StakingP2 Smart Contract Security Review

coinspect

flare

Staking P2

Smart Contract Review

Version: v240220 Prepared for: Flare November 2023

Smart Contract Security Review

Executive Summary

In September 2023, Flare engaged Coinspect to review a merge request responsible for implementing the `Flare Staking Phase 2` feature. The objective of the project was to evaluate the security of a staking system that allows users to mirror operations made on Flare's `P-Chain` to `C-Chain`. This mirroring process grants `P-Chain` stakers a balance increase on the `C-Chain`, which is then taken into account for Flare's rewards distribution process. `Flare Staking Phase 2` implements the mirroring functionality by integrating a voting system where trusted entities vote to commit the merkle root for the transactions made on the `P-Chain`.

The following issues were identified during the initial assessment:

	✓ Solved	⚠ Caution Advised	✗ Resolution Pending
High	0	0	0
Medium	0	0	0
Low	3	0	0
No Risk	3	0	0
Total	6	0	0

Coinspect identified three low-risk issues and three informational issues.

The first low-risk issue, `PTSK-001`, depicts how malicious voters can steal future rewards by committing fake roots, circumventing the currently implemented countermeasures. Also, attackers can create perpetual staking positions that will never be decreased by the Flare Daemon (`PTSK-002`). Lastly, `PSTK-003`, shows how the system does not filter zero-amount stakes, that could potentially overpopulate the staking cleanup queue. This would cause lag for each `daemonize` call, resulting in rewards being distributed for stakes that should have been considered finished.

Summary of Findings

Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
PSTK-001	Insufficient protection against fake stakes	Low
PSTK-002	Infinite mirrored stakes lead to eternal rewards	Low
PSTK-003	Zero-value mirrored stakes increase the lag on daemonize	Low
PSTK-004	Additional vote required after decreasing the voting threshold	None
PTSK-005	Votes of replaced voters count for current root	None
PTSK-006	Never-finalizing voting epoch due to high thresholds	None

Assessment and Scope

The source code review of the `Flare Staking Phase 2` project started on September 4, 2023, and was conducted on the `staking` branch of the git repository located at https://gitlab.com/flarenetwork/flare-smart-contracts/-/merge_requests/665 as of commit `c9796a64227d6ae103fe008e0e0dc5f908cbab18`.

Overall, the code was easy to read. However, Coinspect suggests to include more documentation and specifications regarding its use cases. In addition, the testing suite quality is high.

Design Principle

The `Flare Staking Phase 2` code is based on a proof system where trusted entities, called voters, propose and vote for each epoch's merkle root containing the transactions that occur on the `P-Chain` (the `Platform Chain`). The transactions supported are staking, delegating, and instantiating a validator. Each type of transaction allows users to mirror their position to the `C-Chain` (so called `Contract Chain`). Then, the mirrored transactions are considered by the combined token implementation, `CombinedNat`, to calculate the balance used for rewards distribution. The balance granted by the mirrored transactions is decreased by the `Flare Daemon`, according to their end time (expiration).

Trust Assumptions

A strong assumption during the development of this feature is that voters are trusted entities. While Coinspect considered the voters as trusted, auditors also analyzed the review of the system's components assuming that the inputs (e.g., proofs or actions performed by off-chain components) could potentially be corrupt or malicious.

The assessment's focus was to reduce the risk based on the defense-in-depth principle, performing checks on each critical layer to minimize the reliance on the trust assumptions. Even when some components are trusted, the system might face critical consequences in the event of, for example, a supply chain attack targeting voter's private keys. Attackers could try to compromise trusted voters in order to vote for fake roots (proving stakes or delegations that never happened on the `P-Chain`).

This takeover could be used to increase the attacker's `CombinedNat` balance on the `C-Chain` and receive perpetual rewards or force a chain fork. For example: attackers could prevent any other mirroring by granting themselves all the remaining total supply (triggering an overflow on future mirroring operations) or generate multiple perpetual staking positions. Because of the reasons just mentioned, Coinspect focused on suggesting improvements for the protocol to reduce the likelihood of disrupting the system if the trusted voters are compromised.

Known Limitations

The balance decrease made by the Daemon can revert as it loops over each position and timestamp. Flare implemented several ways to prevent that from happening such as limiting the amount of mapping-updates under two different operating conditions: `normal` and `fallback`. When the system enters a `fallback` state, the maximum amount of updates is reduced by a 20%. This design might cause a delay or lag scenario, because it could happen that a single `daemonize` call is not enough to fully decrease all the staking positions to-be-closed by that timestamp. If that happens, users will be able to unfairly receive rewards for more blocks until the Daemon catches up. This is a limitation known by the Flare team that was discussed during the engagement's kick-off meeting.

Offchain consuming services

During this engagement, Coinspect did not review how the validator uptime data is consumed once the respective event (`PChainStakeMirrorValidatorUptimeVoteSubmitted`) is emitted. As the design of the uptime voting system found in the smart contracts is lightweight, it only emits the event provided by its caller, services consuming its data should handle adversarial scenarios such as double voting, voting for fake `nodeIds`, among others.

Fixes Review

On September 19th, 2023, Coinspect reviewed the fixes made by the Flare team, on commit adecbc6b6bebe95d156b23893d99dc3ea918b2cd of the merge request !665.

Flare addressed all the issues of this report and added a new function to the `AddressBinder` contract.

AddressBinder changes

A new function, `registerPublicKey`, was added where users can link an address of the `P-Chain` and `C-Chain` by only providing the public key, instead of each chain's address and the public key. Coinspect identified that this function is less secure then previously implemented one (`registerAddresses`) as there are no checks to ensure that the derived addresses are the expected ones. In the event of having an issue with the derivation, users might register an unexpected address.

The motivation of this addition, according to the Flare Team is:

```
We added additional method registerPublicKey to AddressBinder contract in
order to simplify the users flow, so that they do not need to do all the
convertions by themselves, but can only provide a public key and later
check the addresses.
```

Detailed Findings

PSTK-001

Insufficient protection against fake stakes

Status

Solved

✓

Resolution

Fixed

Risk

Low

Impact

Medium

Likelihood

Low

Location

`PChainStakeMirrorMultiSigVoting.sol`

Description

Attackers can mirror stakes coming from fake voted proofs before the governance calls `resetVoting` and steal future rewards.

The voting contract has an emergency function that allows the governance to reset a previously voted merkle root. However, once the voting finalizes there will be some time after `resetVoting` is effectively called. This enables attackers to collude to vote a merkle root that includes fake perpetual stakes or delegations and then call mirrorStaking before the governance resets the malicious root.

```
    function resetVoting(uint256 _epochId) external onlyGovernance {
        require(pChainMerkleRoots[_epochId] != bytes32(0), "epoch not
finalized");
        pChainMerkleRoots[_epochId] = bytes32(0);
        emit PChainStakeMirrorVotingReset(_epochId);
    }
```

Once a root has been committed (voting threshold was reached), the verification system will be able to retrieve its value for the specified epoch and will successfully verify the fake staking:

```
    function verifyStake(
        PChainStake calldata _stakeData,
        bytes32[] calldata _merkleProof
    )
        external view override
        returns (bool)
    {
        return _verifyMerkleProof(
            _merkleProof,

_merkleRootForEpochId(pChainStakeMirrorVoting.getEpochId(_stakeData.sta
rtTime)),
            _hashPChainStaking(_stakeData)
        );
    }
```

Recommendation

Include a delay considering the root finalization timestamp at which staking positions can start to be mirrored. This will give the governance a buffer time in case a root needs to be reset before users can mirror their stakes on the C-Chain.

Status

Fixed on commit adecbc6b6bebe95d156b23893d99dc3ea918b2cd.

A revokeStake function was added, according to Flare's Team:

```
The idea was to mirror funds as soon as possible in order to support
fair rewarding and up-to-date voting in Phase 3, so introducing
additional delay is not a desired direction. Instead of this we decided
to add a revokeStake method that can immediately remove fake stakes and
```

```
can be only called by governance. In that case the user will only have
a fake balance/vote power for a short period of time (historical
checkpoints should not be changed as they might already be used
somewhere)
```

While the issue is fixed, Coinspect noted that two threats related to this implementation need to be taken into account moving forward:

1. **Mirroring a malicious position that disrupts the current checkpoint will potentially affect services reading and consuming from the checkpointed-getters (`balanceOfAt`, `stakesOfAt`, among others)**. However, mirroring an overly high position that triggers an overflow on the respective checkpoint is no longer possible thanks to the fixes introduced for `PSTK-002` and `PSTK-003`.

2. The `revokeStake` function poses a centralization risk where the governance can revoke the staking of any user, even if the root of that epoch was not reset. If this function is meant to be used only for malicious mirrored stakes that occur due to reset roots, **Coinspect suggests checking that the epoch's root is `bytes32(0)` for the transaction being revoked**.

PSTK-002

Infinite mirrored stakes lead to eternal rewards

Status
Solved

Resolution
Fixed

Risk
Low

Impact
Medium

Likelihood
Low

Description

A proof with infinite `endTime` will never be cleaned by the Flare Daemon, granting the attacker eternal rewards.

When mirroring a staking, the `PChainStakeMirror` trusts blindly that the provided parameters are not malicious:

```
    function mirrorStake(
        IPChainStakeMirrorVerifier.PChainStake calldata _stakeData,
        bytes32[] calldata _merkleProof
    )
        external override whenActive
    {
        require(verifier.verifyStake(_stakeData, _merkleProof), "stake
not proved");
        // unique hash is combination of transaction hash and source
address as
        // staking can be done from multiple P-chain addresses in one
transaction
        bytes32 txHash = keccak256(abi.encode(_stakeData.txId,
_stakeData.inputAddress));
        require(transactionHashToPChainStakingData[txHash].owner ==
address(0), "transaction already mirrored");
        require(_stakeData.startTime <= block.timestamp, "staking not
started yet");
        require(_stakeData.endTime > block.timestamp, "staking already
ended");

address cChainAddress =
addressBinder.pAddressToCAddress(_stakeData.inputAddress);
        require(cChainAddress != address(0), "unknown staking
address");

PChainStakingData memory pChainStakingData =
            PChainStakingData(cChainAddress, _stakeData.nodeId,
_stakeData.weight);
        transactionHashToPChainStakingData[txHash] = pChainStakingData;
        endTimeToTransactionHashList[_stakeData.endTime].push(txHash);
        _increaseStakeAmount(pChainStakingData, txHash,
_stakeData.txId);
    }
```

A malicious voter or colluded ones can propose a fake root where the stake data has infinite `_stakeData.endTime`, and `daemonize()` will never call `_decreaseStakeAmount` for that position.

Recommendation

Enforce a maximum end time for stakes.

Status

Fixed on commit `adecbc6b6bebe95d156b23893d99dc3ea918b2cd`.

The Flare Team added a set of immutable variables to `PChainStakeMirrorVerifier` to restrict the staking's duration and amounts upon verification, also fixing `PTSK-003`:

```
    uint256 public immutable minStakeDurationSeconds;
    uint256 public immutable maxStakeDurationSeconds;
    uint256 public immutable minStakeAmountGwei;
    uint256 public immutable maxStakeAmountGwei;
```

Coinspect considers that their immutability restricts the mirroring protocol and could cause a malfunctioning if some parameter is modified on the `P-Chain`.

Consider making them mutable, behind an `onlyGovernance` setter.

The Flare Team responded:

```
We considered this option to add the methods to change the parameters
by governance, but we came to the conclusion that it is more gas
efficient to have them as immutable values and in case we need to
change them just redeploy the contract and do one governance
transaction through address updater to actually switch the verifier
contract on PChainStakeMirror Contract
```

PSTK-003

Zero-value mirrored stakes increase the lag on daemonize

Status

Solved

✓

Resolution

Fixed

Risk

Low

▼

🟥 □ □

Impact

Low

Likelihood

Low

Location

`PChainStakeMirror.sol`

Description

When mirroring, the protocol allows processing zero-amount stakes. As a consequence, a proof containing many zero-value stakes could be used to attack the system as the `daemonize()` will also have to decrease the stake amounts for zero-staking positions. Hence, increasing the usage of the `noOfUpdates` and limiting the system to update relevant positions.

The mirroring contract has no mechanism to prevent this side effect of a malicious proof, processing a zero-amount staking:

```
    function mirrorStake(
        IPChainStakeMirrorVerifier.PChainStake calldata _stakeData,
        bytes32[] calldata _merkleProof
    )
        external override whenActive
    {
        require(verifier.verifyStake(_stakeData, _merkleProof), "stake
not proved");
        // unique hash is combination of transaction hash and source
address as
        // staking can be done from multiple P-chain addresses in one
transaction
        bytes32 txHash = keccak256(abi.encode(_stakeData.txId,
_stakeData.inputAddress));
        require(transactionHashToPChainStakingData[txHash].owner ==
address(0), "transaction already mirrored");
        require(_stakeData.startTime <= block.timestamp, "staking not
started yet");
        require(_stakeData.endTime > block.timestamp, "staking already
ended");

address cChainAddress =
addressBinder.pAddressToCAddress(_stakeData.inputAddress);
        require(cChainAddress != address(0), "unknown staking
address");

PChainStakingData memory pChainStakingData =
            PChainStakingData(cChainAddress, _stakeData.nodeId,
_stakeData.weight);
        transactionHashToPChainStakingData[txHash] = pChainStakingData;
        endTimeToTransactionHashList[_stakeData.endTime].push(txHash);
        _increaseStakeAmount(pChainStakingData, txHash,
_stakeData.txId);
    }
```

As a consequence, the `endTimeToTransactionHashList[_stakeData.endTime].push(txHash)` queue is populated with a valueless position that will be later on cleaned up on `daemonize()`. An overpopulation of zero-value positions can generate an outstanding lag when decreasing staking positions or even force a revert:

```
    for (uint256 i =
endTimeToTransactionHashList[nextTimestampToTriggerTmp].length; i > 0;
i--) {
        noOfUpdates++;
        if (noOfUpdates > maxUpdatesPerBlockTemp) {
            break;
        } else {
            bytes32 txHash =
endTimeToTransactionHashList[nextTimestampToTriggerTmp][i - 1];

endTimeToTransactionHashList[nextTimestampToTriggerTmp].pop();

_decreaseStakeAmount(transactionHashToPChainStakingData[txHash],
txHash);
            delete transactionHashToPChainStakingData[txHash];
        }
    }
```

Currently, `GovernanceVotePower.updateAtTokenTransfer()` allows zero-amount updates as the following line is commented:

```
    // require(_amount > 0, "Cannot transfer zero amount");
```

However, if this line is uncommented a revert in `daemonize()` will be triggered. This would allow users to keep their mirrored staking positions forever on the `C-Chain`.

Proof of Concept

The following test shows that the system allows mirroring zero-value staking positions, checking that `daemonize()` does not revert and emits the `StakeEnded` event for both positions.

To run the script, place it on `test/unit/staking/implementation/PChainStakeMirror.ts`.

```
it("Coinspect - Test Zero Stake Amount", async () => {
        let data1 = await setMockStakingData(
          verifierMock,
          pChainVerifier,
          stake1Id,
          0,
          registeredPAddresses[0],
          nodeId1,
          now.subn(10),
          now.addn(10),
          0
        );
        let data2 = await setMockStakingData(
          verifierMock,
          pChainVerifier,
          stake1Id,
          0,
          registeredPAddresses[1],
          nodeId1,
          now.subn(10),
          now.addn(10),
          0
        );
        await pChainStakeMirror.mirrorStake(data1, []);
        await pChainStakeMirror.mirrorStake(data2, []);

await increaseTimeTo(now, 20);
        const signer = await ethers.getSigner(accounts[0]);
        const pChainStakeMirrorEth =
PChainStakeMirror__factory.connect(pChainStakeMirror.address, signer);

const tx = await pChainStakeMirrorEth.daemonize({ from: accounts[0] });
        let receipt = await tx.wait();
const txHash0 = ethers.utils.keccak256(
          ethers.utils.defaultAbiCoder.encode(["bytes32", "bytes20"],
[stake1Id, registeredPAddresses[0]])
        );
        expectEthersEvent(receipt, pChainStakeMirrorEth, "StakeEnded",
{
          owner: registeredCAddresses[0],
          nodeId: nodeId1,
          amountWei: 0 * GWEI,
          txHash: txHash0,
        });

const txHash1 = ethers.utils.keccak256(
          ethers.utils.defaultAbiCoder.encode(["bytes32", "bytes20"],
[stake1Id, registeredPAddresses[1]])
        );
        expectEthersEvent(receipt, pChainStakeMirrorEth, "StakeEnded",
{
          owner: registeredCAddresses[1],
          nodeId: nodeId1,
          amountWei: 0 * GWEI,
          txHash: txHash1,
        });
      });
```

Recommendation

Allow the governance to setup a minimum staking amount (matching the minimum size imposed on the `P-Chain`) and enforce it when mirroring a position.

Status

Fixed on commit `adecbc6b6bebe95d156b23893d99dc3ea918b2cd`.

See `PTSK-002` for more details.

PSTK-004

Additional vote required after decreasing the voting threshold

Status

Solved

✓

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

–

Location

`PChainStakeMirrorMultiSigVoting.sol`

Description

When the governance adjusts the voting threshold downwards for an on-going voting, if the new threshold is below the current amount of votes for a specific root, an additional vote will still be required to commit the root.

```
    function setVotingThreshold(uint256 _votingThreshold) external
onlyGovernance {
        require(_votingThreshold > 1, "voting threshold too low");
        votingThreshold = _votingThreshold;
        emit PChainStakeMirrorVotingThresholdSet(_votingThreshold);
    }
```

This happens because the finalization condition checks that the amount of votes plus one, exceeds the threshold:

```
    if (merkleRootVotes.votes.length + 1 >= votingThreshold) {
        // publish Merkle root
        pChainMerkleRoots[_epochId] = _merkleRoot;
        delete epochVotes[_epochId];
        emit PChainStakeMirrorVotingFinalized(_epochId, _merkleRoot);
    }
```

For example:

Initial State

```
Threshold = 10
Votes[RootA] = 9
Votes[RootB] = 4
```

Final State

```
Threshold = 5
Votes[RootA] = 9
Votes[RootB] = 4
```

The final state will require another call to `submitVote` by a voter that has not voted yet either to commit the `RootA` or `RootB`, even if the `RootA` has more votes than the threshold.

Recommendation

Document this side effect when reducing the threshold while voting is taking place.

Status

Fixed on commit adecbc6b6bebe95d156b23893d99dc3ea918b2cd.

Flare added the following comment on `setVotingThreshold()`:

```
     * **NOTE**: Decreasing threshold will not finalize an ongoing
voting.
     * Additional vote will be required, even if, according to the new
threshold, voting should already be finalized.
```

PTSK-005

Votes of replaced voters count for current root

Status

Solved

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

–

Location

`PChainStakeMirrorMultiSigVoting.sol`

Description

The only way to add or remove voters from the voting contract is by replacing all the voters list. Votes coming from entities that are then replaced will still contribute to the accounting for the current root:

```
function changeVoters(address[] calldata _newVotersList) external
onlyGovernance {
    voters.replaceAll(_newVotersList);
    emit PChainStakeMirrorVotersSet(_newVotersList);
}
```

When accounting for votes, they are only cleared once the voting ends (the threshold is exceeded):

```
if (merkleRootVotes.votes.length + 1 >= votingThreshold) {
    // publish Merkle root
    pChainMerkleRoots[_epochId] = _merkleRoot;
    delete epochVotes[_epochId];
    emit PChainStakeMirrorVotingFinalized(_epochId, _merkleRoot);
```

This means that votes coming from replaced voters will still count.

Recommendation

Document this scenario in the `changeVoters` function.

Status

Fixed on commit `adecbc6b6bebe95d156b23893d99dc3ea918b2cd`.

Flare added the following comment on `changeVoters()`:

```
     * **NOTE**: Already casted votes in an ongoing voting will not be
deleted and will count towards the threshold.
     * **NOTE**: Setting fewer voters than the threshold will disable
finalization of voting.
```

PTSK-006

Never-finalizing voting epoch due to high thresholds

Status

Solved

✓

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

–

Location

`PChainStakeMirrorMultiSigVoting.sol`

Description

The governance can set an overly high threshold value and prevent any voting from concluding.

When setting the threshold, any value can be set, regardless the current amount of allowed voters:

```
function setVotingThreshold(uint256 _votingThreshold) external
onlyGovernance {
    require(_votingThreshold > 1, "voting threshold too low");
    votingThreshold = _votingThreshold;
    emit PChainStakeMirrorVotingThresholdSet(_votingThreshold);
}
```

This means, that setting threshold values over the total amount of voters will prevent a root commitment as there is no way it can be reached:

```
if (merkleRootVotes.votes.length + 1 >= votingThreshold) {
    // publish Merkle root
    pChainMerkleRoots[_epochId] = _merkleRoot;
    delete epochVotes[_epochId];
    emit PChainStakeMirrorVotingFinalized(_epochId, _merkleRoot);
```

It also applies for extreme threshold values such as the ones shown below, disrupting considerably the voting process:

```
threshold_1 = total amount of voters - 1
threshold_2 = 2
```

Recommendation

Document this possibility so that the governance is aware of the total amount of voters before setting the threshold or reset the voters list.

Status

Fixed on commit `adecbc6b6bebe95d156b23893d99dc3ea918b2cd`.

Flare added the following comment on `setVotingThreshold()`:

```
    * **NOTE**: Setting higher threshold than the total number of
voters will disable finalization of voting.
```

Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.

File hashes

Directory `./contracts/governance`:

```
a3163d4d5316c5531bb73809863b69f27dbbf115310ea79165f09df7aa69cf2a
./implementation/GovernorVotePower.sol
0be4020adc4407c68d1555687544e7543b69b6b5ad56143fe87c4fb40388336c
./governance/implementation/PollingFtso.sol
```

Directory `./contracts/staking`:

```
512aa61c2d349695d131f0127e45c9fec008aa77034d8d2ed41c1034978f8485
./interface/IIPChainStakeMirrorVerifier.sol
43990706ed5385bc6158f4f3f02a891e4f207e080142eab0a5d46502f32294ab
./implementation/PChainStakeMirrorMultiSigVoting.sol
718ad1adab1bf81262e8751f77caf37d847c9679d97927da061008c5019a8b51
./implementation/AddressBinder.sol
8ed79f2202855142eb986ae926a08a5a834b2f593f5c36af177b4c71f504de09
./implementation/PChainStakeMirrorVerifier.sol
180987e41c1c79a9b4685f746d64278744f12119f095410b9b49e1d7aef019a6
./implementation/PChainStakeMirror.sol
6fa6158a97053663741e2437c644f92ac1e973c68e41dd68e477f5daa33652fd
./implementation/PChainStake.sol
c7607ac8a7193aa432dae9154ab339f7f6cfed4761d6532fa9b459f9ca776b56
./lib/PChainStakeHistory.sol
```

Directory `./contracts/token`:

```
e8b7e40cd3a20b8f663793453d63c913f3598625b2b5c5dcd9b3dc52565cf12a
./implementation/CombinedNat.sol
ffc78a8e44ec6f94ed3a8ee5bd5189fbf10451aca8be9e205a17ceadb516b5d9
./implementation/GovernanceVotePower.sol
e3bbb05ef6389673dc0489d57088f5c42fdde929c5dd3f11354377109bc2650e
./interface/IICombinedNatBalance.sol
```

```
05fcda973d976a87e81a2189f5974ba2089833da06e653e52bf58f0d495165ca
./interface/IIGovernanceVotePower.sol
```

Directory `./contracts/tokenPools`:

```
4547ec3db8a1f92323570888c2da3f6f869d62ff8d0d7ebf09e09e5c50551f1a
./implementation/DistributionToDelegators.sol
```

Directory `./contracts/userinterfaces`:

```
8f560d1730382027e8bfeb5b3d15763f4c5ec050ea12b20fadd0f0b9d1ca91d5
./IAddressBinder.sol
e496de47d2de3d73cec89059d75e3037203806073df838609bd46bae24c9173
./IPChainStakeMirrorVerifier.sol
6d8f93bc54c37cf51778289dcbd95cb490af1fb3b0090babf5e9cdb39ced8166
./IPChainStakeMirror.sol
8cdae6125d704570f6b5c185f6df8cf96ec7a46e091c29f86f5560d94b7d211b
./IPChainVotePower.sol
a7f4a69f11031ecf9e94bd10bf259ce5c1b8483af784b22156a7fa6618e102b6
./IPChainStakeMirrorMultiSigVoting.sol
```

Directory `./contracts/utils`:

```
be4aa19441689b0bb9bfba0d2ec3c6748daf27aaaad1cfd250629920a0209be3
./implementation/BytesLib.sol
```