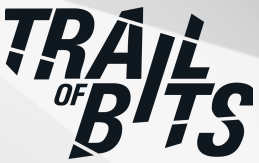

Optimism: Rollup Node and Execution Engine

Security Assessment

May 18, 2022

Prepared for:
Matthew Slipper
Optimism

Prepared by: **David Pokora, Simone Monica, Anish Naik, and Justin Jacob**

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 80+ 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 security of its optimistic rollup node and execution engine. From April 11 to April 29, 2022, a team of four consultants conducted a security review of the client-provided source code, with six 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 target system, including access to the source code and relevant documentation.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Low	3
Informational	4
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	3
Denial of Service	2
Error Reporting	2
Timing	1
Undefined Behavior	4

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-OPT-1**
  A reentrancy vulnerability in the `WithdrawalsRelay` smart contract allows attackers to steal funds.

- **TOB-OPT-3**
  Deposit transactions submitted maliciously may prevent other deposit transactions from being processed, resulting in a loss of funds.

- **TOB-OPT-5**
  Connections between the execution engine and the rollup node through the engine API are not authenticated.

- **TOB-OPT-6**
  An incorrect address value in the rollup node will prevent the `L1Block` contract from being updated with L1 block values.

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:

David Pokora, Consultant
david.pokora@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Project Timeline

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

Date	Event
April 7, 2022	Pre-project kickoff call
April 18, 2022	Status update meeting #1
April 25, 2022	Status update meeting #2
May 2, 2022	Delivery of report draft and report readout meeting
May 18, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the optimistic rollup node and execution engine. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a maliciously crafted transaction on L1 cause the rollup node to crash?
- Is the system robust enough to manage most cases of blockchain reorganizations?
- Could the system's processing of deposits or withdrawals cause funds to be trapped on L2?
- Can the rollup node be tricked into deriving deposits that were not actually made?
- Are there any logical flaws in the rollup node and execution engine smart contracts?
- Can attackers withdraw funds that they do not own from L2?
- Does the system appropriately track the protocol's state across L1 and L2?
- Were any vulnerabilities introduced into the execution engine due to changes made to the underlying `go-ethereum` implementation that it was forked from? Can the new deposit transaction types cause panics in existing RPC handlers?
- Is the rollup node prone to log injection that may result in a phishing attack against rollup node operators?

Project Targets

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

Optimistic Rollup Node

Repository	https://github.com/ethereum-optimism/optimistic-specs
Version	`05136a32b9828b595dde47f767218dec53f19aa4`
Types	Golang, Solidity
Platforms	Linux, macOS, Windows, Solidity

After the audit, Optimism relocated the optimistic rollup node code to a monorepo.

Optimistic Execution Engine

Repository	https://github.com/ethereum-optimism/reference-optimistic-geth
Version	`a7423f3a3167d20e93b6d60e648fbe9fec17f380`
Types	Golang, Solidity
Platforms	Linux, macOS, Windows, Solidity

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:

- We reviewed the smart contract endpoints for withdrawals and deposits. This review uncovered a reentrancy vulnerability within the `WithdrawalsRelay` contract that could allow an attacker to steal funds from the system (TOB-OPT-1).
- We reviewed the overall Optimism state machine. We found that validation of administrator addresses and signatures in the L2 `CheckpointOracle` contract is insufficient (TOB-OPT-2). Additionally, we found that an incorrect address value in the `L1Block` contract may prevent the contract's state from being updated (TOB-OPT-6).
- We assessed the data validation performed within the rollup node. This review uncovered two concerns regarding insufficient validation of deposit transaction parameters (TOB-OPT-3, TOB-OPT-4).
- We reviewed the L2 output submitter component. This review did not uncover any concerns.
- We investigated potential cases of log injection. This investigation did not result in any findings.
- We reviewed the authorization and access controls throughout the system. This review revealed that the execution engine does not employ authentication when leveraging the engine API in communication with the rollup node (TOB-OPT-5).
- We reviewed the error handling throughout the rollup node and execution engine. This review uncovered an issue regarding incorrect error handling when creating an L2 block (TOB-OPT-8).

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:

- A number of code paths and features were not implemented at the time of the audit; therefore, mechanisms such as the rollup node and execution engine's authentication were not evaluated.

- We reviewed the system's handling of blockchain reorganization, but given the complexity of these codepaths, we recommend additional scrutiny of these mechanisms.

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 issues related to the arithmetic used in the system. The accounting performed within the L1 and L2 code is not overly complex or difficult to evaluate.	**Strong**
Auditing	We identified several "TODO" comments to add appropriate error handling during our review of the rollup node and execution engine's auditing and logging mechanisms. Additionally, we discovered an issue regarding incorrect error handling within the rollup node (TOB-OPT-8). Otherwise, error handling appears to be adequate in most cases.	**Moderate**
Authentication / Access Controls	The authentication and access controls in the system are generally appropriate. However, authentication of connections between the rollup node and the execution engine API has not yet been implemented (TOB-OPT-5).	**Moderate**
Complexity Management	The smart contracts are not overly complex, and their functionality is generally fairly simple and easy to understand. Functions within the rollup node and execution engine are well commented to describe expected behavior.	**Satisfactory**
Configuration	The configuration of the rollup node and execution engine seems appropriate. We did not identify any configuration issues.	**Strong**

Cryptography and Key Management	Cryptography is handled by the L1 Ethereum and L2 optimistic execution engines natively. The rollup node does not handle key management, as it speaks directly with the execution engine through the engine API, which is currently unauthenticated.	**Not Applicable**
Data Handling	Data validation throughout the codebases is appropriate in most cases. However, we identified trivial cases of unvalidated deposit transaction event parameters that could lead to a loss of funds (TOB-OPT-3) and a case of ineffective validation of deposit transaction event parameters (TOB-OPT-4). Most notably, we discovered a reentrancy vulnerability in the `WithdrawalsRelay` contract that could allow an attacker to steal funds (TOB-OPT-1).	**Weak**
Documentation	The code is generally well commented. The specification documentation within the repository includes operational and process details. However, this documentation often lacks information regarding security invariants, and some language could be made more clear (TOB-OPT-10).	**Moderate**
Front-Running Resistance	We did not fully evaluate the system for front-running because smart contracts were considered a lower priority for this audit; however, in the code under audit, account ownership is asserted through signed transactions, and data that can be submitted from any account is typically signed or designated for a specific receiver, which cannot be tampered with.	**Satisfactory**
Memory Safety and Error Handling	We did not identify any memory safety concerns. Error handling is generally appropriate throughout the codebase, although it is not yet implemented in some areas (indicated by "TODO" comments), including areas vulnerable to edge cases. We did identify one case of incorrect error handling in the rollup node (TOB-OPT-8) and a high-severity issue related to error handling: if a user submits a deposit transaction in the same block as a malicious transaction, and that block has insufficient	**Moderate**

	error handling, the user could lose funds (TOB-OPT-3).	
Testing and Verification	Unit tests for major components exist throughout the rollup node codebase. The execution engine is tested mostly through a smaller set of end-to-end integration tests against a vanilla L1 Ethereum node, the rollup node, and the execution engine. These tests could be expanded to be able to catch issues such as TOB-OPT-6, which was overlooked due to the use of a hard-coded address. Additional tests and fuzz testing methodology can uncover deep-rooted issues like this.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Risk of theft due to reentrancy vulnerability in WithdrawalsRelay	Timing	High
2	Missing zero address checks in L2 CheckpointOracle	Data Validation	Low
3	Possible failure to parse deposit transactions due to incorrect gasLimit type	Denial of Service	High
4	Incorrect data validation when parsing transaction logs	Data Validation	Informational
5	Execution engine API lacks endpoint authentication	Undefined Behavior	High
6	Pre-deployed L1 attributes contract will never be updated	Undefined Behavior	High
7	Underspecified behavior regarding deposits made through smart contracts	Data Validation	Low
8	Incorrect error handling when creating an L2 block	Error Reporting	Low
9	Incomplete error handling throughout optimistic-specs	Error Reporting	Undetermined
10	Inconsistencies within documentation	Undefined Behavior	Informational
11	Risk of denial of service due to free deposit transactions on L2	Undefined Behavior	Informational

12	Use of time.After() in select statements can lead to memory leaks	Denial of Service	**Informational**

Detailed Findings

1. Risk of theft due to reentrancy vulnerability in WithdrawalsRelay

Severity: **High**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-OPT-1
Target: `optimistic-specs/packages/contracts/contracts/L1/abstracts/WithdrawalsRelay.sol`	

Description

It is possible to steal deposited ETH from the L1 `OptimismPortal` contract due to a reentrancy vulnerability in the `WithdrawalsRelay` contract.

The `OptimismPortal` contract allows users to make deposit transactions to be executed on L2. Users can specify the L2 target address and the calldata and send an amount of ETH that will be locked in the L1 contract and minted on L2.

To withdraw funds from L2, the user first calls `initiateWithdrawal` on the `Withdrawer` contract on L2 and later calls `finalizeWithdrawalTransaction` on the `OptimismPortal` contract on L1. The `finalizeWithdrawalTransaction` function performs a low-level call to send the funds to a user-controlled address. The code checks whether the withdrawal has already been finalized, which is indicated by the `finalizedWithdrawals` value; however, `finalizedWithdrawals` is set after the check and after the funds are transferred, so it is possible to reenter this function with the same arguments and steal ETH locked in L2.

```
   function finalizeWithdrawalTransaction(
        uint256 _nonce,
        address _sender,
        address _target,
        uint256 _value,
        uint256 _gasLimit,
        bytes calldata _data,
        uint256 _timestamp,
        WithdrawalVerifier.OutputRootProof calldata _outputRootProof,
        bytes calldata _withdrawalProof
    ) external {
[...]
        // Check that this withdrawal has not already been finalized.
        if (finalizedWithdrawals[withdrawalHash] == true) {
```

```
            revert WithdrawalAlreadyFinalized();
        }

        l2Sender = _sender;
        // Make the call.
        (bool s, ) = _target.call{ value: _value, gas: _gasLimit }(_data);
        s; // Silence the compiler's "Return value of low-level calls not used"
warning.
        l2Sender = DEFAULT_L2_SENDER;

        // All withdrawals are immediately finalized. If the ability to replay a
transaction is
        // required, that support can be provided in external contracts.
        finalizedWithdrawals[withdrawalHash] = true;
        emit WithdrawalFinalized(withdrawalHash);
[...]
```

Figure 1.1: optimistic-specs/packages/contracts/contracts/L1/abstracts/WithdrawalsRelay.sol#L90-L151

Exploit Scenario

Eve finalizes a transaction to withdraw one ETH locked on L2; the `_target` argument refers to her own malicious smart contract address. When her contract receives the call, it reenters the `finalizeWithdrawalTransaction` function, allowing her to steal additional ETH.

Recommendations

Short term, move the `finalizedWithdrawals[withdrawalHash] = true;` line before the check and the external call so that the check will fail and the function will revert if a user attempts to finalize an already finalized withdrawal.

Long term, follow the checks-effects-interactions pattern and use Slither to discover reentrancy vulnerabilities.

2. Missing zero address checks in L2 CheckpointOracle

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-OPT-2
Target: `optimistic-specs/packages/contracts/lib/optimism/l2geth/contracts/checkpointoracle/contracts/oracle.sol`	

Description

The `optimistic-specs` repository contains a submodule of the `optimism` repository. The `optimism` repository contains the `CheckpointOracle` contract, which allows whitelisted admins to set a checkpoint via a multisignature scheme. However, the whitelist accepts zero address admins; to check whether the admin setting the checkpoint is whitelisted, the multisignature scheme's validation code calls `ecrecover`, which returns zero on invalid signatures. There is no check to determine whether `ecrecover`'s return value indicates an invalid signature.

This means that if an admin whitelists a zero address, the multisignature validation code would identify any invalid signature as a valid whitelisted address.

```
constructor(address[] memory _adminlist, uint _sectionSize, uint _processConfirms,
uint _threshold) public {
    for (uint i = 0; i < _adminlist.length; i++) {
        admins[_adminlist[i]] = true;
        adminList.push(_adminlist[i]);
    }
```

Figure 2.1:
optimism/l2geth/contracts/checkpointoracle/contract/oracle.sol#L19-L23

```
// In order for us not to have to maintain a mapping of who has already
// voted, and we don't want to count a vote twice, the signatures must
// be submitted in strict ordering.
for (uint idx = 0; idx < v.length; idx++){
    address signer = ecrecover(signedHash, v[idx], r[idx], s[idx]);
    require(admins[signer]);
    require(uint256(signer) > uint256(lastVoter));
    lastVoter = signer;
    emit NewCheckpointVote(_sectionIndex, _hash, v[idx], r[idx], s[idx]);
```

Figure 2.2:
optimism/l2geth/contracts/checkpointoracle/contract/oracle.sol#L103-L111

Exploit Scenario

The Optimism team deploys the `CheckpointOracle` contract; however, due to a bug in the off-chain tooling, a zero address is added to the admin list. As a result, a malicious actor is able to gain an additional vote to set a checkpoint.

Recommendations

Short term, add a zero address check to the `CheckpointOracle` constructor to disallow zero address administrators. Alternatively, add a zero address check against the result of the `ecrecover` call in the `SetCheckpoint` method.

Long term, ensure that zero address checks are in place for all calls to the `ecrecover` function to ensure that invalid signatures are not accepted.

3. Possible failure to parse deposit transactions due to incorrect gasLimit type

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-OPT-3
Target: `optimistic-specs/packages/contracts/contracts/L1/abstracts/DepositFeed.sol`, `optimistic-specs/opnode/rollup/derive/payload_attributes.go`	

Description

The code that parses deposit transaction events checks that the gas limit is within the `uint64` range (i.e., it should be less than 2^{64}), but the `gasLimit` value is of the `uint256` type. As a consequence, the code will fail to parse every deposit transaction in a block if one transaction in the block contains a `gasLimit` greater than 2^{64}.

```
event TransactionDeposited(
    address indexed from,
    address indexed to,
    uint256 mint,
    uint256 value,
    uint256 gasLimit,
    bool isCreation,
    bytes data
);
```

Figure 3.1: The `TransactionDeposited` event in DepositFeed.sol#L31-L39

The `DeriveDeposits` function extracts deposit transactions by parsing `TransactionDeposited` events. It first calls the `UserDeposits` function with the receipt of the L1 block to be analyzed. It eventually arrives at the code shown in figure 3.3, which checks that the `gasLimit` value is within the range of `uint64` and returns an error if it is not. In such a case, `DeriveDeposits` also returns an error to indicate a failure to derive all the deposit transactions in the current L1 block.

```
func DeriveDeposits(receipts []*types.Receipt, depositContractAddr common.Address)
([]hexutil.Bytes, error) {
	userDeposits, err := UserDeposits(receipts, depositContractAddr)
	if err != nil {
		return nil, fmt.Errorf("failed to derive user deposits: %v", err)
	}
[...]
```

Figure 3.2: The `DeriveDeposits` function in `payload_attributes.go#L341-L355`

```
if !gas.IsUint64() {
        return nil, fmt.Errorf("bad gas value: %x", ev.Data[offset:offset+32])
}
```

Figure 3.3: The `UnmarshalLogEvent` function in `payload_attributes.go#L117-L119`

Exploit Scenario

Eve, an attacker, sends a deposit transaction with a `gasLimit` greater than 2^{64} on every block. As a consequence, the deposit transactions of other users are never correctly represented on L2, and the deposited ETH is locked in the `OptimismContract` on L1.

Recommendations

Short term, change the `_gasLimit`'s data type from `uint256` to `uint64` in the `depositTransaction` function.

Long term, review all entry points that take user-provided input to ensure that appropriate data validation occurs. Implementing a fail-fast design will ensure that invalid input does not reach deeper portions of the codebase.

4. Incorrect data validation when parsing transaction logs

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-OPT-4
Target: `optimistic-specs/opnode/rollup/derive/payload_attributes.go`	

Description

The data validation that the rollup node performs while parsing deposit transaction events is incorrect. As the code evolves, this incorrect data validation could result in unexpected behavior.

When a user deposits ETH into an L1 contract, the `TransactionDeposited` event is emitted. Before including the transaction on L2, the rollup node parses the `TransactionDeposited` event into a `DepositTx` struct by calling the `UnmarshalLogEvent` function.

During the parsing process, the `UnmarshalLogEvent` function checks the value of `dataOffset`, which represents how far from the start of the encoded log event the dynamic `data` field begins (figure 4.1).

```
event TransactionDeposited(
    address indexed from,
    address indexed to,
    uint256 mint,
    uint256 value,
    uint256 gasLimit,
    bool isCreation,
    bytes data
);
```

Figure 4.1: The `TransactionDeposited` event in `DepositFeed.sol#L31-L39`

However, the `UnmarshalLogEvent` function's check of the `dataOffset` field is incorrect. It checks that `dataOffset` *does not equal* 128 bytes (figure 4.2), but based on the ABI encoding of the `TransactionDeposited` event, `dataOffset` equals 160 bytes.

```
func UnmarshalLogEvent(ev *types.Log) (*types.DepositTx, error) {
	[...]
	var dataOffset uint256.Int
	dataOffset.SetBytes(ev.Data[offset : offset+32])
	offset += 32
	if dataOffset.Eq(uint256.NewInt(128)) {
		return nil, fmt.Errorf("incorrect data offset: %v", dataOffset[0])
```

```
        }
        [...]
}
```

Figure 4.2: The `UnmarshalLogEvent` function in `payload_attributes.go#L79-L153`

Exploit Scenario

After a change is made to the `TransactionDeposited` event, the `gasLimit` value is no longer required, and `dataOffset` now equals 128. As a result, the conditional in `UnmarshalLogEvent` fails, preventing all user-deposit transactions from being included on L2.

Recommendations

Short term, change the conditional so that it returns an error if `dataOffset` *does not equal* the current `offset`.

Long term, follow the guidance in the "Contract ABI Specification" section of the Solidity documentation to ensure that transaction event data is encoded appropriately. Employ unit tests to verify that checks are sufficient.

5. Execution engine API lacks endpoint authentication

Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-OPT-5
Target: `optimistic-specs/opnode/l2/source.go`	

Description

The execution engine API leveraged by Optimism does not authenticate connections, allowing anyone to submit deposit transactions to be added to the L2 chain.

The publicly exposed engine API is used by the rollup node to submit L2 blocks to the execution engine so that they can be added to the canonical L2 chain. As stated in the documentation, this connection must be trusted and authenticated (figure 5.1):

```
Transactions cannot be blindly trusted, trust is established through authentication.
Unlike other transaction types deposits are not authenticated by a signature:
the rollup node authenticates them, outside of the engine.

To process deposited transactions safely, the deposits MUST be authenticated first:

- Ingest directly through trusted Engine API
- Part of sync towards a trusted block hash (trusted through previous Engine API
instruction)

Deposited transactions MUST never be consumed from the transaction pool.
```

Figure 5.1: The "Deposited transaction boundaries" section in `exec-engine.md#L37-L46`

Authentication guarantees that *all* deposits on the L2 chain can be securely derived from the rollup node. However, because the calls to the API are not authenticated, any user can create an L2 block with arbitrary deposit transactions and artificially inflate his or her balance.

It is important to note that a peer-to-peer network runs the execution engine. Thus, the block must be broadcast and subsequently accepted to be added to the canonical chain.

Exploit Scenario

Eve, an attacker, creates an L2 block containing a user-deposit transaction that credits her L2 account with a large amount of ether. Eve broadcasts her block to the execution engine peer-to-peer network. It is accepted and bypasses the rollup node's verification mechanisms.

Recommendations

Short term, implement an API authentication mechanism to ensure secure communication between the rollup node and the engine API.

Long term, ensure that exposed endpoints appropriately use authentication mechanisms and access controls where applicable. Additionally, ensure that logging mechanisms capture critical runtime data that can be used as an audit trail in the event of an attack or system failure.

6. Pre-deployed L1 attributes contract will never be updated

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OPT-6
Target: `optimistic-specs/packages/contracts/contracts/L2/L1Block.sol`, `optimistic-specs/opnode/rollup/derive/payload_attributes.go`	

Description

The pre-deployed L1 attributes contract expects the `msg.sender` to be the `DEPOSITOR_ACCOUNT` address; however, the `msg.sender` is set to `depositContractAddr` on L1. As a result, the L1 attributes contract will never be updated, and it will return incorrect data for handling L1 chain reorganizations and extensions.

```
address public constant DEPOSITOR_ACCOUNT =
0xDeaDDEaDDeAdDeAdDEAdDEaddeAddEAdDEAd0001;
[...]
function setL1BlockValues(
     uint256 _number,
     uint256 _timestamp,
     uint256 _basefee,
     bytes32 _hash
 ) external {
     if (msg.sender != DEPOSITOR_ACCOUNT) {
         revert OnlyDepositor();
     }
     [...]
```

Figure 6.1: The `setl1BlockValues` function in `L1Block.sol#L13-L34`

The L1 attributes contract should hold the block number, timestamp, base fee, and hash of the L1 block that corresponds to the current L2 block. To update this information, the contract adds a call to `setL1BlockValues` with the correct values as the first transaction of every L2 block. The transaction is built by the `L1InfoDeposit` function, in which the `From` address is set to `depositContractAddr` (which is not the same address as `DEPOSITOR_ACCOUNT`). As a result, every `setL1BlockValues` transaction reverts, preventing the L1 attributes contract from being updated with the correct L1 block data.

```
// L1InfoDeposit creates a L1 Info deposit transaction based on the L1 block,
// and the L2 block-height difference with the start of the epoch.
func L1InfoDeposit(seqNumber uint64, block L1Info, depositContractAddr
common.Address) *types.DepositTx {
       [...]
       return &types.DepositTx{
```

```
            SourceHash: source.SourceHash(),
            From:       depositContractAddr,
            To:         &L1InfoPredeployAddr,
            Mint:       nil,
            Value:      big.NewInt(0),
            Gas:        99_999_999,
            Data:       data,
        }
}
```

Figure 6.2: The `L1InfoDeposit` function in `payload_attributes.go#L169-L198`

L2 contracts can retrieve data regarding the current L1 block from the L1 attributes contract. Moreover, the L2 Optimism chain uses the data returned by the L1 attributes contract to handle L1 chain reorganizations and extensions. Due to the L1 attributes contract's failure to update, these contracts and the L2 Optimism chain will retrieve incorrect data.

Exploit Scenario

A reorganization that the L2 Optimism chain should be able to handle occurs in the L1 Ethereum chain; however, because the L1 attribute contract holds invalid data, the L2 chain fails to properly handle the reorganization and is left in an undefined state.

Recommendations

Short term, change the `L1InfoDeposit` function's `From` value from `depositContractAddr` to `DEPOSITOR_ACCOUNT`.

Long term, improve the system's unit test coverage to uncover edge cases and to ensure that the system exhibits the intended behavior.

7. Underspecified behavior regarding deposits made through smart contracts

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-OPT-7
Target: `optimistic-specs/packages/contracts/contracts/L1/abstracts/DepositFeed.sol`	

Description

When a smart contract submits a deposit transaction, the code will transform the contract address to an aliased address by adding a fixed offset. Due to the lack of specification and guidance regarding how smart contracts should manage funds within the system, a naive smart contract that interacts with the `DepositFeed` could lock funds in the system that may not be retrievable later.

```
function depositTransaction(
     address _to,
     uint256 _value,
     uint256 _gasLimit,
     bool _isCreation,
     bytes memory _data
 ) public payable {
     if (_isCreation && _to != address(0)) {
         revert NonZeroCreationTarget();
     }

     address from = msg.sender;
     // Transform the from-address to its alias if the caller is a contract.
     if (msg.sender != tx.origin) {
         from = AddressAliasHelper.applyL1ToL2Alias(msg.sender);
     }

     emit TransactionDeposited(from, _to, msg.value, _value, _gasLimit, _isCreation,
_data);
 }
```

Figure 7.1: The `depositTransaction` function in `DepositFeed.sol#L54-L72`

Because the aliased `from` address will receive the deposited funds on L2 and nobody has access to the keypair associated with the aliased address, a smart contract could erroneously deposit funds that are not sent to other addresses into the system.

The contract could recover these funds by sending another deposit transaction to move the sum of the new and old deposit to another address. However, due to the lack of

guidance around this scenario, a smart contract could erroneously allow some of a deposit to be retained within the alias address and not provide a mechanism to send another deposit transaction to recover it, resulting in a loss of funds.

Exploit Scenario

A user interacts with a smart contract that leverages Optimism's `DepositFeed` to deposit funds into L2. The smart contract allows for deposits that transfer fewer tokens than are minted in L2, but does not allow for deposits that transfer more tokens than were minted in L2. The user sends a deposit transaction requesting only a portion of the deposited funds to be moved to another address. Due to the smart contract's inability to create a second deposit to move previously minted funds, the user cannot recover her funds from L2.

Recommendations

Short term, provide guidance within the documentation to ensure that smart contract developers are aware of this edge case.

Long term, further document the intended behavior throughout the system so that smart contract developers are aware of system edge cases.

8. Incorrect error handling when creating an L2 block

Severity: **Low**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-OPT-8
Target: `optimistic-specs/opnode/rollup/driver/state.go`, `optimistic-specs/opnode/l1/source.go`	

Description

In the code in which the sequencer chooses which L1 block to use as the origin block, the error handling is incorrect and could prevent the creation of L2 blocks.

The sequencer, which is responsible for creating new L2 blocks, must choose an L1 block as the new L2 block's origin. Original blocks allow all L2 blocks to be directly tied to L1 history. The sequencer will always choose the most recently mined L1 block. This choice is performed in the `findL1Origin` function; if a new L1 block has been mined, the function will retrieve it (figure 8.1).

```
func (s *state) findL1Origin(ctx context.Context) (eth.L1BlockRef, error) {
	if s.l2Head.L1Origin.Hash == s.l1Head.Hash {
		return s.l1Head, nil
	}
	currentOrigin, err := s.l1.L1BlockRefByHash(ctx, s.l2Head.L1Origin.Hash)
	if err != nil {
		return eth.L1BlockRef{}, err
	}
	nextOrigin, err := s.l1.L1BlockRefByNumber(ctx, currentOrigin.Number+1)
	if errors.Is(err, ethereum.NotFound) {
		return currentOrigin, nil
	}
	if s.l2Head.Time+s.Config.BlockTime >= nextOrigin.Time {
		return nextOrigin, nil
	}

	return currentOrigin, nil
}
```

Figure 8.1: The `findL1Origin` function in `state.go#L173-L203`

If the retrieval fails and the error returned is `ethereum.NotFound`, the sequencer will continue to mine on the current origin block.

However, the `L1BlockRefByNumber` function never returns an `ethereum.NotFound` error. In fact, it returns the custom error shown in figure 8.2:

```
func (s *Source) L1BlockRefByNumber(ctx context.Context, l1Num uint64)
(eth.L1BlockRef, error) {
	head, err := s.InfoByNumber(ctx, l1Num)
	if err != nil {
		return eth.L1BlockRef{}, fmt.Errorf("failed to fetch header by num %d:
%v", l1Num, err)
	}
	return head.BlockRef(), nil
}
```

Figure 8.2: The `L1BlockRefByNumber` function in `source.go#L305-L311`

Since the failure is never captured, `findL1Origin` will return an empty `eth.L1BlockRef{}` as the current `l1Origin`. Thus, the following check will fail and will prevent the new L2 block from being created (figure 8.3):

```
func (s *state) createNewL2Block(ctx context.Context) error {
	// Figure out which L1 origin block we're going to be building on top of.
	l1Origin, err := s.findL1Origin(ctx)
	if err != nil {
		s.log.Error("Error finding next L1 Origin", "err", err)
		return err
	}
	if l1Origin.Number <= s.Config.Genesis.L1.Number {
		s.log.Info("Skipping block production because the next L1 Origin is
behind the L1 genesis")
		return nil
	}
	[...]
}
```

Figure 8.3: The `createNewL2Block` function in `state.go#L205-255`

Exploit Scenario

The sequencer identifies that a new L1 block has been mined and tries to use it as the origin block for the upcoming L2 block. However, the RPC connection to the L1 node goes down and the `L1BlockRefByNumber` function returns an error. The `findL1Origin` function fails to catch the error, and the L2 block is not created.

Recommendations

Short term, change the error returned by `L1BlockRefByNumber` to be of the `ethereum.NotFound` type.

Long term, improve the system's unit test coverage to uncover edge cases and to ensure that the system exhibits the intended behavior.

9. Incomplete error handling throughout optimistic-specs

Severity: **Undetermined**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-OPT-9
Target: `optimistic-specs`	

Description

Error reporting is insufficient or incomplete in several areas of the `optimistic-specs` repository.

The following is a non-exhaustive list of areas that have error reporting issues:

- `optimistic-specs/opnode/rollup/driver/driver.go#L61-L65`
- `optimistic-specs/opnode/rollup/derive/payload_attributes.go#L227-L231`
- `optimistic-specs/opnode/rollup/derive/payload_attributes.go#L232-L236`
- `optimistic-specs/opnode/rollup/derive/payload_attributes.go#L237-L241`
- `optimistic-specs/opnode/node/node.go#L203-L205`

Recommendations

Short term, ensure that appropriate error handling is implemented in all areas of the codebase. Ensure that all “TODO” comments throughout the codebase are resolved. Additionally, employ Semgrep and `gosec` to ensure that no errors are unhandled.

Long term, consider integrating analyzers such as Semgrep and `gosec` into the CI/CD pipeline to uncover additional cases of unhandled errors.

10. Inconsistencies within documentation

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OPT-10
Target: `optimistic-specs/specs`	

Description

The Optimistic rollup node specification contains inconsistencies and incorrect information. Due to these issues, our review of the codebase required additional effort. Additionally, operators and users may have an incorrect understanding of certain system components.

- References to `engine_executePayloadV1` should refer to the updated API `engine_newPayloadV1`.
- Links to the "deposits spec" throughout the documentation should read "withdrawals spec" instead.
- Some portions of documentation state that the L1 attributes deposited transaction is included only in the first block of a sequencing window, while other portions of the documentation state that it is included in every L2 block. (The latter is the correct behavior.)
- The documentation on the L1 attributes deposit source hash states that the `l1BlockHash` is cast to a `uint256` type and then to a `bytes32` type. However, the `l1BlockHash` is already a `bytes32` object, and the implementation does not perform this casting.

The documentation should include all expected properties and assumptions relevant to the codebase.

Recommendations

Short term, review and correct the documentation issues described in the finding description.

11. Risk of denial of service due to free deposit transactions on L2

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OPT-11
Target: `optimistic-specs/specs`	

Description
Currently, deposit transactions executed on Optimism L2 do not cost gas. The only cost is the gas required to call `depositTransaction` in the `OptimismPortal` contract on Ethereum. If a free transaction requires a large amount of computational power, a denial of service could occur.

```
function depositTransaction(
      address _to,
      uint256 _value,
      uint256 _gasLimit,
      bool _isCreation,
      bytes memory _data
  ) public payable {
[...]
```

Figure 11.1: The `depositTransaction()` function in `DepositFeed.sol#L54-60`

Users can specify the `_gasLimit` that will be used to execute the transaction on L2; depending on the chosen `_gasLimit`, the sequencer may perform a large amount of computation for free. Additionally, Ethereum miners do not have to pay for L1 transactions.

Exploit Scenario
Eve calls `depositTransaction` and sets the `_gasLimit` to the highest possible value and the target to a controlled contract that will perform a useless computation. The sequencer is forced to execute the transaction, causing a denial of service.

Recommendations
Short term, before deploying the system into production, ensure that executing deposit transactions on L2 costs a fee to prevent this attack.

Long term, ensure that computationally expensive operations have an associated cost or limit to prevent attackers from leveraging them as attack vectors in denial-of-service attacks.

12. Use of time.After() in select statements can lead to memory leaks

Severity: **Informational**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-OPT-12
Target: `optimistic-specs/l2os/service.go`, `optimistic-specs/l2os/txmgr/txmgr.go`	

Description

Calls to `time.After` in `for/select` statements can lead to memory leaks because the garbage collector does not clean up the underlying `Timer` object until the timer fires. A new timer, which requires resources, is initialized at each iteration of the `for` loop (and, hence, the `select` statement). As a result, exiting the `select` statement through another `case` condition prevents resources originating from the `time.After` call from being garbage collected.

This issue is prevalent in two locations within the `optimistic-specs` repository, as shown in figure 12.1 and figure 12.2.

```
	for {
		select {

		// Whenever a resubmission timeout has elapsed, bump the gas
		// price and publish a new transaction.
		case <-time.After(m.cfg.ResubmissionTimeout):
[...]

		// The passed context has been canceled, i.e. in the event of a
		// shutdown.
		case <-ctxc.Done():
			return nil, ctxc.Err()

		// The transaction has confirmed.
		case receipt := <-receiptChan:
			return receipt, nil
		}
	}
}
```

Figure 12.1: optimistic-specs/l2os/txmgr/txmgr.go#L203-L231

```
for {
	select {
	case <-time.After(s.cfg.PollInterval):
[...]
```

```
        case <-s.done:
                log.Info(name + " service shutting down")
                return
        }
}
```

Figure 12.2: optimistic-specs/l2os/service.go#L100-L166

Recommendations

Short term, refactor the code that uses the `time.After` function in `for/select` loops using tickers. This will prevent memory leaks and crashes caused by memory exhaustion. Examples of how this could be done can be found in the references section below.

Long term, ensure that the `time.After` method is not used in `for/select` statements with additional exit cases. Periodically use the Semgrep query to check for and detect similar patterns.

References

- Use with caution time.After Can cause memory leak (golang)
- Golang <-time.After() is not garbage collected before expiry

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
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
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. Code Quality Findings

This appendix lists code quality findings that we identified throughout our review.

- **Use `net.JoinHostPort` instead of `fmt.Sprintf` to combine hosts and ports into valid destination strings.** The use of `net.JoinHostPort` is generally preferred over `fmt.Sprintf`.

```
endpoint := fmt.Sprintf("%s:%d", addr, port)
```

Figure C.1: optimistic-specs/opnode/node/server.go#L32