SPEARBIT

Op Enclave Security Review

Auditors

0xIcingdeath, Lead Security Researcher

0xTylerholmes, Security Researcher

CarrotSmuggler, Associate Security Researcher

Zigtur, Security Researcher

Report prepared by: Lucas Goiriz

January 24, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Base is a secure and low-cost Ethereum layer-2 solution built to scale the userbase on-chain.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Op Enclave according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 18 days in total, Coinbase engaged with Spearbit to review the op-enclave protocol. In this period of time a total of **16** issues were found.

Summary

Project Name	Coinbase
Repository	op-enclave
Commit	98d346dd
Type of Project	Infrastructure, L2
Audit Timeline	Dec 23rd to Jan 10th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	3	3	0
Low Risk	5	2	3
Gas Optimizations	1	1	0
Informational	6	5	1
Total	**16**	**12**	**4**

5 Findings

5.1 High Risk

5.1.1 Proposer can steal funds from the portal by including malicious deposits in proofs

Severity: High Risk

Context: stateless.go#L93-L106

Description: The proposer interacts with the enclave to generate output proof by providing the `txs` for the current block. The enclave then verifies that the deposits for the associated L1 block are embedded in the `txs` input. However, the enclave does not ensure that only the deposit transactions from L1 are embedded in `txs`.

```go
func ExecuteStateless(
    ctx context.Context,
    config *params.ChainConfig,
    rollupConfig *rollup.Config,
    l1Origin *types.Header,
    l1Receipts types.Receipts,
    previousBlockTxs []hexutil.Bytes,
    blockHeader *types.Header,
    blockTxs []hexutil.Bytes,
    witness *stateless.Witness,
    messageAccount *eth.AccountResult,
) error {
    // ...

    l1Fetcher := NewL1ReceiptsFetcher(l1OriginHash, l1Origin, l1Receipts)
    l2Fetcher := NewL2SystemConfigFetcher(rollupConfig, previousBlockHash, previousBlockHeader,
    ↪  previousTxs)
    attributeBuilder := derive.NewFetchingAttributesBuilder(rollupConfig, l1Fetcher, l2Fetcher)
    payload, err := attributeBuilder.PreparePayloadAttributes(ctx, l2Parent, eth.BlockID{ // @POC: build
    ↪  payload with deposits from L1
        Hash:   l1OriginHash,
        Number: l1Origin.Number.Uint64(),
    })
    if err != nil {
        return fmt.Errorf("failed to prepare payload attributes: %w", err)
    }

    if txs.Len() < len(payload.Transactions) {
        return errors.New("invalid transaction count")
    }

    for i, payloadTx := range payload.Transactions { // @POC: Ensure that all deposits from L1 are in
    ↪  the `txs` input
        tx := txs[i]
        if !tx.IsDepositTx() {
            return errors.New("invalid transaction type")
        }
        if !bytes.Equal(blockTxs[i], payloadTx) {
            return errors.New("invalid deposit transaction")
        }
    }

        // @POC: a check to ensure that no more deposits are embedded is missing
```

This allows the proposer to add malicious deposits in the `txs` input. These deposits will not be verified by the enclave to ensure that they come from the L1. By exploiting this, the proposer can deposit 1000ETH and then withdraw these 1000ETH in the same block, allowing to steal all funds in the `Portal`.

Recommendation: The enclave must ensure that the number of deposits in the `txs` input corresponds to the number of deposits in `payload`. This is to ensure that no additional deposits not coming from L1 or op-stack upgrades are embedded in `txs`.

Base: Fixed in PR 57.

Spearbit: Fixed. The deposit transactions are not passed in the `txs` input anymore. The `txs` input has been renamed `sequencedTxs` and only include transactions submitted through the sequencer. Deposit transactions are rebuilt from the L1 data passed as input, for which the block root hash will be checked by the portal. The attack vector has been deleted through this refactoring.

5.2 Medium Risk

5.2.1 Lack of parallelization for batches allows attackers to delay batcher from relaying data

Severity: Medium Risk

Context: channel_out.go#L30-L35, driver.go#L341-L343

Description: If withdrawals are submitted one after another, this will delay batches from being sent to the data availability provider, as each withdrawal that is submitted, until they are finished being processed. This means any transactions submitted to the batcher in between the withdrawal executions are dropped.

One of the changes from the existing op-enclave is when a withdrawal comes in, the batch is immediately closed and sent for batching. In either situation where there are a lot of withdrawals or an attacker wishes to block proper execution of this flow, it is a relatively low-cost denial of service attack vector.

Exploit Scenario: Transactions are being batched for batch 1. A series of 0.1 ETH withdrawals are submitted by a user. The first will result in block 1 closing and being batched. Each new transaction will result in a new batch being created, which means there is no chance to add transactions to a block between batches 1-10, and incoming transactions will be delayed

Recommendation: Consider either:

- Parallelizing the batcher to allow for transactions not being missed in this time period.
- Document proper assumptions for what the expected behaviour is.

Base: This will be remediated with PR 41, which ensures that the ErrWithdrawalDetected is only used if the block is within the last 10 seconds. If multiple withdrawals happen in series, it will ensure the block is fresh before it submits the batch.

Spearbit: Fixed.

5.2.2 `DeployChain` calls can be front-run to deny chain deployments

Severity: Medium Risk

Context: DeployChain.sol#L119-L141, DeployChain.sol#L161-L163

Description: The `deploy` function is permissionless. The chain ID given as input is used as a salt to derive the proxies address. However, as this deployment is permissionless, any user can deploy these proxies with an arbitrary chain ID without being tracked off-chain. This vulnerability can be exploited through the `deploy` and through the `deployProxy` functions.

Scenario: Bob wants to deploy a L3 on Base. The chain ID 12345 is given by Coinbase and a call to `deploy` is made. However, Alice (i.e. the attacker) calls `deploy` with this given chain ID before the legit transaction is executed. This call deploys the proxies. Finally, Bob's transaction reverts because the proxies deployment failed.

Recommendation: First, `deployProxy` must not be exposed publicly. Consider making this function internal. Second, consider setting an access control on `deploy` function. If this access control is not expected, do not require let user pass arbitrary chain ID. This can be done through an incremental counter within a given range as chain ID.

Base: Fixed in PR 45.

Spearbit: Fixed. The `DeployChain` contract is now `Ownable` and only the owner can deploy new chains. Also, the `deployProxy` function is now `internal`.

5.2.3 Multiple withdrawals in a single L3 transaction are not provable through op-withdrawer

Severity: Medium Risk

Context: withdrawals.go#L54-L60

Description: The op-withdrawer implements the `provideWithdrawal` command that takes as input a txHash on the L3 in which there is a withdrawal to prove a withdrawal on the Base L2.

This command calls the `withdrawals.ProveWithdrawalParametersForBlock` function from the Optimism repository. It will parse the logs from the given transaction on L3. However, this function from the Optimism repository do not support multiple withdrawals in the same transaction.

This leads the op-withdrawer binary to not being able to prove withdrawals when there are multiple in a single L3 transaction.

Note: The op-stack specifications about withdrawals do not mention that multiple withdrawals in a single transaction is not supported. This behaviour should be supported.

Code snippet: The`withdrawals.ProveWithdrawalParametersForBlock` function parses the withdrawal logs by calling `ParseMessagePassed`. Comments on this function are explaining that it does not support multiple withdrawals.

```
// ParseMessagePassed parses MessagePassed events from
// a transaction receipt. It does not support multiple withdrawals
// per receipt.
func ParseMessagePassed(receipt *types.Receipt) (*bindings.L2ToL1MessagePasserMessagePassed, error) {
    contract, err := bindings.NewL2ToL1MessagePasser(common.Address{}, nil)
    if err != nil {
        return nil, err
    }

        // ...
}
```

Recommendation: The op-withdrawer should support proving and finalizing all withdrawals that are in a single L3 transaction.

Base: Fixed in PR 56 and PR 13568 on Optimism.

Spearbit: Fixed. The withdrawer is now able to handle multiple withdrawals in a single transaction.

5.3 Low Risk

5.3.1 Missing contract size check on implementation can result in _doProxyCall and `_resolveImplementation` succeeding silently

Severity: Low Risk

Context: ResolvingProxy.sol#L96-L99, ResolvingProxy.sol#L137-L143, ResolvingProxy.sol#L149-L152

Description: When using `delegatecall`, account existence must be checked prior to calling, otherwise the outer call will succeed while the target implementation contract's code was never executed. The function used to retrieve the implementation is the following:

```
function _getImplementation() internal view returns (address) {
    address impl;
    bytes32 proxyImplementation = IMPLEMENTATION_SLOT;
    assembly {
        impl := sload(proxyImplementation)
    }
    return impl;
}
```

Note from the Solidity documentation specifically:

> The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

For reference, check the Solidity docs on control structures.

The `_resolveImplementation` function is also vulnerable to the following attack, regardless of whether wrapped in assembly block or not. Thus, the admin address should also be checked to ensure that its code length is greater than zero.

Exploit Scenario:

1. `deployProxy` is called with `proxy` and `salt, where` proxyisaddress(0)‘:

   ```
   function deployProxy(address proxy, bytes32 salt) public returns (address) {
       return ResolvingProxyFactory.setupProxy(proxy, proxyAdmin, salt);
   }
   ```

2. Proxy will be created using `create2` which invokes the `ResolvingProxy` constructor, setting the `IMPLEMENTATION_SLOT` to the zero address:

   ```
   function _setImplementation(address _implementation) internal {
       bytes32 proxyImplementation = IMPLEMENTATION_SLOT;
       assembly {
           sstore(proxyImplementation, _implementation)
       }
   }
   ```

3. When a user invokes a function on the `ResolvingProxy` either through the fallback or another way,, the admin need not even be setup correctly – it will merely return the proxy from the slot above:

   ```
   function _resolveImplementation() internal view returns (address) {
       address proxy = _getImplementation();
       bytes memory data = abi.encodeCall(IResolver.getProxyImplementation, (proxy));
       (bool success, bytes memory returndata) = _getAdmin().staticcall(data);
       if (success && returndata.length == 0x20) {
           return abi.decode(returndata, (address)); // return value of implementation.getproxy
       }
       return proxy; // otherwise return implementation contract
   }
   ```

4. The `_doProxyCall` will use this zero address to execute the delegatecall, which in the most outer transaction, will succeed. This is because developers are expected to check that the address being delegate-called on exists prior to calling any contracts on it.

Recommendation: Add a check to ensure the `_implementation` and `admin` contract provided to the Resolving-Proxy contract is non-zero. This can be done using OpenZeppelin's `isAddress` helper function or through checking `address.code.length > 0` or `extcodesize(addr) > 0` in Yul.

Base: Acknowledged. Given this proxy is aimed to match the implementation of Optimism's Proxy (Proxy.sol#L124), which doesn't check for codesize prior to a delegatecall, and also the fact that this only has impact from a erroneous deploy or upgrade, I'm inclined not to change this behavior. However Optimism's Proxy does have a 0 address check, which we may add a check for.

Spearbit: Acknowledged.

5.3.2 `depositHash` command of the op-withdrawer fails with multiple deposit logs in a single transaction

Severity: Low Risk

Context: main.go#L178-L194

Description: The `depositHash` command in the op-withdrawer binary reads a transaction from Base L2 and returns the calculated deposit transaction hash executed on the L3. However, this command do not support multiple deposits in a single transaction. This is supported in the OP-stack specifications about the derivation process.

Recommendation: The `depositHash` command in the op-withdrawer binary should loop through all the `deposits` and print all of them.

```diff
diff --git a/op-withdrawer/main.go b/op-withdrawer/main.go
index 7dedbee..574bd2a 100644
--- a/op-withdrawer/main.go
+++ b/op-withdrawer/main.go
@@ -189,12 +189,11 @@ func DepositHash(cliCtx *cli.Context) error {
    if err != nil {
        return err
    }
-   if len(deposits) != 1 {
-       return fmt.Errorf("expected 1 deposit, got %d", len(deposits))
-   }

-   hash := types.NewTx(deposits[0]).Hash()
-   fmt.Println(hash.Hex())
+   for _, deposit := range deposits {
+       hash := types.NewTx(deposit).Hash()
+       fmt.Printf("Deposit hash: %s\n", hash.Hex())
+   }

    return nil
 }
```

Base: Fixed in PR 46.

Spearbit: Fixed. The `depositHash` command now logs all the deposits in a transaction.

5.3.3 Malicious actor can make the batcher consume ETH gas

Severity: Low Risk

Context: channel_out.go#L30-L35

Description: The batcher and proposer are pushing a batch and a proof whenever there is a single withdrawal in a L3 block. This behavior enables an attack vector where any user can force the batcher to consume gas. Especially, a L3 withdrawal is way cheaper than a L2 transaction to push the batch or an L2 transaction to push the L3 output to the oracle.

Proof of Concept: An L3 withdrawal transaction (0xdbd9...0c57) on the testnet to withdraw 1 Wei of value consumed 14605115 Wei of gas.

On the Base L2, the batch transaction and the output oracle transaction consumed more ETH.

- Batch transaction (0xa628...0be7): 21159061365145 Wei.

- Output oracle (0x2e22...babe): 149456639281631 Wei.

This means that L2 transactions consume `11_000_000x` more gas than the L3 withdrawal.

```
(170615700646776) / 14605115 = 11681914,22
```

Recommendation: An additional fee could be implemented or the L1 gas cost could be adapted to make users pay this cost.

Base: Acknowledged. We are okay with this risk initially to keep the protocol simple, as the L2 costs end up being sequencer revenue. However, we may implement your L1 gas cost suggestion if certain chains cause DA / proposal submission costs to increase.

Spearbit: Acknowledged. L1 gas cost modification can be used to mitigate the issue.

5.3.4 Bytecode handles 0 address calls different from proxy implementation

Severity: Low Risk

Context: ResolvingProxy.sol#L150-L152

Description: One of the changes under review is the `setupProxy` of the `ProxyFactory` contract, which contains hand-rolled assembly to match an EEIP1167 minimal proxy. The two behave differently when invoking a function on the zero address.

The bytecode implementation calls `addr(0)` and initiates a STOP, then return true. The proxy in-Solidity code implementation merely sets the zero, then REVERT's when a call on the zero is attempted. This breaks the assumption that both contracts must behave in the same way.

Proof of Concept:

```
  [41251] ResolvingProxyDiff::testFuzz_randomCalldata(0xd301746526573b5dfadd55edfd4591deddb4c03b4fe007e ⌟
↪  f29a3281ad596cedf33cb42c444f28a14b7b5ab5d)
    [0] VM::allowCheatcodes(ProxyAdmin: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f])
      ← [Return]
    [17202] 0xCda31DE038a871e5b2dAd5853229b53C5c96a484::d3017465(26573b5dfadd55edfd4591deddb4c03b4fe00 ⌟
  ↪  7ef29a3281ad596cedf33cb42c444f28a14b7b5ab5d)
      [10038] ProxyAdmin::getProxyImplementation(Proxy: [0x2e234DAe75C793f67A35089C9d99245E1C58470b])
  ↪  [staticcall]
        [4489] Proxy::implementation() [staticcall]
          ← [Return] 0x0000000000000000000000000000000000000000
        ← [Return] 0x0000000000000000000000000000000000000000
      [0] 0x0000000000000000000000000000000000000000::d3017465(26573b5dfadd55edfd4591deddb4c03b4fe007 ⌟
  ↪  ef29a3281ad596cedf33cb42c444f28a14b7b5ab5d) [delegatecall]
        ← [Stop]
      ← [Return]
    [7064] ResolvingProxy::d3017465(26573b5dfadd55edfd4591deddb4c03b4fe007ef29a3281ad596cedf33cb42c444 ⌟
  ↪  f28a14b7b5ab5d)
      [1538] ProxyAdmin::getProxyImplementation(Proxy: [0x2e234DAe75C793f67A35089C9d99245E1C58470b])
  ↪  [staticcall]
        [489] Proxy::implementation() [staticcall]
          ← [Return] 0x0000000000000000000000000000000000000000
        ← [Return] 0x0000000000000000000000000000000000000000
      ← [Revert]
    [0] VM::assertEq(true, false, "expected bytecode and resolve success to be the same") [staticcall]
      ← [Revert] expected bytecode and resolve success to be the same: true != false
    ← [Revert] expected bytecode and resolve success to be the same: true != false
```

Recommendation: Overload the Optimism proxy (non bytecode version) `_doProxyCall` function to match the behaviour of the the bytecode version of the proxy. This recommendation has been suggested as they decided that the proxy should not validate proxy address, and instead should merely bubble data up. In order for these two implementations to match, it requires the non-bytecode version to remove the success check and not revert.

```
  /// @notice Performs the proxy call via a delegatecall.
  function _doProxyCall() internal {
      address impl = _getImplementation();
      require(impl != address(0), "Proxy: implementation not initialized");

      assembly {
          // Copy calldata into memory at 0x0....calldatasize.
          calldatacopy(0x0, 0x0, calldatasize())

          // Perform the delegatecall, make sure to pass all available gas.
          let success := delegatecall(gas(), impl, 0x0, calldatasize(), 0x0, 0x0)

          // Copy returndata into memory at 0x0....returndatasize. Note that this *will*
          // overwrite the calldata that we just copied into memory but that doesn't really
          // matter because we'll be returning in a second anyway.
          returndatacopy(0x0, 0x0, returndatasize())

          // Success == 0 means a revert. We'll revert too and pass the data up.
-         if iszero(success) { revert(0x0, returndatasize()) }

          // Otherwise we'll just return and pass the data up.
          return(0x0, returndatasize())
      }
  }
```

5.3.5 Incorrect binary search logic in `getL2OutputIndexAfter` function

Severity: Low Risk

Context: OutputOracle.sol#L161-L186

Description: The `getL2OutputIndexAfter` function uses binary search to find the the first output greater than or equal to the input block number. It incorporates an `offset` value as well, to account for the re-use of the `l2Outputs` array. The issue is that this logic is flawed, giving incorrect results.

Lets assume the array of numbers is $[700, 200, 300, 400, 500, 600]$. 700 was the `l2Outputs[i].l2BlockNumber` latest entry, so `latestOutputIndex = 0`.

In this case, the array stores the ending block number of the proofs stored. So searching for 299 should return index 2, since l2Outputs[2] = 300, which is the first output greater than or equal to the input. However in the current case, it returns index 1 instead.

This behaviour can be verified with the following proof of concept, which implements only the logic of the binary search here.

```
contract Test{
    uint256[6] l2Outputs = [700,200,300,400,500,600];
    uint256 latestOutputIndex = 0;

    function testBinary(uint256 _l2BlockNumber) public view returns (uint256) {
        uint256 lo = 0;
        uint256 hi = l2Outputs.length;
        uint256 offset = latestOutputIndex + 1 % l2Outputs.length;
        while (lo < hi) {
            uint256 mid = (lo + hi) / 2;
            uint256 m = (mid + offset) % l2Outputs.length;
            if (l2Outputs[m] < _l2BlockNumber) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }
}
```

Recommendation: The optimism-bedrock contracts also implement a `getL2OutputIndexAfter` function, without the offset functionality which gives the correct expected result. The usage of offset is erroneous, and the function here returns 1 less than the correct index.

Base: Fixed in PR 49.

Spearbit: Fixed. `lo` now starts from the `offset` itself, giving the correct index.

5.4 Gas Optimization

5.4.1 Duplicated code in `proveAndFinalizeWithdrawalTransaction`

Severity: Gas Optimization

Context: Portal.sol#L279-L289

Description: The `proveAndFinalizeWithdrawalTransaction` function merges the logic of "prove" and "finalize". Now that the two logics have been merged, optimizations can be made as the two logics initially showed the same checks. Some external calls are not needed anymore. The `@audit` comments in the following code snippet show code that can be optimized:

```
function proveAndFinalizeWithdrawalTransaction(
    Types.WithdrawalTransaction memory _tx,
    uint256 _l2OutputIndex,
    Types.OutputRootProof calldata _outputRootProof,
    bytes[] calldata _withdrawalProof
) public whenNotPaused {
    // Prevent users from creating a deposit transaction where this address is the message
    // sender on L2.
    if (_tx.target == address(this)) revert BadTarget();

    // Get the output root and load onto the stack to prevent multiple mloads. This will
    // revert if there is no output root for the given block number.
    bytes32 outputRoot = l2Oracle.getL2Output(_l2OutputIndex).outputRoot;

    // Verify that the output root can be generated with the elements in the proof.
    require(
        outputRoot == Hashing.hashOutputRootProof(_outputRootProof), "OptimismPortal: invalid output
        ↪  root proof"
    );
```

```
// 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 withdrawalHash = Hashing.hashWithdrawal(_tx);
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.
require(
    SecureMerkleTrie.verifyInclusionProof({
        _key: abi.encode(storageKey),
        _value: hex"01",
        _proof: _withdrawalProof,
        _root: _outputRootProof.messagePasserStorageRoot
    }),
    "OptimismPortal: invalid withdrawal inclusion proof"
);

// Emit a `WithdrawalProven` event.
emit WithdrawalProven(withdrawalHash, _tx.sender, _tx.target);

// Make sure that the l2Sender has not yet been set. The l2Sender is set to a value other
// than the default value when a withdrawal transaction is being finalized. This check is
// a defacto reentrancy guard.
if (l2Sender != Constants.DEFAULT_L2_SENDER) revert NonReentrant();

// Grab the OutputProposal from the L2OutputOracle, will revert if the output that
// corresponds to the given index has not been proposed yet.
Types.OutputProposal memory proposal = l2Oracle.getL2Output(_l2OutputIndex); // @audit this has
↪  already been called before

// Check that the output root that was used to prove the withdrawal is the same as the
// current output root for the given output index. An output root may change if it is
// deleted by the challenger address and then re-proposed.
require(
    proposal.outputRoot == outputRoot, // @audit unneeded check, already checked before
    "OptimismPortal: output root proven is not the same as current output root"
);
```

Recommendation: The `proveAndFinalizeWithdrawalTransaction` function can be optimized.

```
diff --git a/contracts/src/Portal.sol b/contracts/src/Portal.sol
index c2db5d5..0f87e8d 100644
--- a/contracts/src/Portal.sol
+++ b/contracts/src/Portal.sol
@@ -276,18 +276,6 @@ contract Portal is Initializable, ResourceMetering, ISemver {
         // a defacto reentrancy guard.
         if (l2Sender != Constants.DEFAULT_L2_SENDER) revert NonReentrant();

-        // Grab the OutputProposal from the L2OutputOracle, will revert if the output that
-        // corresponds to the given index has not been proposed yet.
-        Types.OutputProposal memory proposal = l2Oracle.getL2Output(_l2OutputIndex);
-
-        // Check that the output root that was used to prove the withdrawal is the same as the
-        // current output root for the given output index. An output root may change if it is
-        // deleted by the challenger address and then re-proposed.
-        require(
-            proposal.outputRoot == outputRoot,
-            "OptimismPortal: output root proven is not the same as current output root"
-        );
-
         // Check that this withdrawal has not already been finalized, this is replay protection.
         require(finalizedWithdrawals[withdrawalHash] == false, "OptimismPortal: withdrawal has already
         ↪  been finalized");
```

Base: Fixed in PR 48.

Spearbit: Fixed. Recommended patch has been applied.

5.5 Informational

5.5.1 Add zero checks on incoming addresses in constructor

Severity: Informational

Context: DeployChain.sol#L89-L100

Description: Consider adding zero address checks to the contract deployment that sets the address of different values, including the proxy admin, optimism portal, system config, bridges, factories, messengers, output oracle and super chain config. The accidental setting of zero address on any of these will result in the `DeployChain` contract needing to be redeployed.

Recommendation: Add zero checks and create a document and deployment script with clear guidelines on the allowed and unallowed values of the addresses.

Base: Added zero checks in PR.

Spearbit: Fixed.

5.5.2 Portal uses op-stack version

Severity: Informational

Context: Portal.sol#L99-L103

Description: The `version` function uses the versioning from the op-stack. However, its own versioning should be created as the contract does not follow the op-stack anymore.

Recommendation: Consider adding a string to identify the Base L3 Portals.

Base: Fixed in PR 44.

Spearbit: Fixed. The Portal for `op-enclave` now uses its own versioning.

5.5.3 Unused code causes code bloat

Severity: Informational

Context: driver.go#L191-L196

Description: Remove unnecessary code and unused functions such as the one linked. This helps ensure the code is readable and maintainable.

Recommendation: Run consistent linters to allow go lints to notify of unused code blocks or branches.

Base: Fixed in PR 47.

Spearbit: Fixed.

5.5.4 Proposer can break L2OutputOracle

Severity: Informational

Context: OutputOracle.sol#L105-L147

Description: The `proposeL2Output` function allows the proposer to submit an L2 output. This output can be verified through a proof depending on the configuration. In case of a malicious or compromised proposer, multiple vectors can break the `L2OutputOracle` contract. This would make withdrawals from the L3 impossible. Here are some of these denial of service vectors:

- `_l2BlockNumber` is not included in the signed message, so it can potentially be forged by the proposer to block future l2Output updates if set to a very large number, due to the `require(_l2BlockNumber > latestBlockNumber())` check.
- Proposer provides an output in which there is an additional transaction that is not available in the alt-DA. This can be done by providing incorrect L3 state to the enclave for proof generation.

Recommendation: None, the assumption is taken that the proposer is not malicious or compromised.

Base: `l2BlockNumber` as part of the signature has been fixed in PR 42. For transaction not being available in DA: this requires some deeper thinking, but we hope to fix this by providing some native DA solution on the Base L2 at some point in the future.

Spearbit: Partially fixed. The enclave now build the L2 output payload by including the L2 block number before signing it. The DA availability issue is not fixed.

5.5.5 RSA-4096 can be used

Severity: Informational

Context: server.go#L111-L114

Description: The enclave server uses RSA-2048 as the encryption mechanism. While it has not been shown to be breakable, the NIST in its 800-78-5 indicates that RSA-2048 is not recommended for uses after year 2031.

According to AWS documentation about KMS, RSA-3072 and RSA-4096 are supported. As the enclave server does not have specific performance concerns on the RSA encryption mechanism, a higher key size can be used.

Recommendation: Consider using RSA-4096 in the enclave.

Base: Fixed in PR 50.

Spearbit: Fixed. RSA-4096 is now used.

5.5.6 Channel can be closed without a single withdrawal

Severity: Informational

Context: channel_out.go#L31-L33

Description: The `channelOut.addBlock` function closes a channel after the current block when a withdrawal is found in this block. This is done by setting the `ErrWithdrawalDetected` error when the `Bloom` filter finds the `L2ToL1MessagePasser` address is found in the logs. However, this `Bloom` filter does not ensure that the `L2ToL1MessagePasser` contract has been called. This logic allows an attacker to emit a log on L3 with the `L2ToL1MessagePasser` to trigger a batch submission.

Recommendation: The filter should ensure that the L2ToL1MessagePasser is the emitter of a `MessagePassed` log.

Base: Acknowledged.

Spearbit: Acknowledged. The exploitability of this issue is low and has been reduced by the fix for issue "Lack of parallelization for batches allows attackers to delay batcher from relaying data".