EIP 4788 Beacon Root

Smart Contract Security Assessment

Sept 1, 2023

Review of fixes:
Sept 29, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the bytecode of a smart contract that was introduced to the EIP-4877 in a recent change, enabling the on-chain storing and accessing of the beacon block roots of recent blocks.

SETTING & CAVEATS

The scope of the audit includes the etk implementation of the smart contract described by EIP-4877 (considering the EIP definition at commit hash 5af1bddda4e898808fb38d67661dcd2204c7da55 of the ethereum/EIPs repository).

The audit report covers commit hash `38c114bcd96817989496d6e902c1c5a8679a2eef` of the public repository lightclient/4788asm. The deployed smart contract is defined in src/main.etk and its initcode in src/ctor.etk.

The test suite was consulted during the audit but was not part of it.

We consider the timestamp of future blocks to be produced via the following function, as defined in the ethereum specification:

```
def compute_timestamp_at_slot(state: BeaconState, slot: Slot) -> uint64:
      slots_since_genesis = slot - GENESIS_SLOT
      return uint64(state.genesis_time +
                      slots_since_genesis * SECONDS_PER_SLOT)
```

For the ethereum mainnet `GENESIS_SLOT` is `0`, `state.genesis_time` is `1606824023`, and `SECONDS_PER_SLOT` (i.e. block time) is 12.

We also considered how a future fork changing this functionality could affect the contract’s functionality and have noted our findings in two advisory items. For this exercise we used the reduction of the block time as an example, other changes can have similar effects.

PROJECT OVERVIEW

The audited contract uses the block's timestamp as a key for their parent beacon blocks' roots. In order to bind the contract's storage footprint, while retaining accurate information, a set of two ring buffers are used (using a `HISTORY_BUFFER_LENGTH` with a value of 98304):

1. The first one stores the timestamp (i.e. the key) and is used to ensure that the result for the provided timestamp is the one that is currently stored on-chain. Its value will be stored at storage location `timestamp % HISTORY_BUFFER_LENGTH`.
2. The second one is used to store the beacon root chain value for the timestamp. Its value will be stored at storage location `HISTORY_BUFFER_LENGTH + timestamp % HISTORY_BUFFER_LENGTH`.

The audited contract implements two methods, `set()` and `get()`. As the contract does not adhere to the contract ABI specification, the method to be executed is chosen based on the contract's caller; if called by the special address `0xfffffffffffffffffffffffffffffffffffffffe`, the `set()` function is called, while every other address the calls the `get()` function. Both methods accept the first 32 bytes of the call's calldata as their arguments.

According to the EIP-4788 spec the desired functionality of the two methods is presented below.

`get()`:

- Callers provide the timestamp they are querying encoded as 32 bytes in big-endian format.

- If the input is not exactly 32 bytes, the contract must revert.
- Given `timestamp`, the contract computes the storage index in which the timestamp is stored by computing the modulo `timestamp % HISTORY_BUFFER_LENGTH` and reads the value.
- If the timestamp does not match, the contract must revert.
- Finally, the beacon root associated with the timestamp is returned to the user. It is stored at `timestamp % HISTORY_BUFFER_LENGTH + HISTORY_BUFFER_LENGTH`.

`set()`:

- Caller provides the parent beacon block root as calldata to the contract.
- Set the storage value at `header.timestamp % HISTORY_BUFFER_LENGTH` to be `header.timestamp`
- Set the storage value at `header.timestamp % HISTORY_BUFFER_LENGTH + HISTORY_BUFFER_LENGTH` to be `calldata[0:32]`

AUDIT METHODOLOGY

Two auditors worked on the codebase for 4 days.

Compiling the contract's .etk source, results in the production of an 88 byte EVM bytecode contract.

Because the audited contract is in a bytecode representation we used our tooling to inspect the contract in two more high-level representations (which are available to the public here):

1. The three-address-code (TAC) representation of the gigahorse binary lifter, preserving the original low-level instructions while resolving the control flow graph and the instructions to the variables they define and use.
2. The source-like decompiled representation of the tool powering our contract library blockchain explorer.

The latter managed to condense the program into 11 lines:

```
function __function_selector__(uint256 calldata0_32) public payable {
   if (0xfffffffffffffffffffffffffffffffffffffffe == msg.sender) {
      // set(...)
      STORAGE[block.timestamp % 0x18000] = block.timestamp;
      STORAGE[0x18000 + block.timestamp % 0x18000] = calldata0_32;
      exit;
   } else {
      // get(...)
      require(msg.data.length == 32);
      require(calldata0_32 == STORAGE[calldata0_32 % 0x18000]);
      return STORAGE[0x18000 + calldata0_32 % 0x18000];
   }
}
```

As per the EIP-4788 Contract Audit Request the audit's scope is defined as:

> *The audit should focus **exclusively** on the smart contract bytecode used to store Beacon Roots, referenced in the pull request above. It **should not** encompass all of EIP-4788 (for example, how the beacon roots are passed from the EL to the CL), or client implementations of the EIP, including their interactions with the contract.*

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", and functional correctness.

REVIEW OF POST-AUDIT FIXES

For the post-fix review of the codebase we considered the code at commit hash `bea9744af95953963552057c8a7d2124ec1bd33d` for the public repository lightclient/4788asm, considering the EIP definition at commit hash `949901b52b8be8b57673612de3f83bd9cd0d7924`.

The changes are included in PRs #16, #17 and #19.
Their main changes are:

- The `get()` function now checks against the zero timestamp value. This change resolves item L1.
- The `HISTORY_BUFFER_LENGTH` was changed to a value of 8191. 8191 is a prime number, ensuring that, given a constant block time, the ring buffers will contain the parent beacon blocks' roots of the past 8191 blocks. As a result the contract has a constant storage footprint, resolving issues A1 and A2.
- In addition cosmetic code changes and a minor optimization were included in the changes.

The disassembled, three-address-code, and decompiled representations of the updated contract can be found here.
The decompiled representation condenses the program into 12 lines:

```
function __function_selector__(uint256 calldata0_32) public payable {
   if (0xfffffffffffffffffffffffffffffffffffffffe == msg.sender) {
      // set(...)
      STORAGE[block.timestamp % 8191] = block.timestamp;
      STORAGE[8191 + block.timestamp % 8191] = calldata0_32;
      exit;
   } else {
      // get(...)
      require(msg.data.length == 32);
      require(calldata0_32);
      require(calldata0_32 == STORAGE[calldata0_32 % 8191]);
      return STORAGE[8191 + calldata0_32 % 8191];
   }
}
```

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[NO CRITICAL SEVERITY ISSUES]

HIGH SEVERITY:

[NO HIGH SEVERITY ISSUES]

MEDIUM SEVERITY:

[NO MEDIUM SEVERITY ISSUES]

LOW SEVERITY:

ID	Description	STATUS
L1	`get()` call succeeds for invalid value	**RESOLVED**

Calling the `get()` function with a value of zero will not fail and return the zero value back.

```
// get()
require(msg.data.length == 32);
require(calldata0_32 == STORAGE[calldata0_32 % 0x18000]);
return STORAGE[0x18000 + calldata0_32 % 0x18000];
```

Although this does not affect the contract's functionality for valid timestamps it can potentially lead to misuse. Therefore we suggest adding a special case for the zero value, in the `get()` function or invalidating it by storing a value in the `0th` storage slot during the contract's construction.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Contract's storage footprint can increase for lower block time values	**RESOLVED**

With the current block time interval of 12 seconds each of the two ring buffers will take up 8192 storage slots, for a total of 16384. If the block time is to be reduced in the future, the number of storage slots taken up by the contract can increase by over 10x. While this does not have any security implications we believe it should be acknowledged.

The number of storage slots used by the contract for different values of block time are presented in the following table:

Block Time	Number of storage slots used
1	196608
2	98304
3	65536
4	49152
5	196608
6	32768
7	196608
8	24576

9	65536
10	98304
11	196608
12	16384

A2	Future change in Ethereum’s block time can result in stale ring buffer entries	**RESOLVED**

As described in the previous item, a possible future reduction of Ethereum’s block time interval will affect the storage slots used by the contract’s two ring buffers.

We have found that if the block time was to change to a value of 8 (down from the current 12), 4096 of the block roots stored in the contract’s storage would not be overwritten. This creates a scenario where an outdated beacon block root may be accessed, posing risks not anticipated by developers working with this contract.

For the other values of block timestamp (1 to 11, but not 8), all previously written storage slots would be overwritten.

A3	Future change in Ethereum’s block time can reduce the available history at fork time	**INFO**

This item was added after the post-audit fixes were introduced, changing the `HISTORY_BUFFER_LENGTH` parameter to 8191.
The use of a prime number ensures that the ring buffer will include the past 8191 blocks. Due to this, a possible future reduction of Ethereum’s block time interval will have 2 side effects:

- The contract’s history length decreases for lower block time values.

- The change of the block time interval will also change the sequence in which the ring buffer entries are updated. As a result, after a fork reducing the block time, some beacon root entries will be overwritten much faster than the history length. For instance, for a change to a block time of 5, although the period of block updating will be over 11 hours, some block's entry will be overwritten in just 2hr 16min 31sec. This is the minimum such quantity and is shown in the last column of the table below ("Minimum entry lifetime post-fork").

Block Time	History length	Minimum entry lifetime post-fork
1	2:16:31	2:16:31
2	4:33:02	4:33:02
3	6:49:33	6:49:33
4	9:06:04	9:06:04
5	11:22:35	2:16:31
6	13:39:06	13:39:06
7	15:55:37	2:16:31
8	18:12:08	9:06:04
9	20:28:39	6:49:33
10	22:45:10	4:33:02
11	25:01:41	2:16:31

DISCLAIMER

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.