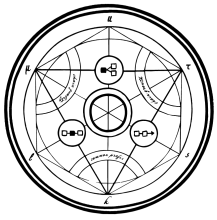
Snowfork

Beefy client audit

Snowfork

May 29, 2023

Common Prefix



Overview

Introduction

Common Prefix was commissioned to perform a security audit on Snowfork's Beefy Client smart contracts, at commit hash [54b62c92445635164d1414af742e26b56a097003]. The files inspected are the following:

`BeefyClient.sol`

`Bitfield.sol`

Description of the protocol

Snowfork is a general-purpose bridge between Polkadot and Ethereum. BEEFY is a light client which allows the bridge to prove on the Ethereum side that a specified parachain header (i.e. blocks on the Polkadot's side of the bridge) has been finalized by the relay chain. An untrusted and permissionless set of relayers regularly transmits commitments signed by relay chain validators. The light client has to verify the signatures before accepting the commitment. The light client achieves an efficient verification by only verifying the signatures for a really small, randomly chosen subset of validators. The commitment contains the number of the finalized block, a unique ID specifying the set of validators who signed the given commitment and a payload (the actual data to be verified). The payload contains an MMR root hash which commits to the Polkadot history.

Each relayer has to sequentially execute the following steps.

1. Submit the hash of the commitment and a bitfield specifying the validators that, according to the relayer's claim, signed the commitment.
2. After a sufficient number of blocks, to ensure that the randomness from RANDAO is not easily manipulated, the relayer has to commit to a `block.prevrandao`, which will play the role of the seed for future random sampling. (Of course, it cannot be guaranteed that the output of the RANDAO is not biased. In this audit we do not investigate the possibility of bad/biased randomness which, of course, would be catastrophic for the protocol).
3. Submits the whole commitment and the contract subsamples the validator set using the seed of the previous step, and verifies the signatures of this subset.

Since the set of validators of the relay chain regularly changes, the relayers of the Beefy contract should also submit the data about the new commit accompanied by proofs of their validity.

In the current codebase version, there is a critical issue (the Merkle tree indices are not verified). The issue was already known to the team, before the audit, and there is also a comment in the `BeefyClient.sol` contract[1] describing the issue. The team has a concrete plan for fixing the issue (will use an older Merkle proof verifier, that indeed did verify the indices).

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

[1] https://github.com/Snowfork/snowbridge/blob/54b62c92445635164d1414af742e26b56a097003/core/packages/contracts/src/BeefyClient.sol#L488

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.[2]

High

No high issues found.

Medium

MEDIUM-1	A relayer can call `submitInitialWithHandover` instead of `submitInitial` to submit a commitment signed by the current validator set to get a lower required number of signatures
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

Any commitment should be signed by at least ⅔ of the validator set. This condition is verified only in the initial submission (`submitInitial` or `submitInitialWithHandover`). Note that at the first step, nothing prevents the relayer from calling `submitInitial` even if his commitment is signed by the next validator set (or `submitInitialWithHandover` if his commitment is signed by the current set).

Suppose that the relayer has a commitment signed by n validators of the current set (let's call the number of validators in the current set m1) and that the number of validators in the next set (let's call it m2) is less than m1. If ⅔*m2 < n < ⅔*m1 then this commitment is not valid. But if the relayer calls `submitInitialWithHandover`, instead of the `submitInitial` which he should

[2] Except for the issue described in the introduction, which was already known to the team.

have been called, the function will be executed without problems. Then he can call `commitPrevRandao` and finally `submitFinal`, in which it is not verified that the `bitfield.length` is at least ⅔ of m1.

The severity of the issue depends on the possible relative values of m1 and m2.

Recommendation

We suggest adding a check of the `bitfield.length` in the `submitFinal` and `submitFinalWithHandover` functions.

MEDIUM-2	A relayer can execute the first two steps of the protocol (`submitInitial(WithHandover)` and `commitPrevRandao`) multiple times to get a random seed in his favor
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

The steps of the protocol are meant to be executed sequentially by the relayer, as follows: `submitInitial(WithHandover)->commitPrevRandao->createFinalBitfield->submitFinal(WithHandover)`. Although, nothing prevents the relayer from re-executing the first step (`submitInitial(WithHandover)`) after he has called `commitPrevRandao`. That way he creates a new ticket, but for the same commitment, with a zero `prevRandao` variable, therefore he is allowed to call again `commitPrevRandao`. A malicious relayer could repeat this procedure as many times as he wishes, till he gets a seed in his favor as an outcome. The only blocking action would be another relayer submitting a commitment for a more recent `blockNumber`, but the protocol cannot rely for its security on this.

Recommendation

We suggest adding a check on `submitInitial` and `submitInitialWithHandover` that the prevRandao variable of the ticket has not been set before, preventing an adversarial relayer from executing multiple times the first two steps. Although this is just a mitigation, since a relayer can

use multiple addresses to submit the same commitment.

MEDIUM-3	`BeefyClient::encodeCommitment` does not exclude collisions and an adversarial relayer could misuse this
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

The relay chain validators sign the hash of the commitment and the BeefyClient contract verifies these signatures. The commitment includes five variables: `blockNumber`, `validatorSetID`, `payload.mmrRootHash`, `payload.prefix` and `payload.suffix`. These variables are first encoded in a single variable of type `bytes`:

```
function encodeCommitment(Commitment calldata commitment) internal pure returns (bytes
memory) {
        return bytes.concat(
            commitment.payload.prefix,
            commitment.payload.mmrRootHash,
            commitment.payload.suffix,
            ScaleCodec.encodeU32(commitment.blockNumber),
            ScaleCodec.encodeU64(commitment.validatorSetID)
        );
    }
```

Then the `keccak256` is applied to that single variable and outputs the `commitmentHash`. The problem is that `payload.prefix` and `payload.suffix` are of type `bytes`, i.e. of arbitrary length, therefore for some commitments a relayer can find collisions, i.e. a different commitment structure with the same hash. Although this new artificial commitment will probably be of no meaningful content, it's a good practice to avoid any kind of collision on the protocol level.

Recommendation

We suggest restricting, if it is possible, the lengths of the prefix and suffix variables of the payload structure to avoid collisions.

MEDIUM-4	A malicious commitment, if accepted, could block the client for an arbitrarily long period
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

The Beefy Client only checks that the commitment is signed by (a subset of the) relay chain validators and does not care about the content of the commitment, since this is handled by other layers e.g. the GRANDPA finality gadget of Polkadot. However, if a number of malicious relay chain validators sign a commitment with a huge `blockNumber` and the relayer manages to get this commitment accepted (the signatures are valid therefore the only obstruction is that these malicious validators are chosen by the Beefy Client for verification, which has a small but not zero probability to happen) then the Beefy Client cannot accept new commitments for a long time due to the restriction:

```
if (commitment.blockNumber <= latestBeefyBlock) {
            revert StaleCommitment();
        }
```

in the `submitFinal` and `submitFinalWithHandover` functions.

Recommendation

There is no easy fix to this issue without re-designing the client and its interactions with other layers. Someone could argue that the probability of this issue is extremely small, but we suggest adding an extra functionality that will allow the client to remove such malicious commitments to avoid being blocked for long periods.

Low

No low issues found.

Informational/Suggestions

INFO-1	Not used custom error
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

The custom error `InvalidTask()` is defined but is never used in the contracts.

Recommendation

We suggest removing this custom error statement.

INFO-2	Missing conditions in `subsample`, `createBitfield`, `isSet` and `set` functions
Contract(s)	`Bitfield.sol`
Status	**Open**

Description

- `subsample`: `length` should be `<= prior.length*256` (this is correctly described in the comments) and n should be <= number of set bits in prior in the (bit) range [0:length], but these conditions are not checked in the body of the function.
- `createBitfield`: `arrayLength*256` should be >= `bitsToSet.length`.
- `isSet`: `self.length` should be $\geq index/2^8$.

- Set: `self.length` should be $\geq index/2^8$.

Recommendation

We suggest adding checks that the above-mentioned conditions are satisfied to make the `Bitfield` library self-contained.

INFO-3	Redundant & operation in `set` and `isSet`
Contract(s)	`Bitfield.sol`
Status	**Open**

Description

In `Bitfield::set, isSet` the & operation in the following line is redundant:

```
uint8 within = uint8(index & 0xFF)
```

since the type casting of the uint256 to uint8 just returns the last 8 digits.

Recommendation

We suggest removing this extra operation.

INFO-4	Code duplication
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

Functions `submitFinal` and `submitFinalWithHandover` have many lines of code in common.

Recommendation

We suggest constructing a separate method implementing these common lines which will be called by these two functions, to avoid code duplication and improve the readability of the code.

INFO-5	Typos in comments
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

In the comments above `struct ValidatorProof` the `addr` variable should be renamed `account`.

In the comments above `struct Ticket`: `sender` should be renamed `account`, `bitfield` should be renamed `bitfieldHash` and a description of the `prevRandao` variable is missing.

In the comment above `randaoCommitDelay` we read that this variable should be set to `MAX_SEED_LOOKAHEAD`, but `MAX_SEED_LOOKAHEAD` counts epochs and `randaoCommitDelay` counts blocks i.e. 32*epochs.

The comments above `submitFinalWithHandover` are not complete and explanations for several arguments of the function are missing.

Recommendation

We suggest correcting the typos and providing the missing definitions to improve readability.

INFO-6	`submitInitial` and `submitInitialWithHandover` are declared payable with no apparent reason
Contract(s)	`BeefyClient.sol`

Status	**Open**

Description

The functions `submitInitial` and `submitInitialWithHandover` are declared `payable`, although it is not described in the comments or in the provided documentation that the relayer should pay ETH to submit a commitment in the light client, therefore we see no apparent reason to declare these functions `payable`.

INFO-7	Implicit type cast
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

`commitment.blockNumber` is of type uint32 and `latestBeefyBlock` is uint64 although they are used to store similar things and moreover, the value of `commitment.blockNumber` is stored in `latestBeefyBlock` in `submitFinal` and `submitFinalWithHandover`.

The argument of `minimumSignatureThreshold` should be of type uint256 although this function is called only once with a uint128 as argument.

Recommendation

We suggest using the same types for consistency as a good practice, although the compiler will automatically type cast them and the above-mentioned cases will not cause any problems.

INFO-8	The signatures should be in the appropriate/non malleable format otherwise the open zeppelin function `recover` will revert
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

There is the known ECDSA signature vulnerability i.e. if (r,s) is a valid signature, (r, n-s) is also valid (n is the order of the elliptic curve group). The `BeefyClient` contract does not use the ecrecover function, which is vulnerable to this problem, but the recover function of the ECDSA library of open zeppelin. This function avoids the vulnerability by requiring the s value to be in the lower half of the order and returning an error message otherwise[3]. Therefore the proofs of the (v,r,s) variables of the ValidatorSet structure should be in the appropriate format.

INFO-9	Minor optimization in `doSubmitInitial`
Contract(s)	`BeefyClient.sol`
Status	**Open**

Description

The exact number of set validators in the bitfield is not needed. What is actually needed is that they are more than ⅔ of the validators' set. Therefore instead of calling the `Bitfield.countSetBits` function, another similar function could be built which will count the set bits in the bitfield till they exceed a given threshold and not all of them.

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/5420879d9b834a0579423d668fb60c5fc13b60cc/contracts/utils/cryptography/ECDSA.sol#L125

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.