Optimism Incident Response Updates Review

January 24, 2025

Prepared for Optimism Labs

Conducted by:

Riley Holterhus (holterhus)

Kurt Willis (phaze)

Richie Humphrey (devtooligan)

About the Optimism Labs - Incident Response Updates Review

Optimism is a Layer 2 blockchain scaling solution for Ethereum that implements optimistic rollups. The system uses a dispute mechanism to ensure the validity of state transitions, with dispute games facilitating the resolution of contested transactions. This review focuses on the incident response improvements to the dispute game and portal system components, including updates to anchor state handling, game validation, and bond distribution mechanisms.

About Offbeat Security

Offbeat Security is a boutique security company providing unique security solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

Summary & Scope

The review covered code from two commits:

1. The changes to packages/contracts-bedrock/src at commit 984bae9:
    - L1/OptimismPortal2.sol
    - L1/OptimismPortalInterop.sol
    - dispute/AnchorStateRegistry.sol
    - dispute/DelayedWETH.sol
    - dispute/FaultDisputeGame.sol
    - dispute/lib/Errors.sol
    - dispute/lib/Types.sol
    - libraries/PortalErrors.sol

2. The changes to packages/contracts-bedrock/src at commit ce2ce43:
    - L1/OPContractsManager.sol
    - dispute/AnchorStateRegistry.sol
    - dispute/lib/Errors.sol

The audit identified 1 low and 3 informational severity issue. The key areas of focus were the dispute game validation changes, anchor state updates, and bond distribution mechanisms. The modifications improve the security model through more rigorous game state validation and bond distribution control, though they also introduce additional complexity that warrants careful testing.

Findings Summary

Identifier	Title	Severity	Status
L-01	Out-of-gas considerations for try/catch blocks	Low	Fixed in PR 14144
I-01	isGameAirgapped() returns true for unresolved games	Informational	Fixed in PR 14139
I-02	DelayedWETH contract lacks standard balance validations and safeguards	Informational	Acknowledged

I-03	Unclear naming for respected game type retirement mechanism	Informational	Acknowledged

Detailed Findings

Low Findings

[L-01] Out-of-gas considerations for try/catch blocks

In the two in-scope PRs, three new try/catch blocks are introduced, each using a generic catch statement that handles all error types. With this approach, it's important to consider how out-of-gas errors are handled, since a caller might intentionally forward insufficient gas to trigger a failure in the external call of the try/catch. Moreover, since EIP-150 reserves 1/64th of remaining gas for execution after an external call, the catch block and other logic after the try/catch may have enough gas to succeed, even though the external call failed.

In two of the three new try/catch blocks, this is not a concern because the catch block reverts anyway:

```
try disputeGameProxy.wasRespectedGameTypeWhenCreated() returns (bool wasRespecte
    if (!wasRespected_) revert InvalidGameType();
} catch {
    revert LegacyGame();
}
```

So, if an attacker intentionally triggers an out-of-gas failure in the external call, the catch block will revert and nothing is accomplished.

The third try/catch block is in the `closeGame()` function of the `FaultDisputeGame`, and attempts to update the anchor state in the `AnchorStateRegistry`:

```
try ANCHOR_STATE_REGISTRY.setAnchorState(IDisputeGame(address(this))) { } catch
```

In this case, an attacker forwarding insufficient gas could theoretically cause the external call to fail, skipping the anchor state update. However, this is unlikely to be a problem in practice.

Firstly, it is currently impossible for 63/64ths of the remaining gas to be insufficient for `setAnchorState()` while 1/64th is sufficient for the rest of `closeGame()`, as the gas cost of `setAnchorState()` is not significantly higher than that of the final portion of `closeGame()`. Secondly, occasionally skipping an anchor state update is not a major issue,

and it can always be corrected by independently calling `setAnchorState()` with the affected game.

So, in all three cases this is not a significant concern, but it may be worth documenting in case of future changes to the codebase.

Recommendation

Consider documenting this behavior in the codebase. Specifically, consider noting that a try/catch block may fail not because of the input/logic of the external call, but due to the overall gas forwarded during the call. If the external call has a relatively high gas cost compared to the remaining logic after the try/catch, it's possible that EIP-150 reserves enough gas for the rest of the function to succeed, which could lead to unexpected results.

Optimism Labs

Acknowledged, as noted in the report we don't believe that there is a gas value that can be provided that can trigger this issue. Given that we are attempting to avoid reverts that would block this function from being executed, we will resolve the issue by (1) adding fuzz testing to validate that no amount of gas can trigger the issue and (2) adding monitoring to alert if the Anchor State is getting too old. The fuzz testing and a new semgrep rule has been added in PR 14144.

Offbeat Security

Verified.

Informational Findings

[I-01] `isGameAirgapped()` returns `true` for unresolved games

In the `AnchorStateRegistry`, the `isGameAirgapped()` function checks if a `_game` has been resolved for longer than `portal.disputeGameFinalityDelaySeconds()`:

```
function isGameAirgapped(IDisputeGame _game) public view returns (bool) {
    return block.timestamp - _game.resolvedAt().raw() > portal.disputeGameFinali
}
```

Note that if the `_game` has not been resolved yet, then `_game.resolvedAt().raw()` will be 0, so this function will return `true`. This isn't a concern in the current codebase, since `isGameAirgapped()` is only called on games that are known to be resolved. However this behavior may be unexpected for someone calling `isGameAirgapped()` independently.

Recommendation

Consider changing `isGameAirgapped()` to return `false` for unresolved games, for example:

```
function isGameAirgapped(IDisputeGame _game) public view returns (bool) {
    return _game.resolvedAt().raw() != 0
        && block.timestamp - _game.resolvedAt().raw() > portal.disputeGameFinali
}
```

Optimism Labs

We are resolving this issue by removing `isGameAirgapped()` entirely given that a fixed version of the function is identical to `isGameFinalized()`. This is done in PR 14139.

Offbeat Security

Verified.

[I-02] DelayedWETH contract lacks standard balance validations and safeguards

Description

The DelayedWETH contract has several potential design considerations that, while not critical issues given its intended usage pattern, could be improved for better robustness and safety:

1. The `unlock()` function lacks balance validation, allowing accounts to unlock more WETH than they own. A malicious or compromised account could unlock `type(uint256).max` for all future withdrawals. While this doesn't create an immediate vulnerability given proper usage by FaultDisputeGames contracts, it would be safer to track pending withdrawal amounts per sender.

2. The `unlock()` function accepts zero amounts without reverting. While not a security issue since `unlock()` is only called once by the FaultDisputeGame contracts, allowing zero amounts could update the `WithdrawalRequest.timestamp` unnecessarily and delay legitimate withdrawals.

3. The withdrawal sub-accounting system is currently implemented at the contract level but relies entirely on proper usage by the caller. This creates a somewhat awkward separation of concerns, where the contract provides functionality that could potentially be implemented more cleanly at the caller level.

Recommendation

Given that this contract is specifically designed for use by FaultDisputeGames contracts and will be replaced with a standard WETH implementation in an upcoming release, no immediate changes are recommended.

Alternatively, the withdrawal sub-accounting could be moved entirely to the FaultDisputeGame contracts, simplifying the DelayedWETH implementation and making the separation of concerns more explicit.

Optimism Labs

Acknowledged, this was an intentional design choice, we will likely be removing this at some point in the future but we don't intend to make any changes to DelayedWETH at this time.

Offbeat Security

Acknowledged.

[I-03] Unclear naming for respected game type retirement mechanism

Description

The OptimismPortal2 contract's naming for the game type retirement mechanism is unclear and potentially misleading. The current `respectedGameTypeUpdatedAt` variable and `setRespectedGameType()` function handle two distinct operations:

1. Setting a new game type
2. Retiring the current game type by updating the timestamp

The current implementation uses `type(uint32).max` as a special value to distinguish between these operations within the same function, obscuring the intention and functionality.

This issue, while not a security vulnerability, reduces code clarity and maintainability. The dual-purpose `setRespectedGameType()` function and ambiguous `respectedGameTypeUpdatedAt` name make it harder to understand the contract's behavior.

Recommendation

Split the functionality into distinct operations and rename variables to better reflect their purpose:

```
contract OptimismPortal2 {
-    uint64 public respectedGameTypeUpdatedAt;
+    uint64 public gameTypeRetirementTimestamp;

-    function setRespectedGameType(GameType _gameType) external {
+    function setRespectedGameType(GameType _gameType) external {
         if (msg.sender != guardian()) revert Unauthorized();
-        if (_gameType.raw() == type(uint32).max) {
-            respectedGameTypeUpdatedAt = uint64(block.timestamp);
-        } else {
-            respectedGameType = _gameType;
-        }
+        respectedGameType = _gameType;
+        emit RespectedGameTypeSet(respectedGameType, Timestamp.wrap(gameTypeReti
+    }

+    function retireCurrentGameType() external {
+        if (msg.sender != guardian()) revert Unauthorized();
+        gameTypeRetirementTimestamp = uint64(block.timestamp);
+        emit GameTypeRetired(respectedGameType, Timestamp.wrap(gameTypeRetirement
+    }

     // For backwards compatibility if needed
+    function respectedGameTypeUpdatedAt() external view returns (uint64) {
+        return gameTypeRetirementTimestamp;
+    }
}
```

The backwards compatibility function ensures no breaking changes to existing integrations.

Optimism Labs

Acknowledged, this was an intentional design choice, we are aiming to remove this function entirely as part of a future upgrade.

Offbeat Security

Acknowledged.

Security Analysis Summary Appendix

Overview of Changes

This analysis covers modifications across three main contracts: `AnchorStateRegistry`, `FaultDisputeGame`, and `OptimismPortal2`. The changes represent an overhaul of the dispute game system, particularly around game validation, bond distribution, and anchor state management.

AnchorStateRegistry Changes

1. Single Anchor Game
    - Replaced mapping of game types to anchor roots with single `anchorGame` reference
    - Added integration with `OptimismPortal2` via `portal` state variable
    - Replaced `StartingAnchorRoot` struct with single `startingAnchorRoot` variable

2. Game Validation
    - New `isGameProper()` function combining multiple validation criteria
    - Added `isGameClaimValid()` for claim validation
    - Implemented `isGameAirgapped()` for airgap period checking
    - Added `isGameFinalized()` and `isGameResolved()` helper functions

3. Blacklist Handling
    - `getAnchorRoot()` reverts if current anchor game is blacklisted
    - System intentionally bricks if anchor game becomes blacklisted

4. Game Type Validation
    - Now uses `wasRespectedGameTypeWhenCreated()` for validation
    - Game type check moved from `isGameProper()` to `isGameClaimValid()`
    - Added retirement check for games created in same block as update timestamp

FaultDisputeGame Changes

1. Credit Tracking
    - Introduced dual credit tracking system (`normalModeCredit` and `refundModeCredit`)
    - Added `hasUnlockedCredit` mapping
    - Implemented `BondDistributionMode` enumeration (`NORMAL` , `REFUND` , `UNDECIDED`)

2. Game Finalization
    - New two-step process: `resolveGame()` and `closeGame()`
    - New `closeGame()` procedure requires airgapped delay
    - Distribution mode determined by game validity: `isGameProper()`
    - Call to `tryUpdateAnchorState()` is replaced by try catch when setting anchor state

3. Game Type Validation
    - Prevention of `type(uint32).max` as game type
    - Added tracking for game type respect at creation time

4. Bond Management
    - Two-stage bond claiming process
    - Separate credit state tracking for normal and refund mode

OptimismPortal2 Changes

1. Validation System
    - New validation of game type at creation time
    - Explicit rejection of legacy games
    - Combined update mechanism for game type and timestamp
    - Uses `type(uint32).max` as special value for timestamp updates
    - Games created in same block as game invalidation timestamp considered invalid