset transfers between the Eclipse blockchain and Ethereum, utilizing the Anchor framework.
sysgy	The Eclipse Canonical Bridge Relayer program encompasses all three canonical bridge relayers and is built utilizing the `cqrs-es` framework.

03 — Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	2
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in chapter 06.

ID	Severity	Status	Description
OS-ECB-ADV-00	MEDIUM	RESOLVED	There is a discrepancy in calculating `MIN_DEPOSIT_LAMPORTS` in `EtherBridge`, failing to account for `ACCOUNT_STORAGE_OVERHEAD`. This oversight may result in an insufficient rent deposit amount.
OS-ECB-ADV-01	LOW	RESOLVED	The `bump` values for account initialization (`deposit_receipt` and `withdrawal_receipt`) are directly supplied as function arguments, risking incorrect assignments and compromising program's integrity.
OS-ECB-ADV-02	LOW	RESOLVED	`sendMessage` incorrectly sets the `toChainId` as the Ethereum chain ID instead of the intended target chain ID (Eclipse) during event emission.

Underestimated Minimum Rent Deposit MEDIUM OS-ECB-ADV-00

Description

The vulnerability in `EtherBridge` arises from an incorrect calculation of `MIN_DEPOSIT_LAMPORTS`, which is necessary for determining the minimum deposit requirement in terms of Solana's rent system. The calculation of `MIN_DEPOSIT_LAMPORTS` does not account for the `ACCOUNT_STORAGE_OVERHEAD` of 128, which is essential in Solana's rent calculation to determine the minimum balance required for rent exemption. Without including `ACCOUNT_STORAGE_OVERHEAD`, `MIN_DEPOSIT_LAMPORTS` may result in an amount that is lower than what is actually required by Solana's rent system. This may result in deposits that are not rent-exempt according to Solana's standards.

>_ EtherBridge.sol SOLIDITY

```solidity
/// @title EtherBridge
/// @dev A bridge contract for depositing and withdrawing ether to and from the Eclipse rollup.
contract EtherBridge is
    [...]
{
    bytes32 public constant ETHER_BRIDGE_ID = keccak256("EtherBridge");

    /// @dev Calculation of constants for minimum deposit requirements.
    /// `MIN_DEPOSIT_LAMPORTS` ensures deposit data on Solar Eclipse is rent-exempt, based on:
    /// These constants ensure deposit amounts meet Solar Eclipse's rent-exemption criteria,
        ↪  adapted for Ethereum's context.
    uint256 public constant MIN_DEPOSIT = MIN_DEPOSIT_LAMPORTS * WEI_PER_LAMPORT;
    [...]
}
```

Remediation

Ensure that `MIN_DEPOSIT_LAMPORTS` is recalculated to include `ACCOUNT_STORAGE_OVERHEAD`. Validate the calculation against Solana CLI tools or official documentation to ensure alignment with Solana's rent requirements.

Patch

Resolved in PR#268 by hard-coding `MIN_DEPOSIT_LAMPORTS = 2_000_000`, which is sufficient to cover the rent for 49 bytes.

Passing Bump Value Via Function Argument LOW OS-ECB-ADV-01

Description

In both `deposit` and `withdraw` (shown below), the `bump` parameter is passed directly as an argument to the functions. This `bump` value is intended to generate a Program Derived Address (PDA) for the `deposit_receipt` and `withdrawal_receipt` accounts. It ensures uniqueness and security by generating a unique address based on the provided `seed` and `bump` combination.

>_ canonical_bridge/src/instructions/withdraw.rs RUST

```rust
pub fn withdraw(
    ctx: Context<Withdraw>,
    destination_address: String,
    withdrawal_nonce: u64,
    amount: u64,
    bump: u8,
) -> Result<()> {
    [...]
}
```

However, due to the fact that the `bump` value is provided as an argument to `deposit` and `withdraw`, there is a risk of accidentally assigning incorrect or unintended `bump` values to the `deposit_receipt` or `withdrawal_receipt` accounts. Incorrect bump values may result in the generation of incorrect addresses, affecting the overall functionality of the program.

Remediation

Ensure that `bump` values are derived from the account's context rather than being directly supplying as input parameters.

Patch

Resolved in PR#7 by retrieving account bumps directly from the accounts rather than relying on user inputs.

Chain ID Misconfiguration LOW OS-ECB-ADV-02

Description

`sendMessage` emits an event with the encoded chain ID, which is set to the Ethereum chain ID (`$.ethereumChainId`). Thus, the chain ID included in the event is the Ethereum chain ID instead of Solana ID. This may confuse off-chain systems that rely on these event logs to determine the destination of the message.

>_ *Mailbox.sol* SOLIDITY

```
/// @dev Modules call this method to send messages out to Eclipse.
function sendMessage(bytes memory, bytes memory receiver_, bytes memory message_, bytes memory)
    [...]
{
    [...]
    // Create security param
    bytes memory param = abi.encode(EthereumSecurityParam({nonce: outboundNonce}));
    emit MessageSent(receiver_, encodeChainId(\$.ethereumChainId), message_, param)
    return true;
}
```

Remediation

Ensure that the correct chain ID for the Eclipse chain is utilized if the event's purpose is to signal the destination chain.

Patch

Resolved in PR#268 by correcting toChainId to eclipseChainId in MessageSent.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-ECB-SUG-00	`log_outbox_message` utilizes Solana logs which are limited to 10 KB, and thus truncate larger logs, potentially resulting in incomplete event data.
OS-ECB-SUG-01	There are several instances where proper validation is not done, resulting in potential security issues.
OS-ECB-SUG-02	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Loss In Event Data Due To Truncation OS-ECB-SUG-00

Description

In `withdraw::log_outbox_message`, the encoded `WithdrawalMessageSent` structure is logged utilizing the `msg!` macro. Due to the utilization of Solana logs, if the encoded message is large, it may exceed the 10 KB limit, resulting in a truncated log. Solana nodes have a default limit of 10 KB for logs. Any logs that exceed this size are truncated, resulting in incomplete logs, which is problematic for tracking detailed events or large amounts of data.

>_ *canonical_bridge/src/instructions/withdraw.rs* RUST

```rust
pub fn log_outbox_message(message: WithdrawalMessageSent) -> () {
    let encoded_message = encode(message);
    msg!("{:?}", BASE64_STANDARD.encode(encoded_message));
}
```

Remediation

Utilize CPI (Cross-Program Invocation) events instead of logs, as they provide a more reliable way to capture and store event data without being subject to the 10 KB log size limitation. Since instruction data stored in `RPC` providers is not subject to the 10 KB truncation limit, this ensures that the full event data is captured and stored.

Missing Validation Logic OS-ECB-SUG-01

Description

1. `Mailbox::sendMessage` generates a nonce (`outboundNonce`) via `generateRandomNonce` , which returns a `uint64` . This nonce is then stored in `StorageV1.sendNonces` to prevent message replay attacks. However, utilizing a smaller data type such as `uint64` (which supports 2^{64} possible values) increases the risk of nonce collisions over time, especially in high-frequency environments. After approximately 600 million messages, there is a theoretical 1% chance of encountering a nonce collision due to the limited range of `uint64` .
2. The current implementation of `addressArrayInitialized` checks that each address in the array is non-zero but does not check if the array itself is empty. If an empty array is passed, the loop will not execute, and `addressArrayInitialized` will proceed without validating any address.

>_ *CommonContainer.sol* SOLIDITY

```solidity
/// @dev Ensures that all addresses are non-zero.
modifier addressArrayInitialized(address[] memory _addresses) {
    for (uint256 i = 0; i < _addresses.length; i++) {
        require(_addresses[i] != address(0), "AddressChecks: zero address in array");
    }
    _;
}
```

3. In the current design, the custody of funds is held in the `relayer` account. Thus, if it gets compromised, it will result in a complete loss of funds since it holds custody over all the funds managed by the bridge.
4. In the current implementation of `EtherBridge` , `Treasury` , and `Mailbox` , it is possible to call `initialize` on the implementation address. This sets the caller as the `owner` , resulting in a potential contract takeover. An attacker may exploit this to `selfdestruct` the implementation contract, effectively bricking the proxy.

Remediation

1. Instead of `uint64`, utilize a larger nonce type such as `uint256` to significantly reduce the likelihood of collisions. Also, update the validation to check if `$.sendNonces[outboundNonce] == false` before storing the nonce to further mitigate risks. This ensures that each nonce utilized is unique within the context of the contract's operation, reducing the chance of unintended behavior due to nonce reuse.
2. implement an `_addresses.length > 0` check to address the edge case where the array is empty.
3. Utilize a Program-Derived Account (PDA) so that control over the funds is governed by the smart contract logic implementing a proper access control mechanism, reducing reliance on a single entity.
4. Prevent `initialize()` from being invoked on the implementation contracts by adding the following constructor.

```solidity
constructor() {
    _disableInitializers();
}
```

Code Maturity

OS-ECB-SUG-02

Description

1. `deposit` and `withdraw` currently utilize `invoke_signed`, which is specifically meant for transactions that require additional signature seeds for validation. Since no signed seeds are involved in this case, `deposit` and `withdraw` may utilize `invoke`.
2. In `state`, the size for `Configuration` and `WithdrawalReceipt` is set to 1024 bytes. However, this allocation is excessive, as `Configuration` only requires 11 bytes, and `WithdrawalReceipt` only requires 57 bytes, assuming 32 bytes for `destination_address`. Reducing the allocated size will optimize storage and ensure efficient resource utilization.

>_ canonical_bridge/src/state.rs RUST

```rust
impl WithdrawalReceipt {
    // TODO: optimize here
    pub fn size() -> usize {
        1024
    }
}
impl Configuration {
    // TODO: optimize here
    pub fn size() -> usize {
        1024
    }
}
```

Remediation

Implement the above-mentioned changes.

Patch

Resolved in #6 and #8.

06 — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

07 — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.