OMNISCIA

November 20, 2024

SMART CONTRACT AUDIT REPORT

Sophon
USDC Bridge

omniscia.io

info@omniscia.io

Online report: sophon-usdc-bridge

Omniscia.io is one of the fastest growing and most trusted blockchain security firms and has rapidly become a true market leader. To date, our team has collectively secured over 370+ clients, detecting 1,500+ high-severity issues in widely adopted smart contracts.

Founded in France at the start of 2020, and with a track record spanning back to 2017, our team has been at the forefront of auditing smart contracts, providing expert analysis and identifying potential vulnerabilities to ensure the highest level of security of popular smart contracts, as well as complex and sophisticated decentralized protocols.

Our clients, ecosystem partners, and backers include leading ecosystem players such as L'Oréal, Polygon, AvaLabs, Gnosis, Morpho, Vesta, Gravita, Olympus DAO, Fetch.ai, and LimitBreak, among others.

To keep up to date with all the latest news and announcements follow us on twitter @omniscia_sec.


omniscia.io


info@omniscia.io

Online report: sophon-usdc-bridge

USDC Bridge Security Audit

Audit Report Revisions

Commit Hash	Date	Audit Report Hash
425dc35f01	November 11th 2024	ec321efd38
51e19fdfe4	November 20th 2024	a1df9d1e65

Audit Overview

We were tasked with performing an audit of the Sophon codebase and in particular their USDC L1 and L2 bridge implementations.

The bridge implementations are based on the zkSync Era `L1SharedBridge` and `L2SharedBridge` implementations with significant adaptations to:

- Remove support of native fund deposit and withdrawal flows
- Remove support of zkSync Era legacy functions and corresponding transaction validation flows
- Simplify logic due to the omission of extraneous logic and the conversion to a single-purpose USDC bridge

The `Bridgehub` implementation that the system interacts with was considered out-of-scope of the audit engagement and the adaptations of the system were evaluated in lieu of their original implementation in the zkSync Era codebase.

Over the course of the audit, we identified the absence of input sanitization across three functions two of which would indicate improper token support at the L2 deployment level.

We advise the Sophon team to closely evaluate all minor-and-above findings identified in the report and promptly remediate them as well as consider all optimizational exhibits identified in the report.

Post-Audit Conclusion

The Sophon team iterated through all findings within the report and provided us with a revised commit hash to evaluate all exhibits on.

We evaluated all alleviations performed by Sophon and have identified that all non-informational exhibits have been adequately dealt with.

A single informational exhibit has been acknowledged (LSE-02S), rendering all items in the audit report properly consumed by the Sophon team with no outstanding remediative actions remaining.

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	0	0	0	0
Informational	4	3	0	1
Minor	2	2	0	0
Medium	0	0	0	0
Major	0	0	0	0

During the audit, we filtered and validated a total of **3 findings utilizing static analysis** tools as well as identified a total of **3 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they can introduce potential misbehaviours of the system as well as exploits.

- 🎯 **Scope**
- 💻 **Compilation**
- 🔍 **Static Analysis**
- 👁️ **Manual Review**
- 🖌️ **Code Style**

Scope

The audit engagement encompassed a specific list of contracts that were present in the commit hash of the repository that was in scope. The tables below detail certain meta-data about the target of the security assessment and a navigation chart is present at the end that links to the relevant findings per file.

Target

- Repository: **https://github.com/sophon-org/custom-usdc-bridge**
- Commit: 425dc35f01409d8edf98ed13f4ec1615622738b4
- Language: Solidity
- Network: Sophon
- Revisions: **425dc35f01**, **51e19fdfe4**

Contracts Assessed

File	Total Finding(s)
src/L1SharedBridge.sol (LSB)	2
src/L2SharedBridge.sol (LSE)	4

Compilation

The project utilizes `foundry` as its development pipeline tool, containing an array of tests and scripts coded in Solidity.

To compile the project, the `build` command needs to be issued via the `forge` CLI tool:

```bash
forge build
```

The `forge` tool automatically selects Solidity version `0.8.24` based on the version specified within the `foundry.toml` file.

The project contains discrepancies with regards to the Solidity version used, however, they are solely located in external dependencies and can thus be safely ignored.

The `pragma` statements have been locked to `0.8.24` (`=0.8.24`), the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `foundry` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

Static Analysis

The execution of our static analysis toolkit identified **17 potential issues** within the codebase of which **11 were ruled out to be false positives** or negligible findings.

The remaining **6 issues** were validated and grouped and formalized into the **3 exhibits** that follow:

ID	Severity	Addressed	Title
LSB-01S	Informational	Yes	Inexistent Sanitization of Input Addresses
LSE-01S	Informational	Yes	Inexistent Sanitization of Input Addresses
LSE-02S	Informational	Acknowledged	Multiple Top-Level Declarations

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Sophon's purpose-built USDC bridges.

As the project at hand implements L1 and L2 bridges akin to zkSync Era, intricate care was put into ensuring that the **flow of funds within the system conforms to the specifications and restrictions** laid forth within each protocol's specification.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed absence of input sanitization** within the system which could have had **minor ramifications** to its overall operation; we advise the relevant minor-severity exhibits of the audit report to be consulted for further details.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to the extent it need be.

A total of **3 findings** were identified over the course of the manual review of which **2 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
LSB-01M	Minor	Yes	Inexistent Sanitization of L2 Value
LSE-01M	Minor	Yes	Inexistent Validation of Input

Code Style

During the manual portion of the audit, we identified **1 optimizations** that can be applied to the codebase that will decrease the operational cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

ID	Severity	Addressed	Title
LSE-01C	Informational	Yes	Inefficient Recalculation of Literal

L1SharedBridge Static Analysis Findings

LSB-01S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Informational	L1SharedBridge.sol:L91-L97

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/L1SharedBridge.sol

SOL

```sol
91  constructor(address _l1UsdcAddress, IBridgehub _bridgehub)
92      reentrancyGuardInitializer
93  {
94      _disableInitializers();
95      L1_USDC_TOKEN = _l1UsdcAddress;
96      BRIDGE_HUB = _bridgehub;
97  }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation:

All input arguments of the `L1SharedBridge::constructor` function are adequately sanitized as non-zero in the latest in-scope revision of the codebase, addressing this exhibit.

L2SharedBridge Static Analysis Findings

LSE-01S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Informational	L2SharedBridge.sol:L42-L46

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/L2SharedBridge.sol

SOL

```sol
42  constructor(address _l1UsdcToken, address _l2UsdcToken) {
43      L1_USDC_TOKEN = _l1UsdcToken;
44      L2_USDC_TOKEN = _l2UsdcToken;
45      _disableInitializers();
46  }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation:

All input arguments of the `L2SharedBridge::constructor` function are adequately sanitized as non-zero in the latest in-scope revision of the codebase, addressing this exhibit.

LSE-02S: Multiple Top-Level Declarations

Type	Severity	Location
Code Style	Informational	L2SharedBridge.sol:L15, L25

Description:

The referenced file contains multiple top-level declarations that decrease the legibility of the codebase.

Example:

src/L2SharedBridge.sol

SOL

```sol
interface MintableToken {
    function mint(address _to, uint256 _amount) external;
    function burn(uint256 _amount) external;
}

/// @author Sophon
/// @notice Forked from ML L1SharedBridge contract
/// @custom:security-contact security@matterlabs.dev
/// @notice The "default" bridge implementation for the ERC20 tokens. Note, that it does not
/// support any custom token logic, i.e. rebase tokens' functionality is not supported.
```

Example (Cont.):

```sol
contract L2SharedBridge is IL2SharedBridge, Initializable {
```

Recommendation:

We advise all highlighted top-level declarations to be split into their respective code files, avoiding unnecessary imports as well as increasing the legibility of the codebase.

Alleviation:

The Sophon team evaluated this exhibit but opted to acknowledge it in the current iteration of the codebase.

L1SharedBridge Manual Review Findings

LSB-01M: Inexistent Sanitization of L2 Value

Type	Severity	Location
Input Sanitization	Minor	L1SharedBridge.sol:L167

Description:

The `L1SharedBridge::bridgehubDeposit` function will not properly reserve memory for the `_l2Value` argument of the `L1SharedBridge::bridgehubDeposit` call disallowing it from being sanitized.

Impact:

An arbitrary `_l2Value` can be supplied to the `L1SharedBridge::bridgehubDeposit` function which may cause upstream complications in the Bridgehub implementation.

Example:

src/L1SharedBridge.sol

SOL

```sol
162 /// @notice Initiates a deposit transaction within Bridgehub, used by
`requestL2TransactionTwoBridges`.
163 function bridgehubDeposit(
164     uint256 _chainId,
165     address _prevMsgSender,
166     // solhint-disable-next-line no-unused-vars
167     uint256,
168     bytes calldata _data
169 )
170     external
171     payable
```

Example (Cont.):

```sol
    override
    onlyBridgehub
    whenNotPaused
    returns (L2TransactionRequestTwoBridgesInner memory request)
{
    address l2Bridge = l2BridgeAddress[_chainId];
    require(l2Bridge != address(0), "USDC-ShB l2 bridge not deployed");

    (address _l1Token, uint256 _depositAmount, address _l2Receiver) =
abi.decode(_data, (address, uint256, address));
    require(_l1Token == L1_USDC_TOKEN, "USDC-ShB: Only USDC deposits supported");
    require(BRIDGE_HUB.baseToken(_chainId) != _l1Token, "USDC-ShB: baseToken
deposit not supported");
    require(msg.value == 0, "USDC-ShB m.v > 0 for BH d.it 2");
```

Recommendation:

We advise a proper variable name to be declared for the third index argument of the `L1SharedBridge::bridgehubDeposit` function and zero-value sanitization to be imposed on it so as to avoid upstream complications in the Bridgehub implementation.

Alleviation:

A `require` check was introduced to ensure that the input `_l2Value` is set to `0`, alleviating this exhibit in full.

L2SharedBridge Manual Review Findings

LSE-01M: Inexistent Validation of Input

Type	Severity	Location
Input Sanitization	Minor	L2SharedBridge.sol:L92-L94, L96-L98

Description:

The referenced `L2SharedBridge::l1TokenAddress` and `L2SharedBridge::l2TokenAddress` functions do not ensure that the input addresses match their respective LX counterparts.

Impact:

The L1-to-L2 and L2-to-L1 mapping functions of the `L2SharedBridge` will presently yield a constant address regardless of what input is passed in, signaling support for tokens that are ultimately not supported by the system.

Example:

src/L2SharedBridge.sol

SOL

```sol
92  function l1TokenAddress(address) external view returns (address) {
93      return L1_USDC_TOKEN;
94  }
95
96  function l2TokenAddress(address) public view override returns (address) {
97      return L2_USDC_TOKEN;
98  }
```

Recommendation:

We advise the `L2SharedBridge::l1TokenAddress` function to ensure the input matches the `L2_USDC_TOKEN` address and the `L2SharedBridge::l2TokenAddress` function to ensure its input matches the `L1_USDC_TOKEN` address, preventing the functions from successfully yielding an invalid address for arbitrary token inputs.

Alleviation:

Both functions were updated to ensure that the input `address` matches the expected L2 and L1 token address for the `L2SharedBridge::l1TokenAddress` and `L2SharedBridge::l2TokenAddress` functions respectively.

As such, we consider this exhibit fully alleviated.

L2SharedBridge Code Style Findings

LSE-01C: Inefficient Recalculation of Literal

Type	Severity	Location
Gas Optimization	Informational	L2SharedBridge.sol:L87

Description:

The referenced operation will add `0x08` to the known value literal `SYSTEM_CONTRACTS_OFFSET` on each invocation of the `L2SharedBridge::withdraw` function.

Example:

src/L2SharedBridge.sol

SOL

```
87  IL2Messenger(address(SYSTEM_CONTRACTS_OFFSET + 0x08)).sendToL1(message);
```

Recommendation:

We advise the result of the calculation itself to be stored as a `constant` declaration, optimizing the function's gas cost.

Alleviation:

The referenced value literal has been relocated as an `L2_MESSENGER` contract-level `constant` optimizing the function's gas cost as advised.

Finding Types

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted if blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Circom language boasts that discerns it from other conventional programming languages.

Curve Specific

Circom defaults to using the BN128 scalar field (a 254-bit prime field), but it also supports BSL12-381 (which has a 255-bit scalar field) and Goldilocks (with a 64-bit scalar field). However, since there are no constants denoting either the prime or the prime size in bits available in the Circom language, some Circomlib templates like `Sign` (which returns the sign of the input signal), and `AliasCheck` (used by the strict versions of `Num2Bits` and `Bits2Num`), hardcode either the BN128 prime size or some other constant related to BN128. Using these circuits with a custom prime may thus lead to unexpected results and should be avoided.

Code Style

In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a toplevel variable in the circuit.

Mathematical Operations

This category is used when a mathematical issue is identified. This implies an issue with the implementation of a calculation compared to the specifications.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.

Privacy Concern

This category is used when information that is meant to be kept private is made public in some way.

Proof Concern

Under-constrained signals are one of the most common issues in zero-knowledge circuits. Issues with proof generation fall under this category.

Severity Definition

In the ever-evolving world of blockchain technology, vulnerabilities continue to take on new forms and arise as more innovative projects manifest, new blockchain-level features are introduced, and novel layer-2 solutions are launched. When performing security reviews, we are tasked with classifying the various types of vulnerabilities we identify into subcategories to better aid our readers in understanding their impact.

Within this page, we will clarify what each severity level stands for and our approach in categorizing the findings we pinpoint in our audits. To note, all severity assessments are performed **as if the contract's logic cannot be upgraded** regardless of the underlying implementation.

Severity Levels

There are five distinct severity levels within our reports; `unknown`, `informational`, `minor`, `medium`, and `major`. A TL;DR overview table can be found below as well as a dedicated chapter to each severity level:

	Impact (None)	Impact (Low)	Impact (Moderate)	Impact (High)
Likelihood (None)	Informational	Informational	Informational	Informational
Likelihood (Low)	Informational	Minor	Minor	Medium
Likelihood (Moderate)	Informational	Minor	Medium	Major
Likelihood (High)	Informational	Medium	Major	Major

Unknown Severity

The `unknown` severity level is reserved for misbehaviors we observe in the codebase that cannot be quantified using the above metrics. Examples of such vulnerabilities include potentially desirable system behavior that is undocumented, reliance on external dependencies that are out-of-scope but could result in some form of vulnerability arising, use of external out-of-scope contracts that appears incorrect but cannot be pinpointed, and other such vulnerabilities.

In general, `unknown` severity level vulnerabilities require follow-up information by the project being audited and are either adjusted in severity (if valid), or marked as nullified (if invalid).

Additionally, the `unknown` severity level is sometimes assigned to centralization issues that cannot be assessed in likelihood due to their exploitation being tied to the honesty of the project's team.

Informational Severity

The `informational` severity level is dedicated to findings that do not affect the code functionally and tend to be stylistic or optimizational in nature. Certain edge cases are also set under `informational` vulnerabilities, such as overflow operations that will not manifest in the lifetime of the contract but should be guarded against as a best practice, to give an example.

Minor Severity

The `minor` severity level is meant for vulnerabilities that require functional changes in the code but tend to either have little impact or be unlikely to be recreated in a production environment. These findings can be acknowledged except for findings with a moderate impact but low likelihood which must be alleviated.

Medium Severity

The `medium` severity level is assigned to vulnerabilities that must be alleviated and have an observable impact on the overall project. These findings can only be acknowdged if the project deems them desirable behavior and we disagree with their point-of-view, instead urging them to reconsider their stance while marking the exhibit as acknowledged given that the project has ultimate say as to what vulnerabilities they end up patching in their system.

Major Severity

The `major` severity level is the maximum that can be specified for a finding and indicates a significant flaw in the code that must be alleviated.

Likelihood & Impact Assessment

As the preface chapter specifies, the blockchain space is constantly reinventing itself meaning that new vulnerabilities take place and our understanding of what security means differs year-to-year.

In order to reliably assess the likelihood and impact of a particular vulnerability, we instead apply an abstract measurement of a vulnerability's impact, duration the impact is applied for, and probability that the vulnerability would be exploited in a production environment.

Our proposed definitions are inspired by multiple sources in the security community and are as follows:

- Impact (High): A core invariant of the protocol can be broken for an extended duration.
- Impact (Moderate): A non-core invariant of the protocol can be broken for an extended duration or at scale, or an otherwise major-severity issue is reduced due to hypotheticals or external factors affecting likelihood.
- Impact (Low): A non-core invariant of the protocol can be broken with reduced likelihood or impact.
- Impact (None): A code or documentation flaw whose impact does not achieve low severity, or an issue without theoretical impact; a valuable best-practice
- Likelihood (High): A flaw in the code that can be exploited trivially and is ever-present.
- Likelihood (Moderate): A flaw in the code that requires some external factors to be exploited that are likely to manifest in practice.
- Likelihood (Low): A flaw in the code that requires multiple external factors to be exploited that may manifest in practice but would be unlikely to do so.
- Likelihood (None): A flaw in the code that requires external factors proven to be impossible in a production environment, either due to mathematical constraints, operational constraints, or system-related factors (i.e. EIP-20 tokens not being re-entrant).

Disclaimer

The following disclaimer applies to all versions of the audit report produced (preliminary / public / private) and is in effect for all past, current, and future audit reports that are produced and hosted under Omniscia:

IMPORTANT TERMS & CONDITIONS REGARDING OUR SECURITY AUDITS/REVIEWS/REPORTS AND ALL PUBLIC/PRIVATE CONTENT/DELIVERABLES

Omniscia ("Omniscia") has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. Omniscia and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

This audit report shall not be printed, saved, disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Omniscia.

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. Omniscia is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. Omniscia's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

Omniscia in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will Omniscia, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

It is the sole responsibility of the Project team to provide adequate levels of test and perform the necessary checks to ensure that the contracts are functioning as intended, and more specifically to ensure that the functions contained within the contracts in scope have the desired intended effects, functionalities and outcomes, as documented by the Project team.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

Omniscia will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

Omniscia will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.