

Audit Report

Proof of Neutrality

Delivered: 2023-03-10



Prepared for Proof of Neutrality Network by Runtime Verification, Inc.



[Summary](#)

[Disclaimer](#)

[Proof of Neutrality Network: Off-Chain Side Overview](#)

[General description](#)

[Schedule](#)

[PoN particularities](#)

[Bidding system](#)

[Protocol agents](#)

[Builder](#)

[Proposer](#)

[Relayer](#)

[Hosted Service](#)

[Reporter](#)

[Behavior risks](#)

[PBS violation rules and penalties](#)

[Assumptions](#)

[Violation rules](#)

[Proposer economic penalty](#)

[Proof of Neutrality Network: Contract Description and Invariants](#)

[Payout Pool](#)

[Proposer Reward and Debt Distribution](#)

[Builder Debt Distribution](#)

[Payout Pool Invariants](#)

[Proposer Registry](#)

[Proposer Registry Invariants](#)

[Builder Registry](#)

[Builder Registry Invariants](#)

[Reporter Registry](#)

[Reports](#)

[Findings](#)

[A01: PayoutPool::processProposerSlashing\(\) incorrectly increases the rewards of the slashed proposer](#)

[A02: PayoutPool:: repayDebt\(\) incorrectly increases the rewards of the debtor proposer](#)

[A03: PayoutPool::processProposerExit\(\) incorrectly updates latestProposerClaim](#)

[A04: PayoutPool:: repayDebt incorrect repayment distribution](#)

A05: PayoutPool:: computeRewardDelta() underflow

A06: PayoutPool::processProposerSlashing() fails if there are no active proposers

A07: setContracts() should only be callable once
Recommendation

A08: ProposerRegistry::kickProposer() missing decrement of active validators

A09: ProposerRegistry::exitClaimAmount may be set too high in kickProposer()

A10: Builders who are slashed in status EXIT PENDING can neither top up nor exit

A11: Builders cannot be reported again while being slashed

A12: Cannot report builder with insufficient stake

A13: Checking that address is not a contract is not reliable

A14: PBSFactory::deployContracts() payoutCycleLength can have different values for different contracts

Informative findings

B01: PayoutPool:: payoutReporter() does not follow the checks-effects-interactions pattern

B02: PayoutPool::receive() emits an event containing only taxed payment

B03: PayoutPool::receive() excessive scheduling of proposers can lead to bad reward distribution

B04: BuilderRegistry.slashBuilder() contains redundant variable

B05: ProposerRegistry.activateProposers() can be simplified

B06: ProposerRegistry::isProposerOperational() inconsistent returns for single and batch functions

B07: ProposerRegistry::positionForRagequit() redundant check

Bo8: Slashing builder who is EXIT PENDING always transitions to SLASHED

Bo9: BuilderRegistry:: isPositionedForExitAndNotSlashed() contains redundant condition

B10: Inconsistent use of < and <= with ReporterRegistry::reporterToLastReportedBlock()

B11: Builders and reporters need to pay gas for activateProposers() and _repayDebt()

B12: isReporterActive and isReporterRagequitted can be combined into single status mapping

B13: ReporterRegistry::submitReport() can be slightly refactored

B14: Contracts do not prevent a proposer from being reported for a violation that occurred after initiating the exit process

B15: Input validation for submitting reports

Appendix: Proof of Neutrality Off-Chain Model

Summary

[Runtime Verification, Inc.](#) has audited the smart contract source code for the [Proof of Neutrality Network](#). The review was conducted from 23-01-2023 to 24-02-2023.

Proof of Neutrality Network engaged Runtime Verification in checking the security for their protocol. The Proof of Neutrality (PoN) is an implementation of the [Proposer-Builder Separation](#) (PBS), where proposers (validators) auction their blockspace to builders who want to include their crafted transaction ordering for MEV.

Scope

The audited smart contracts are:

- `lib/MovingAverage.sol`
- `lib/PBSBase.sol`
- `pbs/BuilderRegistry.sol`
- `pbs/PayoutPool.sol`
- `pbs/ProposerRegistry.sol`
- `pbs/ReporterRegistry.sol`
- `proxy/UpgradeableBeacon.sol`
- `PBSFactory.sol`

The audit has focused on the above smart contracts and has assumed the correctness of the libraries and external contracts they use. The libraries are widely used and assumed to be secure and functionally correct.

The review focused on the `proposer-builder-registry` private code repository, which is a Hardhat and Foundry project with test scripts. The code was frozen for the audit code review at commit `76ef8fa617b0fd59b6ee8d9f058e2c7104a165ce`. Fixes to the audit findings are not part of the scope.

The review is limited in scope to consider only contract code. Although reviewing off-chain and client-side code is not in the scope of this engagement, they have been used for reference in order to understand the design of the protocol and assumptions of the on-chain code, as well as to identify potential issues in the high-level design. Details of the implementation of the off-chain code, as well as the security of the underlying cryptography are not in scope for this audit.

Assumptions

The audit is based on the following assumptions and trust model.

1. Contract owners are trusted and well-behaved. In particular, they
 - a. Set reasonable protocol parameters (e.g., minimum staking amount)
 - b. Rightly distribute the relayers' maintenance fee
 - c. Only register trusted hosted services and signature swappers that satisfy the validity assumptions
 - d. Upgrade contracts responsibly
2. Relayers are honest:
 - a. Always return the bid selected according to the relayer's strategy
 - b. Timely forward the signed header from the proposer to the chosen builder
 - c. Transparently cooperate with the hosted service for report validation
 - d. Do not arbitrarily discriminate against any builders or validators
3. Hosted services are competent and honest, in particular, they
 - a. Respond to requests from reporters in a timely manner
 - b. Do not issue signatures for false or misconstrued reports
4. There is at least one honest and active reporter in the network, ensuring that all violations are reported before they expire


Note that these assumptions roughly assume honesty and competence. However, we will rely less on competence and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

We do not assume full trustworthiness of the governance addresses and do not, for example, trust it with the ability to block certain users or take control of funds that the user did not intend to allow (roughly: theft).

The governance addresses may profit indirectly by disrupting the protocol, such as by registering malicious relayers or hosted services, but such considerations lie outside of the scope of the security review and will only be considered on a best-effort basis.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in [Disclaimer](#), we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to [known security issues and attack vectors](#). Thirdly, we discussed the most catastrophic outcomes with the team and reasoned backward from their places in the code to ensure that they were not reachable in any unintended way. Finally, we regularly participated in



meetings with the Proof of Neutrality Network team and offered our feedback during ongoing design discussions, and suggested development practices as well as design improvements.

This report describes the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practices, and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.



Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Proof of Neutrality Network: Off-Chain Side Overview

This section describes the off-chain side of the Proof of Neutrality Network. This includes their agents, duties, actions, and general considerations for the well functioning of the system.

General description

The agents of the PoN network are

- **Builder:** Builds blocks and bids for blockspace to include the built blocks. Builders receive the EIP1559 fees for that block.
- **Proposer:** Ethereum validator that auctions its blockspace to the builders.
- **Relayer:** Off-chain component where the auction takes place.
- **Reporter:** Checks for PBS violations and reports them to the PoN for rewards.
- **Hosted Service:** Trusted service that gathers data from all relayers to check and sign reports from reporters.

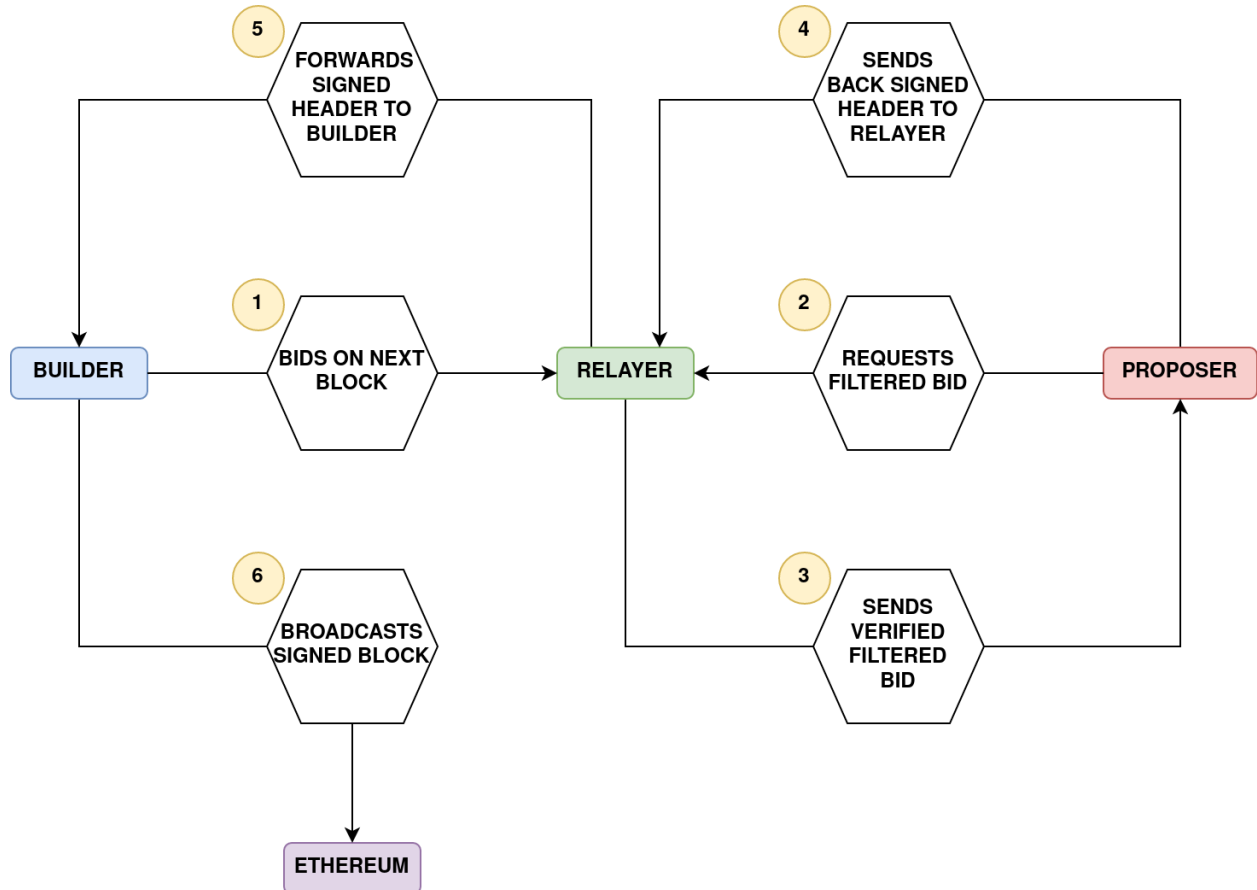
The workflow of the off-chain side of the PoN network is as follows.

1. A PoN validator is selected for block proposal by the Ethereum consensus layer.
2. Builders send the block header (SSZ summary of the Block that does not include the transaction contents) of the block they want to include, together with their bid to a PoN relayer. A bid contains a promised payment and cryptographical proof that the last transaction of the offered block is said payment to the PoN network.
3. PoN relayers check if the submitted bids would win the current auction according to the relayer's criteria. If successful, the validity of the cryptographical proof is also checked to accept the bid.
4. The proposer requests the filtered and verified block header from the different relayers it chooses.
5. After receiving the filtered bids from different relayers, the proposer chooses one bid, signs it, and sends it back to the corresponding relayer.
6. The relayer forwards the signed block header to the builder, who broadcasts to the Ethereum network with the block contents.^{1 2}

¹ Note that during this whole cycle, the builder is the only party that knows the full block contents. As such, it is their duty to broadcast the block.

² In the current implementation, the builder also sends the block contents to the relayer once received the block header. However, this implementation detail plays no role in the protocol's design and could be deprecated in the future.

The following diagram shows (without a 1-1 correspondence with the above items) the PoN workflow. For a more in-depth description of the off-chain side, we refer the reader to the appendix of this report.



Schedule

There is a pre-determined schedule for each step of the PoN workflow. The times are distributed as follows:

1. **Blockspace bids:** 2 seconds. Builders have a 2-second window to place their bids on the PoN relayers. Based on what other builders bid, they can increase their bids one more time per relayer.³
2. **Block header signature:** 8 seconds. The proposer has 8 seconds to request block headers from relayers and return one signed.

³ A sealed bid period may also exist to allow builders who could've bid higher to max out their bid to increase their chances of being chosen. The sealed bid is a final bid that relayers will accept within the first 0.8 seconds after the 2-second auction period has finished. The default strategy must accept any bid offered in this period that is higher than the previous auction-winner bid.

3. **Block broadcasting:** 2 seconds. Once the relayer receives the signed block header, there is a 2 seconds window for the relayer to forward the signed header to the selected builder, so that said builder could broadcast it to the Ethereum network.

A total of $2 + 8 + 2 = 12$ seconds are allocated to complete a block proposal, which is the time that the consensus layer gives for a proposer to submit a block.

PoN particularities

What sets the Proof of Neutrality network apart from other protocols is that the only agent who knows the contents of the block before it's included in the network is the builder that built such block. The relayer and the proposer can only read the payment amount to include the block. Particularly, the only information that is revealed to the relayer is the last transaction of the block, which contains the payment to the PoN by the builder. Refer to [this post](#) for more information on comparing the PoN with other systems.

Bidding system

Each relayer can implement its own bidding system. The PoN relayers come with a default bidding strategy that implements the highest bid auction mechanism (namely, the highest bid wins the auction) followed by a one-bid sealed auction.

Each builder has two bids per relayer during the bidding period. After the usual bidding period has ended, there is a sealed bid period where builders can bid only once. The rationale of this scheme is that if a builder was bidding under what it could offer based on other builders' bids, the sealed bid is an opportunity to bid for its true value in case the auction was not won.

Protocol agents

This section describes the duties that each protocol agent should perform to ensure the correct functioning of the PoN.

Builder

As stated above, the builder constructs a block and places a bid so that this block becomes the next one included in the Ethereum network.

Duties

- Must include payment to the PoN network as the last transaction in the block. This is ensured by the [Restrictive Partially Blind Signatures](#) (RPBS), which allows the builder to

prove that the last transaction of the block is the promised payment without revealing the rest of the transactions.

- Must broadcast the signed block by the proposer. Since only the builder of the chosen block knows its entire contents, it's their responsibility to broadcast it to the network once they receive the signed block header from the proposer.
- Must construct a valid block. The block header submitted to the auction cannot be an invalid block (e.g., wrong slot number, timestamp, validator index...).

Proposer

The chosen validator by the Ethereum consensus layer that offers its blockspace for auction to the PoN. As mentioned before, proposers are not aware of the contents of the chosen block when signing its header.

Duties

- Sign a block header that contains any payment to the PoN network.

Expected behavior

- Once selected by the Ethereum consensus layer, request the block header from a registered PoN relay. The requested block header is the winner of the auction from that relay. Note that the proposer is free to choose any block offered by the requested relays. It's not necessary to select the highest bidder from all the queried relays.
- Return the signed header of the requested block on time. After getting the block header from a registered relay, the proposer must sign it and return it to the same relay before a certain time threshold passes. This threshold ensures the relay and the builder can execute the rest of the PoN workflow steps. Returning a signed header once this period has expired will be considered as not having returned it.

The difference between duties and expected behaviors resides in that as long as the duties of a proposer are fulfilled, not behaving as expected is not considered a violation of the PoN's rules.

Relayer

The relay is the off-chain communication channel. Relays receive offers from the builders, filter and validate them, forward chosen bids to proposers, and send the signed header back to the builder. Relays also store information about past auctions up to 1575 epochs, used to determine if a violation occurred and whose fault it was.

Also, each relay can implement its own bid-sorting strategy. By default, PoN provides a "maximum payment" sorting strategy. This is very customizable, and each relay can

implement a completely different strategy (e.g., a subscription-based prioritization, an on-chain blacklist filtering strategy, or demand a minimum bid depending on the previous block's bids).

Duties

- Transparently offer builders' bids to proposers without censoring or obscuring any offer fulfilling the criteria for the relayer's strategy.
- Keep a bounded record of all the steps taken during a block auction. These include
 - Builders' bids.
 - Requested block from the proposer.
 - Time of return of the signed header.
 - Attestation for signed header forwarding to the correct builder.

With this information, the hosted service can verify who is at fault if the PoN network has not been paid as it should.

Restrictions

There are some actions that relayers are prevented from doing by the design of the PoN off-chain architecture. They are the following:

- Access information (except the last transaction) of the constructed blocks from the builders.
- As a consequence of the above, reuse MEV crafted by a builder.
- Broadcast blocks. Since relayers only have the block header and lack the block contents, they should not be able to broadcast the block on the builder's behalf.

Hosted Service

Relayers do not share data with each other by default. As such, if there are multiple relayers in a PoN network working for a single payout pool, a single relayer will not be able to correctly distinguish if there has been a violation of the protocol rules. To remedy such situations, a *trusted* hosted service is introduced. The hosted service will act as a data-gathering and validation center for the relayers. It is necessary to obtain a signed report from the hosted service to submit it to the PoN network and have the necessary penalties executed.

Duties

- Be up to date on which relayers are active on the network and possibly if they are correctly functioning.
- Verify and sign reports. Reporters will scan the network looking for PBS violations. Once they identify one, the hosted service will be asked to verify and sign a report, sending it back to the reporter.

Reporter

Reporters are agents that scan each PoN slot, checking if a PBS rule violation has been committed. If a reporter detects a PBS rule violation ([Violation rules](#)), they will create a report. Then send it to the hosted service to verify that the reported penalty holds and have it signed. Once the signed report is returned to the reporter, it can be submitted to the on-chain side of the PoN network to collect a bounty for it.

Reporters do not have specific individual duties, although, for the correct functioning of a PoN network, there should at least be one good-functioning reporter so that no violation goes uncaught. A good-functioning reporter is a reporter that satisfies the following:

- Scans every PoN-related consensus layer slot and execution layer blocks in search of PBS violations.
- Timely (within 1575 epochs of the violation) reports observed violations to the PoN.

Behavior risks

There are certain risks and assumptions about the agents.

- Hosted services honesty. Hosted services hold immense control over the PoN network since their signature on reports is needed to execute the appropriate penalties for malfeasance. A malicious hosted service could cause lots of damage to the network in various ways
 - False reporting: modifying the report to change the penalty in any way.⁴
 - Agent alignment: not signing reports issued for a specific (or arbitrary) builder or proposer.
 - Reporter exclusion: not signing reports for a particular reporter or only signing reports for select reporters.
- Relay honesty. A relay holding back or manipulating relevant information could change the interpretation of the hosted service of a penalty. Since no data between relays is shared, a malicious relay cannot be held accountable directly for reporting false information.
- Third-party dominance. If someone controls a relay, a builder, and a validator, it can be unforeseeable what dynamics can arise if malicious behavior is intended.
- Relay uptime and well function. Even if a relay is not malicious, malfunctioning due to power outages or other complications could have severe consequences for the health of the PoN, especially during a PoN auction. But also while being requested information from the hosted service.

⁴ Note that all information used by the hosted service is publicly available. Hence any false reporting could be detected. However, this doesn't imply that the consequences of false reporting could be prevented.

- Reporter availability. Reporters must report within the given timeframe of one week (1575 epochs). Otherwise, a PBS violation will become unreportable.

PBS violation rules and penalties

Here is a description of what is considered a grieving action for the PoN network. The core of what constitutes a fault is always the same: the payout pool was not rightly paid. It can be that the payout pool received no payment or an insufficient amount. Once this is detected, the PBS violation rules determine what agent of the system with economic incentives (i.e., proposer or builder) is at fault.

Assumptions

To rightly enforce the PBS violation rules, the following assumptions must hold

- Relay and hosted service impartiality.
- Relay and hosted service responsiveness. They are assumed to execute their role without flaw. Note that a faulty relay or hosted service could be translated into an incorrect penalty for the builder or proposer, depending on how the relay or hosted service malfunctions.

Violation rules

The rules always have a single entity (proposer or builder) as the party at fault. It cannot happen that both parties are at fault at once. This is ensured because for one party to be at fault, the other party must have fulfilled their due action correctly.

A helpful way of understanding the rules is by looking at the diagram in the Section [General description](#). Possible penalties occur when steps 2, 4, or 6 are executed incorrectly.

Assumption: the PoN network has not received any payment for a slot, and the chosen proposer for the slot was registered in the PoN network.

1. Builder submits a blinded block

Proposer doesn't request a blinded block from **relay**

- a. **Fault:** proposer
- b. **Reason:** The relay received bids for the proposer's blockspace, but the proposer didn't request any blinded block from the relay
- c. **Penalty:** Described in [Proposer economic penalty](#)

2. Builder submits a blinded block

Proposer requests blinded block a from **relay**

Proposer fails to return signed header on time back to **relay**

- a. **Fault:** proposer
 - b. **Reason:** The relay received bids for the proposer's blockspace, but the proposer didn't return the signed header at all or didn't do it on time for the rest of the protocol to work appropriately
 - c. **Explanation:** If the proposer doesn't send back the signed header in the 8 seconds window allocated for it, the system cannot ensure the broadcasting of the block. Hence, it is the responsibility of the proposer to do so timely
 - d. **Penalty:** Described in [Proposer economic penalty](#)
3. **Builder** submits a blinded block
- Proposer** requests a blinded block from **relay**
- Proposer** returns the wrong signed header on time back to **relay**
- a. **Fault:** proposer
 - b. **Reason:** The relay received bids for the proposer's blockspace, but the proposer didn't return the signed header at all or didn't do it on time for the rest of the protocol to work appropriately
 - c. **Explanation:** If the proposer sends back the wrong signed header, the builder won't be able to broadcast the block; thus, the payout pool won't be paid
 - d. **Penalty:** Described in [Proposer economic penalty](#)
4. **Builder** submits a blinded block
- Proposer** requests a blinded block from **relay**
- Proposer** correctly returns signed header on time back to **relay**
- Builder** doesn't broadcast the block
- a. **Fault:** builder
 - b. **Reason:** The builder either didn't request the signed block from the relay or did request it but didn't broadcast it timely, thus grieving the network. Also, if the builder submits an invalid block, even if the proposer signs the block header, it won't be allowed to broadcast it.
 - c. **Penalty:** Twice the promised amount for the slot's blockspace

Assumption: The payout pool has received payment for a slot, and the chosen proposer for the slot was registered in the PoN network.

5. **Builder** submits a blinded block
- Proposer** requests a blinded block from **relay**
- Proposer** correctly returns signed header on time blinded block to the **relay**
- Builder** broadcasts the block, but the pool is underpaid
- a. **Fault:** builder
 - b. **Reason:** The relay checked that the last transaction of the builder's block was the promised payment to the payout pool. However, after the execution layer

updates Ethereum's state, the funds sent to the payout pool are less than expected

- c. **Explanation:** Even if the last transaction included in a block is a payment to the payout pool, it can be that the builder spends that money in the previous unknown transactions, thus making the last transaction fail. That is why if the block is included and the payout pool is underpaid, it's the builder's fault for not allowing the last transaction to succeed
- d. **Penalty:** Twice the promised amount for the slot's blockspace

Assumption: No assumption, the following rule can be applied at any time.

6. Proposer satisfies at least one of these:

6.1. Effective Balance Below 32 ETH

6.2. Proposer is slashed

6.3. Proposer has withdrawn

- a. **Fault:** proposer
- b. **Reason:** Only active and healthy validators can participate in the PoN network. Failure to be so will result in an immediate kick out of the PoN network.
- c. **Penalty:** Forfeit of deferred rewards and expulsion from the PoN

Proposer economic penalty

Proposer penalties emulate the average of the EIP1559 maximum priority fee for the last five blocks. For performance reasons, this value is approximated using the average of the 10th and 90th percentile for the past five blocks is taken. The economic penalty is then the mean of these results.

That is, let $p_{10}, p_{90}: N \rightarrow N$ be the functions that, given a block number, output the 10th and 90th percentile of the maximum priority fee paid for that block, respectively. Then the proposer economic penalty is

$$\frac{1}{5} \sum_{i=1}^5 \frac{p_{10}(n-i) + p_{90}(n-i)}{2} \text{ where block } n \text{ was where the proposer committed the PBS violation.}$$

Proof of Neutrality Network: Contract Description and Invariants

The on-chain side of the PoN is where the economic logic of the protocol is exercised. Every agent that plays a role in the PoN must be reflected on-chain.⁵

The on-chain logic is divided into four main contracts:

- **Payout Pool:** contains reward distribution logic for proposers and reporters.
- **Proposer Registry:** registry where proposers that pledge their blockspace to the PoN are acknowledged.
- **Builder Registry:** registry where builders that want to bid for proposers' blockspace are acknowledged. To be registered in the Builder Registry, builders must stake a determined amount of ETH that will be used to repay proposers and reporters in case of a penalty.
- **Reporter Registry:** registry for reporters to register and submit the violation reports.

This section describes the contracts at a high level and which invariants of their state we expect always to be respected at the end of a contract interaction.

Payout Pool

The Payout Pool is the contract where all rewards from the auctions are deposited. As such, it contains the logic to distribute such rewards amongst all registered proposers eligible for rewards.

It's important to note that for a proposer to start accruing rewards, it is not required to propose any block to the network and, thus, to contribute any rewards to the pool. In fact, the pool is not aware of which proposer signed the header for a given block. As soon as a waiting period of one week is over, any proposer will start accruing rewards.

There are two types of capital inflow to the contract: rewards from auctions and penalty repayment from builders who underpaid the pool. Similarly, there are three types of capital outflow from the contract: proposers withdrawing earned rewards, reporters withdrawing bounties from reporting misbehavior from proposers or builders, and the payout pool owner withdrawing the maintenance bucket for relayers.

It's important to note that the payout pool will only accept block rewards coming from the `block.coinbase`. Thus, a builder must set itself as the coinbase of the crafted block to be able to

⁵ Although currently, relayers are unknown to the on-chain code, they should eventually be known.

pay the pool⁶. This also prevents the pool from accepting ETH transfers from undue parties. Debt repayment from builders —proposer debt accounting and repayment is done within the pool— can only be exercised by the Builder Registry, thus restricting as well who can deposit ETH into the pool.

Proposer Reward and Debt Distribution

When a block reward is transferred to the payout pool, it is not immediately available to the active proposers of the PoN. It is added to a buffer that will transfer all accumulated rewards to the proposers each week. The reward for a block is divided equally among all active proposers. After a week (1575 epochs), proposers can claim all deferred rewards.

If proposers accrue debt from the penalties of violating any PBS rule, the pool will keep a record of how much each proposer owes to the pool. Then, if a proposer is active in the PoN and wants to withdraw their accrued rewards, the debt will be deducted from said rewards and distributed to the rest of the proposers. Anyone can trigger debt repayment from a proposer, even PoN-unregistered parties.

It's important to note that neither penalty nor debt payment is deferred. That is, any penalty or debt paid by a proposer is allocated directly to the claimable rewards of the other proposers. Reporters don't have any buffer in the pool for their rewards, although reporters can only claim them through the reporter registry, which adds a 100-block wait period to withdraw accrued rewards.

Builder Debt Distribution

Debt or penalty payment coming from builders is distributed in the same fashion as with proposers' debt or penalty. Proposers can immediately claim these refunds, and reporters will have the 100-block wait imposed by the Reporter Registry, which may not be active when the refund occurs.

Payout Pool Invariants

The following invariants should hold to ensure the correct functioning of the Payout Pool.

Invariant: If a proposer `p` held any debt, then the rewards available to claim (`claimableRewards`) would be the rewards acknowledged to `p` minus the debt accumulated. In case the debt is greater than the accumulated rewards, there should be zero claimable rewards.

```
proposerToTotalDebt[p] = d > 0 → claimableRewards[p] =  
_computeRewardsDelta[p] ÷ d where a ÷ b = 0 if a <= b
```

⁶ This feature has been deprecated in subsequent iterations of the protocol, giving the builder full agency to specify the coinbase address.

Invariant: If a proposer p doesn't hold any debt to the Payout Pool, then it should not hold any debt to any proposer either, and thus the total debt of proposer p should be zero.

$$\text{proposerToPayoutPoolDebt}[p] = 0 \rightarrow \text{proposerToTotalDebt}[p] = 0$$

Invariant: The last claimed amount of a proposer would never exceed the credited plus deferred rewards.

$$\text{lastProposerClaim}[p] \leq \text{cumulativePerProposerRewards} + \text{cumulativePerProposerDeferredRewards}$$

Invariant: The balance of the Payout Pool is the sum of the maintenance bucket balance, the leftover bucket balance, and the reporter and proposer rewards.

$$\text{PayoutPool.balance} = \text{PayoutPool.maintenanceBucketBalance} + \text{PayoutPool.leftoverBucketBalance} + \text{reporterRewards} + \text{proposerRewards}$$

Where:

- $\text{reporterRewards} = \Sigma \text{PayoutPool.reporterRewardsAccumulated}[r]$
- $\text{proposerRewards} = \Sigma \text{totalRewards}(p)[p \in \text{ActiveOrExitPendingProposers}]$

With

- $\text{ActiveOrExitPendingProposers}$ is the set of all proposers in state `ACTIVE` or `EXIT_PENDING`
- $\text{totalRewards}(p) = \text{upperRewardsBound}(p) - \text{lowerRewardsBound}(p)$
- $\text{upperRewardsBound}(p) =$
 - if p is `ACTIVE` then
 $\text{PayoutPool.cumulativePerProposerRewards} + \text{PayoutPool.cumulativePerProposerDeferredRewards}$
 - else if p is `EXIT_PENDING`
 $\text{ProposerRegistry.getExitClaimAmount}(p)$
 - else
 0
- $\text{lowerRewardsBound}(p) = \text{PayoutPool.latestProposerClaim}[p]$

Proposer Registry

The Proposer Registry has the role of acknowledging what proposers are active in the PoN. Being labeled as active by the Proposer Registry is what it means to be active in the PoN. The

same occurs with the other possible states of a proposer concerning the PoN: UNREGISTERED, REGISTERED, EXIT_PENDING, and KICKED. These are the meanings of each state:

- **UNREGISTERED:** the proposer is unknown to the PoN. Proposers with this label cannot earn PoN rewards nor be held accountable for not acting by the PBS rules.
- **REGISTERED:** the proposer is known to the PoN. Has been accepted by the PoN (via the Signature Swapper⁷) but is still waiting to be activated. Once the current cycle (of an week length) is over, it will become active. Proposers with this label cannot earn PoN rewards nor be held accountable for not acting by the PBS rules.
- **ACTIVE:** the label for proposers pledging all their blockspace to the PoN and earning rewards from other proposers' blockspace auctions. When an active proposer is selected for a slot by the consensus layer, the Payout Pool must receive a payment for that slot. Failure to fulfill this will incur a fault and subsequent penalty by the proposer.
- **EXIT_PENDING:** formerly active proposers that want to leave the PoN. The proposer no longer owes its blockspace to the PoN and, as such, will not earn any more rewards. However, it must remain in this state for a cycle to clear out any possible grieving done by the proposer that has not yet been reported.
- **EXITED:** after waiting for a cycle in the EXIT_PENDING status, the proposer has given the order to exit the PoN. This status entails that the proposer holds no debt to the PoN.
- **KICKED:** the proposer committed more than the maximum amount of faults or is not a healthy validator (is slashed, withdrew balance, effective balance below 32 ETH). This entails immediate expulsion from the PoN.

Note that to achieve a state, proposers must first have the previous one in the list. This is true except for the KICKED status, which can be achieved from the ACTIVE and EXIT_PENDING and cannot be reached from the EXITED status.

An important note is that proposer activation is capped at 100 validators per transaction. So if, for a given block, there are too many scheduled validators to be activated, some may not be activated and thus not distributed the rewards of that block in case there were any. Albeit it is unlikely that 100 validators are scheduled for a PoN block, it is important to highlight this.

Proposer Registry Invariants

Invariant: The `proposers` array contains all registered proposers ever, while the `historicalActivatedProposers` only contains those that at some point had status `ACTIVE`. Thus, the latter must be smaller than the former.

⁷ The Signature Swapper is an off-chain component used to validate the proposer's BLS signature for registration in the PoN since no on-chain BLS signature validation is currently possible. To register the proposer's BLS key in the Proposer Registry, proposers need to obtain a signature from the Signature Swapper, which will also ensure the proposer meets the standards to enter the PoN.

```
proposerRegistry.historicalActivatedProposers <=
proposerRegistry.proposers.length
```

Invariant: No proposers in the `historicalActivatedProposers` can have the `UNREGISTERED` status.

For all `i` with $0 \leq i < \text{historicalActivatedProposers}$ we have: `proposers[i].status ≥ ACTIVE`

Invariant: All proposers that are in the `proposers` array but not in the `historicalActivatedProposers` must have the `REGISTERED` status.

For all `i` with $\text{historicalActivatedProposers} \leq i < \text{proposers.length}$ we have: `proposers[i].status == REGISTERED`

Invariant: The `activeValidators` state variable correctly accounts for the number of active validators.

The number of `ACTIVE` proposers in the `proposers` array equals `activeValidators`

Invariant: All proposers in the `proposers` array are not `UNREGISTERED`.

For all `i` with $0 \leq i < \text{proposers.length}$ we have:
`blsPublicKeyToProposer[proposers[i]].status != UNREGISTERED`

Builder Registry

The Builder Registry holds the builders' status and their stake. To register as a PoN builder, builders must stake a determined amount of ETH. The stake is used to repay any possible grieving from the builders' side. The amount required to stake can vary depending on the average of the rewards offered per block. To offer bids to the PoN, builders must have staked at least the minimum required. That is, when a builder offers a bid to a PoN relayer, said relayer will query the Builder Registry for the status of the builder and the validity of its stake.

The states available for a builder are `UNBEGUN`, `REGISTERED`, `EXIT_PENDING`, `EXITED`, and `SLASHED`. The proposer's `ACTIVE` status is represented in the Builder Registry by being `REGISTERED` and having enough stake. Thus, a builder can be registered but not operational. These are the meanings of each state:

- **UNBEGUN:** any address unknown to the PoN is given the `UNBEGUN` status.
- **REGISTERED:** a builder who has staked the required amount and had previous status `unbegun` will be given the `REGISTERED` status. This means that as long as the builder's stake is enough, it can offer bids to the PoN. In all other statuses, the builder cannot offer bids to the PoN relayers.

- **EXIT_PENDING:** a registered builder can position for exit and will be given the EXIT_PENDING status. After a cycle has passed to clear out any potential unreported wrongdoing from the builder, it can effectively exit the network. Builders with this status will not be able to submit bids.
- **EXITED:** after waiting for a cycle in the EXIT_PENDING status, a builder can trigger the exit from the PoN, and all remaining stake will be refunded.
- **SLASHED:** a registered or positioned for exit builder that is reported from committing a PBS rule violation. If, after paying for the penalty, the remaining stake is below the required threshold, the builder will be given the slashed status. If its previous state was registered, the builder could return to that state by staking the necessary amount⁸.

Similarly to proposers, to acquire a state, builders must have the previous one in the list, except for the slashed status. This can be achieved from the registered or exit pending status. Unlike the kicked status of proposers, if builders staked enough and are in the registered state before being slashed, the slashed status can be reverted.

Builder Registry Invariants

Invariant: A builder will have a positive stake balance only if it has no debt.

```
builder.stakedBalance > 0 → builder.debt = 0
```

Invariant: A builder can be in REGISTERED or EXIT_PENDING status only if it has no debt.

```
builder.status = REGISTERED → builder.debt = 0
```

```
builder.status = EXIT_PENDING → builder.debt = 0
```

Invariant: A builder will only have more stake than the minimum required if it's REGISTERED or EXIT_PENDING.

```
builder.stakedBalance ≥ minimumStakeAmount → builder.status = REGISTERED ∨  
builder.status = EXIT_PENDING
```

Invariant: The builders array contains all builders with a status different than UNBEGUN.

```
builders = {b: b.status ≠ UNBEGUN}
```

Invariant: A builder can only have status EXITED if the debt and stake of the builder are zero.

```
builder.status = EXITED → builder.debt = builder.stakedBalance = 0
```

⁸ The minimum required amount for a builder to be operational is $\min(\text{minStake}, \text{averagePays})$, where minStake is the minimum stake required by the builder registry and averagePays is the average of the last 100 payments to the payout pool.

Invariant: The ETH balance of the builder's registry contract is the sum of all builders' balances that are either `REGISTERED`, `SLASHED`, or `EXIT_PENDING`.

```
balance(BuilderRegistry) =  $\sum$  builder.stakedBalance [builder.status  $\in$  {REGISTERED, SLASHED, EXIT_PENDING}]
```

Reporter Registry

The Reporter Registry is where reporters register and submit reports on PBS rules violations. Anyone can register as a reporter and immediately submit reports. Reporters have three statuses: `INACTIVE`, `ACTIVE`, and `RAGEQUITTED`, which bear the following meaning:

- **INACTIVE:** default status for any address not registered as a reporter.
- **ACTIVE:** status for registered reporters. This is the only state in which reporters can submit reports previously validated and signed by the hosted service and earn rewards for doing it.
- **RAGEQUITTED:** formerly active reporters who declared themselves as no longer active.

Reports for the builders or proposers are filed through the Reporter Registry. The reporter also keeps track of the different hosted services that can sign reports. Note that it is only the owner who can make the registry acknowledge a hosted service. Reports signed by any service that is not registered in this registry will be considered invalid.

Reports

The only action of reporters in the PoN is to submit reports through the Reporter Registry. There are three types of penalties that reporters can report: `ProposerPenalized`, `BuilderPenalized`, or `ProposerDisqualified`. Their meaning is the following:

- **ProposerPenalized:** The reported proposer has violated rules 1, 2, or 3 described in the [Violation rules](#) section. This does not entail expulsion from the PoN by itself. However, should the maximum number of such reports be reached, the proposer would be expelled from the PoN.
- **BuilderPenalized:** The reported builder has violated rules 4 or 5 of the [Violation rules](#) section. Builders can never be expelled from the PoN. However, if the builder's stake is below the minimum amount after being penalized, it will be marked in status `SLASHED`. This will prevent the builder from offering bids until its stake is increased above the minimum required amount.
- **ProposerDisqualified:** The reported proposer has violated rule 6 of the [Violation rules](#) section. This kind of report only forfeits all the future deferred rewards in the pending cycle from the proposer but doesn't decrease the already credited or deferred rewards. The proposer reported with this type of penalty is immediately expelled from the PoN.

There are certain requirements for a report depending on the type of penalty being reported. These are some of the properties that should be satisfied by any report:

- For regular penalties to proposers or builders, the economic amount must be positive:
 $\text{ProposerPenalized} \vee \text{BuilderPenalized} \rightarrow \text{penaltyAmount} > 0$
- For the disqualifying penalty of a proposer, the economic amount must be zero:
 $\text{ProposerDisqualified} \rightarrow \text{penaltyAmount} = 0$
- Only proposers with status `ACTIVE` or `EXIT_PENDING` should be reported:
 $\text{ProposerPenalized} \vee \text{ProposerDisqualified} \rightarrow \text{proposer.status} \in \{\text{ACTIVE}, \text{EXIT_PENDING}\}$
- Only builders with status `REGISTERED`, `EXIT_PENDING`, or `SLASHED` should be reported.
 $\text{BuilderPenalized} \rightarrow \text{builder.status} \in \{\text{REGISTERED}, \text{EXIT_PENDING}, \text{SLASHED}\}$

Findings

AO1: `PayoutPool::processProposerSlashing()` incorrectly increases the rewards of the slashed proposer

[Severity: High | Difficulty: Low | Category: Logic]

When a proposer is slashed, there is an over-distribution of the slashed amount on the proposer side.

When a proposer is slashed, half of the deducted amount goes to the rest of the proposers on [line 290](#).

```
if (activeValidators > 1) {  
    /// The 2 in the denominator will divide the rewards as needed  
    _incrementCumulativeRewards(adjustedPenalty, 2 * activeValidators - 2);  
}
```

However, this increments the rewards available to *all* proposers, including the one being slashed. This presents two major problems:

- The slashed proposer has at its disposal a part of the slashed amount to claim or to repay further debts.
- The total amount available to claim for all proposers has increased by `activeValidators * (penaltyToShare / (activeValidators - 1)) > penaltyToShare` where `penaltyToShare = adjustedPenalty / 2`. Thus potentially creating a discrepancy between the total amount of rewards credited and the funds possessed by the contract.

Recommendation

Credit the shared penalty as claimed by the slashed proposer, as suggested here:

```
if (activeValidators > 1) {  
    /// The 2 in the denominator will divide the rewards as needed  
    _incrementCumulativeRewards(adjustedPenalty, 2 * activeValidators - 2);  
    latestProposerClaim[_blsPublicKey] += adjustedPenalty/(2 *  
                                                activeValidators - 2);  
}
```

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cddc3e42a72446/contracts/pbs/PayoutPool.sol#L312>.

AO2: PayoutPool::_repayDebt() incorrectly increases the rewards of the debtor proposer

[Severity: High | Difficulty: Low | Category: Logic]

This finding is similar to [AO1: PayoutPool::processProposerSlashing\(\) incorrectly increases the rewards of the slashed proposer](#).

Consider the following lines from `PayoutPool._repayDebt()`:

```
/// In case there are other active proposers in the pool we give the reward
to all of them split equally
if (activeValidators > 1) {
    cumulativePerProposerRewards += repayment / (activeValidators - 1);
}
```

If there are any active proposers, then the penalized proposer should repay them. Crucially, the penalized proposer should not repay *itself*. However, this is exactly what is happening.

Recommendation

To prevent this, the `latestProposerClaim` of the penalized proposer must be increased by the same amount as `cumulativePerProposerRewards`:

```
if (activeValidators > 1) {
    cumulativePerProposerRewards += repayment / (activeValidators - 1);
    latestProposerClaim[_blsPublicKey] += repayment / (activeValidators -
1);
}
```

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/PayoutPool.sol#L429>.

A03: PayoutPool::processProposerExit() incorrectly updates latestProposerClaim

[Severity: High | Difficulty: Low | Category: Logic]

The function `PayoutPool.processProposerExit()` is defined as follows:

```
function processProposerExit(bytes calldata _blsPublicKey) external
nonReentrant {
    require(address(PROPOSER_REGISTRY) == msg.sender, 'Only proposer
registry');

    (address feeRecipient,) =
PROPOSER_REGISTRY.getProposerAccounts(_blsPublicKey);

    uint256 claim = _repayDebt(_blsPublicKey);

    latestProposerClaim[_blsPublicKey] = claim; // <---- This should be +=

    if (claim > 0) {
        _transferETH(feeRecipient, claim);
    }

    emit ProposerExitProcessed(_blsPublicKey,
proposerToTotalDebt[_blsPublicKey]);
}
```

`latestProposerClaim` should be *increased* by claim and not set to it. Here, claim denotes the amount of claimable rewards. Thus, to make sure the same rewards cannot be claimed again, `claim` should be added to the already claimed rewards stored in `latestProposerClaim`.

Recommendation

Replace `latestProposerClaim[_blsPublicKey] = claim` with `latestProposerClaim[_blsPublicKey] += claim`.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/PayoutPool.sol#L280>.

AO4: PayoutPool::_repayDebt incorrect repayment distribution

[Severity: High | Difficulty: Low | Category: Logic]

The `PayoutPool::_repayDebt` function may not correctly distribute the debt accrued by a proposer.

In line [362](#) the debt payment is subtracted from the available rewards

```
availableRewards -= totalRepayment;
```

Then, `totalRepayment` is used to update the variables `latestProposerClaim` and `proposerToTotalDebt`. However, instead of distributing `totalRepayment` to the reporters and payout pool, what is being distributed is `availableRewards` (lines [374-413](#)).

Note that this also artificially diminishes the amount of debt-free rewards reported by the function.

Scenario

- Proposer `p` has `proposerToTotalDebt[p] > availableRewards > 0`, hence `totalRepayment = availableRewards` per line [361](#)
- Line [362](#) sets `availableRewards = 0` by doing `availableRewards -= totalRepayment`
- `totalRepayment > 0` is deducted from `proposerToTotalDebt[p]` in line [369](#)
- Since `availableRewards = 0`, the following if clause on line [375](#) is executed and the for loop doesn't distribute rewards to any reporter:

```
if (availableRewards == 0) {  
    break;  
}
```

- Similarly, line [401](#) sets `repayment = _min(proposerToPayoutPoolDebt[p], availableRewards) = 0`. This prevents repaying owed rewards to the payout pool.

Note that this is in the case of `proposerToTotalDebt[p] > availableRewards > 0`, but in general, if `proposerToTotalDebt[p] * 2 > availableRewards > 0`, the distributed debt will still be insufficient.

Recommendation

Either don't decrement `availableRewards` on line [362](#) since the owed debt is decremented throughout the function, or replace `availableRewards` with `totalRepayment` in the lines after line 362.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/PayoutPool.sol#L366>.

A05: PayoutPool::_computeRewardDelta() underflow

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

The function `PayoutPool._computeRewardDelta()` is defined as follows:

```
function _computeRewardDelta(bytes calldata _blsPublicKey) internal view
returns (uint256) {
    /// We are using the property here that upperBound >= lowerBound
    return _getUpperRewardBound(_blsPublicKey) -
    _getLowerRewardBound(_blsPublicKey);
}
```

The comment suggests the subtraction is safe because `upperBound >= lowerBound`. However, this is not always the case: When activating a proposer, the function `PayoutPool.setMinimalClaimBalance()` is called, which performs the following update:

```
uint256 reward = cumulativePerProposerDeferredRewards +
cumulativePerProposerRewards;
latestProposerClaim[_blsPublicKey] = reward;
```

If `cumulativePerProposerDeferredRewards` is non-zero, this means `latestProposerClaim[_blsPublicKey]` will be larger than `cumulativePerProposerRewards`, which in turn implies `_getLowerRewardBound() > _getUpperRewardBound()`. Thus, `_computeRewardDelta()` may fail with an underflow.

The consequence of this is that, for example, the following functions will fail with an underflow: `claimProposerPayout()`, `getClaimableProposerAmount()`, `_repayDebt()`. The failure persists until the next cycle when `cumulativePerProposerDeferredRewards` is added to `cumulativePerProposerRewards`.

While this seems to be only a temporary failure with no long-term consequences, it is unexpected.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cddc3e42a72446/contracts/pbs/PayoutPool.sol#L360>.

AO6: PayoutPool::processProposerSlashing() fails if there are no active proposers

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

When reporting a proposer, the function `PayoutPool.processProposerSlashing()` requires that there is at least one active proposer:

```
function processProposerSlashing(
    bytes calldata _blsPublicKey,
    uint256 _amount,
    address _reporter
) external nonReentrant {
    require(msg.sender == address(REPORTER_REGISTRY), 'Only reporter
registry');

    uint256 activeValidators = PROPOSER_REGISTRY.activateProposers();

    require(activeValidators > 0, 'No proposers for reporting');

    ...
}
```

However, proposers can also be slashed when they are in state `EXIT_PENDING`. Thus, if all proposers are `EXIT_PENDING`, then no proposer can be slashed.

Recommendation

Remove the requirement that there must be at least one active proposer.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cddc3e42a72446/contracts/pbs/PayoutPool.sol#L293>.

A07: `setContracts()` should only be callable once

[Severity: High | Difficulty: High | Category: Security]

The contracts `BuilderRegistry`, `ReporterRegistry`, `ProposerRegistry` and `PayoutPool` each have a function `setContracts()` that needs to be called during registration. However, at the moment, this function can be called any number of times, even after the contracts have been deployed. Since there is no reason to call `setContracts()` after deployment, it makes sense to ensure that it can only be called once.

Moreover, calling `setContracts()` after deployment could effectively change core properties of the network, such as which proposers are active, what hosted services are trusted or what builders are registered.

Recommendation

Update the `setContracts()` function so that it can only be called once at deployment time.

Status

Addressed by client as follows:

- PayoutPool:
<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/PayoutPool.sol#L145>
- ReporterRegistry:
<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ReporterRegistry.sol#L152>
- ProposerRegistry:
<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ProposerRegistry.sol#L133>
- BuilderRegistry:
<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L142>

AO8: `ProposerRegistry::kickProposer()` missing decrement of active validators

[Severity: High | Difficulty: Low | Category: Logic]

The `ProposerRegistry::kickProposer` function does not call the `_decrementActiveValidators` function. This would not be a problem if the proposer kicked were in `EXIT_PENDING` status since `_decrementActiveValidators` would have been called in the `positionForRagequit` function. However, if the kicked proposer were in `ACTIVE` status, the number of active validators would not be decreased.

This would entail incorrect accounting of rewards in the `PayoutPool` contract.

Recommendation

Update the function logic so that when a proposer in `ACTIVE` status is kicked the number of active proposers is decreased by one.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ProposerRegistry.sol#L378>.

A09: `ProposerRegistry::exitClaimAmount` may be set too high in `kickProposer()`

[Severity: High | Difficulty: Low | Category: Logic]

The function `ProposerRegistry::kickProposer()` updates the `exitClaimAmount` of the kicked proposer as follows:

```
blsPublicKeyToProposer[_blsPublicKey].exitClaimAmount =  
_getTotalPayoutUpperBound();
```

This is fine if the proposer being kicked is currently `ACTIVE`. However, if it is `EXIT_PENDING`, then the `exitClaimAmount` has already been set in the function `positionForRagequit()`. This is wrong because

- the `exitClaimAmount` set by `kickProposer()` may be larger than at the time `positionForRagequit()` was called, meaning the proposer essentially earns rewards for the time between `positionForRagequit()` and `kickProposer()`, but
- proposers in state `EXIT_PENDING` should *not* earn rewards anymore.

The rewards that such a proposer in state `EXIT_PENDING` earns are not accounted for, and if it is paid out, other proposers won't get their share.

Recommendation

Update the `kickProposer()` function so that it does not update `exitClaimAmount` if the proposer is already in state `EXIT_PENDING`.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ProposerRegistry.sol#L377>.

A10: Builders who are slashed in status `EXIT_PENDING` can neither top up nor exit

[Severity: Medium | Difficulty: Low | Category: Design]

The function `BuilderRegistry.topUp()` contains the following requirement:

```
require(  
    builder.status == BuilderStatus.REGISTERED || (builder.status ==  
    BuilderStatus.SLASHED && builder.exitBlock == 0),  
    'Builder status invalid'  
);
```

This means that a builder who was slashed while in status `EXIT_PENDING` cannot repay its debt. Since exiting requires that all debts are repaid, this also means that the builder cannot exit.

Recommendation

Allow all `SLASHED` builders to top up.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L214>.

A11: Builders cannot be reported again while being slashed

[Severity: Medium | Difficulty: Low | Category: Design]

The function `BuilderRegistry.slashBuilder()` contains the following conditions:

```
require(isBuilderReportable(_reporter), 'Invalid slasher position');
```

This implies that a builder who is SLASHED cannot be reported.

Scenario

Assume a builder commits two violations. Further, assume that when the first violation is reported, the builder is set to SLASHED. Now, if the builder does not top up within one cycle, then the second violation cannot be reported anymore because violations older than one cycle cannot be reported. Thus, the builder avoided the penalty for the second violation.

One further note: Assume for a second that it is possible to report a builder twice while it is slashed. Even then, the builder can avoid the second report by quickly exiting the PoN before the second report is submitted (SLASHED builders can exit immediately without waiting). In general, this does not seem to represent a problem since a builder who is about to leave has no incentive to top up. However, this might affect the social aspect of the protocol since, in this case, the community would not be aware of the fact that the builder was actually slashed twice and not just once.

Recommendation

Allow slashed builders to be reported.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L425>.

A12: Cannot report builder with insufficient stake

[Severity: Medium | Difficulty: Medium | Category: Design]

When reporting a builder with `ReporterRegistry.submitReport()`, it is required that `BuilderRegistry.isBuilderReportable()` is satisfied for that builder:

```
function isBuilderReportable(address _builder) public view returns (bool) {
    return isBuilderOperational(_builder) ||
    _isPositionedForExitAndNotSlashed(_builder);
}
```

If we want to report a builder that has not positioned for exit this means the builder needs to be operational:

```
function isBuilderOperational(address _builder) public view returns (bool)
{
    bool cond1 = addressToBuilder[_builder].status ==
    BuilderStatus.REGISTERED;
    bool cond2 = addressToBuilder[_builder].balanceStaked >=
    getMinimumStakeAmount();

    return cond1 && cond2;
}
```

As is shown above, it is required that the builder's staking balance needs to match the minimum staking balance. Otherwise, submitting the report fails.

Thus, if the minimum staking amount increases beyond a builder's staking balance after committing a reportable offense and stays that way for one cycle, then it becomes impossible to report that builder.

Recommendation

Allow for builder reporting regardless of their stake.

Status

Addressed by client as follows:

- Builder reportability checks were updated here:
<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L425>

- Additional logic tracking multiple debts for builders can be found here:
<https://github.com/bswap-eng/proposer-builder-registry/blob/6085aba132ca13287fd164b5f4bd228fb2a15c8f/contracts/pbs/BuilderRegistry.sol#L329>

A13: Checking that address is not a contract is not reliable

[Severity: Low | Difficulty: Medium | Category: Input Validation]

Both `BuilderRegistry.registerBuilder()` and `ReporterRegistry.registerReporter()` require that a given address is not a contract using the following requirement:

```
require(!AddressUpgradeable.isContract(addr), 'Must not be a contract');
```

There are ways of bypassing this check, for instance, by calling the `ReporterRegistry.registerReporter()` function in the constructor of a contract.

Recommendation

Since the goal is that these addresses can issue signatures, requiring instead that the addresses sign their own address could be a way of ensuring that only non-contract addresses are allowed to execute these functions.

Status

Acknowledged by client.

A14: PBSFactory::deployContracts() payoutCycleLength can have different values for different contracts

[Severity: High | Difficulty: High | Category: Input Validation]

The function `deployContracts()` from the `PBSFactory` deploys and sets the parameters for the registries and the Payout Pool. They all should share the same payout cycle length.

However, it is never checked that the different values passed to the contracts for initialization is the same.

Recommendation

Require before deployment that the payout cycle length will be the same across all contracts.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/6085aba132ca13287fd164b5f4bd228fb2a15c8f/contracts/PBSFactory.sol#L116>.

Informative findings

BO1: PayoutPool::_payoutReporter() does not follow the checks-effects-interactions pattern

[Severity: - | Difficulty: - | Category: Security]

The function `PayoutPool._payoutReporter()` does not follow checks-effects-interactions pattern:

```
function _payoutReporter(address _reporter) internal {
    uint256 reporterComp = reporterRewardsAccumulated[_reporter];

    /// Trigger the transfer in case the amount is above 0
    if (reporterComp > 0) {
        _transferETH(_reporter, reporterComp);
        delete reporterRewardsAccumulated[_reporter];

        emit ReporterPaid(_reporter, reporterComp);
    }
}
```

Here, `_transferETH()`, which may potentially trigger an untrusted external function call, is executed before the `delete` in the following line.

Recommendation

To follow the checks-effects-interactions pattern, these lines should be swapped:

```
delete reporterRewardsAccumulated[_reporter];
_transferETH(_reporter, reporterComp);
```

Following the checks-effects-interactions pattern ensures that reentrancy does not cause problems. Note that since `_payoutReporter()` is only called by `nonReentrant` functions, problems related to reentrancy should already be prevented and the change suggested here is not absolutely necessary. On the other hand, since the change is so small, one might as well follow the checks-effects-interactions pattern.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/PayoutPool.sol#L628>.

Bo2: `PayoutPool::receive()` emits an event containing only taxed payment

[Severity: - | Difficulty: - | Category: -]

The function `PayoutPool.receive()` is executed whenever a builder pays the bid amount they promised. In this process, the following event is emitted:

```
emit BlockPayment(builder, rest, block.number);
```

where `rest` denotes the amount of ETH sent by the builder with the maintenance fee deducted.

It may make sense to also include the original amount sent by the builder before any fees have been deducted. This value is important to decide whether the builder has paid the full bid amount.

Status

The client states that off-chain this can be easily fetched by simply checking the `value` field (i.e. `msg.sender`).

BO3: PayoutPool::receive() excessive scheduling of proposers can lead to bad reward distribution

[Severity: High | Difficulty: High | Category: Functional Correctness]

Assume there are more than a hundred validators scheduled to be activated for a single block. If that block contains payment to the Payout Pool through the `receive` function, some of the scheduled proposers will not get their due rewards.

This is because only a hundred proposers can be activated at a time. Thus, if there are more than a hundred proposers scheduled for a reward block, it may happen that not all proposers are activated, and hence the non-activated proposers would miss rewards.

Status

Acknowledged by the client.

Bo4: BuilderRegistry.slashBuilder() contains redundant variable

[Severity: - | Difficulty: - | Category: Optimization]

The function `BuilderRegistry.slashBuilder()` contains the following code:

```
uint256 builderStake = addressToBuilder[_builder].balanceStaked;
uint256 amountAvailableToSlash = _min(builderStake, _slashAmount);

unchecked {
    addressToBuilder[_builder].balanceStaked -= amountAvailableToSlash;
}

uint256 expectedReporterReward = getMinimumStakeAmount() / 2;
uint256 expectedPoolReward = _deltaCut(_slashAmount,
expectedReporterReward);

uint256 actualReporterReward = _min(expectedReporterReward,
amountAvailableToSlash);
uint256 actualPoolReward = amountAvailableToSlash - actualReporterReward;

uint256 payment = actualPoolReward + actualReporterReward;
```

Note that `payment` is always equal to `amountAvailableToSlash`:

```
payment == actualPoolReward + actualReporterReward
        == (amountAvailableToSlash - actualReporterReward) + actualReporterReward
        == amountAvailableToSlash
```

Hence, all references to `payment` can be replaced with `amountAvailableToSlash`, and `payment` can be removed.

Status

The client states that the `payment` variable was removed here

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L300>.

Bo5: `ProposerRegistry.activateProposers()` can be simplified

[Severity: - | Difficulty: - | Category: Optimization]

The function `ProposerRegistry.activateProposers()` contains special code for the case where no proposers have been activated yet. However, the function can be simplified such that this special code is not needed.

A simplified version of the code could look as follows:

```
function activateProposers() public returns (uint256) {
    /// Limiting updates to a 100 proposers per transaction
    uint256 l = _min(100, proposers.length - historicalActivatedProposers);

    if (l == 0) {
        return activeValidators;
    }

    uint256 endIndex = historicalActivatedProposers + 1;
    uint256 currentBlock = block.number; /// Saved for gas optimization

    uint256 curIndex = historicalActivatedProposers;
    for (; curIndex < endIndex; ++curIndex) {
        bytes memory proposer = proposers[curIndex];

        if (blsPublicKeyToProposer[proposer].activationBlock > currentBlock) {
            break;
        }

        /// Making sure that newly activated proposers don't get previously earned
        /// rewards (including pending)
        PAYOUT_POOL.setMinimalClaimBalance(proposer);
        blsPublicKeyToProposer[proposer].status = ProposerStates.ACTIVE;

        emit ProposerActivated(proposer);
    }

    // curIndex refers to the first non-active validator
    historicalActivatedProposers = curIndex;
    activeValidators += curIndex - historicalActivatedProposers;

    return activeValidators;
}
```




Status

Acknowledged by client.

Bo6: `ProposerRegistry::isProposerOperational()` inconsistent returns for single and batch functions

[Severity: - | Difficulty: - | Category: -]

In the `ProposerRegistry` contract, the functions `isProposerOperational()` and `checkBatchOperationalStatus()` return different checks. For `isProposerOperational()` it is checked that

```
blsPublicKeyToProposer[_blsPublicKey].status == ProposerStates.ACTIVE;
```

While for a `checkBatchOperationalStatus()` it is checked that

```
blsPublicKeyToProposer[_blsKeys[i]].status == ProposerStates.REGISTERED;
```

Recommendation

Since the only apparent difference in these functions seems to be the batch vs. single, have them return the same check.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/6085aba132ca13287fd164b5f4bd228fb2a15c8f/contracts/pbs/ProposerRegistry.sol#L205>.

B07: `ProposerRegistry::positionForRagequit()` redundant check

[Severity: - | Difficulty: - | Category: -]

In the function `positionForRagequit()` of the Proposer Registry it is first checked that

```
require(proposer.status == ProposerStates.ACTIVE, 'Incorrect Status');
```

However, lines after, it is checked that

```
require(isProposerOperational(_blsPublicKey), 'Non-operational');
```

But the function `isProposerOperational` also requires that the status of the proposer be `ACTIVE`.

Recommendation

Remove one of these two checks.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/f7a1bb9ca4342dddfcedaa7bd7ab598e7fe922a1/contracts/pbs/ProposerRegistry.sol#L323>.

Bo8: Slashing builder who is EXIT_PENDING always transitions to SLASHED

[Severity: - | Difficulty: - | Category: Design]

The function `BuilderRegistry.slashBuilder()` contains the following code:

```
/// In case the minimum stake threshold is not reached after slashing
if (!isBuilderOperational(_builder)) {
    addressToBuilder[_builder].status = BuilderStatus.SLASHED;
}
```

The comment suggests that a builder should only be set to `SLASHED` if its staking balance drops below the minimum amount. However, if the builder is in state `EXIT_PENDING`, then the above code will always set the builder to `SLASHED`, irrespective of the builder's staking balance. This differs from builders in state `REGISTERED`, who will only be set to `SLASHED` if their balance drops below the minimum.

This doesn't seem to affect the exit procedure and thus may not be a problem, but it's a bit inconsistent.

Status

The client states that the updated `isBuilderOperational` function is presented here:

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L389>.

Bo9:

BuilderRegistry::_isPositionedForExitAndNotSlashed()) contains redundant condition

[Severity: - | Difficulty: - | Category: -]

The function `BuilderRegistry::_isPositionedForExitAndNotSlashed()` is defined as follows:

```
function _isPositionedForExitAndNotSlashed(address _builder) internal view
returns (bool) {
    /// Check if the builder exit is pending, and if no debt is present
    (balance can be slashed)
    bool cond1 = addressToBuilder[_builder].status ==
BuilderStatus.EXIT_PENDING;
    bool cond2 = addressToBuilder[_builder].exitBlock < block.number;
    bool cond3 = builderToDebt[_builder].reporter == address(0);

    return cond1 && cond2 && cond3;
}
```

Here, `cond3` is actually redundant and can be removed. This is because if a builder is in state `EXIT_PENDING`, then it cannot have any debt.

Status

The client states that in the updated code, this function was removed all together, and another function checking reportability was introduced:

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/BuilderRegistry.sol#L425>.

B10: Inconsistent use of < and <= with ReporterRegistry::reporterToLastReportedBlock()

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

The function `ReporterRegistry.ragequitReporter()` has the following requirement:

```
require(reporterToLastReportedBlock[reporter] + 100 < block.number,  
'Rewards still pending');
```

In contrast, in `ReporterRegistry.submitReport()`, withdrawals are only allowed if

```
if (_withdraw && reporterToLastReportedBlock[reporter] + 100 <=  
block.number)
```

Note the inconsistent use of < and <= between these two functions.

Recommendation

Use the same comparison operator on both functions.

Status

Addressed by client in:

1. <https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ReporterRegistry.sol#L228>
2. <https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ReporterRegistry.sol#L261>

B11: Builders and reporters need to pay gas for `activateProposers()` and `_repayDebt()`

[Severity: - | Difficulty: - | Category: -]

Submitting a report via `ReporterRegistry.submitReport()` can trigger further calls to `ProposerRegistry.activateProposers()` and `PayoutPool._repayDebt()`. Similarly, builders may trigger a call to `activateProposers()` when paying the payout pool.

Especially `activateProposers()` may be quite gas-expensive, and calling it in the above cases may not be strictly necessary. And it may seem a bit unfair to require a reporter -- who may just want to submit a report -- to also pay for proposer activation.

Status

The client states the following: the reason behind that is all of the off-chain actors act as if each proposer is activated EXACTLY 1575 epochs from registering, and will expect the smart contracts to treat it as such. For this exact reason, we need to activate all possible existing proposers (up to 100 at a time) every time a state-changing function is called to match these expectations.

B12: `isReporterActive` and `isReporterRagequitted` can be combined into single status mapping

[Severity: - | Difficulty: - | Category: -]

`ReporterRegistry` uses the following two mappings to keep track of reporter states:

```
mapping(address => bool) public isReporterActive;  
mapping(address => bool) public isReporterRagequitted;
```

These mappings could be combined into a single mapping like the following:

```
mapping(address => ReporterStatus) public reporter;
```

where `ReporterStatus` is an enum with the fields `INACTIVE`, `ACTIVE`, `RAGEQUITTED`.

This has several advantages:

- It mimics the way that similar state is stored in the other registries
- It makes it impossible to reach invalid program states, like, for example, where `isReporterActive` is false and `isReporterRagequitted` is true
- `isReporterOperational()` can be reduced to a single storage access

Status

The client states the following: Since, currently, the off-chain infrastructure utilizes the existing data structures, we will have to postpone this upgrade to the later versions since this is not a logic error.

B13: ReporterRegistry::submitReport() can be slightly refactored

[Severity: - | Difficulty: - | Category: Gas Optimization]

The function ReporterRegistry.submitReport() contains the following code:

```
if (_report.penaltyType == PenaltyType.ProposerPenalized ||
    _report.penaltyType == PenaltyType.ProposerDisqualified) {
    /// Increment the proposer report count
    kick = PROPOSER_REGISTRY.reportProposer(_report.blsKey);

    /// Call the payout pool to route the funds around, no ETH is
    transferred here only internal accounting adjusted
    PAYOUT_POOL.processProposerSlashing(
        _report.blsKey,
        _report.amount,
        reporter
    );
}

else {
    /// Slashes the builder and transfers the funds to the payout pool
    BUILDER_REGISTRY.slashBuilder(reporter, _report.builder,
    _report.amount);
}

if (_report.penaltyType == PenaltyType.ProposerDisqualified || kick) {
    PROPOSER_REGISTRY.kickProposer(_report.blsKey);
}
```

Note that the second if-statement is only important if we are considering a report for a proposer. Thus, it can be moved into the first if-statement (which also saves a bit of gas):

```
if (_report.penaltyType == PenaltyType.ProposerPenalized ||
    _report.penaltyType == PenaltyType.ProposerDisqualified) {
    /// Increment the proposer report count
    kick = PROPOSER_REGISTRY.reportProposer(_report.blsKey);

    /// Call the payout pool to route the funds around, no ETH is
    transferred here only internal accounting adjusted
    PAYOUT_POOL.processProposerSlashing(
```

```
        _report.blsKey,  
        _report.amount,  
        reporter  
    );  
  
    if (_report.penaltyType == PenaltyType.ProposerDisqualified || kick)  
{  
        PROPOSER_REGISTRY.kickProposer(_report.blsKey);  
    }  
}  
  
else {  
    /// Slashes the builder and transfers the funds to the payout pool  
    BUILDER_REGISTRY.slashBuilder(reporter, _report.builder,  
    _report.amount);  
}
```

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/924642acc177e67cfd840dc768cdcc3e42a72446/contracts/pbs/ReporterRegistry.sol#L248>.

B14: Contracts do not prevent a proposer from being reported for a violation that occurred after initiating the exit process

[Severity: Medium | Difficulty: High | Category: Logic]

Once a proposer initiates the exit process by calling `ProposerRegistry.positionForRagequit()`, the proposer cannot participate in the PoN anymore. Thus, the proposer should never be able to be reported for an action that happened *after* the exit process is initiated.

However, at the moment, this is not enforced on-chain.

Note that this is not about preventing proposers to be reported for any wrongdoing they committed *before* starting the exit process. Rather, to prevent reporting from wrong reports of wrongdoing *after* initiating the exit process.

Scenario

- Proposer is chosen for a block when it's in `EXIT_PENDING`.
- Proposer doesn't pay the PoN in the mentioned block (because it does not have to anymore).
- A report is wrongly generated to penalize the proposer.
- Since the proposer is still in `EXIT_PENDING` the contracts would accept the report and penalty because no check is being made on-chain about if the proposer was `ACTIVE` when for the reported block.

Recommendation

Record the block in which proposers initiate the exit process (or compute it) and require reports that the reported block is prior to the exit block.

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/f7a1bb9ca4342dddfcedaa7bd7ab598e7fe922a1/contracts/pbs/ProposerRegistry.sol#L337>.

B15: Input validation for submitting reports

[Severity: Medium | Difficulty: High | Category: Input Validation]

All reports are submitted through the function `ReporterRegistry::submitReport`. The function performs some checks, such as the report being signed by a valid hosted service or the reported builder being reportable. However, there are no requirements that check that the report is correctly constructed.

Even if the hosted service is a trusted party, it will enhance reporting security to check for properties such as the ones mentioned in the section [Reports](#) or along the following lines:

- If the penalty is `ProposerPenalized` or `ProposerDisqualified`, then `_report.builder = address(0)`
- If the penalty is `BuilderPenalized` then `_report.blsKey = 0`

Status

Addressed by client in

<https://github.com/bswap-eng/proposer-builder-registry/blob/226866dee21a94a5f39d6bdac873468f1f86b433/contracts/pbs/ReporterRegistry.sol#L261>.

Appendix: Proof of Neutrality Off-Chain Model

Contents

1	Agents	61
1.1	Builder	61
1.2	Proposer	61
1.3	Relayer	61
2	State Structures	62
2.1	Bid structures	62
2.1.1	Cryptographic structure	62
2.1.2	Bid	63
2.1.3	Bid sets	63
2.2	Relayer structure	63
2.3	Auction set and function	64
3	State Definition	65
3.1	On-chain state	65
3.2	Off-chain state	66
3.2.1	Derived sets	66
3.3	Initial state	66
4	Agents actions	67
4.1	Builder actions	67
4.1.1	Bid offer	67
4.2	Proposer actions	68
4.2.1	Request block header	68
4.2.2	Sign block header	69
4.3	Relayer actions	70
5	Agent Faults and Penalties	70
5.1	Fault rules	70
5.1.1	Block request	71
5.1.2	Block signature	71
5.1.3	Block broadcasting	72
5.1.4	Proposer health	73
5.2	Rule checking	73

Abstract

The PoN network has off-chain components that affect how the on-chain code is interpreted. We present a mathematical model of the expected behavior of the off-chain side of the PoN.

1 Agents

All off-chain agents of the PoN should also be on-chain parties. Even agents without an active on-chain role, such as relayers, should be acknowledged on-chain. That is, every part of the PoN network –proposers, builders, relayers, reporters, and hosted services– should be known on-chain.

The only agents that the model considers are builders, proposers and relayers. Reporters and hosted service are not modeled. Instead, we give an account of what constitutes a penalty. Penalties should then be validated by the hosted service and reported to the PoN on-chain side.

We define $ts \in \mathbb{Q}$ as the current timestamp in decimal precision. Note that we assume that all agents and states are in sync with this variable.

1.1 Builder

Let **Addr** be the set of all ECDSA addresses and assume a fixed PoN network. We identify any ECDSA address with a builder. That is, we assume that any address can interact with the PoN network as a builder. It is the PoN’s duty to only process rightful interactions. As such, we will consider any potential builder address and discharge all of the builders’ states to the on-chain state.

We will refer to the set of all builders as \mathcal{B} . However, formally we have that $\mathcal{B} = \mathbf{Addr}$.

We will also refer to the “empty” builder as $0_{\mathcal{B}}$. That is, $0_{\mathcal{B}}$ is a placeholder builder to indicate no actual builder (identified with the zero address).

1.2 Proposer

Let **BLS** be the set of all BLS keys. We identify proposers with BLS keys registered in the Ethereum consensus layer. We discharge all proposer book-keeping to the on-chain state. Thus, the status (unregistered, registered, active, exit_pending, exited, kicked) and the validator’s stake are queried to the on-chain part of the PoN.

We will refer to the set of all proposers as \mathcal{P} . However, formally we have that $\mathcal{P} = \mathbf{BLS}$.

We will also refer to the “empty” proposer as $0_{\mathcal{P}}$. That is, $0_{\mathcal{P}}$ is a placeholder proposer to indicate no actual proposer.

1.3 Relayer

Relayers are central to the model. They contain functions that allow for state updates (allowing to compute if certain conditions hold) and functions that represent the state of the off-chain PoN’s side.

While relayers don’t perform any state-updating action, they validate state updates and store an important part of the state.

The relayer structure is defined in Subsection 2.2, once the necessary definitions have been given. We will denote the set of all relayers by \mathcal{R} .

2 State Structures

The following are the necessary structures and functions to describe the off-chain state.

2.1 Bid structures

Bid structures are a formalism to capture the relevant information about a bid. Before defining what is a bid, we need to define its cryptographic components. A crucial component of the PoN is the RPBS scheme, which allows relayers to check that the block offered by a builder contains the promised payment. Thus, before defining a bid, we must define its cryptographic components.

2.1.1 Cryptographic structure

The cryptographic structure \mathbb{C} contains the relevant cryptographic information for the RPBS scheme. This includes:

- Block header of the offered block for the slot.
- Proof of payment from the builder.
- Signed block header by the relayer.
- Signed proof of payment by the relayer.

After the builder submits a block, the relayer must check that the proof of payment guarantees the bid offer. If the check is successful, the relayer signs both the block header and the proof of payment. However, the signed block header and proof of payment are stored in the relayer structure, not on the bid structure.

With this in mind, we define

$$\mathbb{C} = \mathbb{H} \times \text{RPBS}$$

where:

- \mathbb{H} is the set of hashes.
- RPBS is a placeholder set for the necessary components of the RPBS proof of payment. We don't explicitly model nor describe what the necessary components are.

We denote $0_{\mathbb{C}} = (0_{\mathbb{H}}, 0_{\text{RPBS}})$ as the null element of the set \mathbb{C} , and analogously for each of its components.

Given $rpbs \in \mathbb{C}$, we identify its components as follows:

$$\begin{aligned} rpbs = (\text{header} \in \mathbb{H}, & \quad \text{Header of the offered block} \\ \text{PoP} \in \text{RPBS}) & \quad \text{Proof of payment provided by the builder} \end{aligned}$$

We will refer to each component of a $rpbs$ by $rpbs.component$ (e.g., $rpbs.header$ is the first component of the tuple $rpbs$).

2.1.2 Bid

We define the set of bids as

$$Bid = \mathcal{B} \times \mathcal{R} \times \mathbb{N} \times \mathbb{N} \times \mathbb{C}.$$

Given a bid $bid \in Bid$, we will identify its components as follows:

$bid = (builder \in \mathcal{B},$	Builder author of the bid
$relayer \in \mathcal{R},$	Relayer to which the bid is submitted
$promise \in \mathbb{N},$	Reward promised for block inclusion
$slot \in \mathbb{N},$	Slot to bid for
$rpbs \in \mathbb{C})$	Cryptographic components of the PoN

We will refer to each component of a bid bid by $bid.component$ (e.g., $bid.builder$ is the first component of the tuple bid).

Also, we denote the empty bid as

$$0_{Bid} = (0_{\mathcal{B}}, 0_{\mathcal{R}}, 0_{\mathbb{N}}, 0_{\mathbb{N}}, 0_{\mathbb{C}}).$$

2.1.3 Bid sets

- Given a slot $s \in \mathbb{N}$, the *bid set of s* is

$$Bid_s = \{b \in Bid \mid b.slot = s\}$$

that is, the bids corresponding to the slot number.

- Similarly, the *relayer bids* for a slot s and a relayer r is defined as

$$Bid_{s,r} = \{b \in Bid_s \mid b.relayer = r\}.$$

That is, given a relayer r , $Bid_{s,r}$ is the set of bids that r got for slot s .

Now we can define the strategy choice function of a relayer r as:

$$\begin{aligned} c_r : \mathbb{N} &\longrightarrow Bid \\ s &\longmapsto b \in Bid_{s,r} \end{aligned}$$

Note that we're implicitly evaluating the first argument of c to r , the explicit type is

$$c : R \rightarrow \mathbb{N} \rightarrow Bid.$$

2.2 Relayer structure

We identify relayers with a fixed finite set $R \subset \mathbb{N}$. We define the relayer structure as

$$\mathcal{R} = (R, c, schedule, checkRPBS, validBid, signedPoP, signedHeader, request, receive).$$

As mentioned in 1.3, relayers have two kinds of functions: computing and state functions. Computing functions are used to deterministically compute an

output for any given input, whereas state functions will be updated depending on the actions of the other agents. Section 4 explains how state functions can be modified.

Computing functions:

- c choice function representing the selecting strategy implemented by the relayer (type given later).
- $schedule : \mathbb{Q} \rightarrow \{\text{BID, SIGN, BROADCAST}\}$ schedule allocation of a given timestamp. Represents if a timestamp falls under the bid, requesting and signing, or broadcasting blocks time windows.
- $checkRPBS : RPBS \times \mathbb{Q} \times \mathcal{B} \rightarrow \mathbb{B}$ payment commitment check. Given a proof of payment, a promised amount, and a builder, the function checks if the proof of payment for the builder paying that amount is valid.
- $validBid : Bid \rightarrow \mathbb{B}$ a bid validation function. Each relayer may have different criteria to filter which bids are valid and which are not.

State functions:

- $signedPoP : Bid \rightarrow \mathbb{S}$ records the signed proof of payment for a given bid. If $signedPoP(b) \neq 0_{\mathbb{S}}$ it means the proof of payment submitted to the relayer is valid.
- $signedHeader : Bid \rightarrow \mathbb{S}$ records the signed block header by the relayer for a given bid. If $signedHeader(b) \neq 0_{\mathbb{S}}$ it means the bid submitted to the relayer is valid.
- $request : \mathbb{N} \rightarrow \mathbb{B}$ records if the designated proposer requested a block for a given slot.
- $receive : \mathbb{N} \rightarrow \mathbb{S}$ records the signed block returned by the designated proposer for a given slot.

We refer to the “empty” relayer as $0_{\mathcal{R}}$. That is, $0_{\mathcal{R}}$ is a placeholder relayer to indicate no actual relayer. We denote the set of active relayers as $\mathcal{R}^+ = \{r \in \mathcal{R} \mid r \neq 0_{\mathcal{R}}\}$.

Notation remark: Throughout this document, we will use the following convention. Given a set **Carrier** and a structure $S = (\mathbf{Carrier}, f_1, f_2, \dots)$ with $f_i : \mathbf{Carrier} \rightarrow A_i$, we will identify $s \in S$ with $s \in \mathbf{Carrier}$ and $s.f_i$ with $f_i(s)$ for $s \in S$. E.g., for $r \in \mathcal{R}$, we have that $r.receive : \mathbb{N} \rightarrow \mathbb{S}$ and $receive : \mathcal{R} \rightarrow \mathbb{S}^{\mathbb{N}}$. In the above definition, we implicitly omitted the “ $\mathcal{R} \rightarrow$ ” part of the type for clarity. Relayers can be thought of as indexes for the functions described above.

2.3 Auction set and function

An auction for a slot is the set of bids offered for a slot. The auction function represents an indexing by slot number of the relevant auction information for said slot.

We define the auction set $A = (2^{Bid})^{\mathbb{N}}$.

That is, A is the set of functions from the natural numbers to sets of bids 2^{Bid} . Each function $Auction \in A$ represents a possible state of every auction of the PoN network. For each slot, $s \in \mathbb{N}$, $Auction(s)$ represents a possible auction for that slot. We will denote $Auction(s)$ by $Auction_s$.

3 State Definition

In this section, we define the functions that update the state variables. But before defining which are the state-changing functions, we must define what is the state.

3.1 On-chain state

The off-chain state should always be driven by the on-chain state. However, we consider the on-chain state as a separate state from the PoN off-chain state. We represent only the necessary components of the on-chain state that are needed from the off-chain side. We do not model how these on-chain components are updated. That is, we assume they are updated independently of the off-chain state.

The on-chain parameters are modeled as state variables or functions. We distinguish between general beacon chain parameters on the one hand, and parameters specific to the PoN on the other hand.

General beacon chain parameters:

- $slot \in \mathbb{N}$ current slot of the beacon chain.
- $proposerForSlot : \mathbb{N} \rightarrow \mathcal{P}$ chosen proposer by the consensus layer for a given slot. We model it as $\forall n \in \mathbb{N} : n \leq slot, proposerForSlot(n) \neq 0_{\mathcal{P}}$ and $\forall n \in \mathbb{N} : n > slot, proposerForSlot(n) = 0_{\mathcal{P}}$.
- $proposerStake : \mathcal{P} \rightarrow \mathbb{Q}$ effective balance of a proposer.
- $balance : \mathbf{Addr} \rightarrow \mathbb{Q}$ ETH balance of an account.
- $EIP1559Fee : \mathbb{N} \rightarrow \mathbb{Q}$ EIP1559 priority fee for a given slot.
- $signedBlocks : \mathbb{N} \rightarrow \mathbb{N}$ number of signed blocks detected by the consensus layer for a given slot.

PoN-specific parameters:

- $proposerStatus : \mathcal{P} \rightarrow \{\text{UNREGISTERED, REGISTERED, ACTIVE, EXIT_PENDING, EXITED, KICKED}\}$ status in the PoN of a given proposer. Note that $proposerStatus(0_{\mathcal{P}}) = \text{UNREGISTERED}$.
- $isBuilderOperational : \mathcal{B} \rightarrow \mathbb{B}$ returns $\text{BuilderRegistry} :: \text{isBuilderOperational}(builder)$ for $builder \in \mathcal{B}$ at ts . Note that $isBuilderOperational(0_{\mathcal{B}}) = \perp$.
- $paidInSlot : \mathbb{N} \times \mathcal{B} \rightarrow \mathbb{Q}$ amount paid by a builder for a given slot to the payout pool through the $\text{PayoutPool} :: \text{receive}()$ function.

We consider this its own transition system that runs in parallel with the PoN state.

3.2 Off-chain state

We define the Proof of Neutrality Network state as a tuple $(\mathcal{R}, Auction)$ where \mathcal{R} is the current relay structure and $Auction : \mathbb{N} \rightarrow 2^{Bid}$ is the current auction function.

The state represents the configuration of the relayers and how many bids were offered for a given slot. The state transition functions described in this section represent how can the state of the PoN be updated.

Note that the beacon chain state is not included in the off-chain PoN state. This is because we want to separate the off-chain workings of the PoN from the current state of the beacon chain.

3.2.1 Derived sets

From the off-chain state, we can derive the following useful sets.

Relayers with bids

We define the set of relayers who received bids offers for a slot s as

$$\mathcal{R}_s^{bids} = \{bid.relayer \in \mathcal{R} : bid \in Auction_s\}.$$

Relayers with proposers requests

We define the set of relayers who received a request by a proposer for a slot s as

$$\mathcal{R}_s^{request} = \{relayer \in \mathcal{R} : relayer.request(s) = \top\}.$$

Relayers with received signed headers

We define the set of relayers who received a signed header for a slot s as

$$\mathcal{R}_s^{receive} = \{relayer \in \mathcal{R} : relayer.receive(s) \neq 0_S\}.$$

3.3 Initial state

The initial state of the PoN is as follows:

- $\forall r \in \mathcal{R} \forall s \in \mathbb{N} r.request(s) = \perp$ No relayers received requests.
- $\forall r \in \mathcal{R} \forall s \in \mathbb{N} r.receive(s) = 0_S$ No relayers received signed block headers.
- $\forall r \in \mathcal{R} \forall b \in Bid r.signedPoP(b) = 0_S$ No proof of payment is valid.
- $\forall r \in \mathcal{R} \forall b \in Bid r.signedHeader(b) = 0_S$ No bids are valid.
- $\forall n \in \mathbb{N} Auction_s = \emptyset$ No bids have been offered for any slot.

The rest of the relayers' functions compute a state-independent output for any given input.

4 Agents actions

In this section, we describe the different actions that agents can do. Currently, the model contains three state-updating actions: offering bids to a relayer (builder), requesting bids from a relayer (proposer), and sending back to relayers signed bids (proposer).

When there's a state update $(\mathcal{R}, Auction) \rightarrow (\mathcal{R}', Auction')$ by \mathcal{R}' we refer to the relayer structure with the same relayers (R) but with (possibly) different functions. Thus, for any $r \in \mathcal{R}$ we have that $r \in \mathcal{R}'$. When there's room for confusion, we'll represent $r \in \mathcal{R}'$ by $r' \in \mathcal{R}'$. The key difference between r and r' is that we may have (for example) $r.receive(n) \neq r'.receive(n)$ for a slot $s \in \mathbb{N}$, but they represent the same relayer.

4.1 Builder actions

The only action a builder can take in the PoN network from the off-chain side is to submit a bid for a slot.

4.1.1 Bid offer

The function `bid` takes as input $bid \in Bid$, where $bid.builder$ is the builder submitting bid to the PoN. It updates the state as follows:

$$(\mathcal{R}, Auction) \xrightarrow{\text{bid}(bid)} (\mathcal{R}', Auction')$$

Preconditions:

- $bid.slot = slot$ (bid is for the correct slot)
- $proposerStatus(proposerForSlot(p)) = \text{ACTIVE}$ (slot proposer is active in the PoN)
- $isBuilderOperational(bid.builder) = \top$ (builder proposing the bid is operational)
- $bid.relayer.schedule(ts) = \text{BID}$ (PoN relayer is taking bids at the time)
- $bid.promise > 0$ (builder promises positive payment)¹
- $|\{b \in Auction_s : b.builder = bid.builder\}| < 2$ (the builder has already bid the maximum amount of times)²
- $bid.relayer.validBid(b) = \top$ (the relayer to which the bid was submitted considers it valid)³

¹We only require the amount to be non-zero because the state is updated with the bids offered, not with bids that would win the previous leading bid.

²We don't parametrize the maximum number of bids to keep the model simple, but the number 2 is not crucial to the design. Also note that in the model, a builder can call the `bid` function arbitrarily many times with the same bid, but the state doesn't change by calling it multiple times.

³The `validBid` function only represents the check of custom validation requirements chosen by the relayer, such as the builder not being blacklisted.

- $balance(bid.builder) \geq bid.promise$ (the builder has enough balance to fulfill the promised reward)
- $bid.relayer \neq 0_{\mathcal{R}}$ (the relayer is not the zero relayer)
- $bid.relayer.checkRPBS(bid.rpbs, bid.promise, bid.builder) = \top$ (the block offered contains the promised payment to the PoN by the builder)

Postconditions:

- $bid.relayer.signedPoP(bid) = \text{sign}(bid.relayer, bid.rpbs.PoP)$ (the relayer signs the proof of payment)
- $bid.relayer.signedHeader(bid) = \text{sign}(bid.relayer, bid.rpbs.header)$ (the relayer signs the block header)
- $\forall s \neq slot \in \mathbb{N} \text{ Auction}'_s = \text{Auction}_s$ (all auctions for slots different than $slot$ remain the same)
- $\text{Auction}'_{slot} = \text{Auction}_{slot} \cup \{bid\} \in 2^{Bid}$ (the set of bids offered for slot is added bid)

We do not consider block-broadcasting to be a state-updating builder action since it only updates the on-chain state, and this model is only concerned with the off-chain state.

4.2 Proposer actions

Before defining the state updating functions for the proposers, note that we model bid inspection from the chosen proposer as block requesting. I.e., proposers request blocks from the relayers they consider choosing the block header from. Thus, they must only respond back to one of these relayers with a signed header.

Also, proposer bookkeeping (registration, activation, quitting, etc.) is all discharged to the on-chain status. We model the off-chain apparatus as not storing any such information and only querying the on-chain contracts to fetch it. Thus, the only interaction of a proposer with the off-chain is to request and sign block headers.

4.2.1 Request block header

The function `requestBlock` takes as inputs $p \in \mathcal{P}$ and $r \in \mathcal{R}^+$ and updates the state as follows:

$$(\mathcal{R}, \text{Auction}) \xrightarrow{\text{requestBlock}(p,r)} (\mathcal{R}', \text{Auction})$$

Preconditions:

- $proposerForSlot(slot) = p$ (the proposer who requested the block is the one chosen by the consensus layer for the current slot)
- $proposerStake(p) = 32 \text{ ETH}$ (the effective balance of validator p is 32 ETH)

- $proposerStatus(p) = \text{ACTIVE}$ (the proposer is registered as active in the PoN)
- $r \in \mathcal{R}_{slot}^{Bids}$ (the relay has received bids for the blockspace)
- $r.schedule(ts) = \text{SIGN}$ (the relay is currently forwarding block headers)

Postconditions:

- $\mathcal{R} \setminus \{r\} = \mathcal{R}' \setminus \{r\}$ (all relays except for the requested one remain the same)
- $r.request(slot) = \top$ (the requested relay records that it has been requested for the slot)

4.2.2 Sign block header

The function `signBlock` takes as inputs $p \in \mathcal{P}$, $r \in \mathcal{R}^+$, $b \in Bid$, and $signature \in \mathbb{S}$. Then, the state is updated as follows:

$$(\mathcal{R}, Auction) \xrightarrow{\text{signBlock}(p,r,b,signature)} (\mathcal{R}', Auction')$$

Preconditions:

- $r.request(slot) = \top$ (the proposer requested a block from the relay for that slot)
- $r.schedule(ts) = \text{SIGN}$ (the relay is currently accepting signed block headers)
- $proposerForSlot(slot) = p$ (the proposer is the one chosen for $slot$ by the consensus mechanism)
- $proposerStatus(p) = \text{ACTIVE}$ (the proposer is active in the PoN)
- $proposerStake(p) = 32\text{ETH}$ (the effective balance of the proposer is 32 ETH)
- $r.c(slot) = b$ (the signed bid needs to be the one chosen by the relay for $slot$)⁴
- $r.receive(slot) = 0_{\mathbb{S}}$ (the relay has not received a signed header before for the slot)
- $r.signedPoP(b) = \text{sign}(r, b.rpbs.PoP)$ (proof of payment must be validated by the relay)
- $r.signedHeader = \text{sign}(r, b.rpbs.header)$ (bid must be validated by the relay)

Note that the proposer can return an incorrect signature. That is, it may happen that $signature \neq \text{sign}(p, b.rpbs.header)$.

Postconditions:

⁴This condition ensures proposers can only submit signed bids to the relay it was requested from.

- $\mathcal{R} \setminus \{r\} = \mathcal{R}' \setminus \{r\}$ (all relayers different from r remain unchanged)
- $r.receive(slot) = signature$ (the only change in r is updating the *receive* function)

Note that it is not possible for a proposer to execute multiple times the `signBlock` function for the same relayer.

However, even if different block headers are signed, the builders responsible for those blocks should be exempt from any penalty since it's the proposer's wrongdoing.

There could be a race condition for reporting a builder for not having broadcasted a signed block. Namely, if another builder's block was included before the consensus layer declared it invalid because of multiple signed blocks for the same slot, a reporter could report other builders who also received a signed block. However, in practice, slots need to be finalized to be reportable. This should allow enough time for the consensus layer to invalidate the block and thus for the PoN to only allow the reporting of the proposer.

4.3 Relayer actions

In the model, there are no relayer actions. While the actual relayers do perform actions such as forwarding requested and signed block headers, we have included these actions in the actions that the builders or proposers perform. This makes sense since the relayers never initiate an action that does not follow a builder, proposer, or reporter action. Hence, the only role relayers play in this model is that of information verification and storage.

5 Agent Faults and Penalties

Both the builder and the proposer are subjected to the following rules. Non-compliance with these rules will result in a report. The spirit of the rules is: the model is partially responsible for allowing correct updates of the state. For example, not allowing unregistered builders to submit bids or proposers not selected for the current blocks to request block headers. However, responsibility for the correct update of the state is also shared by the parties that update the state (builder and proposer). Failure to do so will result in penalties. These are the rules that builders and proposers must follow when updating the state.

Note that depending on whether the payout pool was paid a positive amount or none, different rules may apply.

Also, if we have that $proposerStatus(proposerForSlot(s)) \neq \text{ACTIVE}$, none of the rules apply. After the rules definition, we indicate in what scenarios each rule must be satisfied.

5.1 Fault rules

Given a state $(\mathcal{R}, Auction)$, if there was a demand for the blockspace of a slot $s < slot$ (i.e., if $Auction_s.B \neq \emptyset$), a proposer should have requested a block header for at least one relayer and returned it signed, and the chosen builder should have broadcasted the block.

5.1.1 Block request

The only rule regarding block requesting is that a proposer must request a block if there is demand for the slot's blockspace.

Block request rule: If there was a demand for a slot's blockspace, it must exist a relay r from which $proposerForSlot(s)$ requested a block.

$$\text{BlockRequestRule}(s) := \text{Auction}_s \neq \emptyset \rightarrow \exists r \in \mathcal{R}_s^{bid} : r.request(s) = \top$$

or, equivalently

$$\text{BlockRequestRule}(s) := \mathcal{R}_s^{bid} \neq \emptyset \rightarrow \mathcal{R}_s^{request} \neq \emptyset$$

Failure to update the state as described will incur a penalty⁵ of

$$2 \cdot \sum_{i=1}^5 \frac{EIP1559Fee(s-i)}{5}.$$

5.1.2 Block signature

There are two rules regarding block signature. The first is that one block signature must be given back. The second is that the signature must be correct.

Block signature rule:⁶ If there was a demand for a slot's blockspace and the designated proposer requested blocks from the relayers, it must return a signed block header to a relay.

$$\begin{aligned} \text{BlockSignatureRule}(s) := & \text{Auction}_s \neq \emptyset \wedge \text{BlockRequestRule}(s) \rightarrow \\ & \exists r \in \mathcal{R}_s^{bids} : r.receive(s) \neq 0_{\mathbb{S}} \end{aligned}$$

or, equivalently

$$\text{BlockSignatureRule}(s) := \mathcal{R}_s^{bid} \neq \emptyset \wedge \text{BlockRequestRule}(s) \rightarrow \mathcal{R}_s^{receive} \neq \emptyset.$$

Failure to update the state as described will incur a penalty of

$$2 \cdot \sum_{i=1}^5 \frac{EIP1559Fee(s-i)}{5}.$$

Correct signature rule: If a signature is returned, it must be a correct signature. That is, it must correspond to one of the blocks offered in the auction.⁷

$$\begin{aligned} \text{CorrectSignatureRule}(s) := & \\ & \text{Auction}_s \neq \emptyset \wedge \text{BlockRequestRule}(s) \wedge \text{BlockSignatureRule}(s) \rightarrow \\ & (\forall r \in \mathcal{R}_s^{receive} \rightarrow (\text{signer}(r.receive(s)) = \text{proposerForSlot}(s) \wedge \\ & \quad \text{signedObject}(r.receive(s)) = r.c(s).rpbs.header)) \end{aligned}$$

⁵Throughout this document, we show the value that is approximated by the protocol. To see how the actual computation is done, we refer to the section "Proposer economic penalty" of the audit report.

⁶Note that a signed block header can only be forwarded to the relay when the schedule is on SIGN. Thus, we only need to check if the block was submitted. An alternative version would be to check if the timestamp where the block was submitted was on time.

⁷The functions "signer" and "signedObject" are ad-hoc functions to compute the signer and the object being signed from a given signature.

Failure to update the state as described will incur a penalty of

$$2 \cdot \sum_{i=1}^5 \frac{EIP1559Fee(s-i)}{5}.$$

5.1.3 Block broadcasting

The builder must broadcast the block. That is, if a block is finalized, the builder must have paid at least the promised amount to the pool. The only exception is if the consensus layer detected that the designated proposer signed multiple block headers to the same slot. In such case, no builder will be penalized, and the proposer will be kicked from the PoN.

Single signing rule: The single signing rule asserts that proposers shall only sign one block header per slot. To avoid any race condition on reporting builders when there is a double sign, as mentioned above, we assume that for every slot $s < slot$, the consensus layer will have realized any double signing penalties. If this happened, the proposer would be kicked out of the PoN, and no proposers would be penalized.

In practice, this assumption is realized by only allowing to report finalized blocks, hence giving the consensus layer enough time to detect double-signing events.

$$\begin{aligned} \text{singleSigned}(s) := & \text{Auction}_s.B \neq \emptyset \\ & \wedge \text{BlockRequestRule}(s) \\ & \wedge \text{BlockSignatureRule}(s) \\ & \wedge \text{CorrectSignatureRule}(s) \rightarrow \text{signedBlocks}(s) = 1 \end{aligned}$$

Note that this rule only covers double signing when the proposer has successfully followed the PoN workflow, and as such, proposers could be flagged as fault for not broadcasting blocks. That is, we have that if the antecedent of $\text{singleSigned}(s)$ and $\text{singleSigned}(s)$ hold, then $|\mathcal{R}_s^{\text{receive}}| = 1$. However, the converse doesn't hold in general.

If a PoN proposer signs multiple blocks without interacting with the PoN, the proposer will only be kicked if it's slashed by the consensus layer.

Also, note that if a proposer has signed multiple blocks but returned them wrongly to the corresponding relayers, the **singleSigned** rule will be successful. This doesn't exempt the relayer from being kicked off the PoN. Since if the consensus layer is made aware of the double signing, the proposer will get slashed and thus kicked. However, we prioritize the incorrect placement of the signed headers because the builders will not be able to broadcast the blocks.

Failure to satisfy this rule will result in the kick of the proposer:

$$\text{proposerStatus}(\text{proposerForSlot}(s)) = \text{KICKED}.$$

Block payment rule: If a proposer has signed a bid, the payout pool must be paid by the builder at least the promised amount.

$$\begin{aligned}
\text{blockPayment}(s) := & \text{Auction}_s \neq \emptyset \\
& \wedge \text{BlockRequestRule}(s) \\
& \wedge \text{BlockSignatureRule}(s) \\
& \wedge \text{CorrectSignatureRule}(s) \\
& \wedge \text{singleSigned}(s) \\
& \rightarrow \forall r \in \mathcal{R}_s^{\text{receive}} \text{paidInSlot}(s, r.c(s).builder) \geq r.c(s).promise
\end{aligned}$$

Note that the if the antecedent holds we have that $|\mathcal{R}_s^{\text{receive}}| = 1$.

Failure to satisfy this rule will result in a penalty of

$$2 \times r.c(s).promise.$$

Note that **blockPayment** covers two different faults. Namely, not broadcasting a block or underpaying the payout pool. However, if the designed proposer didn't satisfy any of the rules that apply to it, the **blockPayment** rule will be vacuously satisfied by builders.

5.1.4 Proposer health

The PoN only accepts healthy proposers to participate and thus earn gains. If a proposer was not of the highest quality, the PoN would kick such a proposer and never distribute the buffered rewards to it.

Healthy Proposer Rule: If a proposer is unhealthy, it can be kicked from the PoN.

$$\text{proposerHealth}(p) := \text{proposerStatus}(p) = \text{ACTIVE} \rightarrow \text{proposerStake}(p) < 32$$

Note that the rule does not depend on any particular slot. Thus, it can be checked, and the penalty enforced at any time.

Also note that this rule also covers the cases when a proposer is slashed or has withdrawn since, as a consequence of these situations, the effective balance will be below 32 ETH.

Failure to satisfy this rule will result in the kick of the proposer:

$$\text{proposerStatus}(p) = \text{KICKED}.$$

5.2 Rule checking

The following table summarizes under which circumstances must each rule be checked. In every case, we assume that $\text{proposerStatus}(\text{proposerForSlot}(s)) = \text{ACTIVE}$. Otherwise, the rules are not checked.

Rule / Scenario	$paidInSlot(s) = 0$	$paidInSlot(s) > 0$
BlockRequestRule(s)	check	don't check
BlockSignatureRule(s)	check	don't check
CorrectSignatureRule(s)	check	don't check
singleSigned(s)	check	don't check
blockPayment(s)	check	check

In short, if the payout pool is not paid, only one rule will not be satisfied, determining the agent at fault. Otherwise, only the **blockPayment** rule applies since we must ensure that the payout pool was paid correctly if the proposer used the PoN.

Note that if the proposer requested blocks from the PoN but didn't sign any of the requested blocks, the **blockPayment** rule is satisfied since that is an allowed action by the PoN.

The following algorithm describes how to determine if a rule was broken for a given slot $s < slot$:

```

Input: Slot  $s \in \mathbb{N}$ 
Output: Infraction[$rule] | BroadcastFailed | NoInfraction
if  $proposerStatus(proposerForSlot(s)) = \text{ACTIVE}$  then
  if  $\exists builder \in \mathcal{B} \text{ } paidInSlot(s, builder) > 0^a$  then
    if blockPayment( $s$ ) then
      | return NoInfraction
    else
      | return Infraction[blockPayment]
    end
  else
    if BlockRequestRule( $s$ ) then
      if BlockSignatureRule( $s$ ) then
        if CorrectSignatureRule( $s$ ) then
          if singleSigned( $s$ ) then
            | return BroadcastFailed
          else
            | return Infraction[singleSigned]
          end
        else
          | return Infraction[CorrectSignatureRule]
        end
      else
        | return Infraction[BlockSignatureRule]
      end
    else
      | return Infraction[BlockRequestRule]
    end
  end
else
  | return NoInfraction
end

```

^aNote that is not required for the builder to be registered in the PoN network.