Audittens

ZKsync Shared Bridge (USDC) Audit

December 17, 2024

Contents

1 About Auditens

Auditens is an audit company with expertise across various sectors of Web3 security.

What truly sets us apart is that our team consists entirely of top participants in mathematics and competitive programming competitions, uniquely equipping us to rapidly analyze codebases, uncover explicit and implicit invariants, and identify unique attack vectors.

You can subscribe and learn more about us at https://x.com/Auditens.

2 About ZKsync

ZKsync is a Layer-2 protocol that scales Ethereum with cutting-edge ZK tech. Their mission is not only to merely increase Ethereum's throughput, but to fully preserve its foundational values — freedom, self-sovereignty, decentralization — at scale.

ZK-rollups like Era are the only scaling solution that can inherit 100% of Ethereum's security. But theory is not enough. ZKsync is committed to go above and beyond to make Era by far the most secure L2, in practice.

You can subscribe and learn more about the project at https://zksync.io.

3 Risk classification

The severities of the issues are determined based on the following properties:

- Critical — results in a significant loss of assets within the protocol, a major deviation from the expected behavior of protocol components, and/or a violation of key security invariants.
- High — causes loss of assets within the protocol and/or general violations of protocol security invariants, with limitations on the variety of potential attacks.
- Medium — enables barely profitable attacks on the protocol, violations of security invariants that pose minimal risk to protocol users, and/or functionality limitations affecting only a relatively small subset of users.
- Low — enables griefing attacks on the protocol, unexpected changes in the protocol's behavior that are imperceptible, and/or functionality limitations with negligible impact on the security of the protocol.
- Informational — issues that have no practical impact on the security of the current version of the protocol but could potentially lead to more serious consequences in future updates, create the possibility for incorrect use of the protocol by users, and/or result in a significant decline in code quality, making maintenance more challenging.

While determining the severity of issues, the constraints required for successful attacks and the potential actions to address their consequences are taken into account in the decision-making process.

4 Executive summary

4.1 Engagement overview

ZKsync Security Council Foundation engaged Audittens to review the security of ZKsync's USDC bridging protocol. From December 9, 2024, to December 13, 2024, a team of four security researchers audited the provided source code.

4.2 Scope

The `matter-labs/usdc-bridge` repository at the `ce4c523b705b73fe28de8d48e2e0277e7fc39b3e` commit was audited.

The following contracts were within the scope of the audit:

```
matter-labs
   usdc-bridge
   └── src
       ├── L1USDCBridge.sol
       └── L2USDCBridge.sol
```

4.3 Summary of findings

Severity \ Status	Acknowledged	Total
Critical	0	0
High	0	0
Medium	1	1
Low	1	1
Informational	2	2
Total	4	4

Table 1. Distribution of found issues.

5 Assumptions and limitations

5.1 Explicit invariants of the codebase usage

The audit was conducted assuming that:

1. No regular users will interact with the bridge smart contracts until the owner of the smart contract issues an official notification in a reliable manner.
2. The pausing functionality implemented in the `L1USDCBridge` contract is not part of the pausing functionality required by the "*Ability to pause USDC bridging*" section of the Circle's documentation.

5.2 Limitations

The codebase was audited as is. Any subsequent changes may introduce new vulnerabilities and require a separate audit.

6 Findings

6.1 Medium severity findings

6.1.1 M-01. Malicious admin can steal `USDC` using incorrect bridge initialization

Context:
src/L1USDCBridge.sol:129

Description: The admin of the `L1USDCBridge` can incorrectly initialize the `l2BridgeAddress` by calling the `L1USDCBridge.initializeChainGovernance` function with wrong `_l2BridgeAddress`. Although this is generally considered an expected potential behavior, there is a possibility to steal user funds while maintaining the natural scenario of the contract initialization process.

Attack scenario:

1. The attacker performs a regular deposit `D1` of `X USDC` tokens through some already working bridge `B` that accepts `USDC` (e.g. the standard `Bridgehub.sharedBridge()` bridge of ZKsync).
2. Once the previous deposit is finalized on `L2`, the attacker performs a regular withdrawal `W1` of `X USDC` tokens through the bridge `B`.
3. Admin of the `L1USDCBridge` contract calls the `L1USDCBridge.initializeChainGovernance` function with `_l2BridgeAddress` equal to the address of `L2`-contract of `B`.
4. The attacker performs a deposit `D2` of `X USDC` tokens through the `L1USDCBridge`. The corresponding `L1USDCBridge.chainBalance` value becomes equal to `X`. This deposit transaction will fail later on `L2`.
5. Without waiting for finalization of `D2` on `L2`, the attacker performs a malicious withdrawal `W2` of `X USDC` tokens through the `L1USDCBridge.finalizeWithdrawal` function, using the proof of the `W1`. The `L1USDCBridge.chainBalance` value becomes equal to `0`.
6. Owner of the `L1USDCBridge` contract calls `L1USDCBridge.reinitializeChainGovernance` function with the correct `_l2BridgeAddress` parameter. On the surface it is difficult to notice that something is wrong with the state of the bridge before such a call, because the transaction history performed on `L1` and both `L1USDCBridge.chainBalance`, `L1USDC.balanceOf(L1USDCBridge)` values look valid.
7. Regular users deposit `USDC` token through the `L1USDCBridge` bridge. Due to these deposits, the `L1USDCBridge.chainBalance` value also increases.
8. The attacker successfully calls the `L1USDCBridge.claimFailedDeposit` with parameters corresponding to `D2`.

At the step 4 of the described scenario, the attacker spends the `X USDC`, while at the steps 5 and 8 he receives a total of `2X USDC`. As a result, the attacker steals a total of `X USDC` from regular users who made deposits at the step 7.

Steps 4-5 can be performed atomically right before the owner's transaction from step 6. In such a case, the owner cannot notice anything wrong before his call of `L1USDCBridge.reinitializeChainGovernance` function is executed.

Recommendation: Either:

- Remove the `reinitializeChainGovernance` function and allow only the owner of the `L1USDCBridge` contract to call the `initializeChainGovernance` function.
- In case of mentioned reinitialization ensure that no deposit transactions have been performed before the reinitialization is completed.

ZKsync: *Acknowledged*. We plan to fix this in future releases.

6.2 Low severity findings

6.2.1 L-01. The `_l2Token` parameter is ignored in the `L2USDCBridge.withdraw` function

Context:
`src/L2USDCBridge.sol:78,80`

Description: In the `L2USDCBridge.withdraw` function, the `_l2Token` parameter should be checked against the `L2_USDC_TOKEN` value. However, the function currently allows calls with arbitrary `_l2Token` values while always withdrawing `USDC`.

The caller of this function may assume that only the requested token can be withdrawn. Therefore, a smart contract that stores multiple assets and allows withdrawals through different bridges could be at risk: it is possible to substitute the token to be withdrawn from this contract's perspective.

Attack scenario: Imagine some passive vault on `L2` that allows bridged tokens to be deposited into it and withdrawn to `L1` (where withdrawals are being processed internally through different whitelisted bridges, including `L2USDCBridge`). This vault has approvals for deposited tokens to the whitelisted bridges. In such a setup, the attacker can steal all the `USDC` from the vault by taking the following steps:

1. The attacker deploys a token `T` on `L2` with big enough initially minted supply.
2. The attacker deposits the minted amount of `T` into the vault.
3. The attacker requests the vault to withdraw `A` tokens of `T` using the `L2USDCBridge`. Here `A` represents the vault's `USDC` balance.
4. The vault calls `L2USDCBridge.withdraw(attackerAddress, T, A)`, withdrawing `USDC` to the attacker.

Instead of custom token `T`, attacker can use some already used in the vault token with price lower than the price of `USDC`.

Recommendation: Modify the `L2USDCBridge.withdraw` function:

```
function withdraw(address _l1Receiver, address _l2Token, uint256 _amount) external override {
    require(_l2Token == L2_USDC_TOKEN);

    ... Already implemented code ...
}
```

ZKsync: *Acknowledged*. We plan to fix this in future releases.

6.3 Informational findings

6.3.1 I-01. The `L2USDCBridge.finalizeDeposit` function ignores the return value of `FiatTokenV1.mint`

Context:
`src/L2USDCBridge.sol:16,71`
`circlefin/stablecoin-evm - contracts/v1/FiatTokenV1.sol:119-127`

Description: In the implementation of the `USDC` contract, the `mint` function returns `bool` — "*True if the operation was successful*". However, this value is ignored in the `L2USDCBridge.finalizeDeposit` function.

Currently this doesn't affect the security of the system, because the `FiatTokenV1.mint` always either returns `true` or reverts. However, if this behaviour will be changed in the future (the `FiatTokenV1.mint` function will in some cases return `false` for unsuccessful mints), calls to the `L2USDCBridge.finalizeDeposit` function may succeed when the token wasn't actually minted.

Recommendation:

- Add a return value to the `mint` function of the `MintableToken` interface:
  `function mint(address _to, uint256 _amount) external returns (bool);`

- Check the returned value in the `L2USDCBridge.finalizeDeposit` function:
  `require(MintableToken(L2_USDC_TOKEN).mint(_l2Receiver, _amount));`

ZKsync: *Acknowledged*. We plan to fix this in future releases.

6.3.2 I-02. The `L1USDCBridge._checkWithdrawal` function checks if the token being withdrawn is a base token

Context:
src/L1USDCBridge.sol:370-371

Description: In the `L1USDCBridge._checkWithdrawal` function, there is a check that the token being withdrawn is a base token — if that's the case, the message is expected to be sent by `L2_BASE_TOKEN_SYSTEM_CONTRACT_ADDR`, otherwise it's expected to be sent by `l2BridgeAddress[_chainId]`. However, the `USDCBridge` is not designed to work with base token withdrawals. Therefore, this check is redundant and could introduce vulnerabilities during future development.

Recommendation: Always set `l2Sender = l2BridgeAddress[_chainId]` in the `L1USDCBridge._checkWithdrawal` function. Also, add the `require(!baseTokenWithdrawal);` statement to ensure consistency with the corresponding invariants of the deposit flow.

ZKsync: *Acknowledged*. We plan to fix this in future releases.