Optimism: Rollup Node and Execution Engine

Fix Review

July 7, 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.

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Table of Contents

About Trail of Bits 1

Notices and Remarks 2

Table of Contents 3

Executive Summary 5

Project Summary 7

Project Methodology 8

Project Targets 9

Summary of Fix Review Results 10

Detailed Fix Review Results 12

1. Risk of theft due to reentrancy vulnerability in WithdrawalsRelay 12

2. Missing zero address checks in L2 CheckpointOracle 14

3. Possible failure to parse deposit transactions due to incorrect gasLimit type 16

4. Incorrect data validation when parsing transaction logs 18

5. Execution engine API lacks endpoint authentication 20

6. Pre-deployed L1 attributes contract will never be updated 21

7. Underspecified behavior regarding deposits made through smart contracts 23

8. Incorrect error handling when creating an L2 block 25

9. Incomplete error handling throughout optimistic-specs 27

10. Inconsistencies within documentation 28

11. Risk of denial of service due to free deposit transactions on L2 29

12. Use of time.After() in select statements can lead to memory leaks 30

A. Status Categories 32

B. Vulnerability Categories 33

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 scope, timeline, test targets, and coverage are provided in the original audit report.

Optimism contracted Trail of Bits to review the fixes implemented for issues identified in the original report. From June 22 to June 23, 2022, a team of two consultants conducted a review of the client-provided source code.

Summary of Findings

The original audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the original findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Informational	2

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Data Validation	3
Denial of Service	1

Overview of Fix Review Results

Optimism has sufficiently addressed most of the issues described in the original audit report.

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
July 7, 2022	Delivery of fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material
- A review of the documentation provided alongside the underlying codebases

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

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Risk of theft due to reentrancy vulnerability in WithdrawalsRelay	Resolved
2	Missing zero address checks in L2 CheckpointOracle	Unresolved
3	Possible failure to parse deposit transactions due to incorrect gasLimit type	Resolved
4	Incorrect data validation when parsing transaction logs	Resolved
5	Execution engine API lacks endpoint authentication	Resolved
6	Pre-deployed L1 attributes contract will never be updated	Resolved
7	Underspecified behavior regarding deposits made through smart contracts	Resolved
8	Incorrect error handling when creating an L2 block	Resolved
9	Incomplete error handling throughout optimistic-specs	Partially Resolved
10	Inconsistencies within documentation	Partially Resolved
11	Risk of denial of service due to free deposit transactions on L2	Resolved
12	Use of time.After() in select statements can lead to memory leaks	Resolved

Detailed Fix Review Results

1. Risk of theft due to reentrancy vulnerability in WithdrawalsRelay	
Status: **Resolved**	
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*

Fix Analysis

This issue has been resolved. The Optimism team has updated the order of state-changing operations to ensure reentrancy is not possible in this case.

2. Missing zero address checks in L2 CheckpointOracle

Status: **Unresolved**	
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`

Fix Analysis

This issue has not been resolved. The Optimism team has indicated this code is maintained by a third party and sourced into their application, but remains unused.

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

Status: **Resolved**	
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`

Fix Analysis

This issue has been resolved. The Optimism team has resolved the parsing error by changing the data type of the affected parameter within the relevant smart contract. They have also changed their error handling to account for denial-of-service attacks against sibling transactions when parsing a bad transaction.

4. Incorrect data validation when parsing transaction logs

Status: **Resolved**	
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`

Fix Analysis

This issue has been resolved. The Optimism team has resolved the validation error by throwing an error only if the data offset field in the transaction does not equal the offset value being tracked in the function.

5. Execution engine API lacks endpoint authentication

Status: **Resolved**	
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.

Fix Analysis

This issue has been resolved. The Optimism team has added authentication mechanisms to HTTP and WebSocket connections.

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

Status: **Resolved**	
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 creats 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.

Fix Analysis

This issue has been resolved. The Optimism team has changed the `From` parameter of the `DepositTx` to reference the hard-coded address expected by receiving the smart contract.

7. Underspecified behavior regarding deposits made through smart contracts	
Status: **Resolved**	
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.

Fix Analysis

This issue has been resolved. The Optimism team has provided developer documentation underscoring potential developer errors with address aliasing when performing deposits.

8. Incorrect error handling when creating an L2 block

Status: **Resolved**	
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).

```go
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`]

Fix Analysis

This issue has been resolved. The Optimism team has resolved the error handling issue by immediately returning from the affected method if any type of error is thrown when looking for the L1 origin block.

9. Incomplete error handling throughout optimistic-specs	
Status: **Partially Resolved**	
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`

Fix Analysis

This issue has been partially resolved. The Optimism team refactored all relevant portions of the codebase flagged for incomplete error handling in this issue. However, additional cases of incomplete error handling remain.

The Optimism team refactored the `NewDriver` method such that it no longer requires the noted error handling. The `BatchesFromEVMTransactions` method was also refactored to perform error handling. Additionally, the code within `node.go` was refactored in depth.

10. Inconsistencies within documentation	
Status: **Partially Resolved**	
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.

Fix Analysis

This issue has been partially resolved. The Optimism team refactored documentation regarding casting, updated engine API references, and provided additional documentation regarding withdrawals. However, the team has not fixed the remaining concerns.

11. Risk of denial of service due to free deposit transactions on L2	
Status: **Resolved**	
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.

Fix Analysis

This issue has been resolved. The Optimism team has added resource metering to the OptimismPortal, which charges a gas fee based on the gas limit provided for L2 deposits.

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

Status: **Resolved**	
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`

Fix Analysis

This issue has been resolved. The Optimism team has resolved the memory leakage error by using a timer that can be reused across multiple iterations without exhausting memory.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Review Status	
Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

B. 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.