Optimism: SystemConfig and Withdrawal Updates

Security Assessment

January 18, 2023

Prepared for:
Matthew Slipper
OP Labs

Prepared by: **Michael Colburn, Anish Naik, and David Pokora**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

OP Labs engaged Trail of Bits to review the security of its system configuration and two-step withdrawal workflows. From November 14 to December 9, 2022, a team of three consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic automated and manual testing of the target system and its codebase.

Summary of Findings

The audit uncovered one significant flaw that could impact system availability. Details on the sole finding are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	0
Low	0
Informational	0
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	1

Notable Findings

A significant flaw that impacts system confidentiality, integrity, or availability is described below.

- **TOB-OPTCW-1**
  An attacker could block the finalization of a user's withdrawal by resubmitting the proof for that withdrawal, as resubmission would reset the seven-day finalization period.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

David Pokora, Consultant
david.pokora@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
November 10, 2022	Pre-project kickoff call
November 21, 2022	Status update meeting #1
November 28, 2022	Status update meeting #2
December 6, 2022	Status update meeting #3
December 13, 2022	Delivery of report draft; report readout meeting
December 13, 2022	Addition of fix review
January 18, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of Optimism's system configuration and two-step withdrawal workflows. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the `op-node`'s parsing and validation of `ConfigUpdate` events sufficient to detect maliciously crafted or malformed events?
- Could an attacker bypass the access controls in the layer 1 (L1) `SystemConfig` contract and update the system configuration?
- Does the `op-geth` execution engine handle system configuration updates correctly, and do all transactions executed after an update use the new configuration?
- Could an attacker spoof or replay a withdrawal request to steal funds from the `OptimismPortal` contract?
- Could an attacker replay a withdrawal proof to cause unexpected behavior?
- Could an attacker finalize a withdrawal without first proving it?
- Could an attacker finalize a withdrawal before the completion of the finalization period?

Project Targets

The engagement involved a review and testing of the targets listed below.

optimism

Repository	https://github.com/ethereum-optimism/optimism
Versions	1bfe79f20b37a77c1158e7c5fdbff2ae109e0a1d ee96ff8585699b054c95c6ff4a2411ee9fedcc87 (contracts) 991120f6d2009b4a96580d7db3021694439c63f5 (Merkle trie)
Types	Golang and Solidity
Platforms	Linux, macOS, Windows, and the EVM

op-geth

Repository	https://github.com/ethereum-optimism/op-geth
Version	68bfa8b473bb8770216517f007b521fab41c5afa
Type	Golang
Platforms	Linux, macOS, and Windows

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **System configuration workflow.** The system configuration workflow is the process for updating the layer 2 (L2) block gas limit, L2 per-transaction gas costs, or batcher address in response to events emitted from L1. When the system configuration is updated, the L1 `SystemConfig` contract emits a `ConfigUpdate` event. The event is then picked up by the `op-node` during the L2 chain derivation process and added to the execution payload as part of an `L1InfoDeposit` transaction. Once `op-geth` executes that transaction, the new system configuration will apply to all subsequent L2 transactions. The components of this workflow are described below.
    - **`SystemConfig`.** The L1 `SystemConfig` contract contains three functions used to update the L2 gas limit, the per-transaction gas costs, and the address that is allowed to submit batched sequencer transactions, respectively. When one of those parameters is updated, the contract emits a `ConfigUpdate` event.
        - We manually reviewed the contract's access controls to ensure that only the owner of the contract can update the system configuration.
        - We manually reviewed the `initialize` function to ensure that it can be called only once.
        - We manually reviewed the bounds on updates to the gas limit to ensure that the `MINIMUM_GAS_LIMIT` required by the `ResourceMetering` contract is not violated.
        - We manually reviewed the contract's emission of events to verify that every parameter update is tied to a unique event.
    - **`op-node`.** When a `ConfigUpdate` event is emitted, the `op-node` parses it, validates it, and then adds it to the `PayloadAttributes` of the next L2 block to be executed by `op-geth`.
        - We manually reviewed the parsing of `ConfigUpdate` events to ensure that a malicious party could not manipulate the system configuration by spoofing an event.
        - We performed dynamic validation of the parsing of `ConfigUpdate` events, checking for edge cases in which a malformed or malicious `ConfigUpdate` event would be accepted as valid.

        - We manually reviewed the creation of `L1InfoDeposit` transactions to ensure that those transactions include all system configuration parameters. Additionally, we verified that all necessary values and the correct primitive types are set for the call to `L1Block.setL1BlockValues`.
        - We manually reviewed the marshaling and unmarshaling of `L1InfoDeposit` transactions to ensure that the operations preserve all system configuration parameters.
        - We performed dynamic validation of the marshaling and unmarshaling of `L1InfoDeposit` transactions to ensure that those operations are symmetric.
    - **`op-geth`.** The calldata of an `L1InfoDeposit` transaction executed by `op-geth` contains a call to `L1Block.setL1BlockValues`. This call applies L1 system configuration changes to L2. All subsequent transactions should then use the system configuration parameters set in that predeployed L2 contract.
        - We manually reviewed the `NewL1CostFunc` function to ensure that it uses the correct storage slots from the `L1Block` contract to calculate per-transaction gas costs.
        - We manually reviewed the gas cost calculations to ensure that the use of the `overhead` and `scalar` values adheres to the documentation.
        - We manually reviewed the state transition function to verify that `L1InfoDeposit` transactions do not carry a gas cost, that those transactions will still be included in the block if they fail, and that all non-deposit transactions use the `L1CostFunc` function to calculate gas costs.

  End-to-end system (E2E) test. We wrote an E2E dynamic test for the system configuration workflow to test situations such as gas price changes. For example, we checked whether setting a high gas price would block the processing of transactions and whether resetting a high gas price to a low one would cause transaction processing to resume.

- **Two-step withdrawal workflow.** To withdraw funds from L2 to L1, a user must first prove that he or she initiated a withdrawal on L2. Then, after the seven-day finalization period, the user can finalize the withdrawal to L1. This process is executed through calls to the L1 `OptimismPortal` contract.

- We manually reviewed the `proveWithdrawalTransaction` function to ensure that a malicious user could not spoof another user's withdrawal request and prevent that user from successfully withdrawing funds to L1. This led us to discover that a malicious user could repeatedly submit proofs for another user's withdrawal request, extending the finalization period and preventing that user from finalizing the withdrawal (TOB-OPTCW-1).

- We manually reviewed the `finalizeWithdrawalTransaction` function to ensure that withdrawals cannot be replayed, that the seven-day finalization period is enforced for all requests, and that the function cannot be reentered. We also verified that a malicious user could not manipulate another user's request and steal that user's funds.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. Specifically, we were unable to perform comprehensive testing of the changes made to the system during the engagement. Details on those changes are provided below.

- During the final week of the engagement, the OP Labs team delivered two updated commits to the `optimism` repository for us to review: `ee96ff8` and `991120f`. The first commit contains changes to the `ResourceMetering`, `L2ToL1MessagePasser`, and `L2OutputOracle` contracts. The second commit contains a refactored `MerkleTrie` library contract. We reviewed those changes but lacked the time to evaluate them in the context of the larger system.
- On the final day of the engagement, we received two additional pull requests (pull requests 4329 and 4337) that included minor changes to the `OptimismPortal` and `CrossDomainMessenger` contracts. We reviewed those changes but lacked the time to evaluate them in the context of the larger system.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We did not identify any issues related to the arithmetic used in either workflow. All of the arithmetic is straightforward and easy to reason about, and some of the gas-metering math has been refactored into a library contract to improve its readability and maintainability.	**Satisfactory**
Authentication / Access Controls	The authentication and access controls on system configuration updates and two-step withdrawals are sufficient. We did not identify any opportunities to bypass those access controls.	**Strong**
Complexity Management	The smart contracts are not overly complex, and their functionality is generally fairly simple and easy to understand. The expected behavior of functions in the `op-node` and `op-geth` code is described in code comments.	**Satisfactory**
Data Handling	The data validation performed in the system configuration workflow is sufficient to prevent the acceptance of a malicious or malformed event. However, there is no validation preventing one user from submitting proofs for another user's withdrawal request; thus, a malicious user could extend the escrow period for another user's withdrawal indefinitely by repeatedly submitting proofs (TOB-OPTCW-1).	**Moderate**
Documentation	The code is generally well commented, and the specification documentation in the repository includes operational and process details. However, that	**Moderate**

	documentation is outdated, and parts of it are incomplete. For example, the specification on the withdrawal process does not mention that the process of proving and finalizing a withdrawal has been split into two separate steps.	
Low-Level Manipulation	The codebase uses inline assembly only where necessary, such as in libraries for data serialization. However, certain uses of assembly lack documentation describing their exact purpose. The commits provided during the last week of the audit (described in the Project Coverage section) include additional assembly code, which is used to effectively bulk-delete elements of an array. We also identified one low-level call, which is safe from reentrancy and memory-expansion attacks.	**Satisfactory**
Memory Safety and Error Handling	We did not identify any memory safety concerns, and both workflows' error-handling mechanisms are generally appropriate.	**Strong**
Testing and Verification	We did not identify any issues that could have been caught through additional unit or dynamic testing. However, we recommend that the OP Labs team continue checking for gaps in its testing (like those in `system_config.go`) and fill any gaps it identifies with unit and fuzz tests. The team should also document all critical system properties (including those listed in appendix C) and perform a manual review and dynamic testing of those properties.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Ability to block withdrawals by resubmitting withdrawal proofs	Data Validation	High

Detailed Findings

1. Ability to block withdrawals by resubmitting withdrawal proofs	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-OPTCW-1
Target: `optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol`	

Description
The withdrawal of funds from L2 to L1 involves a two-step process on L1: a user (having initiated a transaction on L2) must prove that the withdrawal transaction is valid and then trigger its execution after the finalization period has elapsed. During the seven-day finalization period, any user can resubmit the withdrawal proof to the `OptimismPortal` contract to restart the finalization period. This means that by repeatedly resubmitting a proof, one user could effectively prevent another user from finalizing a withdrawal.

The code comments in the `proveWithdrawalTransaction` function mention a "replay check" in the finalization process that prevents a transaction from being finalized and executed more than once. However, there does not appear to be any protection against the replay of a proof for a pending withdrawal. When a proof is replayed, the stored timestamp is overwritten, effectively restarting the delay before the finalization of the withdrawal.

```
// Designate the withdrawalHash as proven by storing the `outputRoot`, `timestamp`,
// and `l2BlockNumber` in the `provenWithdrawals` mapping. A certain withdrawal
// can be proved multiple times and thus overwrite a previously stored
`ProvenWithdrawal`,
// but this is safe due to the replay check in `finalizeWithdrawalTransaction`.
provenWithdrawals[withdrawalHash] = ProvenWithdrawal({
    outputRoot: outputRoot,
    timestamp: uint128(block.timestamp),
    l2BlockNumber: uint128(_l2BlockNumber)
});
```

Figure 1.1: The `OptimismPortal.proveWithdrawalTransaction` method allows the resubmission of withdrawal proofs, which resets the timestamp of the finalization period. (`optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L204-L212`)

```
// Ensure that the withdrawal's finalization period has elapsed.
require(
    _isFinalizationPeriodElapsed(provenWithdrawal.timestamp),
    "OptimismPortal: proven withdrawal finalization period has not elapsed"
);
```

Figure 1.2: The `OptimismPortal.finalizeWithdrawalTransaction` method checks the timestamp.
(`optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L247-L251`)

Exploit Scenario

An Optimism user who has deposited funds on L2 wishes to perform a withdrawal. To do that, he calls the L1 `OptimismPortal.proveWithdrawalTransaction` method with arguments that prove the withdrawal request was processed on L2. An attacker captures the arguments passed to the method and repeatedly replays the request, calling the method directly. Because there is no check ensuring that the withdrawal transaction has not already been proved, the finalization period is reset with each call, blocking the user from withdrawing her funds.

Recommendations

Short term, add a `require` statement to `proveWithdrawalTransaction` to ensure that `provenWithdrawal[withdrawalHash].timestamp` is equal to zero. This will prevent withdrawal transactions from being proved multiple times.

Long term, be mindful of the risk of timing-based attacks such as front-running and replay attacks, which can produce undefined behavior within a system. Additionally, document the system's state machine in as much detail as possible to uncover any other opportunities for such attacks.

Summary of Recommendations

The Optimism optimistic rollup node and execution engine are works in progress with multiple planned iterations. Trail of Bits recommends that OP Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Continue to expand the documentation and specifications, and ensure that they are kept up to date.
- Continue to document important system invariants.
- Use the improved invariant documentation to identify any gaps in the system's testing, and then fill in those gaps.
- Continue reviewing the internal development processes to identify any improvements to the development life cycle that would increase the system's safety and resilience.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. System Configuration and Withdrawal Properties

This appendix lists some of the more critical properties that must hold for the system configuration workflow and two-step withdrawal process.

System Configuration

- All honest `ConfigUpdate` events should be successfully parsed by the `op-node` with no errors. The primary risk to successful parsing is a batcher address that is not correctly padded with 12 zeros (in version 0 of the `SystemConfig` contract).
- Any malicious or malformed `ConfigUpdate` events should be detected by the `op-node` during parsing.
- The `PayloadAttributes` for the next L2 block should always hold the latest system configuration values. The `PayloadAttributes` values can be cross-referenced with those set in the L1 `SystemConfig` contract.
- `L1InfoDeposit` transactions should cost zero gas.
- Deposit transactions should be included in the next L2 block regardless of whether they executed successfully.
- The execution of an `L1InfoDeposit` transaction should *never* revert. This property may require additional analysis and validation.
- The system configuration values in the L1 `SystemConfig` contract and the L2 `L1Block` contract should be the same after the successful execution of an `L1InfoDeposit` transaction.

Two-Step Withdrawal Process

- Withdrawals should not be replayable on L1.
- Withdrawal requests must be proved before their finalization.
- Unless the `outputRoot` value of the L2 `L2ToL1MessagePasser` contract has changed, it should not be possible to prove a withdrawal request more than once.
- Only one withdrawal request can be finalized for each transaction.
- Once a withdrawal has been proved, it cannot be finalized until the finalization period (indicated by `FINALIZATION_PERIOD_SECONDS`) has elapsed.

- All withdrawals must be initiated on L2. This can be tested indirectly through thorough testing of the `MerkleTrie` library.
- A withdrawal can be proved only if the storage root of the L2 block that holds the withdrawal request has been posted to the `L2OutputOracle` L1 contract.

D. Fix Review Results

On December 13, 2022, Trail of Bits reviewed the fix implemented by the OP Labs team for the issue identified in this report.

We reviewed the fix to determine its effectiveness in resolving the associated issue. For additional information, see the Detailed Fix Review Results.

ID	Title	Severity	Status
1	Ability to block withdrawals by resubmitting withdrawal proofs	High	Resolved

Detailed Fix Review Results

TOB-OPTCW-1: Ability to block withdrawals by resubmitting withdrawal proofs
Resolved. The OP Labs team updated the `proveWithdrawalTransaction` function; the function now allows a withdrawal request to be proved multiple times only if the `outputRoot` value of the `L2ToL1MessagePasser` contract has changed. (PR 4057)