OpenZeppelin | security

Optimism Bridge Audit

May 16th, 2022

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

The Optimism team asked us to audit their new bridge mechanism to transfer funds and messages between the Ethereum mainnet (L1) and the Optimism network (L2).

Type	Layer 2
Timeline	From 2022-04-22 To 2022-05-16
Languages	Solidity

Total Issues	26 (0 resolved)
Critical Severity Issues	1 (0 resolved)
High Severity Issues	2 (0 resolved)
Medium Severity Issues	6 (0 resolved)
Low Severity Issues	7 (0 resolved)
Notes & Additional Information	10 (0 resolved)

Scope

For the majority of the audit, we reviewed the all production contracts at commit `0c05488`. However, during the audit the Optimism team undertook a significant rewrite of the code base to ensure backwards compatibility with the live version. To maximize the usefulness of our audit, we spent the last few days reviewing the production contracts at commit `e2da9e2` with the understanding that we would not be able to exhaustively review new functionality that didn't exist in the original scope. We also disregarded all issues that are no longer relevant in the new commit.

Update: After the audit we spent another two days completing our review of the new version. However, the codebase had advanced to commit `8cbaf02` and is still being developed at the monorepo. Consequently, this audit report contains references to the code base at different stages of development. Additionally, the issues relating to the token refund functionality would typically be described together and would likely be addressed together, but they remain distinct to avoid discontinuities with different versions of this report.

All other parts of the system, especially L2 nodes and offchain components, were assumed to work as documented.

System Overview

Documentation about the system is available in the `specs` directory of the repository.

One exception identified during the audit is that L1-to-L2 messages are not charged gas on L2. The Optimism team is working on a metering scheme to prevent L1 users from abusing this property to denial-of-service attack the L2 chain.

By design, a single token on Ethereum may be bridged to multiple tokens on Optimism. This openness comes with the tradeoff that users must be careful not to bridge their tokens to some malicious implementation on Optimism.

Privileged Roles

There are several privileged roles in the system that are worth noting:

- The `L2OutputOracle` is a contract on L1 that tracks the state of the Optimism network. It is initialized with an arbitrary genesis state and associated configuration parameters that should correspond to an existing L2 block. The fault proof mechanism cannot detect an incorrect initialization, which would permit arbitrary L2-to-L1 messages.
- Only the sequencer account can add and delete state roots from the `L2OutputOracle`.
- The `CrossDomainMessenger` contracts (on both domains) have an owner that can pause message relays.
- The `OptimismPortal`, `CrossDomainMessenger` contracts (both domains) and the `StandardBridge` contracts (both domains) are intended to be deployed behind upgradeable proxy contracts, so the administrator can update them, changing any logic within.

Findings

Here are our findings in order of importance.

Critical Severity

C-01
Initiate unauthorized ERC20 bridge

The `finalizeDeposit` function of the `L2StandardBridge` contract is intended to be invoked by the `L1StandardBridge`. This is validated implicitly in the first two scenarios, when the contract attempts to finalize the ETH or ERC20 transfer. In both cases, the finalize function is restricted by the `onlyOtherBridge` modifier. However, if an inconsistency is detected, the contract will refund the deposit by sending it back over the bridge.

In the ERC20 case, this involves initiating an arbitrary token bridge operation. Since the parameters are arbitrary and the function is not authenticated, this means that an attacker can send L2 tokens over the bridge on behalf of a target account.

If the L2 token is not an `OptimismMintableERC20`, the bridge will pull the tokens before sending them to L1. When the target has approved the bridge to spend funds (which would occur before the target initiated a transfer), the attacker could specify their own destination address to receive the funds on L1.

It's worth noting that even if the L2 token is an `OptimismMintableERC20`, the attacker can specify an unrelated L1 token to trigger this attack. However, there are two mitigations:

- the user won't have a natural reason to approve an allowance for the bridge.
- there won't be funds available on L1 to release to the attacker, so this would be a pure griefing attack.

Consider adding access control to the `finalizeDeposit` function to account for all three code paths.

Update: This issue is fixed at the final commit under review. The refund code path has been moved inside the "finalize" functions, where it is protected by the `onlyOtherBridge` modifier.

High Severity

H-01
Inverted finalization validation

The `OptimismPortal` validates that the L2 block has been finalized before allowing a withdrawal contained in that block to be executed. However, the inequality is inverted so it is possible to execute withdrawals before the block is finalized. Consider correcting the validation.

Update: This issue is fixed at the final commit under review.

H-02
Token refunds may use incorrect addresses

If an ERC20 bridge finalization fails, the bridge will attempt to perform a refund on the original sender's chain. However, the "remote" and "local" tokens will be incorrectly flipped before being correctly flipped when constructing the cross-domain message.

Consequently, when the refund message reaches the original chain, the `_localToken` will have the value of the token address *on the opposite chain*. In the unlikely event this succeeds, the wrong token will be refunded. Otherwise, refunds may be continually attempted indefinitely, keeping the same values for `_localToken` and `_remoteToken` regardless of chain.

Consider maintaining the same local and remote tokens in the refund case, and instead leaving that to within `_initiateBridgeERC20Unchecked`. Additionally, since the flipping is subtle, consider including comments to draw the reviewer's attention when the tokens are flipped intentionally.

Medium Severity

M-01

Deposit refund may behave unexpectedly

When finalizing deposits on L2, the `L2StandardBridge` will attempt to refund failed transfers. However, it contains some hidden assumptions that may not hold. Specifically

- it attempts to send funds from the L1 deposit source (the `_from` address) to the L2 deposit recipient (the `_to` address). Since we're on L2, we cannot assume that the `_from` address on L2 or the `_to` address on L1 is controlled by either the depositor or the intended recipient. The refund will attempt to transfer funds to the `_to` address, but on L1 instead of L2.
- when refunding ERC20 tokens, it will either attempt to burn token belonging to the `L2CrossDomainMessenger` or retrieve them from the `_from` address. Either scenario will likely fail, but if the `_from` address on L2 has happened to provide the necessary allowance, its token balance will be unexpectedly deducted. Recall, it is not necessarily associated with the original depositor or recipient. Moreover, the original token transfer has not finalized, so the contract is "burning" tokens (strictly, reducing the available releasable tokens) that were never credited to any address on L2.

Consider merely recording the details of failed transfers. This will make it simpler and clearer to design a mechanism for release that includes the proper access control and validations.

M-02

Donated ETH appears trapped

The `StandardBridge` includes a function to donate ETH to the contract. However, there are no functions that can spend this balance, since they all send exactly what they received in the transaction.

Consider documenting how the ETH would be used during an upgrade or removing the donation function.

M-03

Excessive gas

When relaying a cross-domain message, the gas provided to the recipient is chosen by the relayer, as long as it sufficiently exceeds the value chosen by the sender. This introduces an unnecessary fragility, since the message recipient may behave differently depending on the gas provided. In such a case, the outcome of the call is partially determined by the relayer instead of the message sender, which is counterintuitive and possibly exploitable.

In the interest of predictability, consider sending the exact gas limit specified by the message sender to the recipient.

M-04

Invalid L2 sender

The `finalizeWithdrawalTransaction` function sets the `l2Sender` before executing the withdrawal, and resets it to `DEFAULT_L2_SENDER` afterwards. The intention is to ensure that the sender can be queried during the withdrawal.

However, if a top-level withdrawal transaction re-enters the `finalizeWithdrawalTransaction` function (finalizing a different transaction), `l2Sender` will be set to `DEFAULT_L2_SENDER` after the *inner* withdrawal and will remain incorrectly set during the rest of the *outer* withdrawal.

It is worth noting that there is a guard condition that claims to prevent reentrancy. However, it only prevents the withdrawal transaction from directly invoking a function on the `OptimismPortal`. It does not prevent a withdrawal that finalizes another valid withdrawal.

Consider enforcing non-reentrancy on the `finalizeWithdrawalTransaction` function. Alternatively, if reentrancy is desired, consider caching and then resetting `l2Sender` to the value it was before the call.

M-05

Misleading data parameter

The bridge functions in the `StandardBridge` contract, as well as the legacy versions in the `L1StandardBridge` and `L2StandardBridge` contracts all accept a `_data` parameter to be sent with the transfer. However, the bridge finalization functions do not use that parameter

in any meaningful way. This can be misleading for users who expect the data to be sent to the recipient on the other domain, and may undermine the user's intent.

Although the parameter needs to exist for backwards compatibility reasons, considering ensuring that it is always empty.

Update: During the review of the final commit we realized this description was incomplete. In addition to ensuring empty `_data` values in the legacy functions, consider removing the parameter entirely from the `StandardBridge` functions, which are part of a new interface and do not need to maintain backwards compatability.

M-06

Unpause functionality

The `CrossDomainMessenger` contract has a mechanism for the owner to pause relays. However, there is no corresponding mechanism to unpause the contract, which means the pause functionality is effectively a permanent shutdown (until the contract is upgraded).

Consider introducing an *unpause* function or documenting the reason for the asymmetry.

Low Severity

L-01

Confusing reference to transaction sender

When the `L2StandardBridge` initiates a withdrawal, it uses `msg.sender` instead of the `_from` parameter. Similarly, when the `StandardBridge` initiates an ERC20 bridging, it burns tokens from `msg.sender` rather than the `_from` parameter.

In all supported cases, these are the same value. Nevertheless, in the interest of local reasoning and better encapsulation, consider using the `_from` parameter, since the function shouldn't necessarily know how it will be called.

L-02

Disable implementation contracts

The `CrossDomainMessenger` contract implicitly inherits the OpenZeppelin `Initializable` contract to facilitate upgrades. Consider applying the `initializer` to the constructor, in accordance with recommended usage, to limit the potential attack surface. Note that the latest version of the contract includes a `_disableInitializers` function, which can be used instead.

Similarly, consider initializing the `StandardBridge` contract in its constructor with a non-zero `messenger` to prevent future initializations.

L-03

Implementation contracts have constructor

The `L2OutputOracle` and `OptimismPortal` both claim that they should be deployed behind a proxy, but use a constructor to initialize their variables, which is inconsistent.

Consider removing the comment or using an initializer function instead.

L-04

Nonstandard tokens

Since the bridge is intended to support arbitrary ERC20 tokens, and allow users to add their own, it is worth noting that these may include non-standard ERC20 tokens. For example, users may attempt to transfer tokens with transfer fees, rebasing tokens, tokens with blacklists, or many other variations. Many of these properties, particularly the ones with non-standard accounting, could undermine the basic mechanism of locking tokens in one domain, using counterparts in the other, and then releasing them again on the original domain.

Consider explicitly documenting this risk and define which types of nonstandard tokens are supported.

L-05

Warn about trapped funds

When sending ETH over the bridge, if the transaction on the other domain fails, the ETH may be permanently trapped. This is an intentional design decision, because complex transactions (that could fail) should be executed by contracts, which are assumed to know what they are doing. However, some aspects of the architecture may be surprising to most developers:

- in the L2-to-L1 direction, transactions may be relayed on L1 in a different order to when they were initiated, including two transactions from the same account. In some cases the failed transactions can be replayed, but the initial out-of-order transaction may make them unexecutable.
- developers are used to failed transactions costing them gas, but they typically retain the value of the transaction.

This may make cross-domain ETH transfers to contracts surprisingly risky. Consider including warning in the docstrings for the `bridgeETHTo` function and the corresponding legacy functions with the same risk.

L-06

Zero gas limit for refund transfer

If a token transfer cannot be finalized, the `StandardBridge` initiates a refund transfer with a gas limit of 0. This is reasonable when the refund originates on L2, because the L1 address can choose an arbitrary gas limit when finalizing the refund. However, now that the bridge treats

both domains symmetrically, it should handle refunds that originate on L1, where the chosen gas limit becomes part of the L2 transaction specification.

Consider choosing a non-zero gas limit that will cover a refund transaction in either direction.

L-07

Unlimited message length

The cross domain messaging infrastructure does not limit the size of the message, with the following implications:

- the base gas calculation only considers the bottom 32 bits of the `_message` length field and the bottom 32 bits of the final calculation, so it may be underestimated.
- the `OptimismPortal` will accept an arbitrarily long deposit message, which needs to be reproduced in the corresponding layer 2 transaction, which may be prohibitively expensive.

Consider limiting the acceptable size of cross-domain messages.

Notes & Additional Information

N-01

L2StandardBridge's ETH can be stolen

The `L2StandardBridge` contains a mechanism to let users bridge ETH to L1. However, it does not validate that the user sent sufficient ETH. If the bridge ever has any balance, possibly as the result of a donation, any user can initiate a deposit to send that ETH to their own L1 address.

Since the Optimism team is already aware of this issue, we are simply noting it for reference.

N-02

"TODO" statements in codebase

The codebase has several "TODO" statements, which can rot and lead to code that is less understandable, as well as leading to bugs not being fixed if they are never addressed. We have identified at least one which is a current security issue (see **[L2StandardBridge's ETH can be stolen]**)

Consider tracking these tasks in the project's issues backlog. The "TODO" statements can then be removed or updated to include the issue reference.

N-03

Complex ERC165 implementation

The `OptimismMintableERC20` directly calculates the interface identifiers that it supports. Similarly, the `StandardBridge` hardcodes constants that correspond to particular interface identifiers.

For clarity and simplicity, consider defining an interface contract and using solidity's `interfaceId` keyword to generate the identifier. Additionally, consider using the introspection utilities of the OpenZeppelin contracts library where appropriate.

N-04

Hardcoded values

Within the codebase there are a few spots where values are hard-coded and not adequately explained.

- In `OptimismPortal`'s `receive` function the value `100000` is used as a `_gasLimit` for a call to `depositTransaction`.
- In `OptimismPortal`'s `finalizeWithdrawalTransaction` function the value `20000` is added to the `_gasLimit` to ensure enough gas is being forwarded for the following call.
- In `L2ToL1MessagePasser`, the `receive` function passes `100000` into the `initiateWithdrawal` function.
- In `StandardBridge`, the `receive` function passes `200_000` into the `_initiateBridgeETH` function.
- In `CrossDomainMessenger`'s `relayMessage` function, the value `45000` and `40000` are hardcoded and used to ensure enough gas is included with calls.

Although in most cases, comments do explain the intent of these values enough, consider declaring `constant`s for them. This will give them meaningful names and consolidate sensitive values at the top of the files, making them easier to locate and spot-check in the future.

N-05

Misleading, missing or unclear comments or messages

Within the codebase, the following misleading comments were found:

- The `@return` comment for the `_deriveOutputRoot` function implies that the return value is a boolean, when it is actually just a hash.
- The error message in the `finalizeBridgeETH` function does not match the actual error.
- In both versions of the ERC20 deposit function, the `@param _l2Token` is an incomplete thought.
- The `_to` parameter of the `depositETHTo` and `depositERC20To` functions refers to a "withdrawal" instead of a "deposit".
- The comment regarding storage slots in `_verifyWithdrawalInclusion` does not specify which contract the storage slot is a part of.

- The comment above the `receive` function in `OptimismPortal` explaining that the function is only for EOAs does not explain why contracts should not call this function.
- Several functions in the `L1StandardBridge` contract claim to enforce a maximum length on the `data` parameter (for example, above the `depositETH` function), but the particular length is not specified and there doesn't appear to be any length restriction.
- The `Lib_WithdrawalVerifier` contract refers to an obsolete withdrawer predeploy contract rather than the `L2ToL1MessagePasser` contract.
- The `L2OutputOracle` contract's constructor is missing a `@param` comment for the `sequencer` parameter.
- The `OptimismMintableTokenFactory` contract's `createStandardL2Token` function is missing a `@return` statement.

N-06

Typographical errors

The following typographical errors were identified:

- In `OptimismPortal`:
  - "initated" should be "initiated"
  - "reentrency" should be "reentrancy"

N-07

Unused error

The `NotYetFinal` error in the `OptimismPortal` is unused. Consider removing it or emitting it where relevant.

Update: This issue is fixed at the final commit under review.

N-08

Unused functions

The `CrossDomainHashing` library has several functions that are not used outside the library:

- `L2TransactionHash`
- `sourceHash`
- `L2Transaction`

- `bytes32ToBytes`
- `getVersionedEncoding`

Consider removing them, or documenting their intended usage within the library.

N-09

Unverifiable Genesis

The `L2OutputOracle` is initialized with a genesis state, which should represent the state of Optimism chain at the time of deployment. However, there is no convenient way for a user to validate that this parameter was set correctly. In the interest of verifiability, consider emitting an event with the genesis state.

N-10

Incomplete Documentation

During the audit the Optimism team indicated that L2-to-L1 messages are not charged gas on L2, and they intend to introduce a metering mechanism to prevent L1 addresses from abusing this behavior. Consider explaining this design decision and its rationale in the deposit specification.

Conclusion

1 Critical and 2 High severity issues were found. Some suggestions to improve code cleanliness and quality were made.