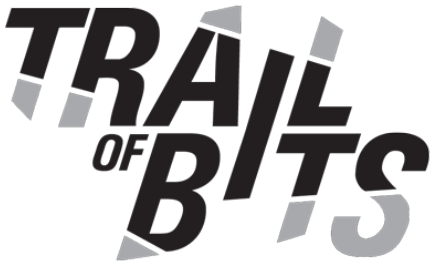

Optimism

Security Assessment

October 30th 2020

Prepared For:
Ben Jones | Optimism
ben@optimism.io

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Dominik Teiml | *Trail of Bits*
dominik.teiml@trailofbits.com

Executive Summary

From October 12 through October 30, 2020, Optimism engaged Trail of Bits to review the security of the OVM. Trail of Bits conducted this assessment over the course of 6 person-weeks with two engineers working from f6f5f3a.

The first week, we focused on gaining an understanding of the codebase. We reviewed the Safety Checker and message passing contracts against the most common Solidity flaws, and started to look how to perform property-based testing in the codebase.
In the second week, we focused on the detection of high-severity issues related to fraud protection. We continued reviewing the message passing contracts, and started looking into the rollup contracts and lower-level libraries. In the final week we reviewed the most complex interactions between the components and privileged users like the sequencer.

Our review resulted in 20 findings ranging from high to informational severity. Many of the high-severity issues are of low difficulty and would allow an attacker to subvert or disrupt the expected behavior of the OMV, such as:

- Incorrect implementation of safety checks when deploying contracts (TOB-OVM-001)
- Lack of proper access control for certain on-chain components (TOB-OVM-003, TOB-OVM-013)
- Incorrect or invalid computation of arithmetic code (TOB-OVM-004, TOB-OVM-007, TOB-OVM-010)
- Insufficient validation of empty Merkle tree proofs (TOB-OVM-008)
- Incorrect implementation of the fraud verification procedure (TOB-OVM-014, TOB-OVM-017, TOB-OVM-020).

We also found the contracts were not robust against different types of denial-of-service attacks either using reentrancy (TOB-OVM-006), transaction spamming (TOB-OVM-009), front-running (TOB-OVM-015)

Appendix C contains additional code quality issues.

Overall, OVM represents a significant work in progress—it's a complex codebase with many interacting components. Most of these components lack documentation (TOB-OVM-016) and many edge cases are untested (TOB-OVM-008).

Trail of Bits recommends addressing the findings presented in this report. We also recommend a feature freeze until the existing features are properly documented and their assumptions tested in depth. Finally, due to the prevalence of high-severity, low-difficulty findings, we recommend additional focused security reviews once the associated specification is written.

Project Dashboard

Application Summary

Name	Optimism
Version	f6f5f3a
Type	Solidity
Platforms	Ethereum

Engagement Summary

Dates	October 12 through October 30, 2020
Method	Whitebox
Consultants Engaged	3
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	9	■■■■■■■■■
Total Medium-Severity Issues	4	■■■■
Total Low-Severity Issues	5	■■■■■
Total Informational-Severity Issues	2	■■
Total	20	

Category Breakdown

Data Validation	12	■■■■■■■■■■■■
Access Controls	3	■■■
Undefined Behavior	4	■■■■
Auditing and Logging	1	■
Total	20	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Weak.** Lack of specification on where access control is really needed caused several issues. (TOB-OVM-003, TOB-OVM-013)
Arithmetic	**Weak.** Lack of industry-standard approach to avoid integer overflows and extensive usage of unsafe downcastings (TOB-OVM-004, TOB-OVM-007).
Assembly Use	**Weak.** Assembly code is used in several components. While in some cases is justified because Solidity does not allow some operation, most of the cases are gas optimizations of code that can be expressed within the language.
Decentralization	**Weak.** The owner of the contract has the power to modify the behavior of important components (TOB-OVM-002). Additionally, the sequencer user has the privilege to insert transactions at arbitrary points of the queue. There is no clear documentation on the privileges and roles of each of these users.
Upgradeability	**Weak.** No explicit upgradeability procedure, but the Address Manager contract could be used for that. If the ownership of that contract is renounced, the components will be immutable, but there is no documentation about this procedure.
Function Composition	**Moderate.** The code is divided into folders with contracts grouped according to their functionality. The use of Solidity inheritance and libraries correctly separates different layers of abstraction. However, the lack of extensive documentation and careful testing makes the code more difficult to review than expected.
Front-Running	**Weak.** We found several issues that allow attackers to disrupt on-chain components using transaction from-running (TOV-OVM-015).
Key Management	**Not Considered.**
Monitoring	**Weak.** We found that there are missing events to monitor the contracts (TOB-OVM-012), Additionally, there is no documentation detailing an incident response plan.
Specification	**Moderate**. Some components like the SafetyCheck contract are precisely specified, while others only have a very brief description.
Testing &	**Weak.** We found an issue that would enable an attacker to validate

Verification	empty proofs (TOB-OVM-008), highlighting the lack of corner cases in the merkle tree testing.

Engagement Goals

The engagement was scoped to provide a security assessment of OVM smart contracts in the contracts-v2 repository.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for the user/controller roles?
- Is there any arithmetic overflow or underflow affecting the code?
- Can participants manipulate or block transactions in L1 or L2?
- Is it possible to manipulate the contracts by front-running transactions?
- Can participants perform denial-of-service or phishing attacks against any of the components?
- Is the safety checker only allowed to deploy contracts with safe opcodes?
- Can users always append new transactions to the chain, and that doing so always results in the finalization of a corresponding, unique state root?
- Is it possible to manipulate, falsify or block L1 or L2 messages?

Coverage

The engagement was focused on the following components:

- **Lib_MerkleUtils, Lib_RLPReader, Lib_RLPWriter, Lib_MerkleTrie, Lib_SecureMerkleTrie.** This set of libraries allows to construct, transverse, validate and modify Merkle/Patricia trees. We manually reviewed these contracts, as well used automatic tools to verify that the data structures are consistently parsed and outputted. We also checked that invalid or empty proof are always rejected.
- **OVM_SafetyChecker.** This contract has a single function which accepts some EVM bytecode and returns whether or not it is "safe," where safe means if a particular subset of opcodes is used. We manually reviewed this contract, as well used automatic tools to verify that no unsafe bytecode can be crafted to bypass these checks.
- **OVM_CanonicalTransactionChain and dependencies:** Append-only log of transactions which should be applied to the rollup state. Allows for a privileged role, the "sequencer," to submit their own transactions to the rollup state that they are forced to include. We manually reviewed the contract's interactions with privileged and unprivileged users when submitting and verifying new transactions.
- **OVM_StateCommitmentChain and dependencies:** List of proposed state roots which parties have asserted are a result of each transaction in the Canonical Transaction Chain. Elements here have a 1:1 correspondence with those transactions, and should be the unique state root calculated off-chain by applying the canonical transactions one by one. We manually reviewed the contract to make sure its invariants hold and users cannot block important operations.
- **OVM_FraudVerifier:** Manages any "fraud proofs", disputes which demonstrate that a proposal in the State Commitment Chain is a malicious proposal which is NOT a result of applying the given canonical transaction to the previous state. We manually reviewed the contract to make sure no one could manipulate, fake or block the fraud verification procedure.
- **OVM_StateTransitioner:** Manages the OVM state which is accessed and updated during a fraud proof. This contract basically is populated with the OVM storage slots used in the transaction whose state commitment is being proven fraudulent, and stores the updated storage slots when the OVM transaction is played out on L1, so they can be compared to the proposed state commitment to check for fraud. We manually review the contract to make sure that every valid state can be transitioned into the next one.
- **Access controls.** Many parts of the system expose privileged functionality, such as setting parameters or managing transactions. We reviewed these functions to

ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed arithmetic calculations for logical consistency where overflows may negatively impact use of the OVM.

Important OVM components outside the scope of this assessment are:
- The execution manager and its dependencies.
- The bond manager and its dependencies.
- The L2 precompiled contracts.
- The Solidity compiler fork to compile contracts using only safe opcodes.
- Any off-chain code components, such as validators.

Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the contracts, including:

- Slither, a Solidity static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties of the low-level.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property. To mitigate these risks, we generate 50,000 test cases per property with Echidna and then manually review all results.

Automated Testing with Echidna

This is the list of properties that we tested using Echidna.

#	Property	Result
1	`isBytecodeSafe` returns true if and only if the bytecode is safe	FAILED (TOB-OVM-001)
2	Every address converted to an RLP value can be parsed back	PASSED
3	Every uint converted to an RPL value can be parsed back	PASSED
4	Every bytestring converted to its nibble representation be parsed back	PASSED

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

☐ **Review the SafetyCheck contract, its specification and make sure they match.**This will avoid any mismatch. TOB-OVM-001

☐ **Disallow any future updates on the values used in the AddressManager and make sure the privileged users are using a multisig wallet.** This will mitigate a single-point of failure. TOB-OVM-002

☐ **Rethink how batches are added and stored.** This will make sure sure that adding new batches cannot fail. TOB-OVM-003

☐ **Refactor code to use SafeMath.** This will ensure integer overflow is not possible. TOB-OVM-004

☐ **Wrap all external contract calls in a `require` or retrieve transaction success, and emit events after execution to ensure users are aware of transaction success.** This will avoid any unexpected behavior when the target contracts fail to run. TOB-OVM-005

☐ **Apply the checks-effects-interaction-pattern to all functions to ensure changing of state before invoking a function that makes an external call and emit events after execution to ensure users are aware of transaction success.** This will avoid any potential reentrancy to be exploited. TOB-OVM-006

☐ **Avoid integer downcasts and rewrite the impacted code to revert if the inputs are larger than expected**. This will avoid introducing an unexpected behavior that could be exploited TOB-OVM-007

☐ **Reject the empty proof for every function that validates Merkle trees.** This will avoid any validation of incorrect Merkle trees. TOB-OVM-008

☐ **Use `gasprice` to set a minimum price to pay for the burned gas and properly document how this measure protects against different types of denial-of-service attacks.** This will mitigate any potential denial-of-service using transactions in L2 TOB-OVM-009

☐ **Fix the `push` method to correctly update the `deletionOffset`.** This will avoid any incorrect updates in the core data structures. TOB-OVM-010

☐ **Ensure users are aware of Optimism's deployed contract address. Additionally, closely analyze all aspects of the message passing architecture and identify the risks associated with an attacker deploying different versions**. This will mitigate any phishing attack to the end users. TOB-OVM-011

☐ **Add events for users to easily identify if their message is properly sent.** This will allow external users to easily track down the on-chain results. TOB-OVM-012

☐ **Disallow the creation of `OVM_StateTransitioner` by any user and include a specific event so it is easy to determine when the fraud verification process started.** This will mitigate any phishing attack to the end users. TOB-OVM-013

☐ **Validate that the transaction provided is actually related to the state root. Alternatively, use the state root and the transaction to keep in the update the mapping.** This will avoid blocking fraud verification. TOB-OVM-014

☐ **Make sure the sequencer has dedicated slots to insert batches.** This will avoid users to interfere with the sequencer operations. TOB-OVM-015

☐ **Review and properly document the missing documentation.** This will help to make the code easier to understand, maintain, and review. TOB-OVM-016

☐ **Change the variable that is employed in `_verifyQueueTransaction` to use `_queueIndex` instead of `_inclusionProof.index`.** This will make sure that the batch validation works as expected. TOB-OVM-17

☐ **Make sure `appendSequencerBatch` succeeds even if the queue is empty.** This will ensure that the transactions are always properly inserted. TOB-OVM-18

☐ **Make sure the code enforces full monotonicity in the timestamp and block numbers.** This will ensure that the transactions in L2 will work as expected. TOB-OVM-019

☐ **Make sure the fraud process can be initialized under all expected circumstances.** This will ensure that the fraud verification is available, in case it is needed. TOB-OVM-20

Long term

⛶ **Use Echidna or Manticore to make sure:**

- **important system properties hold.** TOB-OVM-001, TOB-OVM-004, TOB-OVM-010, TOB-OVM-019
- **all functions properly validate their inputs.** TOB-OVM-007, TOB-OVM-008

⛶ **Review and minimize the permissions assigned to each privileged user.** This will mitigate any potential compromises of private keys and increase trust in the system by its users. TOB-OVM-002

⛶ **Review the access control for every function that changes the state of a component.** This will mitigate potential attacks to disrupt any of the system components**.** TOB-OVM-003

⛶ **Use Slither on the codebase to detect and prevent if:**

- **return values are ignored.** TOB-OVM-005
- **potential re-entrancy attacks are possible.** TOB-OVM-006

⛶ **Actively monitor the blockchain to detect any potential attacks on the on-chain components.** This will help to quickly react to any potential attacks. TOB-OVM-009, TOB-OVM-015

⛶ **Review the risks of third-party contract deployments on all aspects of the system.** This will make sure that third-party interactions work as expected. TOB-OVM-011

⛶ **Always add sufficient logging to ensure users are aware of all state updates.** This will help to monitor your contract interactions and react to any potential attacks. TOB-OVM-012

⛶ **Review and minimize unprotected public functions.** This will reduce the attack surface of unprivileged users. TOB-OVM-013

⛶ **Review all the public functions and make sure inputs are properly validated.** This will reduce the attack surface of unprivileged users. TOB-OVM-014

⛶ **Consider writing a formal specification of the protocol.** This will ensure that the behavior of the protocol is easy to understand and review TOB-OVM-016

⛶ **Carefully review the use unit tests to verify correctness of the system.** This will reduce the likelihood of introducing known issues during the development process. TOB-OVM-17, TOB-OVM-18

⛶ **Review all corner cases in the fraud verification to make sure it cannot be blocked.** This will make sure every possible transaction can be challenged if it is fraudulent**.**
TOB-OVM-20

Findings Summary

#	Title	Type	Severity
1	SafetyChecker allows deployment of bytecode with unsafe instructions	Data Validation	High
2	The owner of the AddressManager contract can backdoor any contract	Access Control	High
3	No access control in appendStateBatch allows to block proof verification	Access Control	High
4	Integer overflow is possible while checking fraud proof window	Data Validation	Medium
5	Result of target.call(msg) execution is ignored	Undefined Behavior	Medium
6	Re-entrancy risk in message passing contracts	Undefined Behavior	Low
7	Downcasting of integer can lead to incorrect update of state variable	Data Validation	High
8	Lib_MerkleUtils.verify accepts empty proofs	Data Validation	High
9	Denial-of-service protections in enqueue can be ineffective	Data Validation	Low
10	Lib_TimeboundRingBuffer.push incorrectly updates deletionOffset	Data Validation	High
11	Allowing multiple message passing deployments may result in unexpected behavior	Undefined Behavior	Informational
12	Insufficient Logging	Auditing and Logging	Low
13	Any user can create backdoored OVM_StateTransitioner contracts	Access Control	Low

14	Fraud verification allows any transaction to be used regardless of the state root	Data Validation	High
15	Sequencer calls to append new batches can be front-runned	Data Validation	Low
16	Without extensive contract documentation the codebase is error-prone	Undefined Behavior	Informational
17	_verifyQueueTransaction uses incorrect variable in its body	Data Validation	High
18	appendSequencerBatch reverts if queue is empty	Data Validation	Medium
19	Monotonicity of L2's timestamp and block number can be violated	Data Validation	High
20	Fraud verification reverts if submitted within the force inclusion period	Data Validation	Medium

1. SafetyChecker allows deployment of bytecode with unsafe instructions

Severity: High
Type: Data Validation
Target: `OVM_SafetyChecker.sol`

Difficulty: Low
Finding ID: TOB-OVM-001

Description
The SafetyChecker does not properly validate and filter unsafe instructions.

Before deploying any contract in L2, the OVM_SafetyCheck contract scans the bytecode which should restrict the use of only a subset of EVM instructions to avoid unexpected behavior. The `isBytecodeSafe` function is responsible to check if the bytecode contains only safe instructions:

```
contract OVM_SafetyChecker is iOVM_SafetyChecker {


  /*******************
   * Public Functions *
   *******************/
  /**
   * Returns whether or not all of the provided bytecode is safe.
   * @param _bytecode The bytecode to safety check.
   * @return `true` if the bytecode is safe, `false` otherwise.
   */
  function isBytecodeSafe(
      bytes memory _bytecode
  )
      override
      external
      view
      returns (bool)
  {
```

Figure 1.1: `isBytecodeSafe` *function in* `OVM_SafetyChecker`.

The list of unsafe instructions includes:

- `ADDRESS`
- `BALANCE`
- `ORIGIN`
- `CALLVALUE`
- `EXTCODESIZE`
- `EXTCODECOPY`
- `EXTCODEHASH`

among others. However, despite the `CALLVALUE` opcode (52) is an unsafe instruction, a bytecode with only that instruction will not be filtered:

```
Analyzing contract:
safety-checker-freeze/contracts/crytic/cryticSafetyChecker.sol:OVM_SafetyChecker
crytic_isBytecodeSafe: failed!💥
  Call sequence:
    addByte(52)
```

Figure 1.2: `echidna_isBytecodeSafe` *test failure*.

Exploit Scenario
Eve deploys a contract with unsafe bytecode in the OVM. Then, she uses it to manipulate other privilege contracts and produce unexpected effects in L2.

Recommendation
Short term, review the SafetyCheck contract, its specification and make sure they match.

Long term, use Echidna or Manticore to make sure important system properties hold.

2. The owner of the AddressManager contract can backdoor any contract

Severity: High
Type: Access Control
Target: `Lib_AddressManager.sol`

Difficulty: High
Finding ID: TOB-OVM-002

Description
The code that is used to link together OVM contracts on-chain can be abused by a privileged user to backdoor any of the components.

Some on-chain components of the OVM interact with each other using contract calling. In order to set up these pointers, the Address Manager is used:

```
contract Lib_AddressManager is Ownable {

    ...

    function setAddress(
        string memory _name,
        address _address
    )
        public
        onlyOwner
    {
        addresses[_getNameHash(_name)] = _address;
    }
```

Figure 2.1: `setAddress` *function in* `Lib_AddressManager`.

However, it is important to note that the owner of this contract can silently change any pointer in any OVM component at any time.

For instance, in the OVM_L1CrossDomainMessenger, the owner could change the result of calling `resolve("OVM_L2CrossDomainMessenger")` to any value:

```
contract OVM_L1CrossDomainMessenger is iOVM_L1CrossDomainMessenger,
OVM_BaseCrossDomainMessenger, Lib_AddressResolver {
    ...

    /**
     * Sends a cross domain message.
```

```
     * @param _message Message to send.
     * @param _gasLimit OVM gas limit for the message.
     */
    function _sendXDomainMessage(
        bytes memory _message,
        uint256 _gasLimit
    )
        override
        internal
    {
        ovmL1ToL2TransactionQueue.enqueue(
            resolve("OVM_L2CrossDomainMessenger"),
            _gasLimit,
            _message
        );
    }
}
```

Figure 2.2: `_sendXDomainMessage` *function in* `OVM_L1CrossDomainMessenger`.

Exploit Scenario
A malicious admin can silently change the results of the call to `resolve` to manipulate the results of the OVM.

Recommendation
Short term, disallow any future updates on the values used in the AddressManager. Additionally, make sure the privileged users are using a multisig wallet to mitigate a single-point of failure.

Long term, review and minimize the permissions assigned to each privileged user. This will mitigate any potential compromises of private keys and increase trust in the system by its users.

3. No access control in appendStateBatch allows to block proof verification

Severity: High | Difficulty: Low
Type: Access Control | Finding ID: TOB-OVM-003
Target: `OVM_BaseChain.sol, OVM_StateCommitmentChain.sol`

Description
The lack of access control when adding state batches allows to block the verification of proof during the message relay.

When a message is relayed, the caller must provide a proof, which is validated by a series of checks. One such check is in the verification of an element in the proof:

```
function verifyElement(
    bytes calldata _element,
    Lib_OVMCodec.ChainBatchHeader memory _batchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _proof
)
    override
    public
    view
    returns (
        bool _verified
    )
{
    require(
        _hashBatchHeader(_batchHeader) == batches[_batchHeader.batchIndex],
        "Invalid batch header."
    );
  ...
```

Figure 3.1: header of the `verifyElement` function in `OVM_BaseChain`

This code will read the `batches` list to validate the existence of an element. However, this state variable can be modified by any user using the `appendStateBatch` function:

```
function appendStateBatch(
    bytes32[] memory _batch
)
    override
    public
{
```

```
        require(
            _batch.length > 0,
            "Cannot submit an empty state batch."
        );

        require(
            getTotalElements() + _batch.length <=
ovmCanonicalTransactionChain.getTotalElements(),
            "Number of state roots cannot exceed the number of canonical transactions."
        );

        bytes[] memory elements = new bytes[](_batch.length);
        for (uint256 i = 0; i < _batch.length; i++) {
            elements[i] = abi.encodePacked(_batch[i]);
        }

        _appendBatch(elements);
    }
```

Figure 3.2: `appendStateBatch` function in `OVM_StateCommitmentChain`

Moreover, once the batches are added, they cannot be easily removed, until `ovmFraudVerifier` calls `deleteStateBatch`.

Exploit Scenario
Alice submits certain batches to be added. Eve sees the unconfirmed transaction and front-runs it to submit invalid results. As a result of that, Alice cannot validate her proofs.

Recommendation
Short term, rethink how batches are added and stored. Make sure that adding new batches cannot fail.

Long term, review the access control for every function that changes the state of a component to make sure potential attackers cannot disrupt any of the system components.

4. Integer overflow is possible while checking fraud proof window

Severity: Medium
Type: Data Validation
Target: `OVM_StateCommitmentChain.sol`

Difficulty: Low
Finding ID: TOB-OVM-004

Description
Integer overflow is possible when validating a submitted state commitment proof's timestamp.

When a message is relayed, the caller must provide a proof, which is validated by a series of checks. One such check ensures that the proof header's timestamp is within the `FRAUD_PROOF_WINDOW`:

```
function insideFraudProofWindow(
    Lib_OVMCodec.ChainBatchHeader memory _batchHeader
)
    override
    public
    view
    returns (
        bool _inside
    )
{
    require(
       _batchHeader.timestamp != 0,
        "Batch header timestamp cannot be zero"
    );
    return _batchHeader.timestamp + FRAUD_PROOF_WINDOW > block.timestamp;
}
```

Figure 4.1: `insideFraudProofWindow` function in `OVM_StateCommitmentChain.sol`

However, as the `_batchHeader.timestamp` is provided by the caller and the arithmetic calculation does not use SafeMath, integer overflow is possible.

Exploit Scenario
An attacker, Eve, submits a proof with a significantly large `_batchHeader.timestamp` that when added to the current fraud proof window causes the check to overflow and fail, despite the timestamp being in the future.

Recommendation
Short term, refactor code to use SafeMath. This will ensure integer overflow is not possible.

Long term, use Manticore or Echidna to ensure that no overflows/underflows are possible.

5. Result of `target.call(msg)` execution is ignored

Severity: Medium | Difficulty: Medium
Type: Undefined Behavior | Finding ID: TOB-OVM-005
Target: `L1_CrossDomainMessenger`, *`OVM_L2CrossDomainMessenger`*

Description
The message passing contracts assume that a call to an external contract is successful regardless whether it has failed or not.

To relay messages, the submitted proof is validated, then checked to ensure it has not been previously processed. Assuming this is true, it calls the external contract:

```
_target.call(_message);
[...]
receivedMessages[keccak256(xDomainCalldata)] = true;
```

Figure 5.1: `relayMessage` function in `OVM_L2CrossDomainMessenger.sol#L81-88`

However, the code does not check to ensure that the `_target.call(_message)` call is successful. Instead, it assumes the call is successful and marks the message as received.

Additionally, functions in the message passing flow would benefit from events being emitted, as these functions currently fail silently and a user needs to retrieve the value of `sentMessages` or `receivedMessages`.

Exploit Scenario
Alice is sending a message from L1 to L2. Once the message has been sent to L2, the external contract gets called but fails. The contract still marks the messages as executed, so Alice needs to re-submit the message.

Recommendation
Short term, wrap all external contract calls in a `require` or retrieve transaction success, and emit events after execution to ensure users are aware of transaction success.

Long term, use Slither on the codebase to prevent future ignored values.

6. Re-entrancy risk in message passing contracts

Severity: Low | Difficulty: Medium
Type: Undefined Behavior | Finding ID: TOB-OVM-006
Target: `OVM_L2CrossDomainMessenger`, *`OVM_BaseCrossDomainMessenger`*

Description
The message passing functions allow nonces to be reused and fails to save the sent message.

After generating calldata to transfer to the other chain, the `sendMessage` function invokes `_sendXDomainMessage` function, then executes state changes on its own contract.

```
_sendXDomainMessage(xDomainCalldata, _gasLimit);

messageNonce += 1;
sentMessages[keccak256(xDomainCalldata)] = true;
```

Figure 6.1: `sendMessage` function in `OVM_BaseCrossDomainMessenger.sol#L49-L52`

The target contract invokes an external call to add the transaction to the queue for processing:

```
    function _sendXDomainMessage(
        bytes memory _message,
        uint256 _gasLimit
    )
        override
        internal
    {
        ovmCanonicalTransactionChain.enqueue(
          resolve("OVM_L2CrossDomainMessenger")
            _gasLimit,
            _message
        );
    }
```

Figure 6.2: `_sendXDomainMessage` function in `OVM_L1CrossDomainMessenger.sol#L246-L258`

As the external call occurs before changing the contract state, the `sendMessage` function is subject to re-entrancy.

Exploit Scenario
Eve uses `sendMessage` to initiate a message from L1 to L2. She calls this function multiple times in the middle of execution, and can send multiple duplicate messages without updating the `nonce` and `sentMessages`.

Recommendation
Short term, apply the checks-effects-interaction-pattern to all functions to ensure changing of state before invoking a function that makes an external call. Also, emit events after execution to ensure users are aware of transaction success.

Long term, use Slither to detect and prevent future potential re-entrancy attacks.

7. Downcasting of integer can lead to incorrect update of state variable

Severity: High | Difficulty: Low
Type: Data Validation | Finding ID: TOB-OVM-007
Target: `OVM_StateCommitmentChain.sol, OVM_CanonicalTransactionChain.sol`

Description
The `indexBatch` field is not properly validated and can be used to update the state variables of the `OVM_StateCommitmentChain` contract with invalid values.

Any user can call `setLastOverwritableIndex` function in order to update important variables such as `lastDeletableIndex`:

```
function setLastOverwritableIndex(
    Lib_OVMCodec.ChainBatchHeader memory _stateBatchHeader,
    Lib_OVMCodec.Transaction memory _transaction,
    Lib_OVMCodec.TransactionChainElement memory _txChainElement,
    Lib_OVMCodec.ChainBatchHeader memory _txBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _txInclusionProof
)
    override
    public
{
    require(
        _isValidBatchHeader(_stateBatchHeader),
        "Invalid batch header."
    );


    require(
        insideFraudProofWindow(_stateBatchHeader) == false,
        "Batch header must be outside of fraud proof window to be overwritable."
    );


    require(
        _stateBatchHeader.batchIndex > lastDeletableIndex,
        "Batch index must be greater than last overwritable index."
    );


    require(
```

```
            ovmCanonicalTransactionChain.verifyTransaction(
                _transaction,
                _txChainElement,
                _txBatchHeader,
                _txInclusionProof
            ),
            "Invalid transaction proof."
        );


        lastDeletableIndex = _stateBatchHeader.batchIndex;
        lastDeletableTimestamp = _transaction.timestamp;
    }
```

Figure 7.1: `setLastOverwritableIndex` function in `OVM_StateCommitmentChain.sol`

One important input, the batch header, is initially validated by `_isValidBatchHeader`:

```
    function _isValidBatchHeader(
        Lib_OVMCodec.ChainBatchHeader memory _batchHeader
    )
        internal
        view
        returns (
            bool
        )
    {
        return Lib_OVMCodec.hashBatchHeader(_batchHeader) ==
batches.get(uint40(_batchHeader.batchIndex));
    }
}
```

Figure 7.2: `insideFraudProofWindow` function in `OVM_StateCommitmentChain.sol`

However, using a `batchIndex` larger than `2**40` will not necessarily fail, since the `uint40` function will clear any extra bits from the input. A similar issue is present in `_verifyElement`:

```
    function _verifyElement(
        bytes32 _element,
        Lib_OVMCodec.ChainBatchHeader memory _batchHeader,
```

```
        Lib_OVMCodec.ChainInclusionProof memory _proof
    )
        internal
        view
        returns (
            bool
        )
    {
        require(
            Lib_OVMCodec.hashBatchHeader(_batchHeader) ==
batches.get(uint32(_batchHeader.batchIndex)),
            "Invalid batch header."
        );


        require(
            Lib_MerkleUtils.verify(
                _batchHeader.batchRoot,
                _element,
                _proof.index,
                _proof.siblings
            ),
            "Invalid inclusion proof."
        );


        return true;
    }
}
```

Figure 7.3: `_verifyElement` function in `OVM_CanonicalTransactionChain.sol`

A similar issue is present in `_appendBatch`, `_deleteBatch` and `getQueueElement`.

Exploit Scenario
Eve calls `setLastOverwritableIndex` with a batch header containing a specially-crafted index that will be validated by all the checks despite being invalid. As a result, `lastDeletableIndex` will be updated with a very large value, potentially blocking any future calls to `setLastOverwritableIndex`.

Recommendation

Short term, avoid integer downcasts and rewrite the impacted code to revert if the inputs are larger than expected.

Long term, use Echidna or Manticore to make sure all your functions properly validate their inputs.

8. `Lib_MerkleUtils.verify` accepts empty proofs

Severity: High | Difficulty: Low
Type: Data Validation | Finding ID: TOB-OVM-008
Target: `OVM_FraudVerifier.sol`, `OVM_StateCommitmentChain.sol`, `Lib_MerkleUtils.sol`

Description
Lib_MerkleUtil's verify function verifies the validity of a Merkle tree, but does not check for empty proofs and can be bypassed by arguments that attackers provide.

Any user can initiate the fraud verification process by calling `initializeFraudVerification`:

```
function initializeFraudVerification(
    bytes32 _preStateRoot,
    Lib_OVMCodec.ChainBatchHeader memory _preStateRootBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _preStateRootProof,
    Lib_OVMCodec.Transaction memory _transaction,
    Lib_OVMCodec.TransactionChainElement memory _txChainElement,
    Lib_OVMCodec.ChainBatchHeader memory _transactionBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _transactionProof
)
    override
    public
    contributesToFraudProof(_preStateRoot)
{
    if (_hasStateTransitioner(_preStateRoot)) {
        return;
    }


    require(
        ovmStateCommitmentChain.verifyStateCommitment(
            _preStateRoot,
            _preStateRootBatchHeader,
            _preStateRootProof
        ),
        "Invalid pre-state root inclusion proof."
    );
...
```

Figure 8.1: header of the `initializeFraudVerification` function in `OVM_FraudVerifier.sol`

This function relies on `verifyCommitStatement`:

```
function verifyStateCommitment(
    bytes32 _element,
    Lib_OVMCodec.ChainBatchHeader memory _batchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _proof
)
    override
    public
    view
    returns (
        bool
    )
{
    require(
        _isValidBatchHeader(_batchHeader),
        "Invalid batch header."
    );


    require(
        Lib_MerkleUtils.verify(
            _batchHeader.batchRoot,
            _element,
            _proof.index,
            _proof.siblings
        ),
        "Invalid inclusion proof."
    );


    return true;
}
```

Figure 8.2: `verifyStateCommitment` function in `OVM_StateCommitmentChain.sol`

which is implemented using `Lib_MerkleUtils.verify`:

```
function verify(
```

```
        bytes32 _root,
        bytes32 _leaf,
        uint256 _path,
        bytes32[] memory _siblings
    )
        internal
        pure
        returns (
            bool _verified
        )
    {
        bytes32 computedRoot = _leaf;


        for (uint256 i = 0; i < _siblings.length; i++) {
            bytes32 sibling = _siblings[i];
            bool isRightSibling = uint8(_path >> i & 1) == 1;


            if (isRightSibling) {
                computedRoot = _getParentHash(computedRoot, sibling);
            } else {
                computedRoot = _getParentHash(sibling, computedRoot);
            }
        }


        return computedRoot == _root;
    }
```

Figure 8.3: `verify` function in `Lib_MerkleUtils.sol`

However, this function will not check if the `_siblings` list is empty, allowing it to verify the validity of the Merkle tree just with the `computedRoot == _root` return statement. This allows the proof to be trivially validated.

Exploit Scenario
Eve initializes a fraud verification process but supplies an empty proof, and she is able to produce fake fraud proof for valid states.

Recommendation
Short term, reject the empty proof for every function that validates Merkle trees.

Long term, use Echidna or Manticore to make sure inputs are properly validated.

9. Denial-of-service protections in enqueue can be ineffective

Severity: Low
Type: Data Validation
Target: `OVM_CanonicalTransactionChain.sol`

Difficulty: High
Finding ID: TOB-OVM-009

Description
Enqueuing new transactions is protected by a minimal amount of gas to consume, but there is no check that gas price is not negligible or zero.

Any user is allowed to enqueue any number of transactions into the L2 chain. To avoid a potential denial-of-service, users are asked to burn a certain amount of gas:

```
    function enqueue(
        address _target,
        uint256 _gasLimit,
        bytes memory _data
    )
        override
        public
    {
        require(
            _data.length <= MAX_ROLLUP_TX_SIZE,
            "Transaction exceeds maximum rollup transaction data size."
        );


        require(
            _gasLimit >= MIN_ROLLUP_TX_GAS,
            "Transaction gas limit too low to enqueue."
        );


        // We need to consume some amount of L1 gas in order to rate limit transactions going
into
        // L2. However, L2 is cheaper than L1 so we only need to burn some small proportion
of the
        // provided L1 gas.
        uint256 gasToConsume = _gasLimit/L2_GAS_DISCOUNT_DIVISOR;
        uint256 startingGas = gasleft();
```

```
        // Although this check is not necessary (burn below will run out of gas if not true),
it
        // gives the user an explicit reason as to why the enqueue attempt failed.
        require(
            startingGas > gasToConsume,
            "Insufficient gas for L2 rate limiting burn."
        );


        // Here we do some "dumb" work in order to burn gas, although we should probably
replace
        // this with something like minting gas token later on.
        uint256 i;
        while(startingGas - gasleft() < gasToConsume) {
            i++;
        }
    ...
```

Figure 9.1: header of the `enqueue` function in `OVM_CanonicalTransactionChain.sol`

However, it is important to note that the actual price of the gas (`gasprice`) is never checked for a minimal value. This allows the gas price to be set to zero. Moreover, miners can introduce any number of transactions in the current block without actually paying for the gas.

Exploit Scenario
An attacker, Eve, submits a very large amount with a very low amount of gas, even zero and waits for the Ethereum blockchain to go through a low-congestion period. Her transactions are eventually confirmed, flooding L2 with useless messages.

Recommendation
Short term, use `gasprice` to set a minimum price to pay for the burned gas. Properly document how this measure protects against different types of denial-of-service attacks.

Long term, actively monitor the blockchain to detect any potential attacks on the on-chain components.

10. `Lib_TimeboundRingBuffer.push` incorrectly updates `deletionOffset`

Severity: High
Type: Data Validation
Target: `Lib_TimeboundRingBuffer.sol`

Difficulty: Low
Finding ID: TOB-OVM-010

Description
`deleteElementsAfter` improperly updates the `deletionOffset` variable making it impossible to read or delete any elements once `maxSize` is reached.

Users are allowed to push batches of transactions. When k new elements are pushed, the upper bound increases by k. The lower bound stays the same if `deletionOffset > 0` (i.e. there are still deleted elements). Since `length` is increased by k, the way to achieve this is to decrease `deletionOffset` by k to `max(0, deletionOffset - k)`. The `push2` function correctly implements this:

```
        if (_self.deletionOffset != 0) {
            _self.deletionOffset = _self.deletionOffset == 1 ? 0 : _self.deletionOffset - 2;
        }
```

Figure 10.1: `push2` is correct

However, `push` increments it instead:

```
        if (_self.deletionOffset != 0) {
            _self.deletionOffset += 1;
        }
```

Figure 10.2: `push` is incorrect

As a result, the lower bound on the index for `get` will be higher than expected, preventing from reading valid elements.

Exploit Scenario
Alice deploys a contract that depends on Canonical Transaction Chain or State Commitment Chain that reads a valid element from the batch list. The transaction is reverted instead, which can lead to unexpected consequences.

Recommendation
Short term, fix the `push` method to correctly update the `deletionOffset`.

Long term, use Echidna to check for invariants in data structure libraries and other critical logic.

11. Allowing multiple message passing deployments may result in unexpected behavior

Severity: Informational
Type: Undefined Behavior
Target: `bridge/`

Difficulty: Undetermined
Finding ID: TOB-OVM-011

Description
Optimism's current architecture allows third parties to deploy different versions of message passing contracts which may result in unexpected behavior.

By allowing third parties to deploy these contracts, it may have unintended consequences such as:

- Malicious contract changes `sendMessage()` to replace L1's msg.sender with the attacker's address and submits cross-domain with the injected value
- Malicious contract tricks users into paying for transactions but replaces the target contract and message values with values of their choice
- Deploy a series of message passing contracts looking identical to Optimism's with an upgradeability feature allowing attackers to change code anytime they wish
- Replace `xDomainSender` with an address that an attacker chooses
- Change and update the storage and state proof verification

Recommendation
Short term, ensure users are aware of Optimism's deployed contract address. Additionally, closely analyze all aspects of the message passing architecture and identify the risks associated with an attacker deploying different versions.

Long term, review the risks of third-party contract deployments on all aspects of the system.

12. Insufficient Logging

Severity: Low | Difficulty: High
Type: Auditing and Logging | Finding ID: TOB-OVM-012
Target: *OVM_L2CrossDomainMessenger, OVM_L1CrossDomainMessenger, OVM_BaseCrossDomainMessenger*

Description
When sending and relaying messages, these functions fail silently and do not alert users to failed transactions.

When relaying a message, the function executes an external call and updates `receivedMessages`:

```
        require(
            successfulMessages[keccak256(xDomainCalldata)] == false,
            "Provided message has already been received."
        );

        xDomainMessageSender = _sender;
        _target.call(_message);
        [...]
        receivedMessages[keccak256(xDomainCalldata)] = true;
```

Figure 12.1: `relayMessage` function in `OVM_L2CrossDomainMessenger.sol#L81-88`

However, this code does not emit an event to broadcast the message has been processed so a user is unaware of the execution of their function call.

Exploit Scenario
Alice relies on a relayer to send her message from L1 to L2. While her message is relayed on L2, the call fails. Due to the lack of events, Alice is unaware that her transaction was unsuccessful, and must directly retrieve `receivedMessages`.

This issue is present in relayMessage on L1 and L2's CrossDomainMessenger as well as BaseDomainMessenger's sendMessage.

Recommendation
Short term, add events for users to easily identify if their message is properly sent.

Long term, always add sufficient logging to ensure users are aware of all state updates.

13. Any user can create backdoored OVM_StateTransitioner contracts

Severity: Low | Difficulty: High
Type: Access Control | Finding ID: TOB-OVM-013
Target: `OVM_StateTransitionerFactory.sol`, `OVM_StateTransitioner.sol`, `OVM_FraudVerifier.sol`

Description
The code that is used to create new OVM_StateTransitioner contracts can be called by any user with some parameters that allow to alter how the contract works.

Any user can initiate the fraud verification process by calling `initializeFraudVerification`:

```
function initializeFraudVerification(
    bytes32 _preStateRoot,
    Lib_OVMCodec.ChainBatchHeader memory _preStateRootBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _preStateRootProof,
    Lib_OVMCodec.Transaction memory _transaction,
    Lib_OVMCodec.TransactionChainElement memory _txChainElement,
    Lib_OVMCodec.ChainBatchHeader memory _transactionBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _transactionProof
)
    override
    public
    contributesToFraudProof(_preStateRoot)
{
    if (_hasStateTransitioner(_preStateRoot)) {
        return;
    }

    require(
        ovmStateCommitmentChain.verifyStateCommitment(
            _preStateRoot,
            _preStateRootBatchHeader,
            _preStateRootProof
        ),
        "Invalid pre-state root inclusion proof."
    );

    require(
```

```
            _verifyTransaction(
                _transaction,
                _transactionBatchHeader,
                _transactionProof
            ),
            "Invalid transaction inclusion proof."
        );


        transitioners[_preStateRoot] = iOVM_StateTransitionerFactory(
            resolve("OVM_StateTransitionerFactory")
        ).create(
            address(libAddressManager),
            _preStateRootProof.index,
            _preStateRoot,
            Lib_OVMCodec.hashTransaction(_transaction)
}
```

Figure 13.1: `initializeFraudVerification` function in `OVM_FraudVerifier.sol`

This function will create a new `OVM_StateTransitioner` contract, in order to allow the user to submit the requested information to confirm an invalid state.

However, it is still possible to call `create` to deploy a fresh `OVM_StateTransitioner` contract using the `OVM_StateTransitionerFactory`. Moreover, it also allows the caller to specify critical parameters to use in the contract, such as the pointers to other ones like `OVM_StateManager`.

```
    function create(
        address _libAddressManager,
        uint256 _stateTransitionIndex,
        bytes32 _preStateRoot,
        bytes32 _transactionHash
    )
        override
        public
        returns (
            iOVM_StateTransitioner _ovmStateTransitioner
        )
    {
        return new OVM_StateTransitioner(
            _libAddressManager,
```

```
            _stateTransitionIndex,
            _preStateRoot,
            _transactionHash
        );
    }
```

Figure 13.2: `create` function in `OVM_StateTransitionerFactory.sol`

Exploit Scenario
Alice notices a fraudulent transaction. Eve tricks Alice to accept a freshly created `OVM_StateTransitioner`, but using incorrect parameters. Since the contract has invalid values, Alice will be unable to successfully prove the fraud.

Recommendation
Short term, disallow the creation of `OVM_StateTransitioner` by any user and include a specific event so it is easy to determine when the fraud verification process started.

Long term, review and minimize unprotected public functions. This will reduce the attack surface of unprivileged users.

14. Fraud verification allows any transaction to be used regardless of the state root

Severity: High
Type: Data Validation
Target: `OVM_FraudVerifier.sol`

Difficulty: Medium
Finding ID: TOB-OVM-014

Description
The fraud verification procedure does not correctly validate the relationship between the state root and the transaction provided.

Any user can initiate the fraud verification process by calling `initializeFraudVerification`. This function requires to provide the state root and transaction:

```
function initializeFraudVerification(
    bytes32 _preStateRoot,
    Lib_OVMCodec.ChainBatchHeader memory _preStateRootBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _preStateRootProof,
    Lib_OVMCodec.Transaction memory _transaction,
    Lib_OVMCodec.TransactionChainElement memory _txChainElement,
    Lib_OVMCodec.ChainBatchHeader memory _transactionBatchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _transactionProof
)
    override
    public
    contributesToFraudProof(_preStateRoot)
{
    if (_hasStateTransitioner(_preStateRoot)) {
        return;
    }

    require(
        ovmStateCommitmentChain.verifyStateCommitment(
            _preStateRoot,
            _preStateRootBatchHeader,
            _preStateRootProof
        ),
        "Invalid pre-state root inclusion proof."
    );
```

```
        require(
            _verifyTransaction(
                _transaction,
                _transactionBatchHeader,
                _transactionProof
            ),
            "Invalid transaction inclusion proof."
        );


        transitioners[_preStateRoot] = iOVM_StateTransitionerFactory(
            resolve("OVM_StateTransitionerFactory")
        ).create(
            address(libAddressManager),
            _preStateRootProof.index,
            _preStateRoot,
            Lib_OVMCodec.hashTransaction(_transaction)
}
```

Figure 14.1: `initializeFraudVerification` function in `OVM_FraudVerifier.sol`

However, there are no checks to ensure that the pair of state root and transaction provided actually make sense. The initialization will write the `transationers` mapping, where only the state root is considered, and this mapping is never cleaned or overwritten.

Exploit Scenario
Eve starts the fraud verification procedure using a valid state root, but an unrelated transaction. This will block any subsequent attempts to start the fraud verification with a relevant transaction.

Recommendation
Short term, validate that the transaction provided is actually related with the state root. Alternatively, use the state root and the transaction to keep in the update the mapping.

Long term, review all the public functions and make sure inputs are properly validated.

15. Sequencer calls to append new batches can be front-runned

Severity: Low
Type: Data Validation
Target: `OVM_FraudVerifier.sol`

Difficulty: High
Finding ID: TOB-OVM-015

Description
Any user can block the sequencer front-running its function to add new batches.

The `OVM_CanonicalTransactionChain` allows for a privileged role, the sequencer to insert their own transactions to the rollup state using the `appendSequencerBatch`:

```
    function appendSequencerBatch()
        override
        public
    {
        uint40 shouldStartAtBatch;
        uint24 totalElementsToAppend;
        uint24 numContexts;
        assembly {
            shouldStartAtBatch    := shr(216, calldataload(4))
            totalElementsToAppend := shr(232, calldataload(9))
            numContexts           := shr(232, calldataload(12))
        }


        require(
            shouldStartAtBatch == getTotalElements(),
            "Actual batch start index does not match expected start index."
        );
        ...
```

Figure 15.1: header of `appendSequencerBatch` function in `OVM_CanonicalTransactionChain.sol`

However, the first of the parameters required, `shouldStartAtBatch`, is vulnerable to front-running, since it will be immediately checked with the result of `getTotalElements`. Any user can call `appendBatch` can increase the number of total elements:

```
function appendQueueBatch(
        uint256 _numQueuedTransactions
    )
```

```
    override
    public
{
    require(
        _numQueuedTransactions > 0,
        "Must append more than zero transactions."
    );


    uint40 nextQueueIndex = _getNextQueueIndex();
    bytes32[] memory leaves = new bytes32[](_numQueuedTransactions);
    for (uint256 i = 0; i < _numQueuedTransactions; i++) {
        leaves[i] = _getQueueLeafHash(nextQueueIndex);
        nextQueueIndex++;
    }


    _appendBatch(
        Lib_MerkleUtils.getMerkleRoot(leaves),
        _numQueuedTransactions,
        _numQueuedTransactions
    );


    emit QueueBatchAppended(
        nextQueueIndex - _numQueuedTransactions,
        _numQueuedTransactions,
        getTotalElements()
    );
}
```

Figure 15.2: `appendQueueBatch` function in `OVM_CanonicalTransactionChain.sol`

Exploit Scenario

Alice is the sequencer of OVM and wants to include a new batch. Eve front-runs her call to increase the number of total elements before Alice's transaction is confirmed. So, Alice's call to `appendSequencerBatch` reverts because Eve's transaction is confirmed first.

Recommendation

Short term, make sure the sequencer has dedicated slots to insert batches.

Long term, actively monitor the blockchain to identify and mitigate front-running attacks.

16. Without extensive contract documentation the codebase is error-prone

Severity: Informational
Type: Undefined Behavior
Target: .

Difficulty: Low
Finding ID: TOB-OVM-016

Description
Overall, the codebase lacks code documentation, high-level description, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes.

The current documentation would benefit from more details, including:

- An overall protocol walkthrough, showing the different users, their interactions, and the inputs and outputs.
- High-level description of the use cases of L1/L2 messages.
- A low-level description of the ring buffer is used by each component.
- A thorough description of the fraud verification steps, including their hashing schemas and the transaction/storage representations.
- The sequencer role and exactly how it impacts the ordering of batches.
- Time-related actions (fraud deadlines).

The documentation for each of these items should include their expected properties and assumptions.

Recommendation
Short term, review and properly document the missing documentation.

Long term, consider writing a formal specification of the protocol.

17. `_verifyQueueTransaction` uses incorrect variable in its body

Severity: High
Type: Data Validation
Target: `OVM_CanonicalTransactionChain.sol`

Difficulty: Low
Finding ID: TOB-OVM-17

Description
The verification of transactions in the Canonical Transaction Chain is incorrectly implemented, potentially blocking the fraud verification procedure.

The Canonical Transaction Chain provides a public view function `verifyTransaction` that is used in the State Commitment Chain and during fraud proofs.

```
function verifyTransaction(
    Lib_OVMCodec.Transaction memory _transaction,
    Lib_OVMCodec.TransactionChainElement memory _txChainElement,
    Lib_OVMCodec.ChainBatchHeader memory _batchHeader,
    Lib_OVMCodec.ChainInclusionProof memory _inclusionProof
)
    override
    public
    view
    returns (
        bool
    )
{
    if (_txChainElement.isSequenced == true) {
        return _verifySequencerTransaction(
            _transaction,
            _txChainElement,
            _batchHeader,
            _inclusionProof
        );
    } else {
        return _verifyQueueTransaction(
            _transaction,
            _txChainElement.queueIndex,
            _batchHeader,
            _inclusionProof
        );
    }
```

```
    }
```

Figure 17.1: `verifyTransaction` function

This function is implemented using `_verifyQueueTransaction` in the case that the transaction in question comes from the queue. This function receives as parameters the transaction to be verified, the index of the corresponding queue element, as well as the batch header where it was included and a Merkle proof.

```
function _verifyQueueTransaction(
        Lib_OVMCodec.Transaction memory _transaction,
        uint256 _queueIndex,
        Lib_OVMCodec.ChainBatchHeader memory _batchHeader,
        Lib_OVMCodec.ChainInclusionProof memory _inclusionProof
    )
        internal
        view
        returns (
            bool
        )
    {
        bytes32 leafHash = _getQueueLeafHash(_inclusionProof.index);
```

Figure 17.2. `_verifyQueueTransaction` function

However, since transactions from the queue are saved (as a leaf of the Merkle tree) in a format following `_getQueueLeafHash`, the function must convert it to that format. The `_queueIndex` is the correct parameter to pass in the following line, not the `_inclusionProof.index`.

Since the two variables will be equal in the long run with negligible probability, then it is likely this line will always return an incorrect leaf hash.

Exploit Scenario
Eve submit arbitrary state roots to trigger this issue and make the inclusion proof fail. So, no fraud verification will be able to be initialized.

Recommendation
Short term, change the variable that is employed in `_verifyQueueTransaction` to use `_queueIndex` instead of `_inclusionProof.index`.

Long term, carefully review the use unit tests to verify correctness of the system.

18. `appendSequencerBatch` reverts if queue is empty

Severity: Medium | Difficulty: Low
Type: Data Validation | Finding ID: TOB-OVM-18
Target: `OVM_CanonicalTransactionChain.sol`

Description
If the sequencer tries to append a batch when the queue is empty, this operation will fail.

The sequencer user has the privilege to insert transactions at arbitrary points of the queue. This user should call `appendSequencerBatch` to append transactions to the Canonical Transaction Chain. In the body, there is a jump to `_validateBatchContext`:

```
function appendSequencerBatch()
    override
    public
{
    uint40 shouldStartAtBatch;
    uint24 totalElementsToAppend;
    uint24 numContexts;
    assembly {
        shouldStartAtBatch    := shr(216, calldataload(4))
        totalElementsToAppend := shr(232, calldataload(9))
        numContexts           := shr(232, calldataload(12))
    }


    ...
    uint40 nextQueueIndex = _getNextQueueIndex();

    for (uint32 i = 0; i < numContexts; i++) {
        BatchContext memory context = _getBatchContext(i);
        _validateBatchContext(context, nextQueueIndex);

    ...
}
```

Figure 18.1. part of the appendSequencerBatch function

`_validateBatchContext` then retrieves the element's hash and its metadata:

```
function _validateBatchContext(
    BatchContext memory _context,
    uint40 _nextQueueIndex
)
    internal
    view
{
    if (queue.getLength() == 0) {
        return;
    }


    Lib_OVMCodec.QueueElement memory nextQueueElement = getQueueElement(_nextQueueIndex);
    ...
```

Figure 18.2. part of the `_validateBatchContext` function

However, if the queue is "empty" (i.e. `_nextQueueIndex == queue.getLength()`), then this function will revert. Hence in at least two situations, the `appendSequencerBatch` will revert, even though successful execution would be expected:

1. Queue is empty in the pre-state.
2. All queued elements are included by the sequencer, and there is another batch context.

Exploit Scenario
Alice is a sequencer that submits a batch when the queue is empty. She is expecting the call to succeed, but instead it reverts, leading to unintended consequences.

Recommendation
Short term, make sure `appendSequencerBatch` succeeds even if the queue is empty.

Long term, use extensive unit tests to ensure correctness of all interactions in all situations.

19. Monotonicity of L2's timestamp and block number can be violated

Severity: High
Type: Data Validation
Target: `OVM_CanonicalTransactionChain.sol`

Difficulty: Medium
Finding ID: TOB-OVM-019

Description
The sequenced transactions can be enqueued in L2 using invalid timestamps and block numbers that are impossible in L1.

The sequencer user has the privilege to insert transactions at arbitrary points of the queue. This user should call `appendSequencerBatch` to append transactions to the Canonical Transaction Chain. In the body, there is a jump to `_validateBatchContext`:

```
function appendSequencerBatch()
    override
    public
{
    uint40 shouldStartAtBatch;
    uint24 totalElementsToAppend;
    uint24 numContexts;
    assembly {
        shouldStartAtBatch    := shr(216, calldataload(4))
        totalElementsToAppend := shr(232, calldataload(9))
        numContexts           := shr(232, calldataload(12))
    }


    ...
    uint40 nextQueueIndex = _getNextQueueIndex();

    for (uint32 i = 0; i < numContexts; i++) {
        BatchContext memory context = _getBatchContext(i);
        _validateBatchContext(context, nextQueueIndex);

    ...
}
```

Figure 19.1. part of the `appendSequencerBatch` function

`_validateBatchContext` performs checks on the timestamp and block number before allowing a transaction to be inserted:

```
function _validateBatchContext(
    BatchContext memory _context,
    uint40 _nextQueueIndex
)
    internal
    view
{
    if (queue.getLength() == 0) {
        return;
    }

    Lib_OVMCodec.QueueElement memory nextQueueElement = getQueueElement(_nextQueueIndex);

    require(
        block.timestamp < nextQueueElement.timestamp + forceInclusionPeriodSeconds,
        "Older queue batches must be processed before a new sequencer batch."
    );

    require(
        _context.timestamp <= nextQueueElement.timestamp,
        "Sequencer transactions timestamp too high."
    );

    require(
        _context.blockNumber <= nextQueueElement.blockNumber,
        "Sequencer transactions blockNumber too high."
    );
}
```

Figure 19.2. `_validateBatchContext` function

However, this does not fully enforce monotonicity of the timestamp and block number. In particular, sequenced transactions may have a lower value than a previous queue element, and sequenced transactions in a later batch context may also have a lower value.

Exploit Scenario

The sequencer appends transactions in an order that violates the guarantee of increasing block timestamp and block numbers. As a result, execution on L2 leads to unexpected results.

Recommendation
Short term, make sure the code enforces full monotonicity.

Long term, use Manticore or Echidna to ensure invariants hold under all situations.

20. Fraud verification reverts if submitted within the force inclusion period

Severity: Medium | Difficulty: Medium
Type: Data Validation | Finding ID: TOB-OVM-20
Target: `OVM_CanonicalTransactionChain.sol`

Description
Certain transactions submitted by the sequencer are impossible to challenge during the fraud verification procedure.

The sequencer user has the privilege to insert transactions at arbitrary points of the queue. This user should call `appendSequencerBatch` to append transactions to the Canonical Transaction Chain. In the body, there is a jump to `_validateBatchContext`:

```
function appendSequencerBatch()
    override
    public
{
    uint40 shouldStartAtBatch;
    uint24 totalElementsToAppend;
    uint24 numContexts;
    assembly {
        shouldStartAtBatch    := shr(216, calldataload(4))
        totalElementsToAppend := shr(232, calldataload(9))
        numContexts           := shr(232, calldataload(12))
    }


        ...
        for (uint32 j = 0; j < context.numSubsequentQueueTransactions; j++) {
            leaves[transactionIndex] = _getQueueLeafHash(nextQueueIndex);
            nextQueueIndex++;
            transactionIndex++;
        }
        ...
}
```

Figure 20.1. part of the `appendSequencerBatch` function

`_getQueueLeafHash` performs some checks on the timestamp of the next queue:

```
function _getQueueLeafHash(
    uint256 _index
)
    internal
    view
    returns (
        bytes32
    )
{
    Lib_OVMCodec.QueueElement memory element = getQueueElement(_index);


    require(
        msg.sender == sequencer
        || element.timestamp + forceInclusionPeriodSeconds <= block.timestamp,
        "Queue transactions cannot be submitted during the sequencer inclusion period."
    );


    return _hashTransactionChainElement(
        Lib_OVMCodec.TransactionChainElement({
            isSequenced: false,
            queueIndex: _index,
            timestamp: 0,
            blockNumber: 0,
            txData: hex""
        })
    );
}
```

Figure 20.2. part of the `_getQueueLeafHash` function

The intentions of the require are so that enqueued elements can be included by the sequencer (through `appendSequencerBatch`) within a time period (after they can be included by anyone, including the sequencer, through `appendQueueBatch`). However, this require will run every time this getter is called, even from a different mechanism.

Exploit Scenario

Bob notices an incorrect state root and initializes the fraud proof process. In particular, it will run during initialization of the fraud proof involving a queued transaction:

1. `FraudVerifier.initializeFraudVerification`

2. Calls `ovmCTC.verifyTransaction`
3. Jumps to `_verifyQueueTransaction`
4. Jumps to `_getQueueLeafHash`

He expects the call to succeed, but it reverts, which could lead to unexpected results.

Recommendation
Short term, make sure the fraud process can be initialized under all expected circumstances.

Long term, review all the corner cases in the fraud verification to make sure it cannot be blocked.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program
Code Quality	Related to conforming to industry best practices of code

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is

	important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Initialize the canOverwrite variable in `push` (`optimistic-ethereum/libraries/utils/Lib_RingBuffer.sol#L94-L95`)**. Explicit initialization will make the code easier to understand, maintain, and review.
- **Initialize the local variables in `_getUpdatedTrieRoot` (`optimistic-ethereum/libraries/trie/Lib_MerkleTrie.sol#L495-L497`)**. Explicit initialization will make the code easier to understand, maintain, and review.
- **Consider validating the inputs for the `fromNibbles` function (`optimistic-ethereum/libraries/utils/Lib_BytesUtils.sol#L212-L226`)**. This function accepts inputs that are not produced by the `toNibbles` function and produces unexpected results. Additional validation will protect against untrusted inputs, if this function is re-used in the future.
- **Consider renaming `l1TxOrigin` as `l1MsgSender`. (`optimistic-ethereum/libraries/codec/Lib_OVMCodec.sol#L82`)** `Lib_OVMCodec` defines a struct `Transaction` with a field called `l1TxOrigin`. If the transaction is a sequenced one, it is equal to `0x0`. If the transaction is from the queue, it is set to the `msg.sender` of that transaction, not the `tx.origin`. As such, we feel this field should be called l1MsgSender.