PUBLIC

Code Assessment

of the EIP-4788 Contract

Smart Contracts

September 28, 2023

Produced for

ethereum foundation

by

CHAINSECURITY

Contents

1 Executive Summary

To the Ethereum Foundation,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our EIP-4788 Contract audit according to the Scope to support you in forming an opinion on the contract's security and correctness.

EIP-4788 introduces a mechanism for the execution layer of Ethereum mainnet to access the beacon roots of the consensus layer. This access is provided through a regular smart contract which acts as a temporary database. This particular smart contract is the scope of this audit.

The most critical subjects covered in our audit are the security and the correctness of the smart contract storing the beacon roots. Herein, the most important security property is that:

- Only the privileged `SYSTEM_ADDRESS` can store beacon roots

Among other properties the correctness properties include:

- Only previously stored beacon roots can be retrieved
- The ring buffer correctly overwrites old beacon roots

Our most significant finding here is that the Zero-Timestamp can be queried successfully, which has been corrected in the code.

In another finding, we suggested to make the ring buffer size a prime number which provides different benefits as described in Implications of Ring Buffer Size. This has been adopted, as in Version 2 the ring buffer size became a prime number.

Furthermore, we investigated possible gas savings and made some recommendations which focused on reducing the execution cost of the contract's usual execution path.

Additionally, in the Informational Section we highlighted how changes in the specification could allow a more efficient contract implementation. Lastly, we wrote Notes for smart contract developers, planning to interact with this contract, so that they can avoid mistakes.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but do not replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	6
• Code Corrected	1
• Acknowledged	4
• No Response	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the following source code files inside the EIP-4788 Contract's code repository:

- *src/main.etk*
- *src/ctor.etk*

The tables below indicates the commits relevant to this report and when they were received. The EIP-4788 was used as a source of documentation and specification.

Contract Code

V	Date	Commit Hash	Note
1	August 29 2023	38c114bcd96817989496d6e902c1c5a8679a2eef	Initial Version
2	September 26 2023	bea9744af95953963552057c8a7d2124ec1bd33d	Version 2

EIP-4788

V	Date	Commit Hash	Note
1	August 29 2023	46f8d5bc9593f096cd92f66692e115218ec1c633	Initial Version
2	September 26 2023	761df83432837603da12879885c8b19381fa0c1d	Version 2

For assembly, `etk-ask` assembler version `0.3.0` was used.

2.1.1 Excluded from scope

This review does not include all of EIP-4788. In particular, it does not include how the beacon roots are passed from Consensus Layer to the Execution Layer or correctness of client implementations of the EIP. Furthermore, it does not include the interactions of clients with the contract.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview. Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

EIP-4788 adds the possibility to retrieve beacon block roots inside the EVM. As explained in the Motivation, this allows decentralized applications based on consensus-level data with reduced trust assumptions. This functionality is provided through a beacon roots contract. The beacon roots contract is a regular EVM contract. Logically, the beacon roots contract acts as a temporary database where block producers can store beacon roots and anyone can retrieve them. That database allows beacon root lookup based on a provided timestamp.

Concretely, the storage layout of the contract is as follows. The contract has two ring buffers, one for timestamps and one for beacon roots. Those buffers are aligned so that element `i` from the first buffer corresponds to element `i` from the second buffer:

1. Timestamps: Storage slots `0 - HISTORY_BUFFER_LENGTH`
2. Beacon Roots: Storage slots `HISTORY_BUFFER_LENGTH` - `HISTORY_BUFFER_LENGTH * 2`

The functionality of the beacon roots contract is as follows:

1. At the beginning of each block, the block producer calls the beacon roots contract using a special `SYSTEM_ADDRESS`. When called with the `SYSTEM_ADDRESS` the contract will perform a `set` operation. The `set` operation has two effects:
    1. `sstore(timestamp % HISTORY_BUFFER_LENGTH, timestamp)`, setting the current timestamp in the timestamp ring buffer
    2. `sstore(timestamp % HISTORY_BUFFER_LENGTH + HISTORY_BUFFER_LENGTH, calldata[0:32])`, setting the provided beacon root in the beacon root ring buffer
2. During regular EVM execution the beacon roots contract can receive calls by any other address. Note that these calls are implicitly `get` calls, so that no calldata signature has to be provided. Instead only the desired timestamp is provided as calldata. As part of a `get` call the following checks are made:
    1. `calldatasize == 32`
    2. `calldataload(0) != 0`
    3. `sload(calldataload(0) % HISTORY_BUFFER_LENGTH) == calldataload(0)`, this checks that the provided timestamp needs to match the timestamp stored in the timestamp ring buffer, indicating that the desired beacon root is available.

    If all of these checks pass, the stored beacon root is returned, otherwise the contract reverts without revert reason.

For a more precise description of the functionality, see the Pseudocode.

As ring buffers are used the beacon root for a particular timestamp is generally only available for a limited time. Furthermore, note that not all slot timestamps might return beacon roots as slots could be missed. For more information on possible pitfalls see Integration Guidelines for Developers.

2.2.1 Trust Model

We make the following trust assumptions:

- A valid signature for the `SYSTEM_ADDRESS` is never found.
- Nobody can successfully deploy a contract to the `SYSTEM_ADDRESS`.
- The consensus checks that block producers correctly insert the beacon roots. In particular, they are not allowed to overwrite old beacon roots.
- No future Ethereum hard fork will allow two blocks to have the same timestamp. That would lead to undefined behavior for this contract.
- No future Ethereum hard fork will allow a subsequent block to have a smaller timestamp.
- If the block interval changes or becomes variable, some of the assumptions will no longer hold. See Changes in Block Interval for more details.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5

5.1 Push3 Used Where Push2 Would Be Sufficient

Design Low Version 2

CS-EIP4788-010

The ring buffer size (8191 in Version 2), is pushed to the stack using PUSH3, however, as it only occupies two bytes, a PUSH2 would also be sufficient. This would result in a minor size reduction of the bytecode.

5.2 Merge Failure Cases for Gas Savings

Design Low Version 1 Acknowledged

CS-EIP4788-002

Generally, we expect the most common `get` execution case to be the one, where the `get` call will succeed. Hence, we try to optimize the gas cost for this case. In the current implementation, the successful `get` contains two executions of `JUMPI` that branch off to the corresponding revert statements. If the two conditions were combined using an `OR` operation, only a single `JUMPI` would be needed. Thereby execution gas costs could be lowered in the successful `get` execution.

5.3 Negate Failure Conditions to Save Gas

Design Low Version 1 Acknowledged

CS-EIP4788-003

The contract has two conditions that can revert the execution:

1. `calldatasize == 32`
2. `sload(calldataload(0) % HISTORY_BUFFER_LENGTH) == calldataload(0)`

Usually, we expect the good case where both conditions will evaluate to true to be the most common one. However, on EVM-level the conditions are currently implemented so that in the good case the `JUMPI` performs a jump. As a result, in the good case two `JUMPDEST` operations are executed, which consume one gas each. If the conditions would be negated on EVM level by using `SUB` instead of `EQ`, the gas consumed by the `JUMPDEST` operations could be saved.

5.4 Reorder Operations to Save Gas

Design | Low | Version 1 | Acknowledged

CS-EIP4788-004

The contract's source code contains a `swap1` operation, which swaps out elements of the stack. Through a different order for the preceding operations, the `swap1` instruction can be omitted. This results in a lower execution gas cost and smaller contract size.

5.5 Replace Push Command With Msize for Gas Savings

Design | Low | Version 1 | Acknowledged

CS-EIP4788-005

Towards the end of the `get()` function there is the following instruction:

```
push1 32
```

It is supposed to push the current memory size (32 bytes) onto the stack, so that this can be used as size input for the return statement. Instead the `msize` instruction could be used, which achieves the same and reduces execution gas costs as well as contract deployment costs.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Zero-Timestamp Can Be Queried Successfully Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	1

- Implications of Ring Buffer Size Code Corrected Specification Changed

Informational Findings	1

- Inconsistent Comment Code Corrected

6.1 Zero-Timestamp Can Be Queried Successfully

Correctness High Version 1 Code Corrected

CS-EIP4788-001

The `get()` function can be queried with the Zero-Timestamp even though no value has ever been set for the Zero-Timestamp. This violates the important property that all returned values must have previously been set. This happens because the EVM storage is initialized with zeroes which allow the timestamp check to pass.

Note that this remains possible until the corresponding storage slot is first used.

With 0 as an argument for `get()`, The returned beacon root would be zero. This leads to integrators being tricked into accepting the Zero-Hash as a valid beacon root which might allow exploits depending on the protocol.

Code corrected:

An explicit check has been added to make sure that the `get()` function reverts when a Zero-Timestamp is provided as calldata.

6.2 Implications of Ring Buffer Size

Design Low Version 1 Code Corrected Specification Changed

CS-EIP4788-008

The EIP-4788 states:

> The ring buffer data structures are sized to hold 8192 roots from the consensus layer at current slot timings.

In Version 1 the code implements the circular buffer, however out of 98304 slots, only 8192 will be utilized at a current `SECONDS_PER_SLOT = 12` on the mainnet.

Effectively the ring buffer behaves as a ring of integers modulo `n`, where `n` is its size. The `( current_timestamp + X * SECONDS_PER_SLOT ) mod 98304` function will produce a cyclic subgroup of order 8192 if `SECONDS_PER_SLOT` is 12. However, if, in the future, the `SECONDS_PER_SLOT` would change to 16, the cyclic subgroup will have order 6144, which is less than 8192. Furthermore, many old entries from the 12-second interval would uselessly remain in the ring buffer.

Thus, the requirement of the EIP-4788 to have 8192 roots available in the ring buffer will not be satisfied if the `SECONDS_PER_SLOT` changes to 16 seconds. If the `SECONDS_PER_SLOT` changes to 13 seconds, the cyclic subgroup will have order 98304, thus increasing the storage requirements for the ring buffer by 12 times.

To summarize, the 98304 as a group order for the ring buffer is not an ideal choice, as it is not a prime number. Potential changes to the `SECONDS_PER_SLOT` will drastically change the behavior of the ring buffer.

If the `( current_timestamp + X * SECONDS_PER_SLOT ) mod 8209` function is used instead, the cyclic subgroup will always have order 8209, since it is a prime number. That would have two key advantages:

- The ring buffer could always hold the most recent 8209 beacon roots independent of `SECONDS_PER_SLOT`
- The storage consumption would remain constant even when `SECONDS_PER_SLOT` changes

If the primary objective is to make sure that the ring buffer can hold all beacon roots of the past 24 hours, then a prime ring buffer size still makes sense, but a bigger one has to be chosen, according to the lowest value `SECONDS_PER_SLOT` might have in the future.

Please note that the changes discussed here would require a change in the specification.

Code corrected:

The specification has been changed to make the ring buffer size 8191, which is a prime number. The code has been changed accordingly. Hence, the new implementation benefits from the positive effects described above.

6.3 Inconsistent Comment

Informational Version 1 Code Corrected

CS-EIP4788-009

In the code comments inside `src/main.etk` for the `get()` function, the same loaded calldata is once referred to as `time` and once as `input`. To avoid confusion a consistent label could be used for it in both places.

Code corrected:

The comments have been updated and are more consistent.

7 Informational

We utilize this section to point out informational findings that are technically not issues. As the EIP served as a specification, we primarily check whether the code correctly and securely implements the EIP. Here, however, we also point out possible improvements to the EIP. Furthermore, an inconsistency in the comments of the source code.

7.1 Changes in Block Interval

Informational **Version 2**

CS-EIP4788-011

As long as the block interval (*SECONDS_PER_SLOT*) remains at its current value of 12 seconds two properties will hold:

1. A successfully written beacon root can only be overwritten after 8191 blocks have passed.
2. All successfully written beacon roots from past 24 hours are available in the contract.

However, different changes in the block interval are possible:

Different, Fixed Block Interval

If, at a block *X*, the block interval changes to a different value, e.g., 8 seconds, the following holds regarding the properties mentioned above:

1. This property is temporarily violated. A beacon root written in the blocks [*X - 8190*, *X*] might be overwritten sooner due to the change in the interval.
2. This property is temporarily violated for the beacon roots written in the 24 hours before *X*. Furthermore, if the new block interval is smaller than 11 seconds, the property will no longer hold as more than 8191 beacon roots are produced in 24 hours.

Variable Block Interval

If the block interval becomes variable, e.g. there are 10 seconds between blocks *X* and *X+1*, but 9 seconds between blocks *X+1* and *X+2*, then the following holds regarding the properties mentioned above:

1. This property is permanently violated.
2. This property is permanently violated.

However, in all cases mentioned above, the following property always holds:

A successfully written beacon root can only be overwritten after 8191 **seconds** have passed. Hence, even during network changes, each beacon root will be available for more than two hours.

7.2 Gas Savings in Ring Buffer Layout

Informational **Version 1**

CS-EIP4788-007

Currently, the two ring buffer storage location segments are laid out as follows:

1. Timestamps: [0 , `HISTORY_BUFFER_LENGTH` - 1]
2. Beacon Roots: [`HISTORY_BUFFER_LENGTH` , 2 * `HISTORY_BUFFER_LENGTH` - 1]

Hence, to compute the beacon root storage slot based on the timestamp storage slot, the code currently adds `HISTORY_BUFFER_LENGTH`. However, a more efficient approach would be to use the EVM's `NOT`

opcode on timestamp slot for the beacon root storage slot computation. By doing this, both the execution gas cost and the overall contract size could be reduced.

Please note that this would require a change in the specification.

7.3 Minor Inconsistencies in Specification

Informational **Version 2**

CS-EIP4788-012

A few documentation parts are outdated in **Version 2**:

- The subsection `Size of ring buffers` in the EIP-4788 contains an outdated ring buffer size.
- The bytecode in the `README.md` of the code repository is incorrect based on the command presented above it.
- The `cfg.png` showing the control-flow graph in the code repository is outdated as it does not contain the latest code.

8 Notes

We leverage this section to highlight issues that could potentially arise when interacting with this contract. First from the perspective of smart contract developers and then from the perspective of execution clients.

8.1 Integration Guidelines for Developers

Note Version 1

Smart contract developers who aim to interact with the beacon roots contract should be aware of the following pitfalls:

1. Unless most smart contracts, this contract has **no 4-byte abi calldata signature** to select the function. As you are not calling from the `SYSTEM_ADDRESS` your call is automatically calling the `get` function.
2. The calldata has to be **exactly 32 bytes** and should only contain the timestamp, in big-endian format, which is the EVM default.
3. Due to the points above, smart contract developers might perform a low-level call in the respective smart contract language. That low-level call generally does not perform checks, such as **sufficient return data**. Hence, smart contracts performing such a low-level call should check that the `RETURNDATASIZE == 32` using the features of the respective language.
4. Beacon roots are only available for a **limited time**. This is because a ring buffer is used where old values are overwritten. In particular this also implies that a timestamp which was queried successfully in one block might not be available in the next block.
5. The ring buffer might contain **outdated entries**. At the time of writing the `HISTORY_BUFFER_LENGTH` will be chosen so that roughly one day of beacon roots is available. However, developers *cannot* assume that successfully queried beacon roots are from the past day. They might be as old as the `FORK_TIMESTAMP`.
6. There is a beacon root available for the **current timestamp**, but it is **not** the beacon root of the current block. It is the beacon root of the preceding block.
7. There might be no beacon root for **current timestamp - 12** or more generally: there might be no beacon root for the timestamp of a particular slot that occurred recently. This is because slots can be missed. Then no blocks will be produced and no beacon root will be inserted.
8. To `get` the beacon root from time `X` do **not query** `X`. Instead the timestamp of the succeeding block must be used. That timestamp is generally unknown it could be `X+12`, `X+24`, or something completely different once the block interval changes.
9. **Do not send ETH** to the contract. The ETH will be lost. A `STATICCALL` can be used, as it does not allow the transfer of ETH.

Please note that we have only covered pitfalls related to the `get` function as this is the only one smart contract developers should be able to call.

8.2 Note for Execution Clients

Note Version 1

Execution clients need to perform the `set` operation as part of block construction. They should be aware that:

`set` **never reverts**. Hence, a non-reverting execution of the `set` function does not imply a correct call. In particular `set` would not revert if called with no calldata or with too much calldata. If less than 32 bytes of calldata are provided, the `calldataload` operation will pad the missing bytes with zeros.