Optimisim Interop Proofs

Security Review

Cantina Managed review by:

Zigtur, Lead Security Researcher

LonelySloth, Lead Security Researcher

Sujith Somraaj, Security Researcher

May 7, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Optimism is a fast, stable, and scalable L2 blockchain built by Ethereum developers, for Ethereum developers. Built as a minimal extension to existing Ethereum software, Optimism's EVM-equivalent architecture scales your Ethereum apps without surprises. If it works on Ethereum, it works on Optimism at a fraction of the cost.

From Mar 18th to Mar 29th the Cantina team conducted a review of optimism on commit hash 9d86edb5. The team identified a total of **25** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	2	2	0
High Risk	2	1	1
Medium Risk	1	1	0
Low Risk	1	1	0
Gas Optimizations	0	0	0
Informational	19	9	10
Total	**25**	**14**	**11**

3 Findings

3.1 Critical Risk

3.1.1 Incorrect error type when initiating message not found leads to state transition failing, unprovable new state

Severity: Critical Risk

Context: consolidate.go#L294

Description: When the log index in an execute message log is too big (more than number of logs in the block), the `Contains` check returns an error that isn't correctly handled. The error message is untyped and and the check in `isInvalidMessageError` will return `false` for them. That means the error will not trigger the replacement of the block with a "deposits only" version (expected behavior).

Instead, the state transition fails. This allows a single invalid execute message log in a single block to prevent the super chain state transition from executing -- and thus make the new state uprovable.

Proof of Concept: This test case can be added to `interop_test.go`:

```
{
    name: "ReplaceChainB-LogIndexTooBig",
    testCase: consolidationTestCase{
        logBuilderFn: func(includeBlockNumbers map[supervisortypes.ChainIndex]uint64, config
        ↪ *staticConfigSource) map[supervisortypes.ChainIndex][]*gethTypes.Log {
            init1 := &gethTypes.Log{
                Address: initiatingMessageOrigin,
                Topics:  []common.Hash{initiatingMessageTopic},
            }
            init2 := &gethTypes.Log{
                Address: initiatingMessageOrigin2,
                Topics:  []common.Hash{initiatingMessageTopic},
            }
            exec := createExecMessage(includeBlockNumbers[chainA], config, chainA)
            exec.Identifier.Origin = init2.Address
            exec.Identifier.LogIndex = 1_000_000
            return map[supervisortypes.ChainIndex][]*gethTypes.Log{
                chainA: {init1, init2},
                chainB: {convertExecutingMessageToLog(t, exec)},
            }
        },
        expectBlockReplacements: func(config *staticConfigSource) []supervisortypes.ChainIndex {
            return []supervisortypes.ChainIndex{chainB}
        },
    },
},
```

Recommendation: The error returned should be of a type that can trigger a block replacement. For example:

```
return supervisortypes.BlockSeal{}, fmt.Errorf("log not found: %w", supervisortypes.ErrConflict)
```

Optimism: Fixed in PR 15376.

Cantina Managed: Fixed. The `ErrConflict` error is now returned to trigger a block replacement when initiating message is not found.

3.1.2 Incorrect error type when initiating message block not found leads to state transition failing, unprovable new state

Severity: Critical Risk

Context: consolidate.go#L350

Description: When the block number in an execute message is too big (more than the height of the canonical chain plus one), an error message will be returned by the error message is untyped and the check in `isInvalidMessageError` will return false for them. That means the error will not trigger the replacement of the block with a "deposits only" version (expected behavior).

Instead, the state transition fails. This allows a single invalid execute message log in a single block to prevent the super chain state transition from executing -- and thus make the new state uprovable.

Proof of Concept: This test case can be added to `interop_test.go`:

```
{
    name: "ReplaceChainB-BlockNumberTooBig",
    testCase: consolidationTestCase{
        logBuilderFn: func(includeBlockNumbers map[supervisortypes.ChainIndex]uint64, config
        ↪  *staticConfigSource) map[supervisortypes.ChainIndex][]*gethTypes.Log {
            init1 := &gethTypes.Log{
                Address: initiatingMessageOrigin,
                Topics:  []common.Hash{initiatingMessageTopic},
            }
            init2 := &gethTypes.Log{
                Address: initiatingMessageOrigin2,
                Topics:  []common.Hash{initiatingMessageTopic},
            }
            exec := createExecMessage(1_000_000, config, chainA)
            return map[supervisortypes.ChainIndex][]*gethTypes.Log{
                chainA: {init1, init2},
                chainB: {convertExecutingMessageToLog(t, exec)},
            }
        },
        expectBlockReplacements: func(config *staticConfigSource) []supervisortypes.ChainIndex {
            return []supervisortypes.ChainIndex{chainB}
        },
    },
},
```

Recommendation: The error returned should be of a type that can trigger a block replacement. For example:

```
return nil, fmt.Errorf("head not found for chain %v %w", chainID, supervisortypes.ErrConflict)
```

Additionally I recommend a thorough review of all error messages, their types, and whether they should trigger a block replacement or a state transition failure.

Optimism: Fixed in PR 15376.

Cantina Managed: Fixed. The `ErrConflict` error is now returned to trigger a block replacement when initiating message block is not found.

3.2 High Risk

3.2.1 Checking for message expiration after receipt processing prevents pruning indexes, increase resources

Severity: High Risk

Context: hazard_set.go#L85

Description: In the present implementation the initiating message/s timestamp is checked for expiration after the message is retrieved from the block. This defeats the purpose of limiting the need of keeping receipts indexed forever -- and actually make nodes that prune their indexes unable to prove a state transition.

Running the op-program would require having all receipts from all blocks available regardless of expiration settings, since any message from any receipt from any block might need to be retrieved for validation.

Recommendation: The check for expiration should be moved to just after the block header is retrieved in `CanonBlockByNumber` (or in `Contains`) and before any further data from the block is retrieved (especially the receipts).

Note: this might also affect other systems such as supervisor/sequencer.

Optimism: Fixed in PR 15553.

Cantina Managed: Fixed. The new `checkMessageForExpiry` function verifies the executed message timestamp before processing it to stop processing early if the message is expired.

3.2.2 The same message can be marked as executing multiple times without being executed

Severity: High Risk

Context: consolidate.go#L25-L42

Description: Currently, the `op-program` retrieves executing messages by parsing logs from the block receipts. Each log in the block is passed to `DecodeExecutingMessageLog`. If an `ExecutingMessage` log coming from the `CrossL2Inbox` contract is found, it is collected in `execMsgs` for processing. However, the current implementation of `CrossL2Inbox` allows emitting multiple times the same `ExecutingMessage` log without actually executing the message.

Code snippet: The `CrossL2Inbox` smart contract allows emitting multiple times the same log:

```
contract CrossL2Inbox is ISemver {
    // ...
    function validateMessage(Identifier calldata _id, bytes32 _msgHash) external {
        // We need to know if this is being called on a depositTx
        if (IL1BlockInterop(Predeploys.L1_BLOCK_ATTRIBUTES).isDeposit()) revert NoExecutingDeposits();

        emit ExecutingMessage(_msgHash, _id);
    }
}
```

Then, the `op-program` will individually collect and process all these occurrences of the same message.

Exploit scenario: A malicious party creates a smart contract that calls `CrossL2Inbox.validateMessage` in a loop. This will create thousands of `ExecutingMessage` logs. This can be done simultaneously on all chains in the dependency set. All these `ExecutingMessage` logs will be collected and processed by the `op-program`. This could lead to unprovable fault in the FPVM due to heavy computations.

Recommendation: Executing multiple times the same processing should be avoided. `op-program` could check for duplicated executing messages. Another way to fix the issue is to limit the `ExecutingMessage` log in `CrossL2Inbox` to one per message.

Optimism: A benchtest was conducted to see if this is an issue. The setup involves 1000 messages initiated and executed in the same block, for each chain. This results in 21M gas per chain, while not near the gas limit, it's large enough that we can make projections.

Doing this requires roughly 2.7 billion steps to run the consolidate routine in Cannon. Which is about 10 minutes on a Ryzen 7 3700X. So this case is also pretty cheap to generate a fault proof.

For future reference, the benchtest was conducted using the code in PR 15642.

Cantina Managed: Acknowledged.

3.3 Medium Risk

3.3.1 Quadratic resource consumption in validating existence of initiating messages

Severity: Medium Risk

Context: consolidate.go#L267

Description: `Contains` iterates through each log in the entire block, requiring all receipts from the block being processed, that is a $\mathcal{O}(n)$ complexity with `n` being the block size. As this function is called for each executing message to check if the corresponding initiating message exists, in the worst case scenario this is $\mathcal{O}(n^2)$ with the size of the block. With the cost of producing logs and executing messages being low in terms of gas, this is likely to result in significant resource consumption above the necessary.

Since the consolidation process is already $\mathcal{O}(m^2)$ with the number of chains, this could lead to the impossibility of proving a state transition as the number of chains scale. If other issues are found that further compound resource consumption, this could lead to more severe impact even with small number of chains in the dependency set.

Recommendation: This should be corrected to consume sublinear resources. For example a simple proof of existence would be $\mathcal{O}(\log(n))$. Other approaches such as indexing all logs by message hash could be explored.

Optimism: Fixed in PR 15379.

Cantina Managed: The PR above improved performance in some scenarios. For other scenarios, project team reported running a benchmark and verifying the performance is acceptable despite the quadratic resource consumption.

3.4 Low Risk

3.4.1 Invalid modulo check in `unmarshalSuperRootV1` leads to panic

Severity: Low Risk

Context: super_root.go#L98-L101

Description: `unmarshalSuperRootV1` executes a modulo operation to check that the input data has complete output roots. However, this check is invalid. It should check for complete chain ID and output root pairs:

```go
func unmarshalSuperRootV1(data []byte) (*SuperV1, error) {
    // ...
    // Must contain complete chain output roots
    if (len(data)-9)%32 != 0 { // @audit invalid, 32 should be 64
        return nil, ErrInvalidSuperRoot
    }
```

Proof of Concept: The following patch can be applied to import the unit test:

```diff
diff --git a/op-service/eth/super_root_test.go b/op-service/eth/super_root_test.go
index ba873985b..8009f6734 100644
--- a/op-service/eth/super_root_test.go
+++ b/op-service/eth/super_root_test.go
@@ -2,6 +2,7 @@ package eth

 import (
    "encoding/binary"
+   "fmt"
    "testing"

    "github.com/stretchr/testify/require"
@@ -33,6 +34,31 @@ func TestSuperRootV1Codec(t *testing.T) {
        require.Equal(t, superRoot, *unmarshaledV1)
    })

+   t.Run("OutOfRangeRead", func(t *testing.T) {
+       chainA := ChainIDAndOutput{ChainID: ChainIDFromUInt64(11), Output: Bytes32{0x01}}
+       chainB := ChainIDAndOutput{ChainID: ChainIDFromUInt64(12), Output: Bytes32{0x02}}
+       chainC := ChainIDAndOutput{ChainID: ChainIDFromUInt64(13), Output: Bytes32{0x03}}
+       superRoot := SuperV1{
+           Timestamp: 7000,
+           Chains:    []ChainIDAndOutput{chainA, chainB, chainC},
+       }
+       // @POC: byteArray is superRoot encoded, with 32 bytes removed
+       byteArray := []byte{
+           1,
+           0, 0, 0, 0, 0, 0, 27, 88,
+           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11,
+           1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12,
+           2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 13,
+       }
+       unmarshaled, err := UnmarshalSuperRoot(byteArray)
+       require.NoError(t, err)
+       unmarshaledV1 := unmarshaled.(*SuperV1)
+       fmt.Println("unmarshaled", unmarshaledV1)
+       require.Equal(t, superRoot, *unmarshaledV1)
+   })
+
    t.Run("BelowMinLength", func(t *testing.T) {
        _, err := UnmarshalSuperRoot(append([]byte{SuperRootVersionV1}, 0x01))
        require.ErrorIs(t, err, ErrInvalidSuperRoot)
```

Then, run the test with the following command:

```
go test -timeout 30s -run ^TestSuperRootV1Codec$/^OutOfRangeRead$
↪  github.com/ethereum-optimism/optimism/op-service/eth -v
```

It outputs these results:

```
=== RUN   TestSuperRootV1Codec
=== RUN   TestSuperRootV1Codec/OutOfRangeRead
--- FAIL: TestSuperRootV1Codec (0.00s)
    --- FAIL: TestSuperRootV1Codec/OutOfRangeRead (0.00s)
panic: runtime error: slice bounds out of range [:201] with capacity 169 [recovered]
        panic: runtime error: slice bounds out of range [:201] with capacity 169
```

Recommendation: Consider replacing `% 32` by `% 64` check if there are complete `ChainIDAndOutput` structures:

```
diff --git a/op-service/eth/super_root.go b/op-service/eth/super_root.go
index dbe80991a..51bcb2410 100644
--- a/op-service/eth/super_root.go
+++ b/op-service/eth/super_root.go
@@ -95,8 +95,8 @@ func unmarshalSuperRootV1(data []byte) (*SuperV1, error) {
    if len(data) < SuperRootVersionV1MinLen {
        return nil, ErrInvalidSuperRoot
    }
-   // Must contain complete chain output roots
-   if (len(data)-9)%32 != 0 {
+   // Must contain complete chain chainIDs and output roots
+   if (len(data)-9)%64 != 0 {
        return nil, ErrInvalidSuperRoot
    }
    var output SuperV1
```

Optimism: Fixed in PR 15708.

Cantina: Fixed. The modulo operation is now executed with `chainIDAndOutputLen = 64`.

3.5 Informational

3.5.1 Invalid `SuperRootVersionV1MinLen` constant

Severity: Informational

Context: super_root.go#L93-L113

Description: `unmarshalSuperRootV1` checks that the input data length is not lower than `1 + 8 + 32`. If it is, an error indicating that the super root data is invalid is returned.

However, this length check is incorrect. The `SuperRootVersionV1MinLen` constant does not account for the expected `ChainIDAndOutput.ChainID` field which adds an additional 32 bytes.

Code snippet: The `SuperRootVersionV1MinLen` constant is `41`. According to the associated comment, it corresponds to the version (1 byte), the timestamp (8 bytes) and one chain output root hash (32 bytes):

```
const (
    // SuperRootVersionV1MinLen is the minimum length of a V1 super root prior to hashing
    // Must contain a 1 byte version, uint64 timestamp and at least one chain's output root hash
    SuperRootVersionV1MinLen = 1 + 8 + 32
)
```

However, the decoding in `unmarshalSuperRootV1` shows that it expects an additional `ChainID` field which is 32 bytes long:

```go
func unmarshalSuperRootV1(data []byte) (*SuperV1, error) {
    // Must contain the version, timestamp and at least one output root.
    if len(data) < SuperRootVersionV1MinLen {
        return nil, ErrInvalidSuperRoot
    }
    // Must contain complete chain output roots
    if (len(data)-9)%32 != 0 {
        return nil, ErrInvalidSuperRoot
    }
    var output SuperV1
    // data[:1] is the version
    output.Timestamp = binary.BigEndian.Uint64(data[1:9])
    for i := 9; i < len(data); i += 64 {
        chainOutput := ChainIDAndOutput{
            ChainID: ChainIDFromBytes32([32]byte(data[i : i+32])), // @audit read 32 bytes
            Output:  Bytes32(data[i+32 : i+64]),                   // @audit read another 32 bytes
        }
        output.Chains = append(output.Chains, chainOutput)
    }
    return &output, nil
}
```

Recommendation: The `SuperRootVersionV1MinLen` constant must account for the `ChainID` field:

```diff
diff --git a/op-service/eth/super_root.go b/op-service/eth/super_root.go
index dbe80991a..56fd104c1 100644
--- a/op-service/eth/super_root.go
+++ b/op-service/eth/super_root.go
@@ -20,8 +20,8 @@ var (

 const (
    // SuperRootVersionV1MinLen is the minimum length of a V1 super root prior to hashing
-   // Must contain a 1 byte version, uint64 timestamp and at least one chain's output root hash
-   SuperRootVersionV1MinLen = 1 + 8 + 32
+   // Must contain a 1 byte version, uint64 timestamp and at least one chain's chainId and output root hash
↪  pair
+   SuperRootVersionV1MinLen = 1 + 8 + 32 + 32
 )

 type Super interface {
```

Optimism: Fixed in PR 15708.

Cantina Managed: Fixed. The `SuperRootVersionV1MinLen` now uses `chainIDAndOutputLen = 64` instead of 32.

3.5.2 Bedrock contracts test setup fails due to incorrect Solady library mapping

Severity: Informational

Context: ForkLive.s.sol#L17

Description: The `ForkLive.s.sol` file imports the Solady library with the following:

```solidity
import { LibString } from "solady/src/utils/LibString.sol";
```

However, the Foundry remapping already accounts for the `src` directory. This makes compilation through `forge build` fail.

Recommendation: Fix the import in `ForkLive.s.sol`.

```
diff --git a/packages/contracts-bedrock/test/setup/ForkLive.s.sol
↪  b/packages/contracts-bedrock/test/setup/ForkLive.s.sol
index 40e9c94e3..7ac455bd5 100644
--- a/packages/contracts-bedrock/test/setup/ForkLive.s.sol
+++ b/packages/contracts-bedrock/test/setup/ForkLive.s.sol
@@ -14,7 +14,7 @@ import { Deploy } from "scripts/deploy/Deploy.s.sol";
 // Libraries
 import { GameTypes, Claim } from "src/dispute/lib/Types.sol";
 import { EIP1967Helper } from "test/mocks/EIP1967Helper.sol";
-import { LibString } from "solady/src/utils/LibString.sol";
+import { LibString } from "solady/utils/LibString.sol";

 // Interfaces
 import { IFaultDisputeGame } from "interfaces/dispute/IFaultDisputeGame.sol";
```

Optimism: Fixed in PR 15249.

Cantina Managed: Fixed. The import has been fixed.

3.5.3 Incorrect documentation regarding number of blocks per state transition

Severity: Informational

Context: consolidate.go#L195

Description: The high-level overview says:

> State transitions on the Superchain happen once per second. Each individual chain within that time can produce a block, several blocks, or no block.

(See `https://hackmd.io/@clabby/ByF1fLivJe`).

However, that assertion is incorrect since only at most one block per chain per state-transition can be processed by the state transition function.

Recommendation: All documentation should be reviewed and corrected to remove the false assertion regarding number of blocks per state transition.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.4 Useless version check in `parseAgreedState`

Severity: Informational

Context: interop.go#L133-L135

Description: The `parseAgreedState` function executes the following check.

```
func parseAgreedState(bootInfo *boot.BootInfoInterop, l2PreimageOracle l2.Oracle) (*types.TransitionState,
↪  *eth.SuperV1, error) {
    // ...
    if transitionState.Version() != types.IntermediateTransitionVersion {
        return nil, nil, fmt.Errorf("%w: %v", ErrIncorrectOutputRootType, transitionState.Version())
    }
```

However, `TransitionState.Version()` function always return the `IntermediateTransitionVersion` constant. This makes the version check useless.

```
func (i *TransitionState) Version() byte {
    return IntermediateTransitionVersion
}
```

Recommendation: The useless check can be removed.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.5 Invalid pending progress length check in `stateTransition`

Severity: Informational

Context: interop.go#L110-L112

Description: During the consolidation step, the following length check is executed.

```go
func stateTransition(logger log.Logger, bootInfo *boot.BootInfoInterop, l1PreimageOracle l1.Oracle,
↪ l2PreimageOracle l2.Oracle, tasks taskExecutor) (common.Hash, error) {
    // ...
    } else if transitionState.Step == ConsolidateStep {
        logger.Info("Running consolidate step")
        // sanity check
        if len(transitionState.PendingProgress) >= ConsolidateStep { // @audit length check here is incorrect
            return common.Hash{}, fmt.Errorf("pending progress length does not match the expected step")
        }
        // ...
    }

        // ...
}
```

This sanity check is incorrect as it will fail when `transitionState.PendingProgress == ConsolidateStep` with `ConsolidateStep` being 127. However, there might be 127 elements because each step from 0 to 126 may add an element to `PendingProgress`.

Note: The current dependency set is not expected to reach the 127 chains hard limit, making this issue unlikely to occur.

Recommendation: The length check must use the > operand instead of >=:

```diff
diff --git a/op-program/client/interop/interop.go b/op-program/client/interop/interop.go
index 5174ab04f..04e463004 100644
--- a/op-program/client/interop/interop.go
+++ b/op-program/client/interop/interop.go
@@ -107,7 +107,7 @@ func stateTransition(logger log.Logger, bootInfo *boot.BootInfoInterop, l1Preima
    } else if transitionState.Step == ConsolidateStep {
        logger.Info("Running consolidate step")
        // sanity check
-       if len(transitionState.PendingProgress) >= ConsolidateStep {
+       if len(transitionState.PendingProgress) > ConsolidateStep {
            return common.Hash{}, fmt.Errorf("pending progress length does not match the expected step")
        }
        expectedSuperRoot, err := RunConsolidation(
```

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.6 Invalid `unmarshalTransitionSate` function name

Severity: Informational

Context: roots.go#L62

Description: The `unmarshalTransitionSate` function name is incorrect. It has a typographical error in it.

Recommendation: Replace `unmarshalTransitionSate` by `unmarshalTransitionState`.

Optimism: Fixed in PR15708.

Cantina Managed: Fixed. The function is now correctly named.

3.5.7 Dependency set is retrieved at each `singleRoundConsolidation` call

Severity: Informational

Context: consolidate.go#L111-L124

Description: The `singleRoundConsolidation` function is called in a for loop from `RunConsolidation`. At each round, `singleRoundConsolidation` retrieves the constant dependency set. This dependency set could be retrieved only once in `RunConsolidation` and passed as an input argument for efficiency purposes.

Recommendation: Consider retrieving the dependency set only once in `RunConsolidation` and then passing it as a parameter to `singleRoundConsolidation`.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.8 Incorrect error type when timestamp invariant is broken -- might possibly lead to state transition failing

Severity: Informational

Context: hazard_set.go#L180

Description/Description: When the timestamp invariant is broken the error message is untyped and and the check in isInvalidMessageError will return false for them. While no path has been so far found to exploit this particular error -- this should be changed to a properly typed error message.

Optimism: Fixed in PR 15149.

Cantina Managed: Fix verified.

3.5.9 Untyped error message in cycle detection

Severity: Informational

Context: cycle.go#L104

Description/Description: This error message in `cycle.go` is untyped and thus might plausibly result in state transition failure. No path for exploiting has been found though. However, it's still recommended that the error be properly typed.

Optimism: Fixed in PR 15422.

Cantina Managed: Fixed. Optimism changed the code to return explicitly the typed error message that causes program failure. That seems reasonable given it's an unexpected condition, and likely unrecoverable.

3.5.10 Optimization suggestion: `block.timestamp` can be checked before receipts

Severity: Informational

Context: consolidate.go#L281

Description: Since the timestamp is a property of the block, there is no need to retrieve the receipt before checking it. The timestamp validation could be moved to the line before the retrieval of receipts. This would save considerable resources if there's an execution with an invalid timestamp.

Recommendation: The timestamp validation should be done before receipts retrieval.

Optimism: Fixed in PR 15709.

Cantina Managed: Fix verified.

3.5.11 Unbounded `ConsolidateStep` execution

Severity: Informational

Context: interop.go#L107

Description: The `RunConsolidation` function can be invoked multiple times once the transition state reaches step 127 (`ConsolidateStep`), as the implementation does not increment the step counter after consolidation occurs. In the current implementation of `stateTransition`, there's a logical fork that handles different step processing:

For steps less than the length of `superRoot.Chains`:

- An optimistic block is derived.
- The step counter is incremented before returning.

For step 127 (consolidation):

- The consolidation process is executed.
- The result is returned directly, resulting in the step being executed multiple times.

```go
func stateTransition(logger log.Logger, bootInfo *boot.BootInfoInterop, l1PreimageOracle l1.Oracle,
↪  l2PreimageOracle l2.Oracle, tasks taskExecutor) (common.Hash, error) {
    // Initial checks and setup...

    if transitionState.Step < uint64(len(superRoot.Chains)) {
        logger.Info("Deriving optimistic block")
        block, err := deriveOptimisticBlock(/*...*/);
        // Process block...
        expectedPendingProgress = append(expectedPendingProgress, block)
    } else if transitionState.Step == ConsolidateStep {
        logger.Info("Running consolidate step")
        // sanity check
        if len(transitionState.PendingProgress) >= ConsolidateStep {
            return common.Hash{}, fmt.Errorf("pending progress length does not match the expected step")
        }
        expectedSuperRoot, err := RunConsolidation(
            logger, bootInfo, l1PreimageOracle, l2PreimageOracle, transitionState, superRoot, tasks)
        if err != nil {
            return common.Hash{}, err
        }
        return common.Hash(expectedSuperRoot), nil  // Early return with no step increment
    }

    // Step increment only happens for non-consolidation steps
    finalState := &types.TransitionState{
        SuperRoot:       transitionState.SuperRoot,
        PendingProgress: expectedPendingProgress,
        Step:            transitionState.Step + 1,
    }
    return finalState.Hash(), nil
}
```

Recommendation: To avoid consolidating multiple times, increment the step post-consolidation and add a terminal state check to panic (or) return an error.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.12 `HazardUnsafeFrontierChecks` function call is not needed

Severity: Informational

Context: consolidate.go#L220-L222, consolidate.go#L302-L310, unsafe_frontier.go#L25-L62

Description: The consolidation executed by the op-program makes the assumption that the blocks are cross-unsafe. This strong assumption makes all execution happening in `HazardUnsafeFrontierChecks` useless as `IsCrossUnsafe` and `IsLocalUnsafe` will never return any error.

Recommendation: The call to `HazardUnsafeFrontierChecks` can be removed from `checkHazards`.

Optimism: Fixed in PR 15709.

Cantina Managed: Fixed. The call to `HazardUnsafeFrontierChecks` has been removed.

3.5.13 Bypass of consolidation step due to unbounded `superRoot.Chains` length

Severity: Informational

Context: interop.go#L98

Description: A sanity check limits `PendingProgress` length to less than `ConsolidateStep` (127), but no similar validation exists for `superRoot.Chains` length in the `stateTransition()` function. If `superRoot.Chains` length exceeds 127, consolidation is never reached, causing potential system stalling. In `stateTransition()`, function, the code uses the following logic to decide between deriving an optimistic block or consolidation:

```
if transitionState.Step < uint64(len(superRoot.Chains)) {
    // Derive optimistic block
    // ...
} else if transitionState.Step == ConsolidateStep {
    // Run consolidation step
    // ...
}
```

The vulnerability arises because:

- It should enter consolidation when `transitionState.Step` reaches `ConsolidateStep` (127).
- If `superRoot.Chains` length is over 127, the first condition is always true.
- Thus, the system keeps deriving optimistic blocks or increments the step.

Recommendation: Implement a validation check early in the process to ensure superRoot.Chains length is bounded:

```
func stateTransition(logger log.Logger, bootInfo *boot.BootInfoInterop, l1PreimageOracle l1.Oracle,
↪   l2PreimageOracle l2.Oracle, tasks taskExecutor) (common.Hash, error) {
    // Existing code...

    transitionState, superRoot, err := parseAgreedState(bootInfo, l2PreimageOracle)
    if err != nil {
        return common.Hash{}, err
    }

    // Add this validation
    if len(superRoot.Chains) > ConsolidateStep {
        return common.Hash{}, fmt.Errorf("chains length exceeds maximum allowed value of %d", ConsolidateStep)
    }

    // Rest of the function...
}
```

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.14 No validation of chain order in `SuperRoot` during state transition

Severity: Informational

Context: interop.go#L137

Description: The `stateTransition()` function currently assumes that `SuperRoot.Chains` are properly ordered without explicitly validating this assumption. While `UnmarshalSuperRoot()` may expect an ordered list, there is no explicit check during the state transition process to ensure the ordering is correct. An improperly ordered list of chains could lead to inconsistent derivation results. In the `parseAgreedState()` function, the code unmarshals the `SuperRoot`:

```
super, err := eth.UnmarshalSuperRoot(transitionState.SuperRoot)
if err != nil {
    return nil, nil, fmt.Errorf("invalid super root: %w", err)
}
```

Recommendation: Add an explicit validation step in the `parseAgreedState()` function to verify the chains are properly ordered to ensure the validity of `SuperRoot` before running the derivation.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.15 Optimization suggestion: exit hazard checks loop after first block replacement

Severity: Informational

Context: consolidate.go#L161

Description: The current "eager" implementation of the hazard checking within a consolidation step keeps iterating over all chains even after a block is marked for replacement (but before actual replacement). While this shouldn't cause any problems in the final result, the blocks found to be valid will still need to be rechecked after the block replacement is actually processed. As most blocks are expected to pass the hazard check this results in unnecessary extra processing.

Recommendation: A more efficient approach would be to break the loop after the first failed hazard check, replace the block, and advance to the next step -- with all blocks not yet replaced are checked, again exiting early if any hazard check fails.

Optimism: Addressed in issue 15203.

Cantina Managed: Verified.

3.5.16 Executing message timestamp should be checked before any further execution

Severity: Informational

Context: hazard_set.go#L161-L181

Description: The `HazardSet.build` function executes the message timestamp check after checking the source chain ID and retrieving the source block. This leads to unnecessary computations when the message timestamp is invalid (message is expired or is in the future).

Recommendation: Consider executing timestamp checks before any further processing. The following properties should be checked:

- The message is not expired.
- The message is not in the future.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.17 New chain in the dependency set should create a block on activation time to avoid deposits-only block

Severity: Informational

Context: hazard_set.go#L54-L60

Description: `HazardSet.checkChainCanExecute` function is called during `CrossUnsafeHazards` execution. This function ensures that a given chain can execute messages based on the `block.Timestamp`:

```go
// hazard_set.go
func (h *HazardSet) checkChainCanExecute(depSet depset.DependencySet, chainID eth.ChainID, block
↪  types.BlockSeal, execMsgs map[uint32]*types.ExecutingMessage) error {
    if len(execMsgs) > 0 {
        if ok, err := depSet.CanExecuteAt(chainID, block.Timestamp); err != nil {
            return fmt.Errorf("cannot check message execution of block %s (chain %s): %w", block, chainID, err)
        } else if !ok {
            return fmt.Errorf("cannot execute messages in block %s (chain %s): %w", block, chainID,
            ↪  types.ErrConflict)
        }
    }
    return nil
}

// static.go
func (ds *StaticConfigDependencySet) CanExecuteAt(chainID eth.ChainID, execTimestamp uint64) (bool, error) {
    dep, ok := ds.dependencies[chainID]
    if !ok {
        return false, nil
    }
    return execTimestamp >= dep.ActivationTime, nil
}
```

This timestamp check can lead to invalidating a block right after the activation.

Scenario: The initial dependency set is configured with 3 chains:

- OP mainnet with a 2s block time.
- Unichain with a 1s block time.
- Randomchain with a 3s block time.

The dependency activation time is set to `dep.ActivationTime = 100`. Then, during the consolidation step at timestamp `100`, the following blocks will be passed:

- OP mainnet: block produced at timestamp 100 (new block).
- Unichain: block produced at timestamp 100 (new block).
- Randomchain: block produced at timestamp 99 (already known).

This already known block will not pass the `checkChainCanExecute` as the block it provides is at timestamp 99. This block will be replaced to a deposits-only block. However, the block could have already been checked and verified after timestamp 99.

Recommendation: Activation time for the dependency set should be set such as all chains in the dependency set are producing new blocks. Another way to fix the issue would be to only process deposits-only blocks around the activation timestamp.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.18 Blocks are retrieved multiple times from oracle

Severity: Informational

Context: consolidate.go#L215-L227, cycle.go#L93-L98, hazard_set.go#L124-L131

Description: `checkHazards` is called to verify executing messages of a candidate block and their dependencies on other blocks of the dependency set.

Currently, the `checkHazards` function executes 3 different functions:

- `CrossUnsafeHazards`: the hazard set is built for the candidate. This function calls the oracle to open blocks that will be added to the hazard set.
- `HazardUnsafeFrontierChecks`.
- `HazardCycleChecks`: builds a graph for the hazard set. It re-opens every block in the hazard set by calling the oracle.

When `CrossUnsafeHazards` adds a block to the hazard set, it opens this block by calling the oracle but only tracks the `BlockSeal` data (i.e. the hash, number and timestamp of the block) in the `HazardSet.entries`. This leads to the `HazardCycleChecks` logic to interact a second time with the oracle for every block in the hazard set.

```go
type BlockSeal struct {
    Hash      common.Hash
    Number    uint64
    Timestamp uint64
}
```

Recommendation: The `HazardSet.entries` could directly store the opened block such that multiple interactions with the oracle are avoided.

Note: this will be more memory consuming.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.19 Improve `transitionState` sanity check during consolidation

Severity: Informational

Context: interop.go#L110

Description: The current implementation compares the length of `transitionState.PendingProgress` against the constant `ConsolidateStep` (value: `127`), unrelated to the actual length requirement.

```go
} else if transitionState.Step == ConsolidateStep {
    logger.Info("Running consolidate step")
    // sanity check
    if len(transitionState.PendingProgress) >= ConsolidateStep {
        return common.Hash{}, fmt.Errorf("pending progress length does not match the expected step")
    }
    // ...
```

This check fails to validate that the `PendingProgress` array has the correct number of elements to match the `superRoot.Chains` array. In the subsequent `RunConsolidation()` function, the code accesses elements from both arrays using the same indices, assuming a one-to-one correspondence.

```go
func RunConsolidation(
    logger log.Logger,
    bootInfo *boot.BootInfoInterop,
    l1PreimageOracle l1.Oracle,
    l2PreimageOracle l2.Oracle,
    transitionState *types.TransitionState,
    superRoot *eth.SuperV1,
    tasks taskExecutor,
) (eth.Bytes32, error) {
    consolidateState := consolidateState{
        TransitionState: &types.TransitionState{
            PendingProgress: make([]types.OptimisticBlock, len(transitionState.PendingProgress)),
            SuperRoot:       transitionState.SuperRoot,
            Step:            transitionState.Step,
        },
        replacedChains: make(map[eth.ChainID]bool),
    }
    // We will be updating the transition state as blocks are replaced, so make a copy
    copy(consolidateState.PendingProgress, transitionState.PendingProgress)

       // ...

    var consolidatedChains []eth.ChainIDAndOutput
    for i, chain := range superRoot.Chains {
        consolidatedChains = append(consolidatedChains, eth.ChainIDAndOutput{
            ChainID: chain.ChainID,
            Output:  consolidateState.PendingProgress[i].OutputRoot,
        })
    }

        // ...
}
```

This could lead to:

- Array Index Out of Bounds: If `PendingProgress` has fewer elements than `superRoot.Chains`, the program will attempt to access non-existent elements in the `PendingProgress` array.
- Silent Data Inconsistency: If `PendingProgress` has more elements than `superRoot.Chains`, the extra elements will be ignored, potentially leading to state inconsistencies.

Recommendation: Add sanity checks to directly validate the relationship between `PendingProgress` and `superRoot.Chains`:

```go
if len(transitionState.PendingProgress) != len(superRoot.Chains) {
    return common.Hash{}, fmt.Errorf("pending progress length (%d) does not match chains length (%d)",
                                     len(transitionState.PendingProgress), len(superRoot.Chains))
}
```

This ensures that there is exactly one progress entry for each chain, which is the invariant required for the consolidation process to work correctly.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.20 Error handling inconsistency in super root type validation

Severity: Informational

Context: interop.go#L142, super_root.go#L89

Description: The codebase uses different error variables (`ErrIncorrectOutputRootType` and `ErrInvalid-SuperRootVersion`) to represent the same error in two different contexts, which creates ambiguity in the error diagnosis. In `interop.go`:

```
func parseAgreedState(bootInfo *boot.BootInfoInterop, l2PreimageOracle l2.Oracle) (*types.TransitionState,
↪   *eth.SuperV1, error) {
    // ....
    if super.Version() != eth.SuperRootVersionV1 {
    return nil, nil, fmt.Errorf("%w: %v", ErrIncorrectOutputRootType, super.Version())
    }
    // ...
}
```

In `super_root.go`.

```
func UnmarshalSuperRoot(data []byte) (Super, error) {
    if len(data) < 1 {
        return nil, ErrInvalidSuperRoot
    }
    ver := data[0]
    switch ver {
    case SuperRootVersionV1:
        return unmarshalSuperRootV1(data)
    default:
        return nil, ErrInvalidSuperRootVersion
    }
}
```

Recommendation: Consider returning the same error for handling version incompatibility of super roots across the entire codebase to facilitate easier debugging.

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.5.21 Remove unused parameter `candidate` from `checkChainCanInitiate` function

Severity: Informational

Context: hazard_set.go#L67

Description: The `candidate` parameter (of type types.BlockSeal) is declared but never used within the `checkChainCanInitiate()` function body. The function only uses:

- `depSet`: To call `CanInitiateAt`.
- `initChainID`: Passed to `CanInitiateAt` and used in error messages.
- `msg`: To access its Timestamp and for error messages.

Recommendation: Consider removing the unused parameter if unnecessary, or utilize it appropriately.

Optimism: Fixed in commit faac706e.

Cantina Managed: Fix verified.