Snowfork

Beefy client audit

Snowfork

Nov 17, 2023

Common ␣ Prefix

Overview

Introduction

Common Prefix was commissioned to perform a security audit on Snowfork's Beefy Client smart contracts, at commit hash 08c5817009931f7ed9c21f2be0b8eed6d4b3a3d8.

This is the second audit we have conducted on the Beefy Client. The report for the first audit can be found here. Key modifications in this updated version of the contracts include::

- A more sophisticated formula for the computation of the minimum number of required signatures, which takes into account not only the total number of validators but also off-chain data related to the validators and the number of times a validator's signature has been previously used during the same session. Details can be found here.
- A more detailed parsing of the payload of the commitment, to avoid possible ambiguities.
- Introduction of the `Verification.sol` library, facilitating commitment verification within the Beefy Client.

Except for the liveness issue (Medium-4), all the other issues detected during the first audit have been resolved in this new version.

The files inspected are the following:

`BeefyClient.sol`

`Verification.sol`

`Math.sol`

`Uint16Array.sol`

Disclaimer

Note that this audit does not give any warranties on the bug-free status of the given smart contracts, i.e. the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. Functional correctness should not rely on human inspection but be verified through thorough testing. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

Findings Severity Breakdown

The findings are classified under the following severity categories according to the impact and the likelihood of an attack.

Level	Description
Critical	Logical errors or implementation bugs that are easily exploited and may lead to any kind of loss of funds
High	Logical errors or implementation bugs that are likely to be exploited and may have disadvantageous economic impact or contract failure
Medium	Issues that may break the intended contract logic or lead to DoS attacks
Low	Issues harder to exploit (exploitable with low probability), issues that lead to poor contract performance, clumsy logic or seriously error-prone implementation
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable and easier to maintain

Findings

Critical

No critical issues found.

High

No high issues found.

Medium

No medium issues found.

Low

LOW-1	Inconsistency between implementation and documentation regarding the minimum number of signatures
Contract(s)	`BeefyClient.sol`
Status	**Resolved**

Description

In the last step of the verification algorithm of the Beefy client, N validators are randomly chosen and the light client verifies that they have signed the commitment. The formula for N is the following:

$$N = \lceil log_2 \left(RV\frac{1}{S}(75+E)172.8 \right) \rceil + 1 + 2\lceil log_2(C) \rceil$$

$$= \lceil minimumNumRequiredSignatures + log_2(V) \rceil + 1 + 2\lceil log_2(C) \rceil$$

where R,V,S are terms available only off-chain and used to compute the minimum number of required signatures, a value passed to the contract during its construction, V is the total number of validators in the current session and C is the number a validator has been used within this section. The docs provide more details and motivation for this formula.

In the code, however, in the BeefyClient::computeNumOfRequiredSignatures function, number N is computed as:

$$minimumNumOfRequiredSignatures + \lceil log_2(V - minimumNumOfRequiredSignatures) \rceil + 1 + \lceil log_2(C) \rceil$$

which could be a number less than the N of the documentation.

Recommendation

We suggest fixing the `BeefyClient::computeNumOfRequiredSignatures` function to align with the formula of the documentation. This adjustment ensures that the analysis provided in the documentation accurately corresponds to the implemented logic within the BeefyClient.

Alleviation

The team fixed the formula at commit hash e4c913521f50f6350100399f18e0cae529ad067a.

LOW-2	The functions of the `Uint16Array` library can access and modify out of (the real) bounds positions
Contract(s)	`Uint16Array.sol`
Status	**Resolved**

Description

`Uint16Array` is a utility library designed for the efficient storage of a `uint16` array in an array of type `uint256`. Each `uint256` entry holds 16 `uint16 values`. If the length l of the `uint16` array is not a multiple of 16, the resulting `uint256` array generated by the `create` function can hold more `uint16` entries than initially anticipated. There is no way to determine l given the `uint256` array. In instances where the `get(self, index)` function returns zero, it becomes impossible to differentiate whether the corresponding `uint16` value is genuinely zero or if the index is out of bounds, i.e. greater than or equal to l.

Recommendation

While our analysis did not reveal any immediate issues with the Beefy Client associated with the described problem, we express concern about the error-prone design of `Uint16Array`. We recommend fixing this potential issue by incorporating an additional variable, `uint156 lengthOfThe16BitArray`, into the `Array` structure. This modification enhances the clarity and reliability of the `Uint16Array` design, providing a more robust design.

Alleviation

The team addressed the issue at commit hash [e4c913521f50f6350100399f18e0cae529ad067a](e4c913521f50f6350100399f18e0cae529ad067a) by introducing an extra variable at the `Array` structure. They also validate that the index does not surpass the Array length in the `get` and `set` functions.

LOW-3	Missing input validation
Contract(s)	`BeefyClient.sol, Verification.sol`
Status	**Partially resolved**

Description

- In the constructor of BeefyClient a check that `nextValidatorSet.id == currentValidatorSet+1` is missing.
- In `BeefyClient::submitInitial` should be validated that `commitment.blockNumber > latestBeefyBlock`.
- In a Merkle tree with input vectors of not fixed length it is trivial to find collisions (one vector of length 2n and another of length n), therefore the standard practice is to hash not only the leaf but the length of the vector/width of the tree as well. Additionally, it is advisable to verify if the proof has the proper length i.e. log2(width). These standard practices are not followed in the Beefy Client. Although the design is still secure, because of the way the input data are encoded, we suggest incorporating these extra measures.

  Specifically, in `BeefyClient::isValidatorInSet`, the hashed leaf should be the hash of the concatenation of `vset.length` and `account` –instead of the hash of the account only- or other extra identifiers should be hashed to distinguish nodes and leaves (check EIP-712) and also it should verified that `proof.length == log2(vset)`. Similar validations should take place in `BeefyClient::verifyMMRLeafProof` and `Verification::verifyCommitment`. Even better these validations could be added in the libraries `SubstrateMerkleProof::computeRoot` and `MMRProof::verifyLeafProof`.

Recommendation

We strongly recommend incorporating these sanity checks into the relevant functions to prevent potential mistakes.

Alleviation

The team fixed the first two issues at commit hash e4c913521f50f6350100399f18e0cae529ad067a. They decided against incorporating extra validations on the Merkle proofs, as doing so would increase the complexity of the code. Furthermore, even the existing design is protected against the possible issues we mentioned through the encoding of the data which will be used as leaves in the Merkle trees.

Informational/Suggestions

INFO-1	`commitPrevRandao` could be callable by anyone
Contract(s)	`BeefyClient.sol`
Status	**Info**

Description

The second step a relayer should take to submit a commitment is to call the `commitPrevRandao` function with the commitment hash as an argument. This function uses the provided commitment hash and the address of the caller to compute the index of the ticket, therefore the randomness for a ticket can only be produced by the relayer who constructed it. This fact gives the relayer some flexibility to manipulate the randomness.

While the protocol mitigates this risk by increasing the minimum number of required proofs in a specific way described here , we believe it could be a good extra measure to allow anyone to call `commitPrevRdanao` and produce the randomness for a ticket. That way other relayers can frontrun a malicious one, calling `commitPrevRandao` and producing randomness for his ticket.

This prevents the malicious relayer from iteratively executing steps 1 and 2 until obtaining favorable randomness.

Recommendation

We suggest making the prevRandao function callable with the ticket index and not the commitment hash as its argument. That way anyone can produce the randomness for a ticket and not only its creator.

INFO-2	Manually passing the Beefy Client address is error-prone
Contract(s)	`Verification.sol`
Status	**Info**

Description

The `Verification::verifyCommitment` function verifies a commitment using the latest stored root in the Beefy Client. But the address of the Beefy Client should be passed to this function manually, as one of its arguments. This is error-prone e.g. an EOA trying to verify a commitment could be misled related to the Beefy Client address, a contract trying to verify a commitment should have the Beefy Client address stored and should have implemented a procedure to update it in case this address changes.

Recommendation

We recommend the implementation of a registry smart contract tasked with maintaining the Beefy Client address, allowing a trusted entity to update it. To achieve this, the address of the registry contract should be stored as a constant in the Verification contract. Subsequently, the Verification contract should call the registry to obtain the Beefy Client address whenever necessary. This approach enhances flexibility, security, and ease of maintenance in managing the Beefy Client address.

INFO-3	Redundancy
Contract(s)	`BeefyClient.sol`
Status	**Resolved**

Description

In `BeefyClient::submitInitial` the check

```
if (commitment.validatorSetID != vset.id) {
            revert InvalidCommitment();
        }
```

is redundant, since the `vset` by its definition will always have the same id as the `commitment`:

```
ValidatorSetState storage vset;
uint16 signatureUsageCount;
        if (commitment.validatorSetID == currentValidatorSet.id)
           . . .
            vset = currentValidatorSet;
        } else if (commitment.validatorSetID == nextValidatorSet.id) {
          . . .
            vset = nextValidatorSet;
        } else {
            revert InvalidCommitment();
        }
```

Alleviation

The team addressed the issue at commit hash e4c913521f50f6350100399f18e0cae529ad067a removing the redundant if-statement.

INFO-4	Code optimization/simplification
Contract(s)	`Verification.sol`,`Math.sol`
Status	**Resolved**

Description

- In `Verification::encodeDigestItem` the first 3 if-else if branches could be merged. Also, although `encodeDigestItem` is expected to be called only by `verifyCommitment` (through `encodeDigestItems`), if other contracts use this library function for other purposes its `InvalidParachainHeader()` error message could be confusing, since the argument of the function is not a whole parachain header.
- `ScaleCodec::checkedEncodeCompactU32` expects a `uint256` argument, therefore the following type castings are unnecessary since `digestItems.length`, `header.number` and `encodedHeader.length` are of type `uint256`

Verification.sol

```
function encodeDigestItems(DigestItem[] calldata digestItems) internal pure returns (bytes
memory) {// encode all digest items into a buffer
        bytes memory accum = hex"";//PC: check this
        for (uint256 i = 0; i < digestItems.length; i++) {
            accum = bytes.concat(accum, encodeDigestItem(digestItems[i]));
        }
        // Encode number of digest items, followed by encoded digest items
        return bytes.concat(ScaleCodec.checkedEncodeCompactU32(uint32(digestItems.length)),
accum);
    }

function createParachainHeaderMerkleLeaf(bytes4 encodedParaID, ParachainHeader calldata
header)
internal
pure
returns (bytes32)
{
  . . .
  ScaleCodec.checkedEncodeCompactU32(uint32(header.number))
  . . .
}

function createParachainHeader(bytes4 encodedParaID, ParachainHeader calldata header)
internal
Pure
returns (bytes memory)
{
 . . .
 ScaleCodec.checkedEncodeCompactU32(uint32(encodedHeader.length))
 . . .
}
```

- The encoding of a parachain header in `Verification::createParacahinHeader` contains the number of digest items in the parachain header twice

```
function createParachainHeader(bytes4 encodedParaID, ParachainHeader calldata header)
internal
pure
returns (bytes memory)
{
        bytes memory encodedHeader = bytes.concat(
            // H256
            header.parentHash,
            // Compact unsigned int
            ScaleCodec.checkedEncodeCompactU32(header.number),
            // H256
            header.stateRoot,
            // H256
            header.extrinsicsRoot,
            // Vec<DigestItem>
            ScaleCodec.checkedEncodeCompactU32(header.digestItems.length),
            encodeDigestItems(header.digestItems)//CP: encodeDigestItems includes also the
header.digestItems.length in the encoding
        );

        . . .
}
```

- If the Snowfork team fixes the double encoding of the digest items length issue described above, then `Verifications::createParachainHeaderMerkleLeaf` could be simplified, since then it will be just the keccak256 of the output of `Verification::createParachainHeader`.
- In `Math::saturatingSub` the > operator in the if branch could be replaced by >= i.e. immediately return 0 when a and b are equal, instead of computing their difference, which will be 0.

Alleviation

The team incorporated in the contracts all the suggested improvements at commit hash [he4c913521f50f6350100399f18e0cae529ad067a](he4c913521f50f6350100399f18e0cae529ad067a).

INFO-5	Typos
Contract(s)	`Uint16Array.sol`
Status	**Resolved**

Description

- Comment describing the purpose of the library:

    - “Layout of 8 counters…”: 8 should be 32.
    - the computation of the bit-index of the counter with logical index 22 should be 96 (22=1*16+6 and 6*16=96) to 111.
- Comments in `get` and `set`: “Mask out the first 4 bytes…” bytes should be bits.

Alleviation

The team fixed the typos at commit hash [e4c913521f50f6350100399f18e0cae529ad067a](e4c913521f50f6350100399f18e0cae529ad067a).

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.