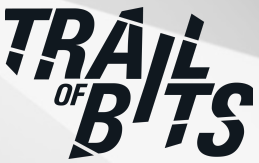

Optimism

Security Assessment

November 10, 2022

Prepared for:
Matthew Slipper
Optimism

Prepared by: **Michael Colburn 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

Optimism engaged Trail of Bits to review the testing strategy of its Optimistic Rollup engine, Optimistic L2 `go-ethereum` fork, and Bedrock smart contracts. From August 22 to September 23, 2022, a team of two consultants conducted a 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 automated analysis against the project targets, as described in the Automated Testing section of the report.

Summary of Findings

The audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of this finding is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	1

Notable Finding

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

- **TOB-OPTEST-1**
  The `GasPriceOracle` contract deployed to L2, which is used to update L1 costs charged on L2, could be misconfigured in a way that sets gas prices high enough to prevent transactions from being processed. Certain misconfigurations may even block future attempts to reset the `GasPriceOracle`.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineers were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

David Pokora, Consultant
david.pokora@trailofbits.com

Project Timeline

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

Date	Event
August 18, 2022	Pre-project kickoff call
August 29, 2022	Status update meeting #1
September 6, 2022	Status update meeting #2
September 19, 2022	Status update meeting #2
September 26, 2022	Delivery of report draft
September 26, 2022	Report readout meeting
November 10, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Optimism team's `op-geth`, `op-node`, and Bedrock smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Which invariants across the project targets should be tested to best ensure the targets' security?
- Does the existing testing methodology contain gaps that could cause tests to miss security-critical issues?
- Could any of the existing unit tests be better served with an accompanying fuzz test?
- How could the Slither API be used to statically analyze smart contracts within Optimism?
- Could the testability of certain targets be improved in any way?
- Generally, does the system behave as expected when tested under various conditions?
    - Are blocks produced in a timely fashion?
    - Are access controls in place to prevent users from submitting deposit transactions through the L2 RPC endpoint?
    - Does the system work end to end? Do the individual components of `op-geth` and `op-node` behave as expected?
    - Are data structures serialized and deserialized without data loss?
    - Are balances and fees charged as expected?
    - Does the system behave as expected when forks are encountered?

Project Targets

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

Optimism (op-node, op-e2e, Bedrock contracts)

Repository	https://github.com/ethereum-optimism/optimism
Version	`b31d35b67755479645dd150e7cc8c6710f0b4a56`
Types	Golang, Solidity
Platforms	Linux, macOS, Windows, Ethereum

Optimistic Execution Engine (op-geth)

Repository	https://github.com/ethereum-optimism/reference-optimistic-geth
Version	`a68e5aa189e14fde92cec03c1abd98cc7f0db263`
Types	Golang, Solidity
Platforms	Linux, macOS, 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 included the following:

- Documentation of invariants within the `OptimismPortal` Bedrock smart contract and its relevant subcomponents, such as `ResourceMetering`
- Documentation of invariants across the `op-node` subproject of the `optimism` monorepo related to smart contracts and `op-node` functionality
- Documentation of invariants across the `op-geth` project
- Verification of various invariants throughout the project's unit, integration, and property tests, which resulted in the following:
    - Testing the round-trip serialization of objects across `op-node` and `op-geth` did not result in the discovery of any new vulnerabilities.
    - Verification of the L1/L2 gas fee computation revealed that the `GasPriceOracle` contract could be misconfigured in a way that sets unreasonably high transaction fees, preventing L2 transaction submissions from being accepted (TOB-OPTEST-1).
    - The block production and fee computations were not tested for all potential configuration permutations of `op-node` and `op-geth`; nonetheless, while running tests during the audit, we did not find these computations to be problematic.
    - The access controls intended to prevent users from submitting deposit transactions through the L2 RPC endpoint were found to be effective.
    - Fuzz testing the system's data structures, such as `go-ethereum` transactions (including the new deposit transaction type) and `BatchData` in the `op-node` subproject, found that they are encoded/decoded successfully without data loss.
    - Verification of transfer-related invariants found that the system handles transfers as expected: attempting to transfer more ETH on L2 than an account owner holds results in errors, while attempting to transfer less than an account owner holds results in the expected transfer of the requested ETH.

- Verification of the `OptimismPortal` contract's deposit routines and inherited contract methods found that they behave as expected in terms of burning ETH, hashing, constructing proofs, aliasing addresses for deposits, enforcing gas metering, and more.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We were unable to perform additional testing of system interactions in a concurrent fashion (e.g., cascading deposits/withdrawals asynchronously to ensure the state machine behaves as expected).
- Not all invariants across the system could be documented.

We recommend that the Optimism team take the following steps to mitigate coverage limitations in future audits and testing:

- Further derive invariants from any off-chain smart contract tests and follow up on additional invariants related to the operation of transaction pools, block construction, P2P operations, payload attribute derivation, and fork conditions.
- Continue writing fuzz tests for all existing unit tests that do not have an accompanying fuzz test. This will ensure that additional conditions or values that are hard-coded within the unit tests undergo additional scrutiny.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation
`go test`	A first-party unit- and property-testing framework for Golang

Test Results

The results of testing the system properties that we enumerated during the audit are detailed below.

contracts-bedrock

This section details the property tests written for the `contracts-bedrock` project located in the `optimism` monorepo under the `packages/contracts-bedrock/` directory.

OptimismPortal: This section details security invariants drawn from the `OptimismPortal` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
The `initialize()` function cannot be called more than once.	**Property test (Echidna):** `echidna_never_initialize_twice`	Passed

The contract cannot be deployed with an invalid `L2OutputOracle` contract address (such as the zero address) and continue to function as intended.	-	**Not Tested**
The amount of ETH taken by the `depositTransaction()` function always equals or exceeds the amount to be minted on L2.	**Property test (Echidna):** `echidna_mint_less_than_taken`	**Passed**
A nonzero `_to` address cannot be supplied to `depositTransaction()` when `_isCreation` is set to `true`.	**Property test (Echidna):** `echidna_never_nonzero_to_creation_deposit`	**Passed**
	Unit test (Slither): `test_deposit_transaction_integrity`	
Gas metering always burns at least the gas cost calculated from the `_gasLimit` argument when `depositTransaction()` is called.	-	**Not Tested**
The `from` parameter in the `TransactionDeposited` event emitted by `depositTransaction()` is aliased if the caller is a contract address.	**Property test (Echidna):** `echidna_alias_from_contract_deposit`	**Passed**
The `from` parameter in the `TransactionDeposited` event emitted by `depositTransaction()` is not aliased if the caller is an externally owned address.	**Property test (Echidna):** `echidna_no_alias_from_EOA_deposit`	**Passed**
Calling the `L1CrossDomainMessenger.sendMessage`	-	**Not Tested**

function results in the same operation as calling `depositTransaction` directly with similar parameters.		
Calls to the `finalizeWithdrawalTransaction` function cannot reenter the function.	-	**Not Tested**
A withdrawal cannot be finalized until after the finalization period has concluded.	**-**	**Not Tested**
A withdrawal can be finalized only once.	-	**Not Tested**
Withdrawal finalization fails if the L2 oracle has no output root for the relevant block number.	-	**Not Tested**
Withdrawal finalization fails if the expected output root cannot be generated from the provided proof.	-	**Not Tested**
Withdrawal finalization fails if the withdrawal request is not accompanied by a valid inclusion proof.	-	**Not Tested**
A gas cost of at least the sum of `_tx.gasLimit` and `FINALIZE_GAS_BUFFER` (a weak lower bound) is required for calls to `finalizeWithdrawalTransaction`.	-	**Not Tested**

ResourceMetering: This section details security invariants drawn from the `ResourceMetering` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
Given a block that uses more gas than the `TARGET_RESOURCE_LIMIT` value, the `basefee` used in the immediate next block is greater than that of the given block.	**Property test (Echidna):** `echidna_high_usage_raise_basefee`	Passed
Given a block that uses less gas than the `TARGET_RESOURCE_LIMIT` value, the `basefee` used in the immediate next block is less than that of the given block (or is equal to `MINIMUM_BASE_FEE`).	**Property test (Echidna):** `echidna_low_usage_lower_basefee`	Passed
The `basefee` of a given block is never less than `MINIMUM_BASE_FEE`.	**Property test (Echidna):** `echidna_never_below_min_basefee`	Passed
`prevBoughtGas` does not exceed `MAX_RESOURCE_LIMIT`.	**Property test (Echidna):** `echidna_never_above_max_gas_limit`	Passed
Given two or more empty blocks, the reduction of the `basefee` is greater than the `basefee` reduction for one or fewer empty blocks (down to `MINIMUM_BASE_FEE`).	-	Not Tested
A block's `basefee` cannot increase by more than a factor of `(1+1/BASE_FEE_MAX_CHANGE_DENOMINATOR)` multiplied by the immediately preceding block's `basefee`.	**Property test (Echidna):** `echidna_never_exceed_max_increase`	Passed
A block's `basefee` cannot decrease by more than a factor of `(1-1/BASE_FEE_MAX_CHANGE_DENOMINATOR)` multiplied by the immediately preceding	**Property test (Echidna):** `echidna_never_exceed_`	Passed

block's `basefee`.	`max_decrease`	

L2OutputOracle: This section details security invariants drawn from the `L2OutputOracle` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
L2 block numbers are monotonically increasing.	-	**Not Tested**
A proposal's block number cannot correspond to a timestamp in the future.	-	**Not Tested**
A proposal with an empty output root is invalid.	**-**	**Not Tested**

AddressAliasHelper: This section details security invariants drawn from the `AddressAliasHelper` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
The L1-to-L2 address aliasing process is able to encode any address and decode the original address without failure.	**Property test (Echidna):** `echidna_round_trip_aliasing`	**Passed**

Burn: This section details security invariants drawn from the `Burn` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
Calls to `eth` to burn ETH remove exactly `_value` ETH from the calling contract.	**Property test (Echidna):**	**Passed**

	`echidna_burn_eth`	
Calls to `gas` to burn gas burn at minimum the amount of gas passed as a parameter.	**Property test (Echidna):** `echidna_burn_gas`	**Passed**

Encoding: This section details security invariants drawn from the `Encoding` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
Versioned nonce encoding and decoding operations succeed for all inputs and are inverse operations of each other.	**Property test (Echidna):** `echidna_round_trip_encoding`	**Passed**

Hashing: This section details security invariants drawn from the `Hashing` smart contract, the tests they underwent, and the results of this testing.

Property	Test	Result
Calls to `hashCrossDomainMessage` never succeed when an invalid nonce (i.e., one whose version is greater than 1) is passed as an argument.	**Property test (Echidna):** `echidna_hash_xdomain_msg_high_version`	**Passed**
Calling `hashCrossDomainMessage` with a version `0` nonce results in the same operation as calling `hashCrossDomainMessageV0` directly.	**Property test (Echidna):** `echidna_hash_xdomain_msg_0`	**Passed**
Calling `hashCrossDomainMessage` with a version `1` nonce results in the same operation as calling `hashCrossDomainMessageV1` directly.	**Property test (Echidna):** `echidna_hash_xdomain_msg_1`	**Passed**

op-node

This section details the property tests that we wrote for the `op-node` subproject located in the `optimism` monorepo under the `op-node/` directory. All unit and fuzz tests are written for use with `go test`.

Property	Test	Result
Various `op-node` configurations cannot introduce undefined behavior into the system (such as the inability to finalize deposits or withdrawals).	-	Not Tested
L2 block creation fails if the new L2 block (with a timestamp of the current L2 block header's timestamp summed with the `BlockTime` value) has a timestamp less than the L1 origin block that it is derived from.	**Property test:** `FuzzRejectCreateBlockBadTimestamp`	Passed
	Unit test: `TestRejectCreateBlockBadTimestamp`	Passed
Logs other than the `TransactionDeposited` log do not have inadvertent effects on the system.	**Property test:** `FuzzDeriveDepositsRoundTrip`	Passed
	Unit test: `TestDeriveUserDeposits`	Passed
Deposit logs can be encoded and decoded with their original values intact.	**Property test:** `FuzzDeriveDepositsRoundTrip`	Passed

	Unit test: `TestDeriveUserDeposits`	Passed
An incorrectly parsed `TransactionDeposited` log for a single deposit does not affect the processing of other deposits.	-	Not Tested
Unknown `DEPOSIT_VERSION` values specified by `TransactionDeposited` events are rejected.	**Property test:** `FuzzDeriveDepositsBadVersion`	Passed
Deposits are not derived from failed transactions on L1.	**Property test:** `FuzzDeriveDepositsRoundTrip`	Passed
	Unit test: `TestDeriveUserDeposits`	Passed
Failures in the system do not cause funds deposited after such failures to be lost.	-	Not Tested
Deposit transactions can be derived from `L1Info` structures.	**Property test:** `FuzzParseL1InfoDepositTxDataValid`	Passed
	Unit test: `TestParseL1InfoDeposi`	Passed

	`tTxData`	
Deriving `L1Info` data from deposit transaction data of an invalid length always fails.	**Property test:** `FuzzParseL1InfoDepositTxDataBadLength`	Passed
	Unit test: `TestParseL1InfoDepositTxData`	Passed
The correct L1 origin is always selected when the `createNewL2Block` function constructs an L2 block.	-	Not Tested
Encoding and decoding the `BatchData` struct preserves the struct's original values.	**Property test:** `FuzzBatchRoundTrip`	Passed
	Unit test: `TestBatchRoundTrip`	Passed
The `BatchQueue` struct ignores batches with a timestamp prior to the safe L2 header's timestamp during the stepping process.	-	Not Tested
The `BatchQueue` struct immediately updates the `BatchQueueOutput` variable with `BatchData` submitted with consecutive timestamps after the safe L2 header's timestamp.	**Unit test:** `TestBatchQueueEager`	Passed
The `BatchQueue` progress is open if the previous progress was open before the current	**Unit test:**	Passed

progress started and if the current progress is closed before the stepping process.	`TestBatchQueueFull`	
The `BatchQueue` progress is closed if the previous progress was closed before the stepping process.	**Unit test:** `TestBatchQueueFull`	**Passed**
Batches are considered invalid if their timestamps are outside of the minimum/maximum L2 time window.	**Unit test:** `TestValidBatch`	**Passed**
Batches are considered invalid if they were tagged with an epoch number that is not the current one.	**Unit test:** `TestValidBatch`	**Passed**
Batches are considered invalid if their timestamps are not aligned to the block time step.	**Unit test:** `TestValidBatch`	**Passed**
Batches are considered invalid if they contain a `DepositTx` type transaction.	**Unit test:** `TestValidBatch`	**Passed**
Batches are considered invalid if they do not contain any transactions.	**Unit test:** `TestValidBatch`	**Passed**
Batches are considered invalid if their epoch hash does not match the current one.	**Unit test:** `TestValidBatch`	**Passed**
A batch is dropped if a reset rolls back a full sequence window or if the batch's timestamp otherwise precedes the safe L2 header.	-	**Not Tested**

op-geth

This section details the property tests that we wrote for the `op-geth` project. Some tests exist within the `op-geth` repository directly, while others exist in `op-e2e` within the `optimism` monorepo. The location of each test is indicated in parentheses in the "Test" column.

Property	Test	Result
`op-geth` configurations with different values for parameters, such as sequence windows and other time durations, do not introduce undefined behavior into the system, such as the inability to finalize deposits or withdrawals.	-	Not Tested
L1 costs set in the `GasPriceOracle` contract are appropriately enforced in L2 transaction fees.	-	Not Tested
L1 fees are appropriately awarded to the `BaseFeeRecipient` address.	-	Not Tested
The nonce of a deposit sender is incremented on L2, regardless of whether an L1 deposit transaction receipt reported a failure status.	**Unit test (op-e2e):** `TestMintOnRevertedDeposit`	Passed
When L2 processes a deposit transaction, failure to transfer ETH to another address does not result in the unexpected loss of ETH minted during the transaction.	**Unit test (op-e2e):** `TestMintOnRevertedDeposit`	Passed
The L1 costs set in the `GasPriceOracle` contract cannot be incorrectly set to values that prevent the contract from being further updated.	**Unit test (op-e2e):** `TestGasPriceOracleFeeUpdates`	Failed

The L1 costs set in the `GasPriceOracle` contract cannot be incorrectly set to values that prevent any transactions from being processed on L2.	**Unit test (op-e2e):** `TestGasPriceOracleFeesL2Lock`	Failed
With the addition of the `DepositTx` type transaction, transaction serialization is not prone to data loss or misinterpretation.	**Property test (op-geth):** FuzzTransactionMarshallingRoundTrip	Passed
The L2 sequencer/verifier does not accept `DepositTx` transaction types submitted through the RPC endpoint.	**Unit test (op-e2e):** `TestL2SequencerRPCDepositTx`	Passed
RPC endpoints appropriately enforce size limits when various deposit transactions are included.	-	Not Tested
In the event of an L1 reorganization, L2 makes appropriate state updates, such as to account balances.	-	Not Tested
The L2 output submitter is updated after an L2 block is committed.	**Unit test (op-e2e):** `TestL2OutputSubmitter`	Passed
The L2 output submitter is resistant to reorganization.	-	Not Tested
L2 nodes sync blocks from other nodes before they are confirmed on L1.	**Unit test (op-e2e):** `TestSystemMockP2P`	Passed
The transaction pool appropriately enforces the `NoTxPool` flag and pushes forced	-	Not Tested

transactions through as expected.		
In the event of a large number of forced transactions, the transaction pool continues to operate, and standard Ethereum transactions in the transaction pool do not expire or become stale.	-	**Not Tested**
Deposit transactions that fail to transfer value on L2 (e.g., because of insufficient balance) do not negatively affect valid deposit transactions.	**Unit test (op-e2e):** `TestMixedDepositValidity`	**Passed**
Failed withdrawal transactions do not prevent valid withdrawal transactions from executing (end to end).	**-**	**Not Tested**
Withdrawals that specify an invalid timestamp, such as one for which an L2 output root does not exist or is not `FINALIZATION_PERIOD` seconds old, are rejected.	**Unit test (op-e2e):** `TestMixedWithdrawalValidity`	**Passed**
The `sender`, `target`, `message`, `value`, and `gasLimit` fields cannot be modified in a withdrawal request without failure.	**Unit test (op-e2e):** `TestMixedWithdrawalValidity`	**Passed**
Failed deposits on L1 that are then reorganized to be successful deposits are handled appropriately by L2.	-	**Not Tested**
Different verifiers are not able to derive different fees.	-	**Not Tested**

Summary of Findings

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

ID	Title	Type	Severity
1	Risk of misconfigured GasPriceOracle state variables that can lock L2	Data Validation	**Undetermined**

Detailed Findings

1. Risk of misconfigured GasPriceOracle state variables that can lock L2	
Severity: **Undetermined**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-OPTEST-1
Target: `optimism/packages/contracts/L2/predeploys/OVM_GasPriceOracle.sol`, `op-geth/core/rollup_l1_cost.go`	

Description

When bootstrapping the L2 network operated by `op-geth`, the `GasPriceOracle` contract is pre-deployed to L2, and its contract state variables are used to specify the L1 costs to be charged on L2. Three state variables are used to compute the costs—`decimals`, `overhead`, and `scalar`—which can be updated through transactions sent to the node.

However, these state variables could be misconfigured in a way that sets gas prices high enough to prevent transactions from being processed. For example, if `overhead` were set to the maximum value, a 256-bit unsigned integer, the subsequent transactions would not be accepted.

In an end-to-end test of the above example, contract bindings used in `op-e2e` tests (such as the `GasPriceOracle` bindings used to update the state variables) were no longer able to make subsequent transactions/updates, as calls to `SetOverhead` or `SetDecimals` resulted in a deadlock. Sending a transaction directly through the RPC client did not produce a transaction receipt that could be fetched.

Recommendations

Short term, implement checks to ensure that `GasPriceOracle` parameters can be updated if fee parameters were previously misconfigured. This could be achieved by adding an exception to `GasPriceOracle` fees when the contract owner calls methods within the contract or by setting a maximum fee cap.

Long term, develop operational procedures to ensure the system is not deployed in or otherwise entered into an unexpected state as a result of operator actions.

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. Testing the Project Targets

This section describes how to execute the tests that Trail of Bits ran during the engagement.

Echidna Fuzz Tests (Bedrock Contracts):

Echidna is an Ethereum smart contract fuzzer that allows users to write on-chain property tests to verify the expected states of their applications.

We provided Git patches for each project target alongside the report containing the tests generated during the course of the assessment. To prepare the environment for fuzz testing, we removed the `optimism/packages/contracts-bedrock/contracts/test` directory, as it contained unlinked libraries that are incompatible with Echidna in a default deployment scheme. Additionally, we updated the Hardhat and Foundry compilation configurations so that they did not strip bytecode hash metadata, which is required by Echidna to match deployed contracts.

To run the Echidna fuzz tests, take the following steps:

- To compile the project, invoke Hardhat in the `optimism/packages/contracts-bedrock` directory by running the following commands:
    - `npx hardhat clean`
    - `npx hardhat compile`
- Invoke Echidna against a contract containing property tests by running the following command. This will tell Echidna to use the previously created compilation and to target the provided contract in a fuzzing campaign:

  `echidna-test --contract <contract_name> --crytic-args --hardhat-ignore-compile .`

Go Test Tests (op-node, op-e2e, op-geth)

The `go test` command invokes unit, integration, and fuzz tests written using Golang's native `testing` package. We produced unit and fuzz tests for `op-node` and `op-geth`, as described in the Automated Testing table.

- To run the end-to-end tests within the `optimism/op-e2e` directory and the `op-node` unit tests within the `optimism/op-node` directory alongside any existing tests, run the following command from the respective directories:

    - `go test -v ./...`

- To run individual unit tests, use the following command instead:
    - `go test -v -run <TestName>`
- To run fuzz tests written for `op-node` and `op-geth`, run the following command from the directory containing the test file. The tests will run until the process is killed or interrupted:
    - `go test -v -fuzz <TestName>`

C. Recommendations for Improving Testability

This section includes recommendations for improving the testability of the codebase.

Solidity Smart Contract Testing

- To use on-chain property fuzzers, such as Echidna, property tests are written in Solidity. However, on-chain property tests cannot access various aspects of the chain state or results. Therefore, we recommend designing functionality in a way that allows the results to be tested on-chain.

- To ensure that all of the routines in a given contract can be tested, verify that the relevant inputs, state changes, and outputs can be captured by a separate method in the contract. For instance, emitted events cannot be queried on-chain; they can be verified only off-chain.

    - If a test intends to verify values within an emitted event, consider splitting the relevant method into a helper function that returns the values rather than emitting them in an event. The original method could use this helper function to perform the underlying work and later emit the output data in an event itself, while test methods could target the helper directly to verify output methods.

        - For example, one could split the `OptimismPortal.depositTransaction` logic into a helper method that returns values rather than simply emitting a log, as these values can be validated by a test using the helper method. Alternatively, one could wrap the emitted event in a separate virtual function that can be overridden by a test contract derived from `OptimismPortal` so that it can capture these values.

- Ensure that the contracts can be easily deployed from a separate contract where possible. Echidna deploys compiled contracts with no constructor arguments and executes transactions against publicly accessible methods in an attempt to produce state changes.

    - For contracts that take constructor arguments, consider either creating a deriving contract that satisfies the constructor arguments with hard-coded values or creating a separate contract to deploy such contracts with the appropriate constructor arguments used for testing.

        - Carefully consider the tested contracts' code composition when making such decisions.

    - For complex contract developments, consider using Etheno alongside Echidna.

- Consider integrating the project's Echidna fuzz tests to the project's CI/CD pipeline, possibly through the use of Echidna GitHub Actions.

    - Leverage the `test-limit` configuration variable to limit the duration of the fuzzing campaign in the CI process.

    - Ensure that the fuzz tests are run at regular intervals. Tests that pass do not necessarily indicate a lack of vulnerabilities. The constraints required to violate a property test may not be found in one run, but the fuzzer may catch the latent issue in a later run.

- Consider adding rules to Slither's existing set of static analysis rules by plugging detectors or other custom scripts into Slither's detector API. To observe how the Slither API can be used to verify the integrity of a codebase, run the following command from the `optimism/packages/contracts-bedrock` directory:

    ```
    python3 ./slither_api_example.py
    ```

    As a proof of concept, this script discovers all Echidna property tests and the contracts they live within, specifies the contracts that they immediately inherit from, and performs a check against `OptimismPortal.depositTransaction` to ensure that no high-level calls were added/removed and that an `if` statement exists for `_isCreation` that contains only a `require` statement comparing `_to` to `address(0)`.

    Note that the test against `OptimismPortal` was written to check every AST node and its underlying IR to show how Slither can be used to iterate over every statement or expression in a method and detect specific patterns or variables across multiple expressions. However, the test could be simplified to instead check the source text for specific segments.

    The use of the Slither API can enable the CI/CD pipeline to catch issues that arise from changes mistakenly introduced by developers, such as changes that violate some property of a given method. For instance, a Slither script could be written to differentiate between internal and external calls to ensure that no external calls are performed in a given method.

L1/L2/op-node Testing

- We recommend creating an API that simplifies the project's end-to-end testing.:

    - The API should provide methods to initialize accounts with different balances on L1 and L2 and provide a simplified test account structure with the key path, private key, and `TransactOpts`, alongside other account properties.
    - The API should ensure that timeout-based tests do not fail simply because the timeouts are set too low. For example, throughout the `op-e2e` tests, various statements wait one second for a block to propagate, which may not be long enough. Increasing the timeout may reduce the likelihood of false-positive test failures for slower systems (such as the CI process).
    - The API should include methods to execute actions such as sending deposit transactions, sending withdrawal requests, creating arbitrary transfer transactions on L1/L2, and causing fork conditions on L1/L2. The testing harness could automatically execute these actions and update expected values, such as expected balances/nonces on L1/L2, which are automatically asserted at the end of the test alongside any conditions that the tester asserts within the test immediately.
        - Simulating fork conditions may require support for rolling back previous actions (and their changes to expected values).
        - Ideally, the system should allow these actions to be invoked in parallel (from goroutines) to simulate typical network behavior (e.g., multiple L1 deposit transactions submitted at once).
        - The system should ensure that blocks produced in tests simulate conditions for multiple Optimism system–related transactions included both simultaneously and individually in a single block.
        - Consider writing all relevant end-to-end tests so that they can be run against various system configurations, such as differing sequence windows and gas fees.
    - Finally, the API should ensure that the same test can be easily rerun with differing system configurations; this will ensure that the system does not exhibit undefined behavior as a result of edge cases arising from various configurations.
- For unit tests that depend on the result of processing certain data and making sure routines succeed or fail as expected, ensure that as many permutations of the input data as possible are tested. Review the existing unit tests to identify hard-coded values that would increase test coverage if they were randomized or fuzzed values instead.

- For example, some unit tests within `op-node` depend on the `MarshalDepositLogEvent` method to produce a deposit event that is used as input to test deposit derivation functions. By reviewing this method, we can see that deposit versions are hard-coded to valid values. Modifying the unit tests' helper methods to accept additional fields (such as the deposit version field) will add the flexibility necessary to test additional invariants (such as whether deposit logs with an invalid version produce derived deposits).