CANTINA

Interop portal

Competition

June 4, 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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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 24th to Apr 7th Cantina hosted a competition based on interop-portal. The participants identified a total of **26** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 7
- Gas Optimizations: 0
- Informational: 19

The fixes implemented by the Op Labs team were reviewed by the Lead Security Researcher Phaze.

3 Findings

3.1 Low Risk

3.1.1 The challenger does not receive the defender's bond if the protocol is paused at the time of game closure

Severity: Low Risk

Context: FaultDisputeGame.sol#L1027-L1035

Description: The `FaultDisputeGame.closeGame` function is essentially used to determine the bond distribution mode. This mode depends on whether the game is considered proper at the time of closure, as shown in the following code snippet:

```
bool properGame = ANCHOR_STATE_REGISTRY.isGameProper(IDisputeGame(address(this)));

// If the game is a proper game, the bonds should be distributed normally. Otherwise, go
// into refund mode and distribute bonds back to their original depositors.
if (properGame) {
    bondDistributionMode = BondDistributionMode.NORMAL;
} else {
    bondDistributionMode = BondDistributionMode.REFUND;
}
```

The problem arises because a game can be temporarily deemed improper if the protocol is paused, as seen in AnchorStateRegistry.sol#L222-L224:

```
function isGameProper(IDisputeGame _game) public view returns (bool) {
    // Must be registered in the DisputeGameFactory.
    if (!isGameRegistered(_game)) {
        return false;
    }

    // Must not be blacklisted.
    if (isGameBlacklisted(_game)) {
        return false;
    }

    // Must be created at or after the respectedGameTypeUpdatedAt timestamp.
    if (isGameRetired(_game)) {
        return false;
    }

    // Must not be paused, temporarily causes game to be considered improper.
    if (paused()) {
        return false;
    }

    return true;
}
```

This means that if the challenger wins but the protocol is paused at the moment `closeGame` is called, the bond distribution mode is set to `REFUND`, and the challenger does not receive the bond. As a result, despite acting honestly the challenger incurs a direct financial loss, having spent gas fees without any compensation.

The specification also confirms that the game might be only temporarily improper and may become proper again once the pause is lifted:

> ALL Dispute Games TEMPORARILY become Improper Games while the Pause Mechanism is active. However, this is a temporary condition such that Registered Games that are not invalidated by blacklisting or retirement will become Proper Games again once the pause is lifted.

In this context, setting the bond distribution mode to `REFUND` when the game's final propriety is unknown seems incorrect, as it harms honest challengers and violates the following invariant from the specification:

> The honest challenger is refunded the bond for every claim it posts and paid the bond of the parent of that claim.

Note: *The same issue exists in `SuperFaultDisputeGame`, but it is out of scope for this contest.*

Proof of Concept: Add this test to `FaultDisputeGame_Test` in `test/dispute/FaultDisputeGame.t.sol` and run it:

```
function test_poc_pause_triggers_refund_mode() public {
    // Confirm that the anchor state is older than the game state.
    (Hash root, uint256 l2BlockNumber) = anchorStateRegistry.anchors(gameProxy.gameType());
    assert(l2BlockNumber < gameProxy.l2BlockNumber());

    // Challenge the claim and resolve it.
    (,,,, Claim disputed,,) = gameProxy.claimData(0);
    gameProxy.attack{ value: _getRequiredBond(0) }(disputed, 0, _dummyClaim());
    vm.warp(block.timestamp + 3 days + 12 hours);
    gameProxy.resolveClaim(1, 0);
    gameProxy.resolveClaim(0, 0);
    assertEq(uint8(gameProxy.resolve()), uint8(GameStatus.CHALLENGER_WINS));

    // Wait for finalization delay.
    vm.warp(block.timestamp + 3.5 days + 1 seconds);

    // Pause the protocol
    vm.prank(superchainConfig.guardian());
    superchainConfig.pause("testing");

    // Close the game.
    gameProxy.closeGame();

    // Confirm that the anchor state is the same as the initial anchor state.
    (Hash updatedRoot, uint256 updatedL2BlockNumber) = anchorStateRegistry.anchors(gameProxy.gameType());
    assertEq(updatedL2BlockNumber, l2BlockNumber);
    assertEq(updatedRoot.raw(), root.raw());

    // The bond distribution mode was set to REFUND despite the fact that the game is
    // technically only temporarily improper. Challenger wins, but doesn't receive the bond.
    assert(gameProxy.bondDistributionMode() == BondDistributionMode.REFUND);
}
```

Recommendation: Considering that some games may become proper again once the pause is lifted, it likely makes sense to revert the execution of `closeGame` if the protocol is paused. This would delay game closure but ensure the correct outcome.

Phaze: Fixed in commit b671b67. `closeGame()` in `FaultDisputeGame` and `SuperFaultDisputeGame` now check the `AnchorStateRegistry`'s `paused()` state and revert if it is paused.

3.1.2 Missing recency check in `setAnchorState()`

Severity: Low Risk

Context: AnchorStateRegistry.sol#L293-L315

Summary: `AnchorStateRegistry.setAnchorState()` lacks a check to ensure the L1 timestamp of a dispute game's L2 sequence number is within 6 months, violating recency requirements and risking fault proof failures.

Finding Description: `setAnchorState()` does not verify that the L1 timestamp associated with a dispute game's `l2SequenceNumber` (via an L1 timestamp oracle) is no older than 6 months, as required by `iASR-005`. This requirement ensures fault proofs can traverse L1 blocks efficiently. Without this check, an attacker could supply a finalized dispute game with an outdated L1 origin (e.g., > 6 months old), which `setAnchorState` accepts as a valid anchor. This breaks the security guarantee of reliable fault proofs, as verifying such an anchor would require traversing an impractical number of L1 blocks, and resolution would require intervention from the Proxy Admin Owner.

The issue arises because `setAnchorState()` only validates game finality, registration, and status, but omits recency. A malicious actor could exploit this by:

1. Creating or referencing a finalized game with an old `l2SequenceNumber`.
2. Passing it to `setAnchorState()`, which updates the anchor without checking the L1 timestamp age.

See the specification for more details.

Impact Explanation: High impact because it undermines fault proof verification, a critical security mechanism in the Optimism ecosystem. If an anchor's L1 origin is too old, fault proofs may fail due to excessive

block traversal, potentially allowing invalid states to persist unchallenged. While not an immediate funds loss, it compromises the system's integrity and reliability, warranting a high severity.

Likelihood Explanation: Medium. Exploiting this requires an attacker to control or reference a finalized dispute game with an L2 sequence number tied to an L1 timestamp > 6 months old, which isn't trivial but feasible in a long-running system where old games exist. The absence of a recency check makes this a passive vulnerability, exploitable when conditions align.

Proof of Concept: Add the following test to `test/dispute/AnchorStateRegistry.t.sol::AnchorStateRegistry_SetAnchorState_TestFail`:

```
/// @notice Tests that setAnchorState accepts an old game without recency check, violating iASR-005.
function test_anchorGameRecencyViolation() public {
    // Set a current anchor for context
    vm.mockCall(address(gameProxy), abi.encodeCall(gameProxy.resolvedAt, ()), abi.encode(block.timestamp - 7));
    vm.mockCall(address(gameProxy), abi.encodeCall(gameProxy.status, ()), abi.encode(GameStatus.DEFENDER_WINS));
    vm.mockCall(address(gameProxy), abi.encodeCall(gameProxy.l2SequenceNumber, ()), abi.encode(500));
    vm.warp(block.timestamp + anchorStateRegistry.disputeGameFinalityDelaySeconds() + 1);
    anchorStateRegistry.setAnchorState(gameProxy);

    // Mock a new game with an old L2 block (L1 origin > 6 months ago)
    IDisputeGame oldGameProxy = IDisputeGame(address(0x1234));
    vm.mockCall(address(oldGameProxy), abi.encodeCall(gameProxy.resolvedAt, ()), abi.encode(block.timestamp -
    ↪  7));
    vm.mockCall(address(oldGameProxy), abi.encodeCall(gameProxy.status, ()),
    ↪  abi.encode(GameStatus.DEFENDER_WINS));
    uint256 oldL2Block = 600; // Higher than current anchor (500)
    vm.mockCall(address(oldGameProxy), abi.encodeCall(gameProxy.l2SequenceNumber, ()), abi.encode(oldL2Block));
    vm.mockCall(
        address(oldGameProxy),
        abi.encodeCall(gameProxy.gameData, ()),
        abi.encode(GameType.wrap(0), gameProxy.rootClaim(), gameProxy.extraData())
    );
    vm.mockCall(
        address(oldGameProxy), abi.encodeCall(gameProxy.wasRespectedGameTypeWhenCreated, ()), abi.encode(true)
    );
    vm.mockCall(address(oldGameProxy), abi.encodeCall(gameProxy.createdAt, ()), abi.encode(block.timestamp));
    vm.mockCall(
        address(disputeGameFactory),
        abi.encodeCall(disputeGameFactory.games, (GameType.wrap(0), gameProxy.rootClaim(),
        ↪  gameProxy.extraData())),
        abi.encode(address(oldGameProxy), block.timestamp)
    );

    // Hypothetical L1 timestamp oracle
    uint256 sixMonthsAgo = block.timestamp - (6 * 30 days + 1); // > 6 months
    address l1TimestampOracle = address(0x5678);
    vm.mockCall(
        address(l1TimestampOracle),
        abi.encodeCall(IL1TimestampOracle.getL1Timestamp, (oldL2Block)),
        abi.encode(sixMonthsAgo)
    );

    // Set the old game as anchor - no recency check exists
    anchorStateRegistry.setAnchorState(oldGameProxy);

    // Verify bug: Old game is set despite > 6 months L1 origin
    assertEq(address(anchorStateRegistry.anchorGame()), address(oldGameProxy));
    // Confirm the L1 timestamp is > 6 months old
    assertGt(block.timestamp - sixMonthsAgo, 6 * 30 days, "L1 timestamp is not older than 6 months");
}
```

Recommendation: Add a recency check in `setAnchorState` to reject games with L1 timestamps older than 6 months.

3.1.3 New `ETHLockbox` address can be a target for withdrawal transactions breaking the invariant

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: `OptimismPortal` has the following invariant:

> The contract MUST NOT allow withdrawals to target the ETHLockbox address.

See the specification for more details.

While this invariant is strictly held for the current `ethLockbox` address set in the `OptimismPortal`, it can be broken for the newly initialized `ETHLockbox` if called from an authorized but not migrated portal.

Finding Description: The root cause of this issue is that many portals may be authorized to the `ETHLockbox` during initialization, and they do not have to migrate at the same time/instantly. Simultaneously, only the currently set `ethLockbox` address is verified in the `_isUnsafeTarget()` function.

Impact Explanation: Broken core invariant, allowing targeting `ETHLockbox` address with a withdrawal. Thanks to the following check, `unlockETH()` cannot be called to misplace funds to a different portal, even after breaking the invariant:

```
// Check that the sender is not executing a withdrawal transaction.
if (sender.l2Sender() != Constants.DEFAULT_L2_SENDER) {
    revert ETHLockbox_NoWithdrawalTransactions();
}
```

It can only be used to call `lockETH()` on the new `ETHLockbox`.

Recommendation: Consider removing `authorizePortal()` call from the `initialize()` function to ensure the invariant is held for **ALL** ETHLockbox contracts. Ensure it is called during `ETHLockbox` migration to avoid other issues.

Proof of Concept: The following test case shows how one can target a newly initialized `ETHLockbox` before the portal migrates to it. Place it in `OptimismPortal2.t.sol`:

```
test_finalizeWithdrawalTransaction_badTarget_DoesNotRevertForNewLockbox() external {
    ProxyAdmin proxyAdmin = ProxyAdmin(artifacts.mustGetAddress("ProxyAdmin"));

    IETHLockbox ethLockbox = IETHLockbox(optimismPortal2.ethLockbox());

    // Deploy the new ETHLockbox
    IETHLockbox newEthLockbox = IETHLockbox(address(new Proxy(address(proxyAdmin))));

    // Can be the same because it's only to simulate the scenario
    vm.prank(address(proxyAdmin));
    address ethLockboxImpl = Proxy(payable(address(ethLockbox))).implementation();

    IOptimismPortal[] memory portalsToAuthorize = new IOptimismPortal[](1);
    portalsToAuthorize[0] = optimismPortal2;

    vm.startPrank(address(ethLockbox.proxyAdminOwner()));
    proxyAdmin.upgradeAndCall(payable(address(newEthLockbox)), ethLockboxImpl,
    ↪  abi.encodeCall(IETHLockbox.initialize,
    (optimismPortal2.superchainConfig(), portalsToAuthorize)));


    // We need to recreate the mock root proofs for new tx
    Types.WithdrawalTransaction memory _ethLockboxTargetTx = Types.WithdrawalTransaction({
        nonce: 0,
        sender: alice,
        target: address(address(newEthLockbox)),
        value: 100,
        gasLimit: 100_000,
        data: hex"1ee116bf"
        });

    (_stateRoot, _storageRoot, _outputRoot, _withdrawalHash, _withdrawalProof) =
        ffi.getProveWithdrawalTransactionInputs(_ethLockboxTargetTx);

    _outputRootProof = Types.OutputRootProof({
        version: bytes32(uint256(0)),
        stateRoot: _stateRoot,
        messagePasserStorageRoot: _storageRoot,
        latestBlockhash: bytes32(uint256(0))
    });

    // And a mock game:
    vm.warp(anchorStateRegistry.retirementTimestamp() + 1);

    GameType respectedGameType = optimismPortal2.respectedGameType();
    game = IFaultDisputeGame(
        payable(
```

```
            address(
                disputeGameFactory.create{ value: disputeGameFactory.initBonds(respectedGameType) }(
                    respectedGameType, Claim.wrap(_outputRoot), abi.encode(_proposedBlockNumber)
                )
            )
        )
    );
    _proposedGameIndex = disputeGameFactory.gameCount() - 1;
    vm.warp(block.timestamp + game.maxClockDuration().raw() + 1 seconds);

    optimismPortal2.proveWithdrawalTransaction({
        _tx: _ethLockboxTargetTx,
        _disputeGameIndex: _proposedGameIndex,
        _outputRootProof: _outputRootProof,
        _withdrawalProof: _withdrawalProof
    });

    // Warp and resolve the dispute game.
    game.resolveClaim(0, 0);
    game.resolve();
    vm.warp(block.timestamp + optimismPortal2.proofMaturityDelaySeconds() + 1 seconds);


    vm.expectEmit();
    emit WithdrawalFinalized(_withdrawalHash, true);
    optimismPortal2.finalizeWithdrawalTransaction(_ethLockboxTargetTx);

    assertEq(address(newEthLockbox).balance, 100);
}
```

3.1.4 Addition of new chains to the super anchor root will not be possible

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: With reference to the `AnchorStateRegistry` contract, let's suppose the Super Anchor Root consisting of 3 chains and we denote the Super Anchor Root as `{A, B, C}`. If we want to add a new chain to the interop set with an anchor root D, such that it gets added to a Super Anchor Root such that it becomes `{A, B, C, D}`, it would not be possible. This is because:

- The super anchor root is only updated through `initialize()` and `setAnchorState()`.
- We are in a scenario where the ASR is already initialized so `initialize()` isn't relevant. Also noting that ASR isn't upgradeable.
- That means `setAnchorState()` must be called if we want to update the super anchor root.
- To even play the Super FDG for the newly added chain, there must be an anchor root D for the newly added chain stored in the Super Anchor Root.
- Therefore, we can conclude it isn't possible to update the anchor root to include the newly added anchor root from `setAnchorState()`.

As such in the current version of the `AnchorStateRegistry`, it would be impossible to add more chains into the interop set.

Recommendation: Have a function `updateAnchorState` callable by the proxy admin to update the anchor root to allow adding more chains into the interop set.

3.1.5 Withdrawal hashes can still be proven after being finalized, allowing spamming of events for the Op Supervisor

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The new update allows multiple chains using Optimism Portals to connect to an Op Supervisor which controls interactions between chains by observing their log events. Proving and finalizing withdrawals are some of the most important aspects of each Optimism portal. Withdrawal hashes can

be proven using a MerkleTrie calculation of the root (which must be part of the Super root), and the withdrawal proof.

After being proven, the withdrawals can be finalized if the dispute game associated with that withdrawal is resolved successfully to the defender. A withdrawal hash can be proven multiple times and by multiple users but can only be finalized once.

However, even after being finalized, withdrawal hashes can still be proven, because the prove function does not confirm that the given withdrawalHash is not finalized.

```
function _proveWithdrawalTransaction(
    Types.WithdrawalTransaction memory _tx,
    IDisputeGame _disputeGameProxy,
    uint256 _outputRootIndex,
    Types.SuperRootProof memory _superRootProof,
    Types.OutputRootProof memory _outputRootProof,
    bytes[] memory _withdrawalProof
) internal {
    // ...

    // Load the ProvenWithdrawal into memory, using the withdrawal hash as a unique identifier.
    bytes32 withdrawalHash = Hashing.hashWithdrawal(_tx);

    //audit: Missing check for if withdrawal hash is finalized

    // Compute the storage slot of the withdrawal hash in the L2ToL1MessagePasser contract.
    // Refer to the Solidity documentation for more information on how storage layouts are
    // computed for mappings.
    bytes32 storageKey = keccak256(
        abi.encode(
            withdrawalHash,
            uint256(0) // The withdrawals mapping is at the first slot in the layout.
        )
    );

    // Verify that the hash of this withdrawal was stored in the L2toL1MessagePasser contract
    // on L2. If this is true, under the assumption that the SecureMerkleTrie does not have
    // bugs, then we know that this withdrawal was actually triggered on L2 and can therefore
    // be relayed on L1.
    if (
        SecureMerkleTrie.verifyInclusionProof({
            _key: abi.encode(storageKey),
            _value: hex"01",
            _proof: _withdrawalProof,
            _root: _outputRootProof.messagePasserStorageRoot
        }) == false
    ) {
        revert OptimismPortal_InvalidMerkleProof();
    }

    // Designate the withdrawalHash as proven by storing the disputeGameProxy and timestamp in
    // the provenWithdrawals mapping. A given user may re-prove a withdrawalHash multiple
    // times, but each proof will reset the proof timer.
    provenWithdrawals[withdrawalHash][msg.sender] = ProvenWithdrawal({
        disputeGameProxy: _disputeGameProxy,
        timestamp: uint64(block.timestamp)
    });

    // Add the proof submitter to the list of proof submitters for this withdrawal hash.
    proofSubmitters[withdrawalHash].push(msg.sender);

    // Emit a WithdrawalProven events.
    emit WithdrawalProven(withdrawalHash, _tx.sender, _tx.target);
    emit WithdrawalProvenExtension1(withdrawalHash, msg.sender);
}
```

The Op Supervisor stores every log event since every log event can potentially initiate a cross domain message. This is stated in the interop docs:

> OP-Supervisor holds a database of all the log events of all the chains in the Superchain interoperability cluster. Every event can potentially initiate a cross-domain message, and it is the job of OP-Supervisor to validate that the log event really happened on the source chain.

See the explainer for more details.

This allows anybody to flood the Op Supervisor with logs of proving withdrawal hashes that have already been finalized, potentially causing resources constraints to the execution clients, Op Supervisor and that chain.

Proof of Concept: Insert the below test into `packages/contracts-bedrock/test/L1/OptimismPortal2.t.sol`. This simple test illustrates how the withdrawal hashes can still be proven after being finalized.

Run with `forge test --mt test_audit_finalizedWithdrawalTransaction_canStillBeProven -vvvv`:

```
function test_audit_finalizedWithdrawalTransaction_canStillBeProven()
        external
    {
        vm.expectEmit(address(optimismPortal2));
        emit WithdrawalProven(_withdrawalHash, alice, bob);
        vm.expectEmit(address(optimismPortal2));
        emit WithdrawalProvenExtension1(_withdrawalHash, address(this));
        optimismPortal2.proveWithdrawalTransaction({
            _tx: _defaultTx,
            _disputeGameIndex: _proposedGameIndex,
            _outputRootProof: _outputRootProof,
            _withdrawalProof: _withdrawalProof
        });

        // Warp and resolve the dispute game.
        game.resolveClaim(0, 0);
        game.resolve();
        vm.warp(
            block.timestamp +
                optimismPortal2.proofMaturityDelaySeconds() +
                1 seconds
        );

        vm.expectEmit(true, true, false, true);
        emit WithdrawalFinalized(_withdrawalHash, true);

        optimismPortal2.finalizeWithdrawalTransaction(_defaultTx);

        //audit: Prove that withdrawal hashes can still be proven after finalizing
        vm.expectEmit(address(optimismPortal2));
        emit WithdrawalProven(_withdrawalHash, alice, bob);
        vm.expectEmit(address(optimismPortal2));
        emit WithdrawalProvenExtension1(_withdrawalHash, address(this));
        optimismPortal2.proveWithdrawalTransaction({
            _tx: _defaultTx,
            _disputeGameIndex: _proposedGameIndex,
            _outputRootProof: _outputRootProof,
            _withdrawalProof: _withdrawalProof
        });
    }
```

See gist 005a0a85 for the output. Note both events are still emitted even for withdrawals which have already been finalized.

Recommendation: Add a validation whether the withdrawalHash has been finalized before proving.

```
  function _proveWithdrawalTransaction(
      Types.WithdrawalTransaction memory _tx,
      IDisputeGame _disputeGameProxy,
      uint256 _outputRootIndex,
      Types.SuperRootProof memory _superRootProof,
      Types.OutputRootProof memory _outputRootProof,
      bytes[] memory _withdrawalProof
  ) internal {
      // ...

      // Load the ProvenWithdrawal into memory, using the withdrawal hash as a unique identifier.
      bytes32 withdrawalHash = Hashing.hashWithdrawal(_tx);

+     if (finalizedWithdrawals[_withdrawalHash]) {
+         revert OptimismPortal_AlreadyFinalized();
+     }

      // Compute the storage slot of the withdrawal hash in the L2ToL1MessagePasser contract.
      // Refer to the Solidity documentation for more information on how storage layouts are
      // computed for mappings.
```

```
        bytes32 storageKey = keccak256(
            abi.encode(
                withdrawalHash,
                uint256(0) // The withdrawals mapping is at the first slot in the layout.
            )
        );

        // Verify that the hash of this withdrawal was stored in the L2toL1MessagePasser contract
        // on L2. If this is true, under the assumption that the SecureMerkleTrie does not have
        // bugs, then we know that this withdrawal was actually triggered on L2 and can therefore
        // be relayed on L1.
        if (
            SecureMerkleTrie.verifyInclusionProof({
                _key: abi.encode(storageKey),
                _value: hex"01",
                _proof: _withdrawalProof,
                _root: _outputRootProof.messagePasserStorageRoot
            }) == false
        ) {
            revert OptimismPortal_InvalidMerkleProof();
        }

        // Designate the withdrawalHash as proven by storing the disputeGameProxy and timestamp in
        // the provenWithdrawals mapping. A given user may re-prove a withdrawalHash multiple
        // times, but each proof will reset the proof timer.
        provenWithdrawals[withdrawalHash][msg.sender] = ProvenWithdrawal({
            disputeGameProxy: _disputeGameProxy,
            timestamp: uint64(block.timestamp)
        });

        // Add the proof submitter to the list of proof submitters for this withdrawal hash.
        proofSubmitters[withdrawalHash].push(msg.sender);

        // Emit WithdrawalProven events.
        emit WithdrawalProven(withdrawalHash, _tx.sender, _tx.target);
        emit WithdrawalProvenExtension1(withdrawalHash, msg.sender);
    }
```

3.1.6 We are not checking for ascending chain ids sorting in output roots when proving transaction with `SuperRoot`

Severity: Low Risk

Context: OptimismPortal2.sol#L513-L533

Summary: `OptimismPortal2.proveWithdrawalTransaction()` does not properly validate the outputRoot's chain ids, as it does not check if they follow the invariant of being sorted in ascending way.

Finding Description: Games can be used to prove withdrawal transactions and start the proveWithdrawal delay (which then allows to finalize a transaction). However we cannot call finalize for a withdraw transaction until the game used to prove the withdrawal is not valid and finalized.

The checks in `_proveWithdrawalTransaction()` ensure that in the cases where we know 100% that we won't ever allow `finalizeWithdrawalTransaction()` for a game (i.e. being retired/blacklisted & etc...) are covered and we do not allow the user to start the `PROOF_MATURITY_DELAY_SECONDS` and wait for nothing. Currently the `OptimismPortal2.proveWithdrawalTransaction()` has insufficient validation:

```
function _proveWithdrawalTransaction() internal
{
    // Make sure that the target address is safe.
    if (_isUnsafeTarget(_tx.target)) {
        revert OptimismPortal_BadTarget();
    }

    // Game must be a Proper Game.
    if (!anchorStateRegistry.isGameProper(_disputeGameProxy)) {
        revert OptimismPortal_ImproperDisputeGame();
    }

    // Game must have been respected game type when created.
    if (!anchorStateRegistry.isGameRespected(_disputeGameProxy)) {
        revert OptimismPortal_InvalidDisputeGame();
    }
```

```
// Game must not have resolved in favor of the Challenger (invalid root claim).
if (_disputeGameProxy.status() == GameStatus.CHALLENGER_WINS) {
    revert OptimismPortal_InvalidDisputeGame();
}

if (block.timestamp <= _disputeGameProxy.createdAt().raw()) {
    revert OptimismPortal_InvalidProofTimestamp();
}

if (superRootsActive) { // <<<
    // Verify that the super root can be generated with the elements in the proof.
    if (_disputeGameProxy.rootClaim().raw() != Hashing.hashSuperRootProof(_superRootProof)) {
        revert OptimismPortal_InvalidSuperRootProof();
    }

    // Check that the index exists in the super root proof.
    if (_outputRootIndex >= _superRootProof.outputRoots.length) {
        revert OptimismPortal_InvalidOutputRootIndex();
    }

    //@audit only verifying the chainId and the array length but not the order of the chains in the
    ↪  outputRoots array.
    // Check that the output root has the correct chain id.
    Types.OutputRootWithChainId memory outputRoot = _superRootProof.outputRoots[_outputRootIndex]; // <<<
    if (outputRoot.chainId != systemConfig.l2ChainId()) {
        revert OptimismPortal_InvalidOutputRootChainId();
    }

    // Verify that the output root can be generated with the elements in the proof.
    if (outputRoot.root != Hashing.hashOutputRootProof(_outputRootProof)) {
        revert OptimismPortal_InvalidOutputRootProof();
    }
} else {
    if (_disputeGameProxy.rootClaim().raw() != Hashing.hashOutputRootProof(_outputRootProof)) {
        revert OptimismPortal_InvalidOutputRootProof();
    }
}
```

It is possible that the following invariant for the super-roots can be broken for a game that is passed, and a withdrawal still to be proven:

See specs:

> The output roots in the super root MUST be sorted by chain ID ascending.

The problem is that once a game has set its root/claim, it cannot be changed, thus this will data will be final and eventually invalidated, however a User might have proven their transaction and wait 7 days for withdraw finalize delay for nothing, as the game would end up with `CHALLENGER_WINS` status.

Note: *Ascending chain id validation can very easily be overlooked, as we only do validation for the output root we care about when we prove a withdrawal*.

Impact Explanation: Some Users may pick seemingly valid SFDGame which turns out to be invalid, due to breaking the chain ID invariant.

Likelihood Explanation: Medium.

Proof of Concept: Modify the below proof of concept, as shown, and run it via: `forge test --mt "test_proveWithdrawalTransaction_superRootsVersion_succeeds" -vv`:

```
  function test_proveWithdrawalTransaction_superRootsVersion_succeeds() external {
      // Enable super roots.
      setSuperRootsActive(true);

      // Set up a dummy super root proof.
-     Types.OutputRootWithChainId[] memory outputRootWithChainIdArr = new Types.OutputRootWithChainId[](1);
+     Types.OutputRootWithChainId[] memory outputRootWithChainIdArr = new Types.OutputRootWithChainId[](3);
      outputRootWithChainIdArr[0] =
+         Types.OutputRootWithChainId({ root: _outputRoot, chainId: systemConfig.l2ChainId() });
+     outputRootWithChainIdArr[1] =
+         Types.OutputRootWithChainId({ root: _outputRoot, chainId: systemConfig.l2ChainId() + 3 });
+     outputRootWithChainIdArr[2] =
+         Types.OutputRootWithChainId({ root: _outputRoot, chainId: systemConfig.l2ChainId() + 1 });
      Types.SuperRootProof memory superRootProof = Types.SuperRootProof({
          version: 0x01,
          timestamp: uint64(block.timestamp),
          outputRoots: outputRootWithChainIdArr
      });

      // Figure out what the right hash would be.
      bytes32 expectedSuperRoot = Hashing.hashSuperRootProof(superRootProof);

      // Mock the game to return the expected super root.
      vm.mockCall(address(game), abi.encodeCall(game.rootClaim, ()), abi.encode(expectedSuperRoot));

      // Should succeed.
      optimismPortal2.proveWithdrawalTransaction({
          _tx: _defaultTx,
          _disputeGameProxy: game,
          _outputRootIndex: 0,
          _superRootProof: superRootProof,
          _outputRootProof: _outputRootProof,
          _withdrawalProof: _withdrawalProof
      });
  }
```

Recommendation: During the `proveWithdrawalTransaction()` check if the chain ids are in properly sorted in the array.

3.1.7 Missing superRoot timestamp validation with the output roots when doing `proveWithdrawal` transaction

Severity: Low Risk

Context: OptimismPortal2.sol#L478

Summary: Currently we are missing the superRoot proof timestamp validation, which in case its invalid would result in invalid Game/not ever validated and the User would have waited finalization delay to finalize its withdrawal for nothing.

Finding Description: The checks in `_proveWithdrawalTransaction()` ensure that the cases where we 100% won't ever allow `finalizeWithdrawalTransaction()` are covered and we do not allow the user to start the `PROOF_MATURITY_DELAY_SECONDS` and wait for a game that will 100% result in being retired/blacklisted/with status `CHALLENGER_WINS` & etc... This ensures that the user does not wait for nothing, in the cases where we can verify that the game will for sure be denied. Currently we do not validate in any way the timestamp from the `SuperRootProof` struct:

```
struct SuperRootProof {
    bytes1 version;
    uint64 timestamp;
    OutputRootWithChainId[] outputRoots;
}
```

```
struct OutputProposal {
    bytes32 outputRoot;
    uint128 timestamp;
    uint128 l2BlockNumber;
}
```

in `Optimism2Portal.proveWithdrawalTransaction()`. Fully described information about its validation requirements can be found in the specs:

> The output root for each chain in the super root MUST be for the block with a timestamp where `TimeSBlockTime<TimeB <=TimeS` where `TimeS` is the super root timestamp, `BlockTime` is the chain block time and `TimeB` is the block timestamp. That is the output root must be from the last possible block at or before the super root timestamp.

Impact Explanation: Some Users may pick seemingly valid SFDGame which turns out to be invalid, due to breaking the timestamp invariant. As we cannot possibly have a new anchor state game with the same or older timestamp.

Likelihood Explanation: Medium likelihood.

Proof of Concept: Just run the modified proof of concepy with 0 set as the timestamp, to verify that we do not validate it, use `forge test --mt "test_proveWithdrawalTransaction_superRootsVersion_succeeds" -vv` to run it:

```
  function test_proveWithdrawalTransaction_superRootsVersion_succeeds() external {
      // Enable super roots.
      setSuperRootsActive(true);

      // Set up a dummy super root proof.
      Types.OutputRootWithChainId[] memory outputRootWithChainIdArr = new Types.OutputRootWithChainId[](1);
      outputRootWithChainIdArr[0] =
          Types.OutputRootWithChainId({ root: _outputRoot, chainId: systemConfig.l2ChainId() });
      Types.SuperRootProof memory superRootProof = Types.SuperRootProof({
          version: 0x01,
-         timestamp: uint64(block.timestamp),
+         timestamp: uint64(0),
          outputRoots: outputRootWithChainIdArr
      });

      // Figure out what the right hash would be.
      bytes32 expectedSuperRoot = Hashing.hashSuperRootProof(superRootProof);

      // Mock the game to return the expected super root.
      vm.mockCall(address(game), abi.encodeCall(game.rootClaim, ()), abi.encode(expectedSuperRoot));

      // Should succeed.
      optimismPortal2.proveWithdrawalTransaction({
          _tx: _defaultTx,
          _disputeGameProxy: game,
          _outputRootIndex: 0,
          _superRootProof: superRootProof,
          _outputRootProof: _outputRootProof,
          _withdrawalProof: _withdrawalProof
      });
  }
```

Recommendation: Validate the timestamps in the way described in the specs, before allowing for a withdrawal to be proved.

3.2 Informational

3.2.1 `getAnchorRoot()` might return blacklisted anchors as valid checkpoints

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `AnchorStateRegistry` fails to validate the current `anchorGame` in `getAnchorRoot()`, allowing blacklisted or invalid games to persist as trusted anchors. This breaks the contract's core security guarantee of being the source of truth for the validity of Dispute Game instances.

The contract promises to provide only finalized, unchallenged, and non-blacklisted state roots as anchors. However `getAnchorRoot()` returns the stored `anchorGame`'s root claim without checking if the game as blacklisted or if the game is still valid.

Attack Path:

- A malicious actor creates a fraudulent game (`GameA`) that resolves incorrectly.
- `GameA` becomes the anchor through `setAnchorstate()`.
- Later the guardian blacklist `GameA` after recognizing the fraud.

- `getAnchorRoot()` still returns `GameA`'s root tricking downstream contracts into accepting invalid state transitions.

Impact Explanation: I consider this a high because bridges, validators, or rollup clients relying on `getAnchorRoot()` will pass on corrupt state as if it were valid which could lead to fraudulent withdrawals and chain integrity failure.

Proof of Concept:

```
function test_PoC_BlacklistedAnchorStillReturned() public {
    // 1. Get the initial anchor state (startingAnchorRoot)
    (Hash startingRoot, uint256 startingBlockNum) = anchorStateRegistry.getAnchorRoot();

    // 2. Setup a valid game and make it the anchor
    vm.mockCall(address(gameProxy), abi.encodeCall(gameProxy.resolvedAt, ()), abi.encode(block.timestamp));
    vm.warp(block.timestamp + optimismPortal2.disputeGameFinalityDelaySeconds() + 1);
    vm.mockCall(address(gameProxy), abi.encodeCall(gameProxy.status, ()), abi.encode(GameStatus.DEFENDER_WINS));
    vm.mockCall(address(gameProxy), abi.encodeCall(gameProxy.wasRespectedGameTypeWhenCreated, ()),
    ↪  abi.encode(true));

    // Set as anchor
    anchorStateRegistry.setAnchorState(gameProxy);
    (Hash initialRoot, uint256 initialBlockNum) = anchorStateRegistry.getAnchorRoot();

    // 3. Verify anchor was set correctly
    assertEq(initialRoot.raw(), gameProxy.rootClaim().raw());
    assertEq(initialBlockNum, gameProxy.l2SequenceNumber());

    // 4. Now blacklist the game
    vm.prank(superchainConfig.guardian());
    anchorStateRegistry.blacklistDisputeGame(gameProxy);
    assertTrue(anchorStateRegistry.isGameBlacklisted(gameProxy));

    // 5. Critical Bug: getAnchorRoot() still returns the blacklisted game's root!
    (Hash currentRoot, uint256 currentBlockNum) = anchorStateRegistry.getAnchorRoot();

    // These assertions should FAIL (demonstrating the bug)
    assertEq(currentRoot.raw(), gameProxy.rootClaim().raw());
    assertEq(currentBlockNum, gameProxy.l2SequenceNumber());

    // 6. Expected behavior would be to revert to starting root
    assertEq(currentRoot.raw(), startingRoot.raw()); // This will fail
    assertEq(currentBlockNum, startingBlockNum); // This will fail
}
```

Logs:

```
Ran 1 test suite in 4.85s (1.83s CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/dispute/AnchorStateRegistry.t.sol:AnchorStateRegistry_PoC
[FAIL: assertion failed: 1 != 0] test_PoC_BlacklistedAnchorStillReturned() (gas: 149318)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation: Modify `getAnchorRoot()` to validate the anchor:

```
function getAnchorRoot() public view returns (Hash, uint256) {
    if (
        address(anchorGame) == address(0) ||
        !isGameClaimValid(anchorGame) // Checks blacklist/finality
    ) {
        return (startingAnchorRoot.root, startingAnchorRoot.l2SequenceNumber);
    }
    return (Hash.wrap(anchorGame.rootClaim().raw()), anchorGame.l2SequenceNumber());
}
```

3.2.2 `ETHLockbox` lacks pause mechanism for locking ETH

Severity: Informational

Context: ETHLockbox.sol#L130-L137

Description: The `lockETH()` function in the ETHLockbox contract allows authorized OptimismPortal contracts to deposit ETH, but there is no mechanism to prevent deposits during emergency scenarios when the system might be paused. While the function verifies the sender is authorized, it doesn't check if the system is in a paused state.

```
function lockETH() external payable {
    // Check that the sender is authorized to trigger this function.
    IOptimismPortal sender = IOptimismPortal(payable(msg.sender));
    if (!authorizedPortals[sender]) revert ETHLockbox_Unauthorized();

    // Emit the event.
    emit ETHLocked(sender, msg.value);
}
```

During a system-wide emergency that requires pausing operations, it would be beneficial to prevent new ETH deposits to ensure consistency and avoid potential complications during recovery procedures.

Recommendation: Consider adding a pause check to the `lockETH()` function that aligns with the system's overall pause mechanism:

```
  function lockETH() external payable {
      // Check that the sender is authorized to trigger this function.
      IOptimismPortal sender = IOptimismPortal(payable(msg.sender));
      if (!authorizedPortals[sender]) revert ETHLockbox_Unauthorized();

+     // Check if the system is paused
+     if (paused()) revert ETHLockbox_SystemPaused();

      // Emit the event.
      emit ETHLocked(sender, msg.value);
  }
```

This would ensure that during emergency scenarios, no new ETH could be deposited into the lockbox, providing more consistent behavior across the system.

3.2.3 Migration of some existing chains can be bricked due to inability to change the `superchainConfig`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When migrating a chain to the superchain, the `upgrade()` and `migrateToSuperRoots()` function is primarily used to reference the new lockbox and ASR contracts. If we look at the above functions, we notice that it does not allow changing the `superchainConfig`, this is a problem during migration, as to migrate a chain to a superchain set, the `superchainConfig` must be the same for all portals in the superchain set. Let's look at two scenarios where this is a problem.

- Case 1: In the `upgrade()` function for non-upgraded portals. Suppose we have a Portal A and want to migrate it to a Superchain B, since Portal A is an existing portal, we call `upgrade()`, but since `upgrade()` does not allow us to modify the `superchainConfig`, what occurs is that some existing chains cannot actually use this function to join the superchain set.

  An example of this is Redstone which has a `superchainConfig` set to `0x4b5b41c240173191425F5928bc6bdd0d43933` which can be verified at mainnet address 0xc7bcb0e8839a28a1cfadd1cf716de9016cda51ae, this is different than the one on Optimism.

```
/// @notice Upgrades the OptimismPortal contract to have a reference to the AnchorStateRegistry.
/// @param _anchorStateRegistry AnchorStateRegistry contract
/// @param _ethLockbox ETHLockbox contract.
function upgrade(
    IAnchorStateRegistry _anchorStateRegistry,
    IETHLockbox _ethLockbox
)
    external
    reinitializer(initVersion())
{
    anchorStateRegistry = _anchorStateRegistry;
    ethLockbox = _ethLockbox;

    // Migrate the whole ETH balance to the ETHLockbox.
    _migrateLiquidity();
}
```

- Case 2: In the `migrateToSuperRoots` for merging two superchain clusters. If we have two superchain clusters Superchain A and Superchain B and would like to join them, it wouldn't be possible even with using the `migrateToSuperRoots` function. This is because it also does not allow modifying the `superchainConfig` function:

```
function migrateToSuperRoots(IETHLockbox _newLockbox, IAnchorStateRegistry _newAnchorStateRegistry)
↪   external {
    // Make sure the caller is the owner of the ProxyAdmin.
    if (msg.sender != proxyAdminOwner()) revert OptimismPortal_Unauthorized();

    // Chains can use this method to swap the proof method from Output Roots to Super Roots
    // without joining the interop set. In this case, the old and new lockboxes will be the
    // same. However, whether or not a chain is joining the interop set, all chains will need a
    // new AnchorStateRegistry when migrating to Super Roots. We therefore check that the new
    // AnchorStateRegistry is different than the old one to prevent this function from being
    // accidentally misused.
    if (anchorStateRegistry == _newAnchorStateRegistry) {
        revert OptimismPortal_MigratingToSameRegistry();
    }

    // Update the ETHLockbox.
    IETHLockbox oldLockbox = ethLockbox;
    ethLockbox = _newLockbox;

    // Update the AnchorStateRegistry.
    IAnchorStateRegistry oldAnchorStateRegistry = anchorStateRegistry;
    anchorStateRegistry = _newAnchorStateRegistry;

    // Set the proof method to Super Roots. We expect that migration will happen more than once
    // for some chains (switching to single-chain Super Roots and then later joining the
    // interop set) so we don't need to check that this is false.
    superRootsActive = true;

    // Emit a PortalMigrated event.
    emit PortalMigrated(oldLockbox, _newLockbox, oldAnchorStateRegistry, _newAnchorStateRegistry);
}
```

Recommendation: Allow changing the `superchainConfig` in `upgrade()` and `migrateToSuperRoots`.

3.2.4 Withdrawals can be proven and finalized based on an older dispute game

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `OptimismPortal2` contract, the proveWithdrawalTransaction function allows a user to submit a proof of a withdrawal transaction tied to a specific dispute game. However, this function does not verify whether the dispute game used in the proof aligns with the current anchor state maintained by the `AnchorStateRegistry`. The anchor state represents the most recent valid state of the L2 chain, as determined by the latest dispute game set via setAnchorState. Without this check, a withdrawal could be proven using an outdated dispute game that does not reflect the current system state. The finalizeWithdrawalTransaction function does validate the dispute game's validity through checkWithdrawal, which calls isGameClaimValid in `AnchorStateRegistry`. This ensures the game is registered, not blacklisted, finalized, and resolved as `DEFENDER_WINS`. However, there is no requirement that the game's L2 sequence

number (indicating its recency) matches or exceeds that of the current anchor state at the time the proof is submitted. This gap allows proofs based on stale or outdated games to proceed, potentially leading to inconsistencies.

Impact: This vulnerability could enable withdrawals to be proven and finalized based on an older dispute game, even if the anchor state has advanced to a newer game reflecting a more recent L2 state. In a system where the anchor state is meant to represent the authoritative L2 state for validation, this could lead to:

- Stale State Withdrawals: Withdrawals based on an outdated state might not reflect the latest L2 transactions or balances, potentially allowing invalid withdrawals.
- Double-Spending Risks: If an older game does not account for subsequent L2 state changes (e.g., spent funds), a user could withdraw funds that no longer exist in the current state.

Evidence:

- In proveWithdrawalTransaction: The function accepts a dispute game (`_disputeGameProxy`) and stores the proof data in the provenWithdrawals mapping without checking its L2 sequence number against the anchor state:

```
function _proveWithdrawalTransaction(
    Types.WithdrawalTransaction memory _tx,
    IDisputeGame _disputeGameProxy,
    uint256 _outputRootIndex,
    Types.SuperRootProof memory _superRootProof,
    Types.OutputRootProof memory _outputRootProof,
    bytes[] memory _withdrawalProof
)
    internal
{
// ... (other checks)
bytes32 withdrawalHash = hashWithdrawal(_tx);
provenWithdrawals[withdrawalHash][msg.sender] = ProvenWithdrawal({
    disputeGameProxy: _disputeGameProxy,
    timestamp: uint64(block.timestamp)
});
// @audit-ok No check against anchorStateRegistry.anchorGame()
```

- In 'AnchorStateRegistry: The setAnchorState function ensures the anchor is updated only if the new game's L2 sequence number is greater than the current anchor's:

```
function setAnchorState(IDisputeGame _game) public {
    // Convert game to FaultDisputeGame.
    // We can't use FaultDisputeGame in the interface because this function is called from the
    // FaultDisputeGame contract which can't import IFaultDisputeGame by convention. We should
    // likely introduce a new interface (e.g., StateDisputeGame) that can act as a more useful
    // version of IDisputeGame in the future.
    IFaultDisputeGame game = IFaultDisputeGame(address(_game));

    // Check if the candidate game claim is valid.
    if (!isGameClaimValid(game)) {
        revert AnchorStateRegistry_InvalidAnchorGame();
    }

    // Must be newer than the current anchor game.
    (, uint256 anchorL2BlockNumber) = getAnchorRoot();
    if (game.l2SequenceNumber() <= anchorL2BlockNumber) {
        revert AnchorStateRegistry_InvalidAnchorGame();
    }

    // Update the anchor game.
    anchorGame = game;
    emit AnchorUpdated(game);
}
```

  However, this check is not mirrored in `proveWithdrawalTransaction`.

- In checkWithdrawal: The validation focuses on the dispute game tied to the withdrawal, not its relation to the anchor:

```
function checkWithdrawal(bytes32 _withdrawalHash, address _proofSubmitter) public view {
    ProvenWithdrawal memory provenWithdrawal = provenWithdrawals[_withdrawalHash][_proofSubmitter];
    IDisputeGame disputeGameProxy = provenWithdrawal.disputeGameProxy;
    if (!anchorStateRegistry.isGameClaimValid(disputeGameProxy))
        revert OptimismPortal_InvalidRootClaim();
    // ... (other checks)
}
```

- The isGameClaimValid function checks registration, blacklist status, finalization, and resolution, but not recency relative to the anchor.

Proof of Concept: Add a test to verify that proving a withdrawal with an outdated game reverts:

```
function test_proveWithdrawalTransaction_outdatedGame_reverts() external {
    // Set anchor to a game with sequence number 100
    IDisputeGame newerGame = new MockDisputeGame(100);
    vm.prank(guardian);
    anchorStateRegistry.setAnchorState(newerGame);

    // Attempt to prove withdrawal with an older game (sequence number 99)
    IDisputeGame olderGame = new MockDisputeGame(99);
    WithdrawalTransaction memory tx = defaultTx;
    bytes32[] memory proof = defaultProof;
    bytes memory extraData = defaultExtraData;

    vm.expectRevert("OptimismPortal: dispute game must be at least as recent as anchor");
    optimismPortal2.proveWithdrawalTransaction(tx, olderGame, proof, extraData);
}
```

Recommendation: Add a check in `proveWithdrawalTransaction` to ensure the dispute game's L2 sequence number is at least as recent as the current anchor state.

3.2.5 Incorrect Timestamp Returned by `respectedGameTypeUpdatedAt()`

Severity: Informational

Context: OptimismPortal2.sol#L330-L335

Summary: The function `respectedGameTypeUpdatedAt()` incorrectly returns the `retirementTimestamp()` from `AnchorStateRegistry`, despite being described as a getter for the timestamp when the respected game type was updated. Additionally, `respectedGameTypeUpdatedAt()` is not defined in `IAnchorStateRegistry.sol`, raising concerns about its correctness and intended purpose.

Finding Description: The function `respectedGameTypeUpdatedAt()` is expected to return the timestamp when the respected game type was updated. However, it instead returns `retirementTimestamp()` from `AnchorStateRegistry`, which likely refers to a different event. This misalignment can lead to incorrect assumptions by dependent contracts or off-chain systems relying on this function for game state tracking.

Additionally, `respectedGameTypeUpdatedAt()` is not defined in `IAnchorStateRegistry.sol`, which suggests it may be an unintended or improperly integrated function. This could indicate a specification mismatch, where external systems expect a distinct timestamp but instead receive an unrelated value.

While this issue does not immediately introduce a direct security vulnerability, it breaks data integrity guarantees by providing misleading information. If a system relies on this function for critical decision-making—such as verifying game validity, enforcing time-based constraints, or managing updates—it could be manipulated or disrupted. A malicious actor could exploit this flaw by assuming an outdated or incorrect timestamp, potentially bypassing expected security conditions.

Impact Explanation: The issue affects data integrity and can mislead dependent systems into making incorrect assumptions about game state updates. While it does not directly enable an exploit like theft or unauthorized access, it could disrupt protocol logic, lead to improper validations, or cause unintended behaviors in time-sensitive operations. If external contracts or off-chain systems rely on this incorrect timestamp for critical functions, it may introduce security risks or operational failures.

Likelihood Explanation: The issue is likely to occur if any system relies on `respectedGameTypeUpdatedAt()` for accurate timestamps, as it incorrectly returns `retirementTimestamp()` instead of a dedicated game type update timestamp. Since this function is publicly accessible and used in contract logic, its in-

correct behavior can propagate errors throughout the system, making the likelihood moderate to high, depending on its usage.

Proof of Concept: Save the file as `RespectedGameTypeUpdatedAt.t.sol` and run:

- `forge test --match-test testBugExists_WrongTimestampReturned -vvv`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";

contract MockAnchorStateRegistry {
    function retirementTimestamp() external pure returns (uint64) {
        return 1234567890; // Mocked return value
    }
}

contract MockOptimismPortal {
    MockAnchorStateRegistry public registry;

    constructor(address _registry) {
        registry = MockAnchorStateRegistry(_registry);
    }

    function respectedGameTypeUpdatedAt() external view returns (uint64) {
        return registry.retirementTimestamp();
    }
}

contract RespectedGameTypeUpdatedAtTest is Test {
    MockAnchorStateRegistry mockRegistry;
    MockOptimismPortal mockPortal;

    function setUp() public {
        mockRegistry = new MockAnchorStateRegistry();
        mockPortal = new MockOptimismPortal(address(mockRegistry));
    }

    function testBugExists_WrongTimestampReturned() public {
        uint64 retirementTime = mockRegistry.retirementTimestamp();
        uint64 respectedGameTime = mockPortal.respectedGameTypeUpdatedAt();

        assertEq(respectedGameTime, retirementTime, "Bug Exists: Function should not return
        ↪  retirementTimestamp!");
    }
}
```

Recommendation: The function `respectedGameTypeUpdatedAt()` should return the correct timestamp specifically related to when the respected game type was updated, rather than mistakenly using `retirementTimestamp()`.

- Introduce a new state variable to track the correct timestamp.
- Ensure `respectedGameTypeUpdatedAt()` retrieves and returns this value.
- Update the logic to set this variable when the game type is updated.

3.2.6 `ETHLockbox.migrateLiquidity()` allows ETH outflow when paused

Severity: Informational

Context: ETHLockbox.sol#L189-L202

Summary: `migrateLiquidity()` in the `ETHLockbox` contract permits ETH to flow out even when the contract is paused, violating the invariant that no ETH should exit during a paused state.

Finding Description: The specs specify that

> No Ether MUST flow out of the contract when in a paused state

There is no check in `migrateLiquidity()` to stop this.

Impact Explanation: Medium: Undermines the pause mechanism's purpose.

Likelihood Explanation: Medium:

- The pause state is a deliberate feature triggered by governance or emergency responses.
- It's not an everyday occurrence (requires pause + specific call), but the absence of a check makes it exploitable under realistic conditions.

Proof of Concept: Steps:

1. Set up a destination lockbox proxy and initialize it.
2. Fund the origin lockbox with ETH.
3. Mock SuperchainConfig.paused() to true.
4. Call migrateLiquidity as proxyAdminOwner.
5. Verify ETH moves out despite the pause.

Add the following test to `test/L1/ETHLockbox.t.sol`:

```
/// @notice Tests that `migrateLiquidity` incorrectly allows ETH to flow out when the contract is paused.
///         This demonstrates a bug where the function does not respect the "No Ether MUST flow out
///         when paused" invariant.
function testFuzz_migrateLiquidity_whenPaused_allowsOutflow_reverts(
    uint256 _originLockboxBalance,
    uint256 _destinationLockboxBalance
)
    public
{
    // Skip test in fork environment since fuzzed addresses may not exist
    if (isForkTest()) vm.skip(true);

    // Bound balances to avoid overflow
    _originLockboxBalance = bound(_originLockboxBalance, 1, type(uint256).max - address(ethLockbox).balance);
    _destinationLockboxBalance = bound(_destinationLockboxBalance, 0, type(uint256).max -
    ↪  _originLockboxBalance);

    // Deploy a new Proxy for the destination lockbox
    address destinationLockbox = address(new Proxy(address(proxyAdmin)));

    // Get the ETHLockbox implementation of the origin `ethLockbox` proxy
    vm.prank(address(proxyAdmin));
    address implementation = Proxy(payable(address(ethLockbox))).implementation();

    // Upgrade the destination lockbox proxy to the `ETHLockbox` implementation
    vm.prank(address(proxyAdmin));
    Proxy(payable(destinationLockbox)).upgradeTo(implementation);

    // Authorize the origin lockbox on the destination lockbox
    vm.prank(proxyAdminOwner);
    IETHLockbox(destinationLockbox).authorizeLockbox(ethLockbox);

    // Mock the calls for checks on the destination lockbox so it can receive the migration
    vm.mockCall(
        address(destinationLockbox),
        abi.encodeCall(IProxyAdminOwnedBase.proxyAdminOwner, ()),
        abi.encode(proxyAdminOwner)
    );
    vm.mockCall(
        address(destinationLockbox), abi.encodeCall(IETHLockbox.authorizedLockboxes, (ethLockbox)),
        ↪  abi.encode(true)
    );

    // Deal balances to both lockboxes
    deal(address(ethLockbox), _originLockboxBalance);
    deal(address(destinationLockbox), _destinationLockboxBalance);

    // Mock the SuperchainConfig to return true for paused status
    vm.mockCall(address(superchainConfig), abi.encodeCall(ISuperchainConfig.paused, ()), abi.encode(true));

    // Verify the contract is paused
    assertTrue(ethLockbox.paused(), "Contract should be paused");

    // Get balances before the migration
    uint256 originLockboxBalanceBefore = address(ethLockbox).balance;
    uint256 destLockboxBalanceBefore = address(destinationLockbox).balance;
```

```
    // Expect the `LiquidityMigrated` event (this happens in the buggy implementation)
    vm.expectEmit(address(ethLockbox));
    emit LiquidityMigrated(IETHLockbox(destinationLockbox), originLockboxBalanceBefore);

    // Call `migrateLiquidity` while paused
    vm.prank(proxyAdminOwner);
    ethLockbox.migrateLiquidity(IETHLockbox(destinationLockbox));

    // Assert that ETH flowed out despite being paused (this is the bug)
    assertEq(address(ethLockbox).balance, 0, "Origin lockbox should not have zero balance if paused");
    assertEq(
        address(destinationLockbox).balance,
        destLockboxBalanceBefore + originLockboxBalanceBefore,
        "Destination lockbox should not have received ETH when paused"
    );

    // Note: In a correct implementation, this should revert with ETHLockbox_Paused
    // and the balances should remain unchanged.
}
```

Recommendation:

```
  function migrateLiquidity(IETHLockbox _lockbox) external {
+     if (paused()) revert ETHLockbox_Paused();

      // Check that the sender is the proxy admin owner.
      if (msg.sender != proxyAdminOwner()) revert ETHLockbox_Unauthorized();

      // Check that the lockbox has the same proxy admin owner.
      if (!_sameProxyAdminOwner(address(_lockbox))) revert ETHLockbox_DifferentProxyAdminOwner();

      // Receive the liquidity.
      uint256 balance = address(this).balance;
      IETHLockbox(_lockbox).receiveLiquidity{ value: balance }();

      // Emit the event.
      emit LiquidityMigrated(_lockbox, balance);
  }
```

3.2.7 Migrating already upgraded `OptimismPortal2` to Super Roots will break pending withdrawals

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: In the current interop design of OP-stack L1 contracts, the `anchorStateRegistry` became the source of truth for fault dispute games. To support this architecture, `upgrade()` function was implemented in `OptimismPortal2`, which allows modifying `anchorStateRegistry` and `ethLockbox` contracts.

Moreover, the `migrateToSuperRoots()` function was implemented to allow the admin to set `superRootsActive` to `true` and allow interoperability in the withdrawals proving process. It is possible that a contract already upgraded to support `AnchorStateRegistry` as a source of truth, and is migrated to use super roots later on:

```
// Chains can use this method to swap the proof method from Output Roots to Super Roots
// without joining the interop set. In this case, the old and new lockboxes will be the
// same. However, whether or not a chain is joining the interop set, all chains will need a
// new AnchorStateRegistry when migrating to Super Roots. We therefore check that the new
// AnchorStateRegistry is different than the old one to prevent this function from being
// accidentally misused.
if (anchorStateRegistry == _newAnchorStateRegistry) {
    revert OptimismPortal_MigratingToSameRegistry();
}
```

As explained in the comment above, it requires a new `AnchorStateRegistry`.

Finding Description: It's impossible to perform the migration safely (without breaking invariants) for several reasons:

1. It would break the validity check for dispute games that use the old `anchorStateRegistry`. For these games, `isGameClaimValid()` will always return `false` because `isGameRetired()` will return `true`, preventing valid withdrawals from finalizing:

```
function isGameRetired(IDisputeGame _game) public view returns (bool) {
    // Must be created after the respectedGameTypeUpdatedAt timestamp. Note that this means all
    // games created in the same block as the respectedGameTypeUpdatedAt timestamp are
    // considered retired.
    return _game.createdAt().raw() <= retirementTimestamp;
}
```

`retirementTimestamp` is set to dispute game creation timestamp before migration. Not to mention a potential supported game type change, blacklist etc...

2. To resolve 1), the admin is forced to wait until the existing dispute games are finalized. This is again impossible because pausing would block the old games from resolving, in `isGameProper()`:

```
if (paused()) {
    return false;
}
```

while re-approving the withdrawal using a new game type would revert any game's progress - which should not happen unless something like the supported game type in `AnchorStateRegistry` has changed.

Impact Explanation: Interop portal (one that was upgraded/initialized to support interoperability in the past) cannot be safely migrated to Super Roots if it has any active dispute games attached to pending proved withdrawals. Doing so would revert the game's progress, while waiting for it to finish is impossible because of the `paused()` check -- new ones will be created.

Likelihood Explanation: Occurs whenever the interop portal is migrated to use Super Roots while having an initialized `anchorStateRegistry` with pending withdrawals.

Proof of Concept: This issue's impact is something that is intended in other scenarios - old dispute games are invalid and their progress is lost - just in a specific migration scenario. For this reason, to avoid complex setup, a simplified `DummyAnchorStateRegistry` was implemented where it's possible to manually set `gameCreatedAt` timestamp of a singular game to simulate what is known to happen when new anchor is used after migration - `isGameRetired()` returns true. Treat this PoC as a visualization of the finding description rather than a proof valid games are retired when new anchor is used in the portal. Add these imports to `OptimismPortal2.t.sol`:

```
import { IDisputeGameFactory } from "interfaces/dispute/IDisputeGameFactory.sol";
import {ISuperchainConfig} from "interfaces/L1/ISuperchainConfig.sol";
import { Proxy } from "src/universal/Proxy.sol";
import { ProxyAdmin } from "src/universal/ProxyAdmin.sol";
import {OptimismPortal2} from "src/L1/OptimismPortal2.sol";
import { GameType, Proposal, Claim, GameStatus, Hash } from "src/dispute/lib/Types.sol";
```

Place the dummy anchor registry in that file:

```
contract DummyAnchorStateRegistry is IAnchorStateRegistry {
    uint64 private _retirementTimestamp;
    uint64 private _dummyGameCreationTimestamp;
    ISuperchainConfig private _superchainConfig;
    IDisputeGameFactory private _disputeGameFactory;
    string private _version;
    IFaultDisputeGame private _anchorGame;

    uint256 public gameCreatedAt;
function __constructor__(
        uint256 /*disputeGameFinalityDelaySeconds*/
    ) external {}

    function initialize(
        ISuperchainConfig superchainConfig_,
        IDisputeGameFactory disputeGameFactory_,
        Proposal memory _startingAnchorRoot,
        GameType /* _startingRespectedGameType */
    )
        external
        override
    {
        require(bytes(_version).length == 0, "Already initialized");

        _superchainConfig = superchainConfig_;
```

```
        _disputeGameFactory = disputeGameFactory_;
        _retirementTimestamp = uint64(block.timestamp);
        _version = "DummyAnchorStateRegistry v1";

        emit Initialized(1);
    }
    function anchorGame() external view override returns (IFaultDisputeGame) {
        return _anchorGame;
    }
    function anchors(GameType /* _gameType */) external view override returns (Hash, uint256) {
        return (Hash.wrap(0xDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF), 0);
    }
    function disputeGameBlacklist(IDisputeGame /* _game */) external pure override returns (bool) {
        return false;
    }
    function getAnchorRoot() external view override returns (Hash, uint256) {
        return (Hash.wrap(0xDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF), 0);
    }
    function disputeGameFinalityDelaySeconds() external pure override returns (uint256) {
        return 302400;
    }
    function disputeGameFactory() external view override returns (IDisputeGameFactory) {
        return _disputeGameFactory;
    }
    function isGameBlacklisted(IDisputeGame /* _game */) external pure override returns (bool) {
        return false;
    }
    function isGameProper(IDisputeGame /* _game */) external pure override returns (bool) {
        return true;
    }
    function isGameRegistered(IDisputeGame /* _game */) external pure override returns (bool) {
        return true;
    }
    function isGameResolved(IDisputeGame /* _game */) external pure override returns (bool) {
        return false;
    }
    function isGameRespected(IDisputeGame /* _game */) external pure override returns (bool) {
        return true;
    }
    function isGameRetired(IDisputeGame /* _game */) external view override returns (bool) {
        return gameCreatedAt <= _retirementTimestamp;
    }
    function isGameFinalized(IDisputeGame /* _game */) external pure override returns (bool) {
        return false;
    }
    function isGameClaimValid(IDisputeGame /* _game */) external pure override returns (bool) {
        return false;
    }
    function paused() external pure override returns (bool) {
        return false;
    }
    function respectedGameType() external pure override returns (GameType) {
        return GameType.wrap(0);
    }
    function retirementTimestamp() external view override returns (uint64) {
        return _retirementTimestamp;
    }
    function superchainConfig() external view override returns (ISuperchainConfig) {
        return _superchainConfig;
    }
    function version() external view override returns (string memory) {
        return _version;
    }
    function blacklistDisputeGame(IDisputeGame /* _disputeGame */) external override {
        // No-op.
        emit DisputeGameBlacklisted(IDisputeGame(address(0)));
    }
    function setAnchorState(IDisputeGame /* _game */) external override {
        // No-op.
        emit AnchorUpdated(IFaultDisputeGame(address(0)));
    }
    function setRespectedGameType(GameType /* _gameType */) external override {
        // No-op.
        emit RespectedGameTypeSet(GameType.wrap(0));
    }
    function updateRetirementTimestamp() external override {
```

```
        _retirementTimestamp = uint64(block.timestamp);
        emit RetirementTimestampSet(block.timestamp);
    }
    function setGameCreatedAt(uint256 timestamp) external {
        gameCreatedAt = timestamp;
    }
}
```

And finally, the simplified test case in `OptimismPortal2_Test` contract in that file:

```
    function test_SuperRootsMigrationIssue() public {
    ProxyAdmin proxyAdmin = ProxyAdmin(artifacts.mustGetAddress("ProxyAdmin"));

    IETHLockbox _oldLockbox = IETHLockbox(optimismPortal2.ethLockbox());

    // No need to call upgrade() because default portal was already initialized to support interop
    console.log(address(optimismPortal2.anchorStateRegistry()));
    console.log(address(optimismPortal2.ethLockbox()));

    vm.warp(block.timestamp + 100 days);

    // New anchor state registry for Super Roots
    DummyAnchorStateRegistry dummyRegistry = new DummyAnchorStateRegistry();
    IAnchorStateRegistry _newAnchorStateRegistry = IAnchorStateRegistry(address(dummyRegistry));

    // Simulate dispute game created before the migration
    dummyRegistry.setGameCreatedAt(block.timestamp);

    vm.warp(block.timestamp + 1 hours);
    _newAnchorStateRegistry.initialize(
        superchainConfig,
        disputeGameFactory,
        Proposal({
            root: Hash.wrap(0xDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF),
            l2SequenceNumber: 0
        }),
        GameType.wrap(0)
    );

    // Migrate to Super Roots
    vm.startPrank(address(_oldLockbox.proxyAdminOwner()));


    optimismPortal2.migrateToSuperRoots(_oldLockbox, IAnchorStateRegistry(_newAnchorStateRegistry));

    assertEq(_newAnchorStateRegistry.isGameRetired(IDisputeGame(address(0))), true);

    vm.stopPrank();
}
```

Recommendation: Allowing safe migration of an interop-supporting portal with existing dispute games may require significant modifications to how the anchor-portal-game relations are handled.

3.2.8 Externally donated funds through `donateETH` will not be instantly usable

Severity: Informational

Context: OptimismPortal2.sol#L356-L359

Description: The `donateETH` function of the optimism portal accepts ETH value without triggering a deposit on L2. This is implemented for the ETHLockbox to send funds to the Portal for withdrawal finalization. It can also be used for external donations. However, funds from external donations will not be usable by the ETH lockbox before the proxy admin calls `migrateLiquidity` on the portal.

Scenario: Alice makes a withdrawal from L2 to L1 to call a specific smart contract. This smart contract verifies that the call comes from the Optimism Portal. However, there is not enough liquidity in the ETHLockbox to support this withdrawal. Alice would like to pay (through a donation) to ensure that her withdrawal is executed on time. But due to the fact that donated funds are not transfered to the ETHLockbox, Alice can't finalize her withdrawal.

Code snippet: All donations done through `donateETH` are kept in the `OptimismPortal2` contract, and not sent to the `ETHLockbox`:

```
/// @notice Accepts ETH value without triggering a deposit to L2.
function donateETH() external payable {
    // Intentionally empty.
}
```

Recommendation: Consider calling `_migrateLiquidity` when a donation is done. Note that this requires setting a new function for the lockbox to send to the portal.

```
diff --git a/packages/contracts-bedrock/src/L1/OptimismPortal2.sol
↪  b/packages/contracts-bedrock/src/L1/OptimismPortal2.sol
index a6478226f..1ff8c68ec 100644
--- a/packages/contracts-bedrock/src/L1/OptimismPortal2.sol
+++ b/packages/contracts-bedrock/src/L1/OptimismPortal2.sol
@@ -157,6 +157,11 @@ contract OptimismPortal2 is Initializable, ResourceMetering, ReinitializableBase
     /// @param ethBalance Amount of ETH migrated.
     event ETHMigrated(address indexed lockbox, uint256 ethBalance);

+    /// @notice Emitted when a donation is done and transferred to the ETHLockbox.
+    /// @param lockbox The address of the ETHLockbox contract.
+    /// @param ethBalance Amount of ETH donated.
+    event ETHDonated(address indexed lockbox, uint256 ethBalance);
+
     /// @notice Emitted when the ETHLockbox contract is updated.
     /// @param oldLockbox The address of the old ETHLockbox contract.
     /// @param newLockbox The address of the new ETHLockbox contract.
@@ -355,7 +360,9 @@ contract OptimismPortal2 is Initializable, ResourceMetering, ReinitializableBase

     /// @notice Accepts ETH value without triggering a deposit to L2.
     function donateETH() external payable {
-        // Intentionally empty.
+        ethLockbox.lockETH{ value: msg.value }();
+
+        emit ETHDonated(address(ethLockbox), msg.value);
     }

     /// @notice Allows the owner of the ProxyAdmin to migrate the OptimismPortal to use a new
```

3.2.9 Incorrect game type check allows to use games from other registry

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: An incorrect check on the games' type allow to use games generated for any registry to any registry, allowing to set invalid anchor and break the withdrawal process. **Finding Description:** Every game's type is associated with an `AnchorStateRegistry`.

`isGameRespected` is expected to check that the provided game used the right game's type (see AnchorStateRegistry.sol#L177-L182).

However this check is incorrect. It checks that the game's type was the one of its associated registry (see the `FaultDisputeGame`'s implementation as an example: FaultDisputeGame.sol#L357-L360), but it does not validate that the game is associated to the registry being used. This means than a game that is valid for a registry `A` can be used in a registry `B`.

One direct impact is that `setAnchorState` can be used to set the root's state of any chain the state of any other chain using a different registry (AnchorStateRegistry.sol#L293). Another impact is that the portal associated with the registry can be manipulated with the wrong root state, and wrong games.

Note for OP labs: hopefully this is not exploitable on the previous version of the code which is deployed, given than the previous version was correctly checking the game's type in OptimismPortal2.sol#L503.

Impact explanation: High. This allow to set invalid root state. This issue breaks the invariant `iASR-001: Games are represented as Proper Games accurately` (see the spec).

Likelihood Explanation: High/Medium. The likelihood was subject to debate. To exploit the issue the following needs to be true:

- Two registries need to be deployed using the same factory.

- The games need to be created after the registry deployment (or after the latest update to retirementTimestamp).

Based on the code, I believe this scenario is likely to happen. In particular if there is a period where a chain exists using the old FaultDisputeGame, while another one uses the SuperFaultDisputeGame. However the client mentioned that this is not their first intent. Given the discussion, I am ok to use medium likelihood instead of high.

Proof of Concept:

- Two registries are deployed, one for FaultDisputeGame and one for SuperFaultDisputeGame, for two separate chains (C0 and C1).
- A valid game advances C0's state.
- Eve reuse the game's contract and use it to set the new root for C1.
- As a result, C1 now operate with an incorrect root.

```
import {Test} from "forge-std/Test.sol";

import {IAnchorStateRegistry} from "interfaces/dispute/IAnchorStateRegistry.sol";
import {AnchorStateRegistry} from "src/dispute/AnchorStateRegistry.sol";
import { Proxy } from "src/universal/Proxy.sol";
import {DisputeGameFactory} from "src/dispute/DisputeGameFactory.sol";
import {SuperchainConfig} from "src/L1/SuperchainConfig.sol";
import {IDisputeGameFactory} from "interfaces/dispute/IDisputeGameFactory.sol";
import {GameType, Proposal, Hash} from "src/dispute/lib/Types.sol";
import {ISuperchainConfig} from "interfaces/L1/ISuperchainConfig.sol";
import {IDisputeGame} from "interfaces/dispute/IDisputeGame.sol";
import {Timestamp, GameStatus, Claim} from "src/dispute/lib/Types.sol";

// Mock game that is always resolved in favor of the defender.
contract MockGame is IDisputeGame {

    GameType immutable  t;
    Timestamp created;

    constructor(GameType _t)  {
        t = _t;
    }

    function initialize() external payable {
        created = Timestamp.wrap(uint64(block.timestamp));
    }

    function createdAt() external view returns (Timestamp){
        return created;
    }
    function resolvedAt() external view returns (Timestamp){
        return created;
    }
    function status() external view returns (GameStatus){
        return GameStatus.DEFENDER_WINS;
    }
    function gameType() external view returns (GameType gameType_){
        return t;
    }
    function gameCreator() external pure returns (address creator_){
        return address(0);
    }
    function rootClaim() external pure returns (Claim rootClaim_){
        return Claim.wrap(0);
    }
    function l1Head() external pure returns (Hash l1Head_){
        return Hash.wrap(0);
    }
    function l2SequenceNumber() external pure returns (uint256 l2SequenceNumber_){
        return 0;
    }
    function extraData() external pure returns (bytes memory extraData_){
        return abi.encode(0);
    }
    function resolve() external returns (GameStatus status_){
        return GameStatus.DEFENDER_WINS;
    }
```

```
    function gameData() external view returns (GameType gameType_, Claim rootClaim_, bytes memory extraData_){
        return (t, Claim.wrap(0), abi.encode(0));
    }
    function wasRespectedGameTypeWhenCreated() external view returns (bool){
        return true;
    }
}

contract Issue77Test is Test {
    AnchorStateRegistry anchorStateRegistryImpl;
    DisputeGameFactory dgfImpl = new DisputeGameFactory();
    SuperchainConfig supConfigImpl = new SuperchainConfig();

    IDisputeGameFactory factory;
    AnchorStateRegistry registry0;
    AnchorStateRegistry registry1;

    MockGame mockGame0 ;
    MockGame mockGame1 ;

    function setUp() public {
        // setup the system with two registries using the same factory

        mockGame0 = new MockGame(GameType.wrap(100));
        mockGame1 = new MockGame(GameType.wrap(200));

        anchorStateRegistryImpl = new AnchorStateRegistry(0);
        dgfImpl = new DisputeGameFactory();
        supConfigImpl = new SuperchainConfig();

        Proxy supConfigProxy = new Proxy(address(1));
        vm.prank(address(1));
        supConfigProxy.upgradeToAndCall(
            address(supConfigImpl), abi.encodeCall(supConfigImpl.initialize, (address(this), false))
        );

        Proxy factoryProxy = new Proxy(address(1));
        vm.prank(address(1));
        factoryProxy.upgradeToAndCall(address(dgfImpl), abi.encodeCall(dgfImpl.initialize, (address(this))));
        factory = IDisputeGameFactory(address(factoryProxy));

        factory.setImplementation(GameType.wrap(100), IDisputeGame(address(mockGame0)));
        factory.setImplementation(GameType.wrap(200), IDisputeGame(address(mockGame1)));

        Proxy registry0Proxy = new Proxy(address(1));
        vm.prank(address(1));
        registry0Proxy.upgradeToAndCall(
            address(anchorStateRegistryImpl),
            abi.encodeCall(
                anchorStateRegistryImpl.initialize,
                (
                    ISuperchainConfig(address(supConfigProxy)),
                    factory,
                    Proposal({ root: Hash.wrap(0), l2SequenceNumber: 0 }),
                    GameType.wrap(100)
                )
            )
        );
        registry0 = AnchorStateRegistry(address(registry0Proxy));

        Proxy registry1Proxy = new Proxy(address(1));
        vm.prank(address(1));
        registry1Proxy.upgradeToAndCall(
            address(anchorStateRegistryImpl),
            abi.encodeCall(
                anchorStateRegistryImpl.initialize,
                (
                    ISuperchainConfig(address(supConfigProxy)),
                    factory,
                    Proposal({ root: Hash.wrap(0), l2SequenceNumber: 0 }),
                    GameType.wrap(200)
                )
            )
        );
        registry1 = AnchorStateRegistry(address(registry1Proxy));
    }
```

```
    function test_issue77() public {

        vm.warp(1000);
        IDisputeGame game0 = factory.create(GameType.wrap(100), Claim.wrap(0), abi.encode(0));
        IDisputeGame game1 = factory.create(GameType.wrap(200), Claim.wrap(0), abi.encode(0));

        vm.warp(10**18);

        // Checks that the game works with their respective registry.
        assertTrue(registry0.isGameClaimValid(game0));
        assertTrue(registry1.isGameClaimValid(game1));

        // Checks that the game does not work with the other registry.
        // This currently reverts due to the bug
        assertTrue(!registry1.isGameClaimValid(game0));
    }
}
```

Recommendation: One way to fix this issue, one is to check than the registry associated to a game is the address of the registry being used:

- Add `function anchorStateRegistry() external view returns (IAnchorStateRegistry registry_);` to the interface `IDisputeGame` (note: it is already implemented on all the interface of the different game's type).
- Check that `require(_game.anchorStateRegistry() == this)` in the `AnchorStateRegistry`.

Phaze: Fixed in commit b671b67. When checking `isGameRegistered()`, the `AnchorStateRegistry` now reverts if the game's `AnchorStateRegistry` contract does not equal the calling contract. Additionally verified whether all FDG (`FaultDisputeGame`, `PermissionedDisputeGame`, `SuperFaultDisputeGame`, `SuperPermissionedDisputeGame`) contracts expose `anchorStateRegistry()`. Perhaps these contracts could inherit from `IFaultDisputeGame` to ensure this (not sure if there's any limiting factor).

Op Labs: Acknowledged. We are planning to improve this interface in a future release.

3.2.10 `AnchorStateRegistry` could have a different `superchainConfig` address than the portal it is attached to

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When `anchorStateRegistry` address is set in the `OptimismPortal2` contract, contrary to the `ETHLockbox`, there are no explicit checks whether the `superchainConfig` addresses in these two contracts match.

If only one of these configs is paused, this could lead to inconsistencies in how the Pause state is handled. Including breaking the iASR-001 invariant. Issue is Low severity due to the inherent very low likelihood of such admin mistake.

Recommendation: Consider verifying that the new `anchorStateRegistry` address set in any of the functions in the portal (`initialize()`, `upgrade()` and `migrateToSuperRoots()`) check whether the `superchainConfig` matches.

3.2.11 Portal `upgrade` function is not access controlled, leading to 800M$ loss in case of mistake

Severity: Informational

Context: OptimismPortal2.sol#L258-L296

Description: The `upgrade` function does not have any access control mechanism. This is to be expected as it is supposed to be called during the proxy upgrade. However, if it is not done during the proxy upgrade for any reason, the `upgrade` function will be callable by anyone. This will allow migrating funds to an untrusted `ethLockbox` contract.

Impact: The `OptimismPortal2` proxy on mainnet holds more than $800M that will be transferred during the call to `upgrade`.

Likelihood: This would require an incorrect upgrade, for example an upgrade through `upgradeTo` instead of `upgradeToAndCall` will lead to losing all funds.

```
function upgradeTo(address _implementation) public virtual proxyCallIfNotAdmin {
    _setImplementation(_implementation);
}

function upgradeToAndCall(address _implementation, bytes calldata _data)
    public
    payable
    virtual
    proxyCallIfNotAdmin
    returns (bytes memory)
{
    _setImplementation(_implementation);
    (bool success, bytes memory returndata) = _implementation.delegatecall(_data);
    require(success, "Proxy: delegatecall to new implementation contract failed");
    return returndata;
}
```

Recommendation: For defense in depth, I would recommend adding other protection mechanisms on the `upgrade` function. Here are some ideas for in depth protections:

- Load the `bytes32(uint256(keccak256('eip1967.proxy.admin')) - 1)` storage slot and ensure that the returned address (the owner of the proxy) is the current caller.
- Add the expected lockbox to the SystemConfig, and then check that the `_ethLockbox` parameter is the correct one.

Phaze: Fixed in PR 15615. All `initialize()` and `upgrade()` functions are now secured with an `onlyProxyAdmin` modifier. These contracts inherit from `ReinitializableBase` and `ProxyAdminOwnedBase`.

3.2.12 There is no check for the validity of the `disputeGameType` input when deploying

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: We use the `OPContractsManage` contract to handle the upgrading and new deployment processes of OP Stack chains. `OPContractsManagerDeployer::deploy()` will get fired to deploy a new Chain contract. When deploying, we deploy `Permissioned` Game type, but when we deploy it we use `_input.disputeGameType`.

- OPContractsManage::deploy()#L870:

```
output.permissionedDisputeGame = IPermissionedDisputeGame(
    Blueprint.deployFrom(
        blueprint.permissionedDisputeGame1,
        blueprint.permissionedDisputeGame2,
        computeSalt(_input.l2ChainId, _input.saltMixer, "PermissionedDisputeGame"),
        encodePermissionedFDGConstructor(
            IFaultDisputeGame.GameConstructorParams({
                gameType: _input.disputeGameType, // <<<
                // ...
            }),
            _input.roles.proposer,
            _input.roles.challenger
        )
    )
);
```

`_input.disputeGameType` is an input provided by the Deployer, there is no anything that restricts this value to be of exact the type of `Permissioned_CANNON` game. In addition to this, when we initialize the anchor state registry we force the initialize to use `Permissioned_CANNON` and we do not use `_input.disputeGameType`.

- OPContractsManager.sol::encodeAnchorStateRegistryInitializer()#L1160:

```
function encodeAnchorStateRegistryInitializer( ... ) ...  {
    Proposal memory startingAnchorRoot = abi.decode(_input.startingAnchorRoot, (Proposal));
    return abi.encodeCall(
        IAnchorStateRegistry.initialize,
        (_superchainConfig, _output.disputeGameFactoryProxy, startingAnchorRoot,
        ↪  GameTypes.PERMISSIONED_CANNON) // <<<
    );
}
```

So The Design allows the owner to deploy his OP contracts by providing the game type as input, but if he provided a type other than `Permissioned_CANNON`, the deployment of the game implementation will be incorrect.

Scenario:

- Deployer provided `CANNON` game type in Input struct (permissionless_cannon).
- The deployed game implementation will be `PERMISSIONED_CANNON`, but the game implementation will have a type of `permissionless_cannon`.
- Anchor State Registry will have Respected game type as `PERMISSIONED_CANNON`.
- The Fault dispute process will not work as Portal will have a respected Game that does no even exist.

Recommendation: in `assertValidInputs()` function, which checks the inputs' validity. Ensure that `_input.disputeGameType` is `Permissioned_CANNON`.

Phaze: Fixed in commit b671b67. `OPContractsManager` now sets the permissioned FDG's `gameType` to `GameTypes.PERMISSIONED_CANNON`. The parameter can/should now be removed from the DeployInput struct in order to avoid confusion, since it's not being used anymore.

Op Labs: Agreed, will mark this as a backlog item.

3.2.13 Incomplete check for the validity of Contract addresses

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When checking contract addresses' validity using `OPContractsManager::assertValidContractAddress()` we just check the length of the code to be > 0.

- OPContractsManager.so::assertValidContractAddress()l#L164:

```
function assertValidContractAddress(address _who) public view {
    if (_who.code.length == 0) revert OPContractsManager.AddressHasNoCode(_who);
}
```

But this check is not complete, where after The Pectra upgrade and EIP 7702, even EOA will have code length. So the validity of contract addresses is not just they have code length > 0. But they should also not be an EOA with a code like EIP7702.

Recommendation: Check that the code is greater than zero as well as is not equals 23 in length (23 is the code length of EOA delegating to a contract according to EIP7702).

3.2.14 There is no way deauthorize Portals or Lockboxes

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `ETHLockbox`, we can authorize portals to allow portal(s) in that supoerchain to interact with the ETH lockbox. The authorization process is the same when authorizing ETH lockboxes. The problem is that there is no way to remove authorized portals or ETHlockboxes in need. where once we authorize a given portal or ETH lockbox we can't unauthorized it.

- This will affect the process of authorizing wrong address input by mistake.
- If a given Portal was in the SuperChain and is to be removed (either get hacked or something else).

- If a given ETHlockbox authorized another one to receive a liquidity from it, we will not be able to remove it.

Recommendation: Add methods that allow the ProxyAdminOwner to unauthorized Portals and Etlockboxes in Lockbox contract.

3.2.15 Inconsistency of the `AnchorStateRegistry` Proposal value can occur when upgrading leading to incorrect StateRoot initializing

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When upgrading our contracts, we deploy a new `AnchorStateRegistry`. In the old implementation, each GameType has its proposal (StateRoot). which represents the root and the l2BlockNumber.

- AnchorStateRegistry__old:

```
/// @inheritdoc IAnchorStateRegistry
mapping(GameType => OutputRoot) public anchors;

function tryUpdateAnchorState() external {
    // ...

    // Actually update the anchor state.
    anchors[gameType] = OutputRoot({ l2BlockNumber: game.l2BlockNumber(), root:
    ↪  Hash.wrap(game.rootClaim().raw()) }); // <<<
}
```

  In the newly deployed version, this feature does not exist. where there will just be one `AnchorStateRegistry` contract that will be used, and whatever the game type is, we update our Root. Since we will use only one Root, we use the current respectedGame Game type root. As it will return the latest state of the L2 blockchain.

- `OPContractsManager.sol::OPContractsManagerUpgrader::upgrade()#L596`:

```
GameType respectedGameType = IOptimismPortal(payable(opChainAddrs.optimismPortal)).respectedGameType();
↪  // <<<

// Separate context to avoid stack too deep.
{
    // Get the existing anchor root from the old AnchorStateRegistry contract.
    // Get the AnchorStateRegistry from the PermissionedDisputeGame.
    (Hash root, uint256 l2BlockNumber) = getAnchorStateRegistry( ... ).anchors(respectedGameType); //
    ↪  <<<
    ...
    upgradeToAndCall(
        ...
        abi.encodeCall(
            IAnchorStateRegistry.initialize,
            (
                superchainConfig,
                dgf,
                Proposal({ root: root, l2SequenceNumber: l2BlockNumber }), // <<<
                respectedGameType
            )
        )
    );
}
```

The problem is that the current GameType (Respected) may not retrieve the latest validated block, here is an example. If we changed the game type and before any game was created or finalized, we upgraded the L2 chain. The respected game type will not retrieve the latest finalized block.

Scenario:

- Game type was permissioned from the start.
- The L2 stills for 1,2,3,... years.
- Last finalized L2block is `10,000,000`.
- Portal respected Game Type is to permissioned.

- The GameType changed into permissionless.
- The L2 stills for 4,5,6,... years.
- Last finalized L2block is `20,000,000`.
- Portal respected Game Type changed to permissioned.
- After a period of time we performed our upgrade.
- Now there is no any new game created and finalized with the new respected Game Type, so the AnchorStateRegistery will return to L2BlockNumber equals 10,000,000 instead of 20,000,000 and the root will be that of block 10,000,000.

Since the Current Block At L2 is `>= 20,000,000` This will make the new Game created accepts These BlockNumbers. This will allow attackers to provide invalid claims by manuilating the first blocks (after `10,000,000`) and This will prevent Challengers to challenging them.

According to `Spec::iASR-005` if the game is older than 6 months this will open the possibility to provide Invalid claims that can not be challenged.

- iASR-005: The Anchor Game is recent enough to be fault provable.

  We require that the Anchor Game corresponds to an L2 block with an L1 origin timestamp that is no older than 6 months from the current timestamp. This time constraint is necessary because the fault proof VM must walk backwards through L1 blocks to verify derivation, and processing 7 months worth of L1 blocks approaches the maximum time available to challengers in the dispute game process.

Recommendation: We should pick the Largest blocknumber out of the Two games (`Permissioned` and `Permissionless`), to guarantee correct upgrading.

3.2.16 Missing pause mechanism spesific for `superRoots`'s `chainId`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `OptimismPortal2`, there is only one method to set `superRootsActive = true`, which is through `migrateToSuperRoots`. Once the `OptimismPortal2` migrates to SuperRoots using `migrateToSuperRoots`, the `superRootsActive` flag will remain permanently set to `true`.

```
function migrateToSuperRoots(IETHLockbox _newLockbox, IAnchorStateRegistry _newAnchorStateRegistry) external {
    superRootsActive = true;
}

function _proveWithdrawalTransaction(
    Types.WithdrawalTransaction memory _tx,
    IDisputeGame _disputeGameProxy,
    uint256 _outputRootIndex,
    Types.SuperRootProof memory _superRootProof,
    Types.OutputRootProof memory _outputRootProof,
    bytes[] memory _withdrawalProof
)
    internal
{
    if (superRootsActive) {
        // Verify that the super root can be generated with the elements in the proof.
        if (_disputeGameProxy.rootClaim().raw() != Hashing.hashSuperRootProof(_superRootProof)) {
            revert OptimismPortal_InvalidSuperRootProof();
        }

        // Check that the index exists in the super root proof.
        if (_outputRootIndex >= _superRootProof.outputRoots.length) {
            revert OptimismPortal_InvalidOutputRootIndex();
        }

        // Check that the output root has the correct chain ID.
        Types.OutputRootWithChainId memory outputRoot = _superRootProof.outputRoots[_outputRootIndex];
        if (outputRoot.chainId != systemConfig.l2ChainId()) {
            revert OptimismPortal_InvalidOutputRootChainId();
        }
    }
}
```

A `superRoot` has a chain-specific `outputRoot.chainId`, which should correspond to `systemConfig.l2ChainId`. Assume that a superRoot with `chainId` "C" encounters an issue that requires pausing the withdrawal mechanism (`proveWithdrawalTransaction`) for that chain. However, there is no way to pause the mechanism for a specific chain within SuperRoots. Consequently, if the `proveWithdrawalTransaction` is paused, it will affect all chains since the pause mechanism is controlled by `superchainConfig`, which manages both `OptimismPortal2` and `ETHLockbox`.

As a result, all withdrawal transactions between chains that are not affected by chain "C" will also be paused. This could impact the revenue fee by transactions between chain.

Proof of Concept: The flow of the issue is as follows:

- A `chainId` "C", which corresponds to the superRoot `chainId` "C", is hacked with unlimited minting.
- The emergency response involves the guardian initiating a pause mechanism using `superchainConfig`.
- However, this also pauses all transactions between chains that are not affected by chain "C".

Recommendation: Implement a mechanism to pause for a specific superRoot `chainId`.

3.2.17 L2 joining interop proof system can steal all the ETH inside the `ETHLockbox` contract

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: An L2 joining the interop proof system can steal all the ETH inside the ETHLockbox contract.

Finding Description: When an L2 joins the interop proof system, it shares an ETHLockbox with all the other L2s. The ETHLockbox contract allows any authorized OptimismPortal contract to unlock as much ETH as they want. To join the interop proof system, the `ETHLockbox` owner has to authorize the L2's `OptimismPortal` contract to lock and unlock ETH. Otherwise, the OptimismPortal wouldn't be able to function since it calls `lockETH()` on each deposit and `unlockETH()` on each withdrawal.

The only thing limiting the L2s `OptimismPortal` contract from withdrawing all the ETH in the lockbox is the current implementation of the portal. But the `OptimismPortal` is an upgradable contract. Therefore, there exists the possibility of a malicious upgrade being executed, which allows the owner of the `OptimismPortal`

to call the `ETHLockbox`'s `unlock` function with an arbitrary value. Hence, they can drain the `ETHLockbox`, which holds the ETH of *all* the L2s in the interop system.

Additionally, there's no guarantee that the L2 upgrades their system in a way where the existing locked ETH in their system is migrated to the interop system's ETHLockbox contract. In that case, any of the withdrawal that's finalized will be taking ETH that originally belonged to the other L2s in the interop system.

Impact Explanation: Loss of ETH stored in ETHLockbox contract. Affects all L2s participating in the interop system.

Likelihood Explanation: This attack depends on an L2 abusing the fact that their OptimismPortal contract is approved to access the `ETHLockbox` contract. This could happen if:

- an L2 dev team turns malicious (for example, if the attack earns them more than operating honestly).
- the L2's system is compromised by a third-party attacker.

Recommendation: The `ETHLockbox` contract should keep track of all the ETH deposited by each `OptimismPortal` and only allow them to withdraw the same amount of funds.

3.2.18 Blacklisted Games can take be at Anchor Roots

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: According to Optimism Specs `iASR-004` After the Upgrade, Blacklisted games should not be used as Anchor Games anymore, as this will affect future games.

- iASR-004: Invalidation functions operate correctly.

  We require that the blacklisting and retirement functions operate correctly. Games that are blacklisted must not be used as the Anchor Game, must not be considered Valid Games, and must not be usable to prove or finalize withdrawals. Any game created before a transaction that updates the retirement timestamp must not be set as the Anchor Game, must not be considered Valid Games, and must not be usable to prove or finalize withdrawals.

Severity: High/Critical. If this invariant is broken, the Anchor Game could be set to an incorrect value, which would cause future Dispute Game instances to use an incorrect starting state. This would lead games to resolve incorrectly and would be considered a High Severity issue. Issues that would allow users to finalize withdrawals with invalidated games would be considered Critical Severity.

In the Old Design The Optimism Portal was the source of truth, and blacklising a game was takin an action on the Portal itself. In the Old implementation if a game is Blacklisted it can be settled as Anchor (no check for game validity). And Portal handles this situation. But in the new implementation this is not allowed.

- FaultDisputeGame__Old::resolve:

```
function resolve() external returns (GameStatus status_) {
    // INVARIANT: Resolution cannot occur unless the game is currently in progress.
    if (status != GameStatus.IN_PROGRESS) revert GameNotInProgress();

    // INVARIANT: Resolution cannot occur unless the absolute root subgame has been resolved.
    if (!resolvedSubgames[0]) revert OutOfOrderResolution();

    // Update the global game status; The dispute has concluded.
    status_ = claimData[0].counteredBy == address(0) ? GameStatus.DEFENDER_WINS :
    ↪  GameStatus.CHALLENGER_WINS;
    resolvedAt = Timestamp.wrap(uint64(block.timestamp));

    // Update the status and emit the resolved event, note that we're performing an assignment here.
    emit Resolved(status = status_);

    // Try to update the anchor state, this should not revert.
    ANCHOR_STATE_REGISTRY.tryUpdateAnchorState(); // <<<
}
```

- AnchorStateRegistry__Old::tryUpdateAnchorState():

```
function tryUpdateAnchorState() external {
    // Grab the game and game data.
    IFaultDisputeGame game = IFaultDisputeGame(msg.sender);
    (GameType gameType, Claim rootClaim, bytes memory extraData) = game.gameData();

    // Grab the verified address of the game based on the game data.
    // slither-disable-next-line unused-return
    (IDisputeGame factoryRegisteredGame,) =
        DISPUTE_GAME_FACTORY.games({ _gameType: gameType, _rootClaim: rootClaim, _extraData: extraData
        ↪  });

    // Must be a valid game.
    if (address(factoryRegisteredGame) != address(game)) revert UnregisteredGame();

    // No need to update anything if the anchor state is already newer.
    if (game.l2BlockNumber() <= anchors[gameType].l2BlockNumber) {
        return;
    }

    // Must be a game that resolved in favor of the state.
    if (game.status() != GameStatus.DEFENDER_WINS) {
        return;
    }

    // Actually update the anchor state.
    anchors[gameType] = OutputRoot({ l2BlockNumber: game.l2BlockNumber(), root:
    ↪  Hash.wrap(game.rootClaim().raw()) }); // <<<
}
```

There is no check weather the game is blacklisted or Not, since this is handled by Portal. The issue is that when upgrading our contracts we take the last `OutputRoot` value (Proposal) and set it as the starting root, making this game as the Valid Claim at this point. But we are not checking weather this Root we are using is for a Blacklisted game or not.

- OPContractsManager.sol::OPContractsManagerUpgrader::upgrade()#L581-L583:

```
(Hash root, uint256 l2BlockNumber) = getAnchorStateRegistry(
    IFaultDisputeGame(address(permissionedDisputeGame))
).anchors(respectedGameType);
...
upgradeToAndCall(
    ...,
    abi.encodeCall(
        IAnchorStateRegistry.initialize,
        (
            superchainConfig,
            dgf,
            Proposal({ root: root, l2SequenceNumber: l2BlockNumber }), // <<<
            respectedGameType
        )
    )
);
```

This will make the the Starting anchorRoot is the root of Blacklisted game and it's L2BlockNumber, which is not allowed and invariant breaking according to the specs. This can also affect the validity of the future games that will be created with that malicious Root, as stated in the Specs.

Recommendation: Before using the Root, we should make sure that the Optimism Portal Is not blacklisting this Root. This may be too complex to get implemented onchain as the old Optimism Portal blacklist by the GameProxy Address, and the Old AnchorStateRegistry stored just the root and l2 blocknumber. It is better to monitor the Root off-chain before doing the upgrade process to prevent this issue from occuaring.

3.2.19 Missing check for `_gasLimit` being lower than `SYSTEM_DEPOSIT_GAS_LIMIT`

Severity: Informational

Context: OptimismPortal2.sol#L53-L54

Description: The `OptimismPortal2::depositTransaction` function enforces a minimum gas limit based on the size of the calldata by invoking `minimumGasLimit`, but it currently does not verify whether the pro-

vided `_gasLimit` meets the fixed minimum requirement defined by the constant `SYSTEM_DEPOSIT_GAS_LIMIT` (200,000).

Recommendation: Add something like the following:

```
if (_gasLimit < SYSTEM_DEPOSIT_GAS_LIMIT) {
    revert OptimismPortal_GasLimitTooLow();
}
```

Op Labs: `SYSTEM_DEPOSIT_GAS_LIMIT` is a dangling unused variable - needs to be removed from the contract.

Phaze: Fixed in commit b671b67. This variable is removed from the contract.