Relic Dencun Audit Report

Prepared by Riley Holterhus

March 14, 2024

Contents

1 Introduction

1.1 About Relic Protocol

Relic Protocol provides trustless access to historical blockhash values from the Ethereum blockchain. By using a combination of zk-SNARKs and merkle proofs, Relic Protocol can be used as a scalable oracle of historical blockchain data. For more information, visit Relic Protocol's website: relicprotocol.com.

1.2 About Riley Holterhus

Riley Holterhus is an independent security researcher that focuses on Solidity smart contracts. Other than conducting independent security reviews, he works as a Lead Security Researcher at Spearbit, and also searches for vulnerabilies in live codebases. Riley can be reached by email at rileyholterhus@gmail.com, by Telegram at @holterhus and on Twitter/X at @rileyholterhus.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the author has made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an "as-is" basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

From March 9th, 2024 through March 13th, 2024, Riley Holterhus conducted an audit of Relic Protocol's Dencun contracts. During this period, a manual analysis was undertaken to identify various security issues and logic flaws.

The Relic Dencun contracts are focused on leveraging the Ethereum network's Dencun hardfork, which went live on March 13th, 2024. This hardfork introduced EIP4788, which allows smart contracts to access historical blockchain information in a much more streamlined fashion.

This audit was conducted on the codebase found in the relic-dencun-audit GitHub repository, specifically on commit 0917a12. The following files were in scope for the audit:

- contracts/lib/SSZ.sol
- contracts/lib/MerkleTree.sol
- contracts/lib/Propogate.sol
- contracts/BeaconBlockHistoryBase.sol
- contracts/BeaconBlockHistory.sol
- contracts/ProxyBeaconBlockHistory.sol

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities are subjective, but aim to capture the impact and feasibility of each potential issue. In total, 2 medium-severity findings, 3 low-severity findings, and 3 informational findings were identified.

All issues identified in the audit have been addressed, with the exception of a minor typo which has already been cemented in a previous version of the code. The subsequent changes were reviewed, specifically up to (and including) commit 3afed0a.

3 Findings

3.1 Medium Severity Findings

3.1.1 Relative proving can verify uninitialized values

Description: Since the `_verifyRelativeBlockRoot()` function utilizes the `block_roots` vector, and since the behavior of this vector has not changed since the inception of the Beacon Chain, it's possible for a sequence of `_verifyRelativeBlockRoot()` calls to prove values from arbitrarily far in the past.

As a result, the contract can use a block root from one of the first 8192 slots of the Beacon Chain. In these early slots, the `block_roots` vector will not have been fully filled, and will still contain uninitialized values. Since there is no logic to account for this, these uninitialized values can be proven as real block roots, which should not be allowed.

Note that to reach the first 8192 slots, the sequence of `_verifyRelativeBlockRoot()` calls would need to occur in a single transaction, and would therefore not be allowed to exceed the block gas limit. This may or may not be feasible. Since this functionality will ultimately be used on several blockchains, it'd be simplest if this issue was explicitly prevented, regardless of its feasibility.

Recommendation: Ensure that the `_verifyRelativeBlockRoot()` function does not return a `blockRoot` equal to one of the default values in the `block_roots` vector. In this case, this default value would be `bytes32(0)`, so this can be checked as follows:

```
function _verifyRelativeBlockRoot(
    bytes calldata rawProof
) internal view returns (bytes32 blockRoot) {
    RelativeBlockRootProof calldata proof = _castRelativeBlockRootProof(rawProof);

    // first verify the base block root
    blockRoot = _verifyBeaconBlockRoot(proof.baseProof);

    // now access the base block's state root
    bytes32 stateRoot = SSZ.verifyBlockStateRoot(proof.stateRootProof, blockRoot);

    uint256 index = proof.index;
    require(index < SLOTS_PER_HISTORICAL_ROOT, "block_roots index out of bounds");

    // verify the target block root relative to the base block root
    blockRoot = SSZ.verifyRelativeBlockRoot(
        proof.relativeRootProof,
        index,
        stateRoot
    );
+   require(blockRoot != bytes32(0));
}
```

Relic: Addressed in commit 1df44a7.

Auditor: Verified.

3.1.2 Incorrect proof length for `verifyRelativeBlockRoot()`

Description: The `_verifyRelativeBlockRoot()` function allows for relative proving, one of the four potential methods of verifying a beacon block root. This approach proves that a specific block root exists within the `block_roots` vector of a known beacon state root. The relevant proving code is as follows:

```
function verifyRelativeBlockRoot(
    bytes32[] calldata proof,
    uint256 index,
    bytes32 stateRoot
) internal view returns (bytes32 blockRoot) {
    require(proof.length == 20, "invalid proof length");
    blockRoot = proof[0];
    bytes32 vectorRoot = MerkleTree.proofRoot(
        index,
        blockRoot,
        proof[1:14]
    );
    bytes32 computedRoot = MerkleTree.proofRoot(
        BLOCK_ROOTS_INDEX,
        vectorRoot,
        proof[14:]
    );
    require(computedRoot == stateRoot, "invalid relative proof");
}
```

Notice that `proof[14:]` is the proof that the `block_roots` vector in question is contained in the `stateRoot`. Since `proof.length == 20`, this component must be of length 6. On the other hand, since the `BeaconState` container has always contained between 17 and 32 elements, it'd be expected that a proof of an element in the `stateRoot` would be of length 5.

This discrepancy will make it impossible to do relative proving, since every proof must contain an extra element, and the `computedRoot` will necessarily be incorrect.

Recommendation: Decrease the proof length requirement by one:

```
function verifyRelativeBlockRoot(
    bytes32[] calldata proof,
    uint256 index,
    bytes32 stateRoot
) internal view returns (bytes32 blockRoot) {
-   require(proof.length == 20, "invalid proof length");
+   require(proof.length == 19, "invalid proof length");
    blockRoot = proof[0];
    bytes32 vectorRoot = MerkleTree.proofRoot(
        index,
        blockRoot,
        proof[1:14]
    );
    bytes32 computedRoot = MerkleTree.proofRoot(
        BLOCK_ROOTS_INDEX,
        vectorRoot,
        proof[14:]
    );
    require(computedRoot == stateRoot, "invalid relative proof");
}
```

Relic: Addressed in commit 1df44a7.

Auditor: Verified.

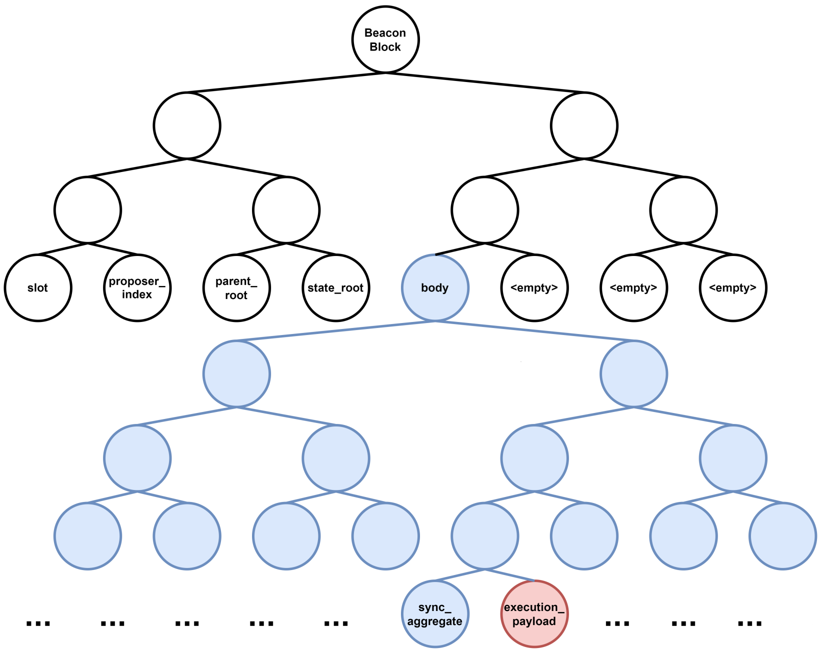
3.2 Low Severity Findings

3.2.1 Risks with merkleization structure changes

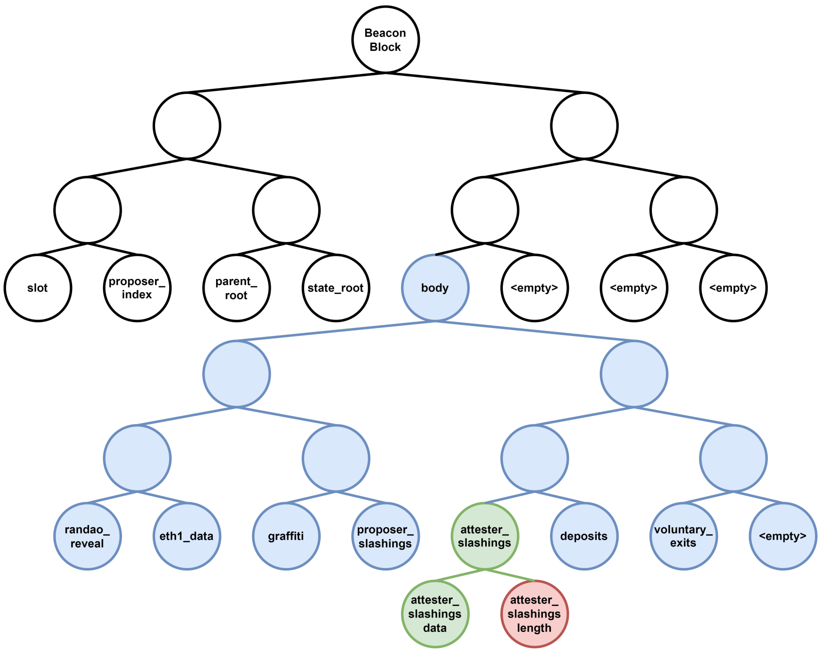
Description: As the Beacon Chain progresses through updates, some containers add new fields, and therefore some containers increase their tree depth during merkleization. For example, notice that the initial version of the `BeaconBlockBody` had 8 elements (merkleization depth of 3) while the newest version has 12 elements (merkleization depth of 4):

```
class BeaconBlockBody(Container):
    randao_reveal: BLSSignature
    eth1_data: Eth1Data  # Eth1 data vote
    graffiti: Bytes32  # Arbitrary data
    # Operations
    proposer_slashings: List[ProposerSlashing, MAX_PROPOSER_SLASHINGS]
    attester_slashings: List[AttesterSlashing, MAX_ATTESTER_SLASHINGS]
    attestations: List[Attestation, MAX_ATTESTATIONS]
    deposits: List[Deposit, MAX_DEPOSITS]
    voluntary_exits: List[SignedVoluntaryExit, MAX_VOLUNTARY_EXITS]
+   sync_aggregate: SyncAggregate
+   # Execution
+   execution_payload: ExecutionPayload  # [Modified in Deneb:EIP4844]
+   bls_to_execution_changes: List[SignedBLSToExecutionChange, MAX_BLS_TO_EXECUTION_CHANGES]
+   blob_kzg_commitments: List[KZGCommitment, MAX_BLOB_COMMITMENTS_PER_BLOCK]  # [New in
    Deneb:EIP4844]
```

Since the merklization depth can change, it's important to reflect on how this would affect the existing proving logic that relies on a hardcoded tree structure. Consider, for example, the `verifyExecutionPayloadFields()` logic, which proves an `execution_payload` value up to a known beacon block root. This logic would involve proving the below red node up to the root of the tree:



As mentioned above, earlier versions of the Beacon Chain would merkleize the `BeaconBlockBody` with a smaller depth. In particular, notice that in an older version, the `attester_slashings` field would occupy the same location as the current parent of the `execution_payload`:



Since the `attester_slashings` field is itself a root of a merklization tree, this field's right child could be attempted to be proven as an `execution_payload`. In this specific scenario, the right child of the `attester_slashings` list would be its length, which would fortunately be impossible to interpret as a valid `execution_payload`. However, if the scenario was slightly different (e.g. if the `graffiti` was the 5th element of the `BeaconBlockBody`) it might have been possible to prove values in unexpected ways.

Also, note that this potential problem does not only apply to previous block roots. If any containers were to increase in depth in a *future* hardfork, similar issues could arise. For example, if the `BeaconBlock` container increases its depth by 1, intermediate hashes might be interpreted as valid `slot` values.

Note: there is also the consideration of the `_verifyHeaderBlockRoot()` function, which may be affected by execution block header changes. Since this involves RLP encoding, and since `blockRoot != bytes32(0)` is checked in the code, this does not appear to be a problem.

Recommendation: To prevent any problems with *previous* merklization changes, add specific checks on the `slot` values of the beacon blocks. The main merklization proving functions have the following considerations:

- `_verifyBlockSummary()` - no change necessary. It's already implicitly checked that the relevant `slot` is past the Capella hardfork, which is sufficient.
- `_verifySummaryBlockRoot()` - no change necessary. This function uses `_verifyBlockSummary()` as a helper function, and this is sufficient.
- `_verifyRelativeBlockRoot()`- no change necessary. This function uses the `BeaconBlock` and `BeaconState` containers, which have had the same depths since the Beacon Chain started.
- `_verifyELBlockData()` - add a requirement that `slot >= CAPELLA_SLOT`. This ensures that an older version of the `BeaconBlockBody` can't be used. Technically, the Bellatrix hardfork could be the cutoff for this check, but using the `CAPELLA_SLOT` would be simpler as this variable already exists.

To help prevent problems with *future* merklization changes, consider adding sanity checks where possible. For example, a `slot` value should always be a `uint64`, so this can be explicitly checked in `verifyBlockSlot()` to potentially catch intermediate hashes that are not real `slot` values. However, note that this does not prevent all future problems, and it appears impossible to do so. Ideally, the Beacon Chain containers do not change their depths in any future versions. If they do, consider deprecating the current contracts, and deploy new versions with updated paths.

Relic: Added the `slot >= CAPELLA_SLOT` check in commit 1df44a7. Added the `slot <= type(uint64).max` partial mitigation for future merklization changes in commit f11e427.

Auditor: Verified.

3.2.2 Prevent calling `blockhash()` with the current block number

Description: In the `BeaconBlockHistory` contract, `blockhash()` is used in the following two functions:

```
function _validCurrentBlock(bytes32 hash, uint256 num) internal view returns (bool) {
    // the block hash must be accessible in the EVM and match
    return (block.number - num <= 256) && (blockhash(num) == hash);
}
```

```
function commitRecent(uint256 blockNum) external {
    require(block.number - blockNum <= 256, "target block must be recent");
    _storeCommittedBlock(blockNum, blockhash(blockNum));
}
```

In both of these functions, using a target block number of `block.number` will not cause any errors, even though the blockhash for `block.number` is not yet accessible in the EVM, and `blockhash(block.number)` will return `bytes32(0)`.

Fortunately, the upstream/downstream logic of both of these functions will catch any potential problems relating to this. However, for a more future-proof implementation, it would be beneficial to remove this potential footgun.

Recommendation: Consider creating a helper function to determine if a block number has an EVM accessible blockhash:

```
function _isBlockhashEVMAccessible(uint256 num) internal view returns (bool) {
    return num < block.number && block.number - num <= 256;
}
```

This can be used to improve the overall logic and give more informative error messages:

```
function _validCurrentBlock(bytes32 hash, uint256 num) internal view returns (bool) {
    // the block hash must be accessible in the EVM and match
    return _isBlockhashEVMAccessible(num) && (blockhash(num) == hash);
}
```

```
function commitRecent(uint256 blockNum) external {
    require(_isBlockhashEVMAccessible(blockNum), "target block not in EVM");
    _storeCommittedBlock(blockNum, blockhash(blockNum));
}
```

Relic: Addressed in commit f11e427.

Auditor: Verified.

3.3 Informational Findings

3.3.1 Duplicate computation

Description: The `verifyHistoricalBlockSummary()` function defines the `numImplicitNodes` variable as `HISTORICAL_BLOCK_SUMMARIES_PROOF_LENGTH - proof.length`. Later in the code, the same computation is done:

```
bytes32 listDataRoot = MerkleTree.rootWithDefault(
    HISTORICAL_BLOCK_SUMMARIES_PROOF_LENGTH-proof.length,
    intermediate,
    defaultValue
);
```

Recommendation: Consider reusing the `numImplicitNodes` variable:

```
bytes32 listDataRoot = MerkleTree.rootWithDefault(
-   HISTORICAL_BLOCK_SUMMARIES_PROOF_LENGTH-proof.length,
+   numImplicitNodes,
    intermediate,
    defaultValue
);
```

Relic: Addressed in commit 3afed0a.

Auditor: Verified.

3.3.2 Unused constant

Description: The `ProxyBeaconBlockHistory` contract defines the `SLOTS_PER_HISTORICAL_ROOT` constant, but does not use it anywhere in the code.

Recommendation: Consider removing this constant.

Relic: Addressed in commit 3afed0a.

Auditor: Verified.

3.3.3 Typos

Description: The following typos have been identified in the code:

1. The word "`precomitted`" is used in the `BeaconBlockHistory` contract. A more correct spelling would be "`precommitted`".
2. A comment in `verifyExecutionPayloadFields()` says "`... otherwise, we have need one extra proof node`". The word "`have`" can be removed from this comment.
3. The `verifySummaryIndex()` function conducts a merkle proof against its `stateRoot` argument. However, this argument actually represents the root of a `HistoricalSummary`, so "`summaryRoot`" would be a more relevant variable name.

Recommendation: Consider correcting these typos.

Relic: The first typo has already been cemented in the `PrecomittedBlock(uint256,bytes32)` event signature, so we will keep as is. The second typo has been addressed in commit 3afed0a. The third typo has been addressed in commit f11e427.

Auditor: Verified.