eth2 security audit report

May 21, 2021 11:18 AM

https://t.me/average_user

Table of Contents

1 Overview

Background

Morphysm is a lean group of researchers and software engineers. The group performed testing and review of the ethereum consensus layer specification, the client implementation, and the underlying cryptographic libraries of eth2 after the Phase 0 mainnet launch in December 2020. In addition, the group researched all four eth2 clients; however, the team spent the most time on Prysm in Go.

Ethereum 2.0 is a nouveau suite of protocols and algorithms for consensus to move towards the scalability of Ethereum's original vision in terms of vertical and horizontal scalability. However, even after several years of research, the Ethereum 2.0. mainnet is a "young" project, s.t. security maturity is currently in its early days.
In preparation for more diligent security testing and prioritization + scoping of the large and diverse set of eth2 assets, the group chose a breadth-first security reconnaissance methodology.

This report summarises the data that was gathered during this reconnaissance phase.

Review Team

- jasperhepp
- kiliankae
- kittyandrew
- 0kSure

Areas of Concern

Our investigation focused on the following areas:

- Economic Safety;
    - Attacks leading to missing validator duties and rewards
    - Attacks leading to a delayed proposal
    - Attacks leading to delayed attestation
    - Attacks leading to slashable attestations/proposals
- Forkchoice and beacon state corruption;
    - Attacks leading to corrupted node state or crash
    - Attacks leading to ill-formed fork choice
    - Attacks intending to misuse resources cause unintended forks and create unwanted or adversarial chains.
- Network attacks, including replays, flooding of or misusing data;
- Authenticity, Integrity and confidentiality guarantees of the deployed cryptographic primitives;

Due to the novelty of eth2, limited tooling for security testing is available. The group, therefore, developed some tooling for that purpose.

The following repos are currently under development to improve and simplify future security testing:

- pyrum scripts
- obtrusive
- voyeur

Target Code and Revision

The following code repositories were considered during this review iteration:

- Core Eth2.0 specifications for Phase 0: https://github.com/ethereum/eth2.0-specs
- The Beacon Chain spec: https://github.com/ethereum/eth2.0-specs/blob/dev/specs/phase0/beacon-chain.md
- The p2p-interface: https://github.com/ethereum/eth2.0-specs/blob/dev/specs/phase0/p2p-interface.md
- The Beacon Chain Altair upgrade: https://github.com/ethereum/eth2.0-specs/blob/dev/specs/altair/beacon-chain.md
- The BLS Signatures RFC: https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-3.3
- The BLS reference implementation: https://github.com/ethereum/py_ecc
- The BLS proof of possession implementation of Prysm, Lighthouse, Nimbus, and Teku in process_deposit:
    - Spec https://github.com/ethereum/annotated-spec/blob/master/phase0/beacon-chain.md#deposits
    - Prysm https://sourcegraph.com/github.com/prysmaticlabs/prysm/-/blob/tools/specs-checker/data/specs/phase0/beacon-chain.md
    - Nimbus https://sourcegraph.com/github.com/status-im/nimbus-eth2/-/blob/beacon_chain/spec/beaconstate.nim
    - Teku https://sourcegraph.com/github.com/ConsenSys/teku/-/blob/ethereum/spec/src/main/java/tech/pegasys/teku/spec/logic/common/block/AbstractBlockProcessor.java
    - Lighthouse https://sourcegraph.com/github.com/sigp/lighthouse/-/blob/consensus/state_processing/src/per_block_processing.rs

2 Findings

We found the eth2 specification to be extensive and comprehensive. However, we found the p2p and messaging layer specification and implementation less detailed; here, one specific issue was identified (Issue A). Additionally, some incentives were found to have implicit security implications that would benefit from being more explicitly balanced (Issue G).

The BLS RFC and reference implementation (py_ecc) were found to be well written and extensively tested. However, due to the novelty of the cipher suites and the functional ambition of aggregatable signatures, weaknesses, vulnerabilities, and flaws are likely and to be expected. Issue C exemplifies a cryptographic vulnerability in BLS that was identified among others. The practical exploitability with current eth2 mainnet parameters is low. However, the security community's standard is to work towards an intact security margin between theoretical and practical vulnerabilities.
BLS was found to require more extensive auditing, testing, and documentation to identify and remediate vulnerabilities, edge cases, and unexpected behavior in order for the BLS suite to mature into a safe suite of cryptographic primitives.

Client diversity was one of the major design goals of Ethereum 2.0, as it fosters decentralization and resilience of the overall network. However, the Prysm client is used by a supermajority of users running an eth2 validator. As Prysm bugs could likely negatively influence eth2 finality and consensus, extensive and automated security testing of the Prysm client is crucial.
Unfortunately, while the go eth2 implementation was already extensively audited and tested, several bugs were found in this and an earlier engagement. One cause of Prysm's bug density was identified as favoring features over safety and the lack of dedicated eth2 security testing tools or CI. Sadly, Prysm’s feature richness over safety paradigm manifested itself as a variety of relatively trivial bugs and security errors, ranging from API crashes over incorrect slot/internal clock processing to consensus/fork choice errors of aggregated attestations (not part of this report, reported earlier).

We list our findings in the order we identified them during this assessment:

Issue A: Libp2p ENR is not Correlated with validator Keys or Stake

author: OkSure

Location

Spec - https://github.com/ethereum/eth2.0-specs/blob/dev/specs/phase0/p2p-interface.md#enr-structure

Summary

The combination libp2p/discv5/gossipsub does not tie p2p identity to a validator's stake. As p2p traffic is not authenticated with the stake, block proposals may be broadcast to the topic by arbitrary discv5/libp2p entities. Thus, the traffic needs to be processed on a protocol level, where actually no reasonable traffic on this topic is expected by entities that are not in `state.validators`.

Impact

Enables DDoS

Remediation

Validators should sign their ENR records with a private key that allows resolving the signature to their ETH 1 stake. This will enable libp2p level rate-limiting for eth2 topics where only validators are supposed to broadcast. Additionally, p2p message/spam slashing could be researched to harden the p2p layer.

Issue B: Inbound Connection Exhaustion

author: kittyandrew

Location

Prysm - any publicly exposed node.

Summary

The Prysm beacon node allows peers to establish up to 5 inbound connections per IP (30 connections total) without constraints. As a result, an attacker can establish connections to a beacon node to fill up the pool of allowed connections and block any legitimate connections from being established.

Libp2p/Gossibsub makes it simple for any validator to figure out the other validator's IP addresses quickly. An attacker might use inbound connections from rotating ips to execute DDoS attacks against proposers or attesters of a slot to stall the chain or keep slots empty.

Impact

The attack is inexpensive enough to perform over a long time on a series of block proposers. How exactly a targeted node will behave is currently being further investigated with the obtrusive tool here.

Feasibility

Due to unconstrained inbound connections, an attacker can perform the attack by recreating peer ids and leveraging a VPN to switch IP addresses; an attacker can perform the attack from a single host.

Technical Details

P2P connection exhaustion was implemented using rumor (fork of Rumor eth p2p scripting tool), code - pyrum-scripts, and obtrusive.

Issue C: Partition of One Attack

author: OkSure

Summary

BLS signatures are vulnerable, if the sum of a set of private keys equals `s_1 + ... + s_n = 1` in the respective field. In this case a valid signature is given by the hash of the message `e(g_1, sig) = e(g_1,H(m))` and is therefore universally forgeable. An equivalent form of this vulnerability is present if the sum of aggregated public keys equals the group generator. An initial review (py-ECC, herumi) suggested that all Eth2 BLS libraries are affected, as the RFC is flawed.

Location

BLS RFC

Impact

Two cases of this attack are considered in this report:

1. The special case `s = 1`, where the field identity element is chosen as a private key (by chance, misconfiguration, or user error). `s = 1` is erroneously specified as a valid output of the private key generation procedure (KeyGen) in the BLS RFC. While this case is easy to detect for an attacker, all cryptographic assumptions are broken. While this error is trivial, similar bugs have been actively exploited and studied on the Bitcoin and Ethereum 1 Blockchain: https://www.ise.io/casestudies/ethercombing/
2. The general case `s_1 + ... + s_n = 1 % curve_order` requires a set of private keys to be a partition of unity, which an attacker can detect without knowledge of any secret keys by observing p2p traffic. The general case can occur due to adversarial behavior or for stochastic reasons.

The exploitability of the vulnerabilities differs for the two cases:

1. For the case `s = 1`, an attacker can forge signatures for all types of messages by a vulnerable validator (withdrawals, slashable proposals, and attestations).
2. For the case, `s_1 + ... + s_n = 1 % curve_order`, an attacker (e.g., a malicious aggregator) that captures attestations and aggregates them, can produce valid signatures for aggregated attestations of a subset of validators whose private keys add up to 1.
   As a griefing attack, an attacker might also maliciously create validators with such private keys and afterwards deny to have signed a given (slashable) message due to the cryptographic error.

Preconditions

An attacker that passively monitors eth2 traffic may detect a vulnerable committee. An attacker might also precompute, or Bruteforce vulnerable committee (sub-)sets ahead of time.

Feasibility/Likelihood

Given the following parameters:

```
curve_order =
52435875175126190479447740508185965837690555005276378226036586999385811845 13

N = 128                          # maximum number of validators in one
committee
S = 64                           # number of shards per slot
T = 5*60*24*365                  # slots per year
```

We estimate the probability of a collision in a subset of a committee and the probability of having a collision in one year in one subset of a committee as follows:

```
# probability of a collision in one committee

p1 = 2^N / curve_order                # this is of order 10**(-39)

# probability of a collision/vulnerable committee in one year on the beacon
chain

p2 = 1 - (1 - 2^N / curve_order)^(S*T) # this is of order 10**(-30)
```

Note that the likelihood of a collision increases with the maximum size of the committee `N` and decreases with slot length in seconds. In summary, we found this vulnerability's exploitability to be difficult to determine, as it involves the number of partitions `p(1) % curve_order`, which is a nontrivial number theoretical question.

Technical Details

Descriptive tests can be found here.

Issue D: Prysm Resource Exhaustion - Host Crash

author: kittyandrew

Location

Prysm API

Summary

The Prysm has a vulnerability that causes the node to rapidly exhaust the host's memory while also loading all CPU cores.

Impact

After a few seconds of processing malicious requests, the victim beacon node and the (physical) server becomes unresponsive. The experiment took ~10 seconds for the script to kill a 16GB 4 core server. Interestingly, there are no logs inside the Prysm beacon chain client to indicate that something is happening, which is dangerous (during tests, logging level was enabled).

Preconditions

The victim client needs to have the flag enabled.

Feasibility

Any actor with access to the API can carry out this attack.

Technical Details

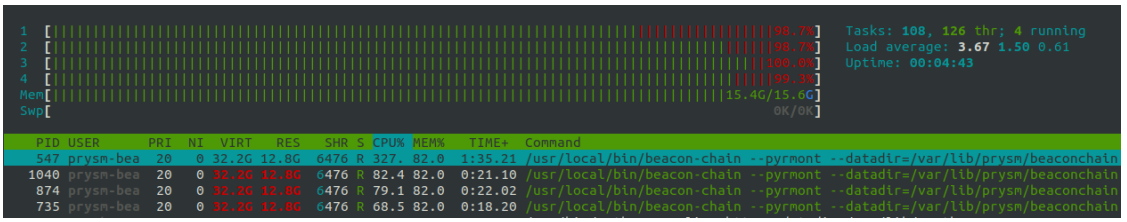
Shell script:

```
seq 1 500 | xargs -I $ -n1 -P50  curl -sX GET -H "accept: application/json"
http://localhost:3500/eth/v1alpha1/debug/state?slot=99999
```

Identified and reproducible via pyrum-scripts.

These images are few seconds apart:

The image below is the result, where the host server became unresponsive.



The fact that the Prysm client propagates the error to the host OS increases the impact of this issue, as the host might host other sensitive or critical applications, which would be affected by this error.

Issue E: Skip Verification Flags in Client Nimbus and Teku

author: jasperhepp

Summary

In Nimbus, the flag and Teku, this allows for skipping the BLS verification and is implemented for testing. Similarly, Teku allows for skipping the verification of the deposit contract merkle proof.

Impact

Skipping the BLS aggregation increases the performance and decreases the required resources of a validator node significantly. Furthermore, due to increased state initialization, attestation, and block processing speed, users are by proxy incentivized to set the flag to speed up their nodes and reduce

host/maintenance cost. Therefore, activating this flag falls under a risk/reward tradeoff, as without BLS verification, validators operate faster and have lower resource requirements. However, skipping the verification allows for attacks, for example, a rogue key attack since proof of possession is skipped in process_deposit [BDN2018].

Feasibility

If a validator activates the flag, an attacker might infer this since the response and processing time of the validator node should be faster.

Technical Details

You find the flag for Nimbus here, for Teku here, and the skipping of the merkle proof in Teku here.

Remediation

We recommend the Nimbus and the Teku team remove the flag in order to remove potential for misconfiguration and to not introduce implicit security/reward incentives increasing validator and network health risks.

Issue F: Prysm API gRPC Panic

author: kittyandrew

Location

Prysm API

Summary

The Prysm has a bug triggered when you send empty data via post request to the endpoint, which causes a go runtime error.

Impact

Spike in CPU usage of the host

Preconditions

No special preconditions are required.

Feasibility

Any actor with access to the API can carry out this attack.

Technical Details

Shell script:

```
curl -sX POST -H "accept: application/json"
http://localhost:3500/eth/v1alpha1/validator/block | json_pp
```

Identified via radamsa fuzzer. Response:

```
{
   "code" : 2,
   "details" : [],
   "error" : "runtime error: invalid memory address or nil pointer
dereference",
   "message" : "runtime error: invalid memory address or nil pointer
dereference"
}
```


Issue G: Validator DOS incentive in get_total_active_balance

author: jasperhepp

Summary

Validators have an incentive to DOS other validators in order to increase their individual rewards. Moreover,

Impact

Adversarial behavior is incentivized, risking chain finalization and liveness.

Preconditions

No special preconditions are required.

Feasibility

Generic/Misaligned incentive

Technical Details

The rewards are calculated based on the total active balance, which is reduced if a validator gets inactive/slashed.

```
def get_base_reward(state: BeaconState, index: ValidatorIndex) -> Gwei:
    total_balance = get_total_active_balance(state)
    effective_balance = state.validators[index].effective_balance
    return Gwei(effective_balance * BASE_REWARD_FACTOR //
integer_squareroot(total_balance) // BASE_REWARDS_PER_EPOCH)
```

In addition, unexpected inactivity of a validator leads to slashing in case the validator misses the assigned slot; see here.

We are planning to simulate rational adversarial validator strategies in a further engagement.

3 Open questions and future testing

The following observations, remarks, and questions arose during the review of the Specification in the lighthouse client is not compatible with annotated spec.
As of the spec, this function is supposed to , this verification couldn't be identified.

- `process_gossip_voluntary_exit` in the lighthouse client is not compatible with annotated spec.
  As of the spec, this function is supposed to

  ```
  assert bls.Verify(
      validator.pubkey,
      signing_root,
      signed_voluntary_exit.signature
      );
  ```

  This verification couldn't be identified.
- In compute_fork_digest, the comment `4-bytes suffices for practical separation of forks/chains` is not clear, as an adversarial eth2 fork might brute force `Version` to produce a colliding `ForkDigest (4 bytes)`, to bypass the p2p layer network separation effectively and cheaply generate large amounts of p2p traffic that needs to be separated in the state machine.
- In compute_proposer_index rename `MAX_RANDOM_BYTE = 2**8 - 1` in the `compute_proposer_index` function to `MAX_RANDOM_BYTE_VALUE`. This avoids confusion about the type of the variable.
- In initialize_beacon_state_from_eth1 the entropy in beacon state initialisation from Eth1 is insecure. The used to seed the in can be influenced by the Eth1 miners. Therefore, miners can influence the validators assigned to the beacon chain committees, the shard chain committees, and the block proposers in the first epoch for a new fork.
- `bls.Verify` is still present in the Spec. It should be replaced with `bls.AggregateVerify` as of Attacks and weaknesses of BLS aggregate signatures.
- In get_seed it is unclear how
  `(epoch + EPOCHS_PER_HISTORICAL_VECTOR - MIN_SEED_LOOKAHEAD - 1)`
  it translates to `(53 - 1 - 4) % 10 = 8` in the explanatory text.
  Should it be `(epoch + EPOCHS_PER_HISTORICAL_VECTOR - MAX_SEED_LOOKAHEAD - 1)`?
- In slash_validator `state.slashings` is the total amount of ETH from which a proportion is slashed. Therefore the naming is not ideal. Could it be something like preSlashedEth?
  In the description of `slash_validator`:

  "This array is used to track the total number of validators slashed, which is used to compute total slashing penalties (often called "anti-correlation penalties")"

  is incorrect; it is the total pre-slashed staking amount.

Future testing

Can this be triggered without a proposer actually proposing two different blocks?

```
def process_proposer_slashing(state: BeaconState, proposer_slashing:
ProposerSlashing) -> None:
```

It should be asserted that all nodes can handle spoofed ProposerSlashings with invalid signatures well, e.g., ProposerSlashing where only the sig verification

```
assert bls.Verify(proposer.pubkey, signing_root, signed_header.signature)
```

fails and all other assertions pass in this codepath.

Possible overflows on Bytes4 Types?

`DomainType`, `ForkDigest`, and `DomainType` are only `Bytes4`. is there any arithmetic involving these?

How expensive is it to compute is_slashable_attestation_data?

```
def is_slashable_attestation_data(
       data_1: AttestationData,
       data_2: AttestationData
   ) -> bool:


"""
Check if ``data_1`` and ``data_2`` are slashable according to Casper FFG
rules.
"""

       return (

              # Double vote
              (data_1 != data_2 and data_1.target.epoch == data_2.target.epoch)

              or

              # Surround vote
              (
                     data_1.source.epoch < data_2.source.epoch and
                     data_2.target.epoch < data_1.target.epoch)
              )
```

Case to test here: long range surrounding attestation. To detect, a node would have to compare every new attestation with any previously seen attestation, with the list basically growing linearly with every new attestation. Is that implemented? Prysm allows for something like 1k slots future/past.

This can become expensive:

- time: compare each new attestation with all previous.
- storage: store all previous
- grows linearly

32 eeks fork

What happens if attestations violate slashing conditions outside of the 32 eeks randaomix store time and part of a fork?