Prysm

Security Assessment

October 23, 2023

Prepared for:
Private Client

Prepared by: **Benjamin Samuels, Dominik Czarnota, and Damilola Edwards**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

A private client engaged Trail of Bits to review the security of Prysm. From March 13 to April 7, 2023, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the full public source code and documentation. We performed both automated and manual processes of the target system and its codebase.

Summary of Findings

The audit did not uncover any significant flaws or defects. Only one notable finding was found; however, it is medium severity and represents an issue with the Ethereum Specification, not with the Prysm software itself.

EXPOSURE ANALYSIS

Severity	*Count*
Medium	2
Low	4
Informational	4
Undetermined	5

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Configuration	1
Data Exposure	1
Data Validation	7
Denial of Service	1
Documentation	1

Timing	1
Undefined Behavior	1

Notable Findings

Notable flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-PRYSM-8**
  Denial-of-service (DoS) attacks can be launched against a block proposer in order to force the block proposer to miss its proposal slot. This finding is not exclusive to Prysm, and it affects all Ethereum consensus client implementations up to and including the Bellatrix hard fork.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Dominik Czarnota, Consultant
dominik.czarnota@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 6, 2023	Pre-project kickoff call
March 17, 2023	Status update meeting #1
March 24, 2023	Status update meeting #2
March 31, 2023	Status update meeting #3
April 10, 2023	Delivery of report draft to client & Prysm team
April 10, 2023	Report readout meeting
May 3, 2023	Delivery of final report
October 23, 2023	Delivery of final report with fix review

Project Goals

The engagement was scoped to provide a security assessment of Prysm. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does Prysm correctly implement protections to prevent validators from being slashed?
- Can DoS attacks be used to disrupt a Prysm validator's operation?
- Does Prysm's implementation adhere to the Ethereum 2.0 specification?
- Does Prysm properly validate all inputs to protect against potential issues?
- What is the optimal system configuration to run a Prysm validator safely and securely?
- What kinds of controls and procedures should a large-scale validator operator follow to run Prysm validators safely and securely?
- How effective are Prysm's fuzzing harnesses, and where can they be improved?
- Is Prysm vulnerable to Rogue BLS key attacks?

Project Targets

The engagement involved a review and testing of the following target:

Prysm

Repository	https://github.com/prysmaticlabs/prysm
Version	`v3.2.2 (e2fa7d40e3f496416283cc1d329a8ff6c048cb8a)`
Type	Blockchain Consensus Software
Platform	Cross platform

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

- **Validator availability**. We began by identifying potential entrypoints for DoS attacks and cross-referencing those entrypoints with the consensus specification. Manual analysis was performed against the specification to identify DoS issues that may prevent a validator from submitting attestations or proposing blocks.

  Next, we performed light threat modeling to understand the crypto-economic incentives for performing a DoS attack against a validator.

  Finally, we reviewed the configuration and throttling capabilities for gossip endpoints to assess whether malicious peers would be penalized/dropped.

- **Slashing protection**. We conducted a manual review of Prysm's safeguards for preventing the validator from being slashed. These safeguards were cross-referenced with the consensus specification's recommended best practices for avoiding slashing.

- **Specification adherence**. We conducted a manual review to compare the consistency and differences between the Ethereum 2.0 consensus specification and the Prysm codebase. We tracked sections of the codebase that were implemented in accordance with the official specifications, taking into account the changes across forks and variations between programming languages used in the specification pseudocode and the matching implementation in Golang. We observed that the codebase appears to have correctly implemented the protocol's blockchain state transition, slot, epoch processing, and attestation rules. Where differences existed, they were primarily due to changes in function naming, inclusion of additional checks, error handling, and the use of helper functions within the implementation code.

- **Implementation quality**. Prysm's code quality was reviewed using an array of static tooling, detailed in appendix H. After triaging the findings raised by the tooling, we conducted a manual review against hot spots in the codebase that the tooling flagged as potentially indicative of a greater problem.

- **Cryptography**. A brief review of Prysm's use of cryptography was performed by reviewing its randomness sources, encryption configuration, and key management code. The Ethereum Consensus specification was reviewed to see if it is vulnerable to common BLS attacks such as rogue key attacks.

- **Fuzzing analysis**. A review of Prysm's fuzzing harnesses was conducted to investigate their code coverage and effectiveness. We developed scripts to run the fuzzing harnesses and generate code coverage reports from them. We also added three new fuzzing harnesses and extended the code coverage of existing harnesses that dealt with gossip/pubsub topics. We ran the fuzzing harnesses for around a week and reported the issues found with them. Appendix G describes the setup of the automated dynamic analysis tools and test harnesses used during this audit.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Compatibility with other Ethereum Proof of Stake consensus clients
- Correctness of the cryptographic primitives used during node operation and encryption at rest (BLS, ECDSA, BIP39, ERC-2335, etc.). These operations are implemented by third-party libraries out of the scope of the audit.
- Correctness and implementation of the Beacon Chain Validator API
- Deposit contracts were not audited.
- The accuracy of Prysm's implementation of the Ethereum 2.0 consensus specification was verified with the best effort that could be achieved during a time-boxed engagement. We manually reviewed only the core components of the beacon-chain implementation, including the code related to block processing, state transitions, validator operations, decoding of gossip messages, and associated helper functions.
- Outside the context of DoS issues, the Ethereum 2.0 Consensus specification itself was not audited for security/correctness.
- Integrations with block construction services and their security implications for validator operators were not reviewed.
- Prysm's third-party dependencies, their maintenance status, and their time since last update were not reviewed.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
go test -fuzz	Go's built-in coverage guided fuzzing engine.	Appendix G
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix H
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix H
Go-sec	A static analysis utility that looks for a variety of problems in Go codebases. Notably, go-sec will identify potential stored credentials, unhandled errors, cryptographically troubling packages, and similar problems.	Appendix H
Go-vet	A popular static analysis utility that searches for more go-specific problems within a codebase, such as mistakes pertaining to closures, marshaling, and unsafe pointers. Go-vet is integrated within the go command itself, with support for other tools through the vettool command line flag.	Appendix H
Staticcheck	A static analysis utility that identifies both stylistic problems and implementation problems within a Go codebase. Note that many of the stylistic problems staticcheck identifies are also indicative of potential	Appendix H

	“problem areas” in a project.	
Ineffassign	A static analysis utility that identifies ineffectual assignments. These ineffectual assignments often identify situations in which errors go unchecked, which could lead to undefined behavior of the program due to execution in an invalid program state.	Appendix H
Errcheck	A static analysis utility that identifies situations in which errors are not handled appropriately.	Appendix H
gokart	A static analysis tool for Go that finds vulnerabilities using the single static assignment (SSA) form of the Go source code.	Appendix H
goleak	A Goroutine leak detector that detects leaks by directly hooking into a project’s test suite.	Appendix H
GCheck	A suite of static detectors to analyze Go software, for example to find goroutine leaks.	We attempted to use it but it failed as it doesn't support new Go modules projects.

Areas of Focus

Our automated testing and verification work focused on detecting the following issues:

- General code quality issues and unidiomatic code patterns
- Issues related to incorrect error handling or incorrect data validation
- Potential goroutine leaks that could lead to DoS scenarios

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Prysm routes many mathematical operations through a dedicated module, `math_helper.go`. This module checks for invalid operations, overflows, underflows, and divide-by-zero conditions. However, more scrutiny should be applied to ensure that all important math operations actually use the math helper, or else code should be documented to justify why it does not need to use the math helper (for purposes such as the computation of attestation deltas).	**Satisfactory**
Auditing	Prysm provides auditing/monitoring capabilities for monitoring the active validator or other validators on the network. Logging capabilities cover all critical validator events, and important events are specifically called out in Prysm's documentation. Prysm also exposes locally accessible Prometheus metrics along with some Grafana dashboards for monitoring one or more validators.	**Strong**
Authentication / Access Controls	The distributed aspect of the system means that certain traffic, realized by the gossip peer-to-peer protocol, does not require authorization. The beacon chain allows authentication with the execution endpoint and its beacon chain API endpoint. However, we found an endpoint that, when explicitly enabled, does not require authentication while it should,	**Strong**

	or else it should be removed or redesigned, as detailed in TOB-PRYSM-9.	
Complexity Management	Prysm takes good measures to manage the complexity of the consensus client. Splitting the consensus client into the beacon chain and validator clients made the code much easier to review. The code also contains inline references to the original consensus specification, which make it easier to understand and reason about how Prysm works.	**Strong**
Configuration	Prysm provides extensive secure configuration options for end users. However, in some places the documentation does provide best practices for users (TOB-PRYSM-14). While Prysm does provide options for strong authentication at its various endpoints, in some cases there is support for allowing secret values to be passed to Prysm over CLI, creating a risk of disclosure (TOB-PRYSM-3).	**Moderate**
Cryptography and Key Management	Prysm's use of cryptography and key management aligns with current best practices. All meaningful sources of entropy come from cryptographically secure random number generators. The underlying implementation for BLS12-381 uses `blst`, a well-known and audited library. Prysm code wrapping `blst` functions is well documented and includes information about assumptions and design considerations. Prysm supports storing key material remotely in an HSM via web3signer.	**Strong**
Data Handling	Prysm takes necessary precautions when validating most types of incoming data. Many parts of the code have clear documentation that calls out when a piece of data is validated and what kinds of validations occur in a given function (e.g., `ConvertToIndexed`, `VerifyBlockSignatureUsingCurrentFork`).	**Moderate**

	However, some parts of the codebase have much less robust data handling, as we observed in TOB-PRYSM-11, TOB-PRYSM-12, and TOB-PRYSM-15.	
Documentation	Prysm provides a comprehensive set of setup, configuration, backup, operations, and monitoring documentation. In particular, Prysm's "Security Best Practices" documentation is high quality and offers a comprehensive guide on how to run a validator securely.	**Strong**
Maintenance	Prysm has thorough inline code documentation that comprehensively outlines special considerations and assumptions for critical code paths. Prysm's comments also include inline code snippets from the consensus specification, which greatly simplify the comparison of specification and implementation. However, Prysm's unit tests are challenging to successfully run, as some of them depend on a very specific network configuration. This may make long-term maintenance more challenging as maintainers rotate in and out of the project. Prysm lacks higher-level developer documentation that describes the system as a whole (e.g., how the validator client and beacon client interact with each other during attestation), making auditing of the project more challenging.	**Satisfactory**
Memory Safety and Error Handling	Prysm is written exclusively in Go, so is not exposed to memory safety issues. Prysm handles errors properly in the vast majority of its codebase. The only exception is several benign issues in unit tests (TOB-PRYSM-1).	**Strong**
Testing and Verification	Prysm has thorough unit test coverage known-good test cases; however, it lacks unit tests covering various error conditions the software may encounter. Although end-to-end tests are present, they can be run only on a Prysm-managed build server.	**Satisfactory**

	Prysm uses several static analysis tools as part of its CI to catch simple bugs and low-quality code (e.g., errcheck, unused, gocognit, unused) Prysm has an array of fuzzing harnesses that were reviewed during the audit. Many are run as part of the unit test suite, but at least one harness appears to be unmaintained and failed (TOB-PRYSM-13).	

Summary of Recommendations

Trail of Bits recommends that Prysmatic Labs address the findings detailed in this report and take the following additional steps to enhance its security posture:

- Negative unit tests to test critical code components that must fail under certain conditions (e.g., incorrect signature verification, attestation using the wrong bitmap slot)
- Add CodeQL rules to detect when important abstractions are bypassed (e.g., someone creates a file without using `fileutil`).
- Code that deviates from the reference specification should be differentially fuzzed against a canonical implementation (see figure G.12 in Appendix G: Automated Dynamic Analysis).
- Add additional high-level developer documentation to help onboard new contributors. Examples of such documentation include:
    - Additional `README` files with developer documentation in each major module directory (e.g., `prysm/validator`, `prysm/beacon-chain`, `prysm/tools`)
    - System diagrams with references to locations where specific implementations may be found
    - Sequence diagrams of critical code paths and how they traverse Prysm's various components
    - Annotations for functions that may only be used by tests, which will prevent these functions from being confused with or accidentally used in production implementations (ex: `func RandKey()` in `crypto/bls/bls.go`)
    - Standardized terminology for function names (e.g., verify vs. validate vs. `IsValid`)
- Work towards standardizing unit/e2e test harnesses to enable developers to run all tests from their local machines. This process may be simplified by running tests inside a standardized container environment.

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Unhandled Errors	Auditing and Logging	Informational
2	os.Create() used without checking for an existing file	Data Validation	Informational
3	Passing sensitive configuration values through the command line may leak to other processes on the system	Data Exposure	Low
4	The configuration files are not checked for permissions while they may contain sensitive values	Access Controls	Low
5	The beacon-chain and validator RPC APIs can panic which is recovered but may lead to crashes due to memory exhaustion	Data Validation	Low
6	Goroutine leaks	Undefined Behavior	Undetermined
7	Potential deadlock if the Feed.Send panic is recovered and the function is retried	Timing	Undetermined
8	Block Proposer DDoS	Denial of Service	Medium
9	The db backup endpoint may be triggered via SSRF or when visiting an attacker website which may cause a denial of service	Data Validation	Medium
10	Maximum gRPC message size of MaxInt32 (2GB) set in beacon-chain/server may lead to denial of service	Configuration	Informational

11	EpochParticipation.UnmarshalJSON may parse invalid data	Data Validation	**Undetermined**
12	Uint256.UnmarshalJSON may parse invalid data	Data Validation	**Undetermined**
13	Failed assertions in the FuzzExecutionPayload fuzzing harness	Data Validation	**Undetermined**
14	The JWT authentication docs should not suggest generating the secret with the use of third-party websites	Documentation	**Low**
15	Potentially insufficient gossip topic validation	Data Validation	**Informational**

Detailed Findings

1. Unhandled errors

Severity: **Informational**	Difficulty: **N/A**
Type: Auditing and Logging	Finding ID: TOB-PRYSM-1
Target: `various`	

Description

The Prysm tests contain multiple ineffectual assignments to `err` error variables. These errors are never checked, which may lead to undefined test behavior given a sufficiently large change set.

Ineffectual assignments to `err` can be viewed in figures 1.1-1.5.

```
wsb, err := blocks.NewSignedBeaconBlock(knownBlocks[i])
err = WriteBlockChunk(stream, chain, p2.Encoding(), wsb)
if err != nil && err.Error() != network.ErrReset.Error() {
       require.NoError(t, err)
}
```

Figure 1.1: The error from `blocks.NewSignedBeaconBlock` is overwritten. (prysm/beacon-chain/sync/rpc_send_request_test.go#242–246)

```
bc := params.BeaconConfig()
altairSlot, err := slots.EpochStart(bc.AltairForkEpoch)
bellaSlot, err := slots.EpochStart(bc.BellatrixForkEpoch)
require.NoError(t, err)
```

Figure 1.2: The error from `slots.EpochStart` on line 136 is overwritten. (prysm/encoding/ssz/detect/configfork_test.go#135–138)

```
bellav := bytesutil.ToBytes4(params.BeaconConfig().BellatrixForkVersion)
altairS, err := slots.EpochStart(params.BeaconConfig().AltairForkEpoch)
bellaS, err := slots.EpochStart(params.BeaconConfig().BellatrixForkEpoch)
require.NoError(t, err)
```

Figure 1.3: The error from `slots.EpochStart` on line 207 is overwritten. (prysm/encoding/ssz/detect/configfork_test.go#206–209)

```
bellav := bytesutil.ToBytes4(params.BeaconConfig().BellatrixForkVersion)
```

```
altairS, err := slots.EpochStart(params.BeaconConfig().AltairForkEpoch)
bellaS, err := slots.EpochStart(params.BeaconConfig().BellatrixForkEpoch)
require.NoError(t, err)
```

Figure 1.4: The error from `slots.EpochStart` on line 298 is overwritten. (prysm/encoding/ssz/detect/configfork_test.go#297–300)

```
signedTx, err := types.SignTx(tx, types.NewLondonSigner(chainid), key)
if err != nil {
       return nil
}
err = backend.SendTransaction(context.Background(), signedTx)
return nil
```

Figure 1.5: The error from `backend.SendTransaction` is overwritten. (prysm/testing/endtoend/components/eth1/transactions.go#115–120)

Exploit Scenario

A large set of changes is made to Prysm that causes one of the overwritten errors to become material, in addition to introducing a bug that the affected test should detect. Since the error goes uncaught, the test may complete successfully despite the software not performing the intended behavior.

Recommendations

Short term, add code to catch the unhandled errors.

Long term, add automated analysis tooling to Prysm's CI pipeline such as `ineffassign`.

2. os.Create() used without checking for an existing file

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-PRYSM-2

Target:
`prysm/tools/interop/convert-keys/main.go`
`prysm/tools/specs-checker/download.go`	

Description

Prysm uses `os.Create()` in several locations without checking for the presence of an existing file. If a file is already present when `os.Create()` is called, the original file will be truncated. The affected code sections use `os.Create()` with the expectation that an existing file will cause an error, or that any existing file will be overwritten.

```
outFile, err := os.Create(os.Args[2])
if err != nil {
	log.WithError(err).Fatalf("Failed to create file at %s", os.Args[2])
}
```

Figure 2.1: `os.Create` used to create a file without checking for its presence first (`prysm/tools/interop/convert-keys/main.go#55–58`)

```
func getAndSaveFile(specDocUrl, outFilePath string) error {
	// Create output file.
	f, err := os.Create(filepath.Clean(outFilePath))
	if err != nil {
		return fmt.Errorf("cannot create output file: %w", err)
	}
```

Figure 2.2: `os.Create` used to create a file without checking for its presence first (`prysm/tools/specs-checker/download.go#41–46`)

Exploit Scenario

The spec checker is run at a specific point in time, then again later after the spec changes. The spec files are created using `os.Create()`, which truncates the original file and creates an invalid spec.

Recommendations

Short term, replace calls to `os.Create()` with `os.OpenFile("file.txt", os.O_CREATE|os.O_EXCL, 0600)`. This alternate call will generate an error if the file already exists.

Long term, consider abstracting calls to `os.Create()` through Prysm's dedicated file management module in `prysm/io/file/fileutil.go`. In addition, add a checklist for pull request reviews for the reviewer to explicitly check for new `os.Create()` calls so that guidance may be given to use the `fileutil.go` module.

3. Passing sensitive configuration values through the command line may leak to other processes on the system

Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-PRYSM-3
Target: Command line flags	

Description

The `--execution-endpoint` and its previous (and deprecated) `--http-web3provider` Prysm consensus node (beacon-chain) command line flags allow users to specify an authentication header (figure 3.1). Specifying sensitive values through command line flags may leak their values to an attacker who can read files on the machine. This can happen when:

- The attacker can list system processes either with the `ps aux` command or by reading relevant files in the `/proc/` filesystem.
- An attacker finds an arbitrary file read vulnerability in another application running on the system and reads the `/proc/$pid/cmdline` flag of Prysm processes.
- The process command line flags are exposed to a monitoring service and the attacker gets access to its data.

```
   --execution-endpoint value                                 An execution client http
endpoint. Can contain auth header as well in the format (default:
"http://localhost:8551")

   --http-web3provider value                                  DEPRECATED: A mainchain
web3 provider string http endpoint. Can contain auth header as well in the format
--http-web3provider="https://goerli.infura.io/v3/xxxx,Basic xxx" for project secret
(base64 encoded) and --http-web3provider="https://goerli.infura.io/v3/xxxx,Bearer
xxx" for jwt use (default: "http://localhost:8551")
```

Figure 3.1: The command line flags that may be used to set an auth header

In addition to those two command line flags, there are others that may be used to pass sensitive values, for example: `--execution-headers` or `--grpc-headers`.

Also please note that the visibility of the process command line flags may be reduced by using the `hidepid=2` and `gid=0` mount options for the proc filesystem. However, this option is not enabled by default on the majority (or all?) of Linux distributions and is usually not used by system administrators.

Exploit Scenario

An attacker finds an arbitrary file read vulnerability in a web application that runs on the same server as a Prysm node. They exploit the vulnerability to read the command line arguments from the Prysm node and find the configured execution endpoint URL along with its authentication header, since the user passed it through the command line flag. The attacker uses this information to further attack the execution client.

Recommendations

Short term, remove the option to pass the execution endpoint authentication header through a command line flag and instruct the users to provide any sensitive configuration values, or, even all the configuration through a YAML config file set with the `--config-file` flag.

Long term, regularly audit the command line options of all Prysm binaries to ensure they disallow or discourage users from providing sensitive values. Also, document the risks of passing sensitive data through the command line flags.

4. Configuration files containing potentially sensitive values are not checked for permissions

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PRYSM-4
Target: configuration files loading	

Description

The Prysm binaries do not check that the configuration files are not readable by other users on the system, but these files may contain sensitive values, such as the execution endpoint authentication header value (figure 4.1). This may allow an attacker to read and use these values if the configuration files are set with overly broad permissions.

```
$ ls -la
total 16
drwxr-xr-x   4 dc  staff   128 Mar 17 12:28 .
drwxr-xr-x  74 dc  staff  2368 Mar 16 05:09 ..
-rwxrwxrwx   1 dc  staff    17 Mar 17 12:28 config.yaml
-rwxrwxrwx   1 dc  wheel    66 Mar 16 05:02 jwt.hex

$ bazel run //cmd/beacon-chain:beacon-chain -- --config-file=`pwd`/config.yaml
--jwt-secret=`pwd`/jwt.hex
…
INFO: Build completed successfully, 1 total action
[2023-03-17 12:30:53]  INFO Finished reading JWT secret from
/Users/dc/src/github.com/prysmaticlabs/prysm/chain/jwt.hex
…
[2023-03-17 12:30:53]  INFO node: Starting beacon node version=Prysm/Unknown/Local
build. Built at: Moments ago
[2023-03-17 12:30:53]  INFO blockchain: Blockchain data already exists in DB,
initializing...
[2023-03-17 12:30:53]  INFO gateway: Starting API middleware
…
```

Figure 4.1: The blockchain starts and operates without ensuring that the configuration files cannot be read by other users.

Although the Prysm node does check file permissions for the wallet password file and when it writes files, this protection does not extend to verifying the permissions of its configuration files when they are read..

Exploit Scenario

An attacker can read files located on the server on which a Prysm node runs as a different user than the one who runs the node. The attacker reads the Prysm node configuration file since it was set to be world-readable. The attacker then uses the secret values read from this file, such as the execution endpoint authentication header, to further attack the user's Prysm node. Note that the attacker does not necessarily require shell access to read files, since they could find another vulnerability that allows them to do so.

Recommendations

Short term, check the permissions of files and directories that may contain sensitive data and warn the user and exit the program if they are overly broad. The warning should suggest that the user rotate any secrets that the files may have stored, since these files may have been read by other users on the system. Such a check should be implemented by using the `File.Stat` function on an already opened file, since performing the check before opening the file leads to time-of-check vs time-of-use (TOCTOU) issues.

Long term, add tests to ensure the Prysm binaries check if the configuration files have overly broad permissions.

5. Panics by the beacon-chain and validator RPC APIs can panic are recovered but may lead to crashes due to memory exhaustion

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRYSM-5
Target: `prysm/api/pagination/pagination.go, API endpoints`	

Description

The beacon-chain RPC APIs that support pagination via the `pagination.StartAndPage` function panic due to an out-of-bounds slice indexing when they receive a negative value of the `PageToken` or `PageSize` parameters. Such panics are later recovered by the gRPC server and so do not crash the node. However, this issue may still allow an attacker to cause a denial of service (DoS) of the node (e.g., through memory exhaustion when many invalid requests are sent).

Figures 5.1-2 show example requests that are intercepted and modified using the Burp Suite proxy. In these examples, a validator run with the `--rpc` flag panics, causing the API to return a 500 Internal Server Error response containing the panic error.

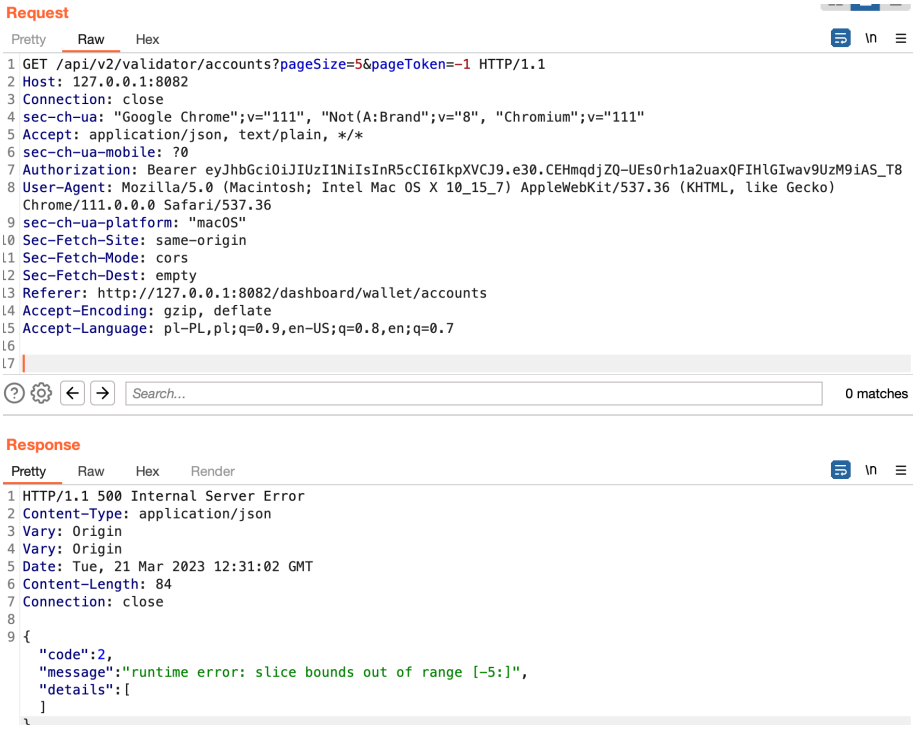

Figure 5.1: Sending a request with `pageToken=-1` causes a panic and a 500 response.

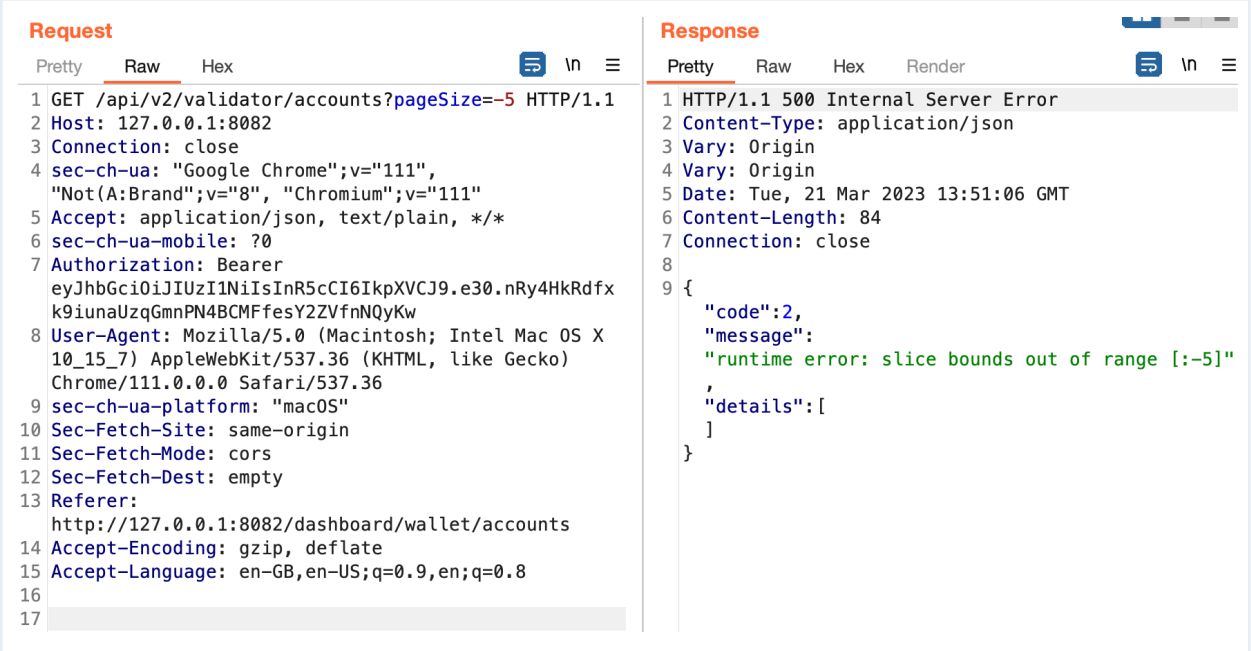

Figure 5.2: Sending a request with `pageSize=-5` causes a panic and a 500 response.

Figure 5.3 shows where one of the API endpoints panics, but all the other endpoints implement similar logic. The `ListAttestations` endpoint calls `StartAndEndPage` and passes in the `req.PageToken` and `int(req.PageSize)`. If these values are sent as negative numbers, they will cause the `StartAndEndPage` function to return a negative `start` value (figure 5.4), which is then used by the `ListAttestations` function to index the `atts` slice.

```
func (bs *Server) ListAttestations(/* (...) */) {
	// (...)
	start, end, nextPageToken, err := pagination.StartAndEndPage(req.PageToken,
int(req.PageSize), numAttestations)
	// (...)
	return &ethpb.ListAttestationsResponse{
		Attestations:  atts[start:end], // <-- out-of-bounds slice panic
		TotalSize:     int32(numAttestations),
		NextPageToken: nextPageToken,
	}, nil
}
```

Figure 5.3:
prysm/beacon-chain/rpc/prysm/v1alpha1/beacon/attestations.go#L100-L105

```
func StartAndEndPage(pageToken string, pageSize, totalSize int) (/* (...) */ ) {
	// (...)
	token, err := strconv.Atoi(pageToken)
	if err != nil { /* (...) */ }

	// Start page can not be greater than set size.
```

```
        start := token * pageSize  // <-- token or pageSize may be negative
        if start >= totalSize { /* (...) */ }

        // End page can not go out of bound.
        end := start + pageSize   // <-- both token and pageSize may be negative
        // (...)
        return start, end, nextPageToken, nil
}
```

Figure 5.4: prysm/api/pagination/pagination.go#L22-L42

The API endpoints that are vulnerable to this issue are as follows:

- In beacon-chain: `ListValidatorAssignments`, `ListAttestations`, `ListIndexedAttestations`, `AttestationPool`, `ListBeaconBlocks` (through `listBlocksForEpoch` and `listBlocksForSlot`), `ListValidatorBalances`, `ListValidators`
- In validator: `ListAccounts`

Recommendations

Short term, fix the API endpoints pagination logic to reject negative values by implementing the following fixes:

1. Change the requests `PageSize` fields types from `int32` to `uint32`.
2. Change the `StartAndEndPage` function to parse the `pageToken` string as a 32-bit unsigned integer through the `strconv.ParseUint` function.

This will prevent the API endpoints from panics that could potentially lead to crashes.

Long term, add tests to ensure that the pagination API rejects negative values.

6. Goroutine leaks can lead to Denial of Service	
Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PRYSM-6
Target: `tests`	

Description

Many tests in the Prysm codebase leak goroutines, meaning that the goroutines spawned by tests do not finish and hang on an IO wait, channel receive or send, or a select state. As a result, the resources used by those goroutines are not freed. If these goroutine leaks can happen continuously throughout the normal Prysm node run, or, if they can be triggered by an attacker, this can lead to DoS through resource exhaustion.

We detected this issue by using the uber-go/goleak package and adding or modifying the `TestMain` function in all Prysm packages that included tests to call the `goleak.VerifyTestMain(m)` function. We include the list of all goroutines that leaked as part of one or many of the tests in appendix E.

The severity of this finding is undetermined, as we have not analyzed or confirmed whether any of those goroutines can leak continuously in a normal run of a Prysm node or whether such a leak can be triggered by an attacker.

Recommendation

Short term, add the uber-go/goleak goroutine leaks detection to Prysm tests and investigate the goroutine leaks that occur. Fix all goroutine leaks that are valid, and document and ignore (by specifying them in a call to `VerifyTestMain`) any that are false positives.

7. Potential deadlock if the Feed.Send panic is recovered and the function is retried

Severity: **Undetermined**	Difficulty: **High**
Type: Timing	Finding ID: TOB-PRYSM-7
Target: `prysm/async/event/feed.go`	

Description

The `Feed.Send` function can cause a deadlock if it is called again on the same object after the first call panics and is recovered.

The severity of this finding is undetermined, as we have not confirmed if this can happen in practice. However, the function should unlock the `f.mu` mutex before panicking to prevent the potential deadlock.

```
// Send delivers to all subscribed channels simultaneously.
// It returns the number of subscribers that the value was sent to.
func (f *Feed) Send(value interface{}) (nsent int) {
	// (...)
	// Add new cases from the inbox after taking the send lock.
	f.mu.Lock()
	f.sendCases = append(f.sendCases, f.inbox...)
	f.inbox = nil

	if !f.typecheck(rvalue.Type()) {
		f.sendLock <- struct{}{}
		panic(feedTypeError{op: "Send", got: rvalue.Type(), want: f.etype})
	}
	f.mu.Unlock()
```

Figure 7.1: prysm/async/event/feed.go#L141-L149

Recommendations

Short term, fix the lack of `f.mu` unlock in the `Feed.Send` function before panic. This will prevent the deadlock that could happen if the `Feed.Send` function is called again after a panic from its previous call.

8. Block Proposer DDoS	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-PRYSM-8
Target: Ethereum 2.0 Specification	

Description

The Ethereum 2.0 Specification defines a “block proposer” for each beacon chain slot [1]. Block proposers are responsible for selecting a set of transactions to include in the block that each satisfies the beacon chain’s state transition function. When validators successfully propose a valid block, they are rewarded an amount of ETH that constitutes a large percentage of validator staking rewards. The block proposers for a given slot in an epoch can be pre-calculated up to one epoch ahead of time.

The IP address of a given validator (and thus, the IP of a block proposer) can be determined by any P2P actor using previously published deanonymization attacks [2].

Given that the identity of block proposers is known ahead of time, and that a block proposer’s IP address may be ascertained by a malicious actor, block proposers may be subject to distributed denial-of-service (DDoS) attacks with the goal of forcing the block proposer to miss their proposal slot.

This attack can be extended to prevent validators from fulfilling other validator responsibilities, such as slot attestations. But preventing block proposals is more likely as it may have a financial incentive, as demonstrated in the exploit scenario below.

This attack is not unique to Prysm, and is present in all Ethereum 2.0 consensus clients at the time of writing.

Exploit Scenario

Attacker Alice and victim Bob operate Ethereum validators. In the upcoming epoch, Bob is assigned to propose a block in slot 1, and Alice is assigned to propose a block in slot 2. While waiting for Bob to propose a block, Alice notices a transaction on the gossip layer that pays an extraordinarily high priority fee, and that this transaction will likely be included in Bob’s block.

Alice then launches a DDoS attack against Bob’s validator in order to knock it offline so it will miss its proposal slot. Bob’s validator misses its proposal slot, and Alice includes the

high-value transaction in her proposed block for slot 2. Through this attack, Alice effectively steals a proportion of the validator awards that were designated for Bob's slot.

Recommendation

Short term, the only known mitigation is to configure validators with a "front-end/"back-end" network topology [3]. In this topology, each validator client (back end) routes its block proposals and attestations to one of two beacon chain nodes (front end). Using this mitigation, an attack against a block proposer will take down the attestation beacon chain node (as that is the IP address that can be revealed using current research), and the block proposal node is free to relay the proposed block to the rest of the network.

It should be noted that this mitigation would require the validator and each beacon chain node to be deployed to separate servers, which complicates deployment and violates the current recommendation to run all the validation software on a single machine.

There may be other mitigations, such as using a sacrificial reverse proxy for attestation gossip or deploying an L3 firewall in front of the validator. However, due to time constraints, we could not verify the effectiveness of such mitigations.

Long term, the Ethereum Foundation should pursue secret leader elections for inclusion in a future hard fork.

References

[1] Ethereum Phase 0 Spec - Block Proposal
[2] Practical Deanonymization Attack in Ethereum Based on P2P Network Analysis
[3] How I learned to stop worrying about the DoS and love the chain

9. The db backup endpoint may be triggered via SSRF or when visiting an attacker website, which may cause a DoS

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRYSM-9
Target: `/db/backup` monitoring API endpoint	

Description

A Prysm node (beacon-chain or validator) run with the `--enable-db-backup-webhook` flag exposes a `GET /db/backup` monitoring API endpoint that saves a backup of the node database onto the disk. Although the monitoring API is hosted on localhost (`127.0.0.1`) by default, an attacker can still reach this API either by performing this request when the user who hosts a Prysm node visits their site, or, if the node is hosted on a server, through a server side request forgery (SSRF) vulnerability.

This can lead to a DoS either by filling up the disk space or by causing an out-of-memory scenario as detailed in the database backups documentation (since the backup loads the whole database into memory first). This may also leave the node database files corrupted.

Additionally, note that there is no authentication for the monitoring API; if it were exposed to the public (by setting the `--monitoring-host` flag) with the database backups enabled, it would be much easier to exploit this vulnerability.

For the sake of complete information, the default monitoring API endpoint ports are 8080 and 8081 for the beacon-chain and validator, respectively.

Exploit Scenario

A Prysm node is hosted with db backups webhook enabled on a server that runs other web applications. An attacker finds a bug in one of the web applications that allows the attacker to perform an SSRF attack. The attacker forges as many requests to the Prysm node monitoring endpoint as possible in order to create db backups and fill up the server disk space, causing the Prysm validator to be slashed once the node stops operating.

Recommendation

Short term, remove the `GET /db/backup` endpoint. This endpoint is not only unauthenticated and potentially reachable using cross-site request forgery (CSRF) or SSRF attacks, but it is also unsafe to use, as it may cause an out-of-memory scenario and corrupt the database.

10. Maximum gRPC message size of MaxInt32 (2GB) set in beacon-chain/server may lead to DoS

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-PRYSM-10
Target: `beacon-chain/server`	

Description

The Prysm 3.2.2 release changed the maximum gRPC message size in the beacon-chain/server program to a value of 2GB (figure 10.1). This may allow an attacker who can send requests to this server to cause a DoS by sending very large messages.

```
216  217          // GrpcMaxCallRecvMsgSizeFlag defines the max call message size for GRPC
217  218          GrpcMaxCallRecvMsgSizeFlag = &cli.IntFlag{
218       -              Name:  "grpc-max-msg-size",
219       -              Usage: "Integer to define max recieve message call size (default: 4194304 (for 4MB))",
220       -              Value: 1 << 22,
     219  +              Name: "grpc-max-msg-size",
     220  +              Usage: "Integer to define max recieve message call size. If serving a public gRPC server, " +
     221  +                      "set this to a more reasonable size to avoid resource exhaustion from large messages. " +
     222  +                      "Validators with as many as 10000 keys can be run with a max message size of less than " +
     223  +                      "50Mb. The default here is set to a very high value for local users. " +
     224  +                      "(default: 2147483647 (2Gi)).",
     225  +              Value: math.MaxInt32,
221  226          }
```

Figure 10.1: The patch that changed the maximum gRPC message size (`github.com/prysmaticlabs/prysm/pull/12072`)

To clarify, the beacon-chain/server is not the beacon-chain client node. The beacon-chain/server is a proxy server one can use to expose the beacon-chain's gRPC API through an HTTP/JSON REST API. Effectively, with the default configuration, the beacon-chain/server listens on `127.0.0.1:8000` and can forward its requests to `localhost:4000`, which is the beacon-chain gRPC API.

Recommendation

Short term, change the default maximum gRPC message size to a sane small value. Alternatively, warn the user in the logs when they run the beacon-chain/server on a non-localhost address. This will help users to avoid this issue when, for example, they decide to host the beacon-chain/server on a public address but do not read the warning from the `grpc-max-msg-size` flag description.

Long term, always set safe defaults and require users to change them if needed.

11. EpochParticipation.UnmarshalJSON may parse invalid data

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRYSM-11
Target: `prysm/beacon-chain/rpc/apimiddleware/structs_marshalling.go`	

Description

The `EpochParticipation.UnmarshalJSON` function used to deserialize a JSON byte slice does not validate the format of the provided JSON string, assuming that it starts and ends with quotation marks, which may not be true. For example, unmarshalling a value of `[]byte("XdHJ1ZQ==X")` with unexpected `X` characters at the beginning and ending would not return an error, although an error would be expected.

This may lead to a situation where a malformed JSON string is successfully parsed as the epoch participation value when it should be rejected instead. This can further cause other problems (e.g., if the JSON string would be parsed by two different systems).

The severity of this finding is undetermined because we have not exhaustively analyzed this case.

```
// EpochParticipation represents participation of validators in their duties.
type EpochParticipation []string

func (p *EpochParticipation) UnmarshalJSON(b []byte) error {
  if string(b) == "null" {
    return nil
  }
  if len(b) < 2 {
    return errors.New("epoch participation length must be at least 2")
  }

  // Remove leading and trailing quotation marks.
  decoded, err := base64.StdEncoding.DecodeString(string(b[1 : len(b)-1]))
  if err != nil {
    return errors.Wrapf(err, "could not decode epoch participation base64 value")
  }

  *p = make([]string, len(decoded))
  for i, participation := range decoded {
    (*p)[i] = strconv.FormatUint(uint64(participation), 10)
  }
  return nil
```

```
}
```

Figure 11.1:
prysm/beacon-chain/rpc/apimiddleware/structs_marshalling.go#L10-L32

Recommendation

Short term, fix the `EpochParticipation.UnmarshalJSON` function so that it rejects invalid inputs that do not fit the expected data format. Add additional test cases to test the function behavior.

12. Uint256.UnmarshalJSON may parse invalid data

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRYSM-12
Target: `prysm/api/client/builder/types.go`	

Description

The `Uint256.UnmarshalJSON` function (figure 12.1) does not validate the format of the provided JSON byte slice. As a result, the function handles cases that start or end with quotes, which allows malformed JSON values, such as those below, to be parsed successfully:

- "123
- 123"

The severity of this finding is undetermined because we have not exhaustively analyzed this case. However, as with TOB-PRYSM-11, parsing of malformed data may lead to problems depending on various factors, including how the function is used, whether it is possible to control the data it takes, or if two systems that would be fed with the same data would be expected to process it in the same way.

```
func (s *Uint256) UnmarshalJSON(t []byte) error {
	start := 0
	end := len(t)
	if t[0] == '"' {
		start += 1
	}
	if t[end-1] == '"' {
		end -= 1
	}
	return s.UnmarshalText(t[start:end])
}
```

Figure 12.1: prysm/api/client/builder/types.go#L129-L139

Recommendation

Short term, fix the `Uint256.UnmarshalJSON` function so that it rejects invalid inputs that do not fit to the expected data format. Add additional test cases to test the function behavior.

13. Failed assertions in the FuzzExecutionPayload fuzzing harness

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRYSM-13
Target: `prysm/beacon-chain/execution/engine_client_fuzz_test.go`	

Description

We found two cases when the `FuzzExecutionPayload` fuzzing harness fails one of its assertions. This fuzzing harness ensures that the Geth's `engine.ExecutableData` structure layout matches the Prysm's `pb.ExecutionPayload` structure. It does so by unmarshalling the input byte slice into the two structures, comparing their errors, and then performing additional marshaling operations.

The first input that fails an assertion about mismatched Geth and Prysm unmarshall errors has a `withdrawals` field that is not supported by Prysm's structure (figure 13.1). This field, added in the Capella fork, has been added to the Geth's go-ethereum package in this patch (see `gen_ed.go` modifications), and a 1.11.0 version including it was released on February 25, 2023.

This issue does not seem to affect the security of the Prysm codebase, since it dispatches much of its logic based on the fork version, and its `ExecutionPayloadCapella` structure supports the Capella fork and its `withdrawals` field.

```
f.Add([]byte("{\"parentHash\":\"0xff01ff01ff01ff01ff01ff01ff01ff0100000000000000000000000000000000\"," +
        "\"feeRecipient\":\"0xff01ff01ff01ff01ffff01ff01ff01ff01ff0000\"," +
        "\"stateRoot\":\"0xff01ff01ff01ff01ff01ff01ff01ff0100000000000000000000000000000000\"," +
        "\"receiptsRoot\":\"0xff01ff01ff01ff01ff01ff01ff01ff0100000000000000000000000000000000\"," +
        "\"logsBloom\":\"0x6a756e6b" + strings.Repeat("0", 504) + "\"," +
        "\"prevRandao\":\"0xff01ff01ff01ff01ff01ff01ff01ff0100000000000000000000000000000000\"," +
        "\"blockNumber\":\"0xffffffffffffffff\"," +
        "\"gasLimit\":\"0xffffffffffffffff\"," +
        "\"gasUsed\":\"0xffffffffffffffff\"," +
        "\"timestamp\":\"0x64\"," +
        "\"extraData\":\"0x\"," +
        "\"baseFeePerGas\":\"0x7fffffffffffffff\"," +
        "\"blockHash\":\"0xff01ff01ff01ff01ff01ff01ff01ff0100000000000000000000000000000000\"," +
        "\"transactions\":[" +
        "\"0xff01ff01ff01ff01ff01ff01ff01ff01\"," +
        "\"0xff01ff01ff01ff01ff01ff01ff01ff01\"," +
        "\"0xff01ff01ff01ff01ff01ff01ff01ff01\"," +
        "\"0xff01ff01ff01ff01ff01ff01ff01ff01\"],"  +
        "\"withdrawals\":9999}"))
```

Figure 13.1: First input that triggers an assertion in the `FuzzExecutionPayload` harness

The second input returns an "empty hex string" error on the Prysm unmarshaller but no error on the Geth unmarshaller. This is due to the empty `baseFeePerGas` field, as shown in figure 13.2.

```
f.Add([]byte("{\"pArentHAsh\":\"0X0000000000000000000000000000000000000000000000000000000000000000\"," +
        "\"feeReCipient\":\"0X00000000000000000000000000000000000000\"," +
        "\"stAteRoot\":\"0X0000000000000000000000000000000000000000000000000000000000000000\"," +
        "\"reCeiptsRoot\":\"0X0000000000000000000000000000000000000000000000000000000000000000\"," +
        "\"logsBloom\":\"\"," +
        "\"prevRAndAo\":\"0X0000000000000000000000000000000000000000000000000000000000000000\"," +
        "\"BloCkNumBer\":\"\"," +
        "\"gAsLimit\":\"\"," +
        "\"gAsUsed\":\"\"," +
        "\"timestAmp\":\"\"," +
        "\"eXtrADAtA\":\"\"," +
        "\"BAseFeePerGAs\":\"\"," +
        "\"BloCkHAsh\":\"0X0000000000000000000000000000000000000000000000000000000000000000\"," +
        "\"trAnsACtions\":[]}"))
```

Figure 13.1: The second input that triggers an assertion in the `FuzzExecutionPayload` harness

Recommendations

Short term, double check that the described issue cannot affect production Prysm code. Also, fix the code or fuzzing harness so it does not crash on the described inputs and assertion.

Long term, run the fuzzing harnesses regularly and fix any bugs they find, especially before Prysm releases.

14. The JWT authentication docs suggest generating the secret using third-party websites

Severity: **Low**	Difficulty: **High**
Type: Documentation	Finding ID: TOB-PRYSM-14
Target: https://docs.prylabs.network/docs/execution-node/authentication	

Description
The Prysm's JWT authentication documentation suggests the JWT token's secret value can be generated using third-party websites:

> *Use an online generator like this. Copy and paste this value into a `jwt.hex` file.*

This may lead to a situation where the secret generated by the user is leaked or used against their node because a third-party service used by the user stored the secrets or generated a non-random value.

Exploit Scenario
The suggested online generator was hacked and now serves predictable random values. The attacker finds a Geth instance that uses authentication and uses the predictable random value to authenticate. They then use the authenticated endpoint to cause a DoS of the node.

Recommendations
Short term, do not recommend the use of third-party websites to generate authentication secret values in the JWT authentication documentation, since such websites may produce non-random values or store the generated values, adding a risk that the secret may leak or be used against the user.

An alternative solution is to host such a token secret generator within Prysm itself, which eliminates the need for users to trust a third-party service to generate the token.

15. Potentially insufficient gossip topic validation	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRYSM-15
Target: Multiple code paths	

Description

Some code paths in the Prysm project process pubsub/gossip messages topics in a potentially insufficient manner. While other validations in other processing steps appear to prevent these code paths from introducing security risks, additional code changes or the introduction of new gossip message formats could make those code paths problematic.

The issue is that according to the p2p interface consensus specs, the pubsub topics must conform to the following format:

```
/eth2/ForkDigestValue/Name/Encoding
```

The Prysm project deals with these pubsub topics in the following ways:

1. In the `decodePubsubMessage` function, it does not validate or check that the `Encoding` part is `ssz_snappy`. Instead, it only trims this encoding from the topic and leaves the topic string intact if the encoding part is not there. This is realized by the following code line:

   ```
   topic = strings.TrimSuffix(topic, s.cfg.p2p.Encoding().ProtocolSuffix())
   ```

2. Multiple functions check for a given pubsub topic using the `strings.Contains(topic, <some string literal>)` code, which can lead to incorrect topic attribution if a topic of a given kind could contain a string used by another kind of topic. The following functions demonstrate this behavior:
   - `decodePubsubMessage`
   - `topicScoreParams`
   - `addDigestToTopic`
   - `addDigestAndIndexToTopic`
   - `updateMetrics`
   - `registerRPCHandler`

We have discussed these potential issues with the Prysm team to validate whether they may affect the functioning of Prysm nodes. A Prysm node would not accept a message with

an unexpected pubsub message topic, since the messages will be filtered by their topics by this libp2p library, according to the filter set by the beacon-chain code.

Recommendation

Short term, consider refactoring the code of the `decodePubsubMessage` and other functions that deal with pubsub message topics to centralize and simplify the validation of pubsub topics. Avoid choosing the message topic with the constructions like `strings.Contains(topic, <some string literal>)`.

Long term, implement end-to-end tests or fuzzing harnesses that will validate that all pubsub topics that do not fit into the expected formats are properly rejected by the system.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. System Diagram

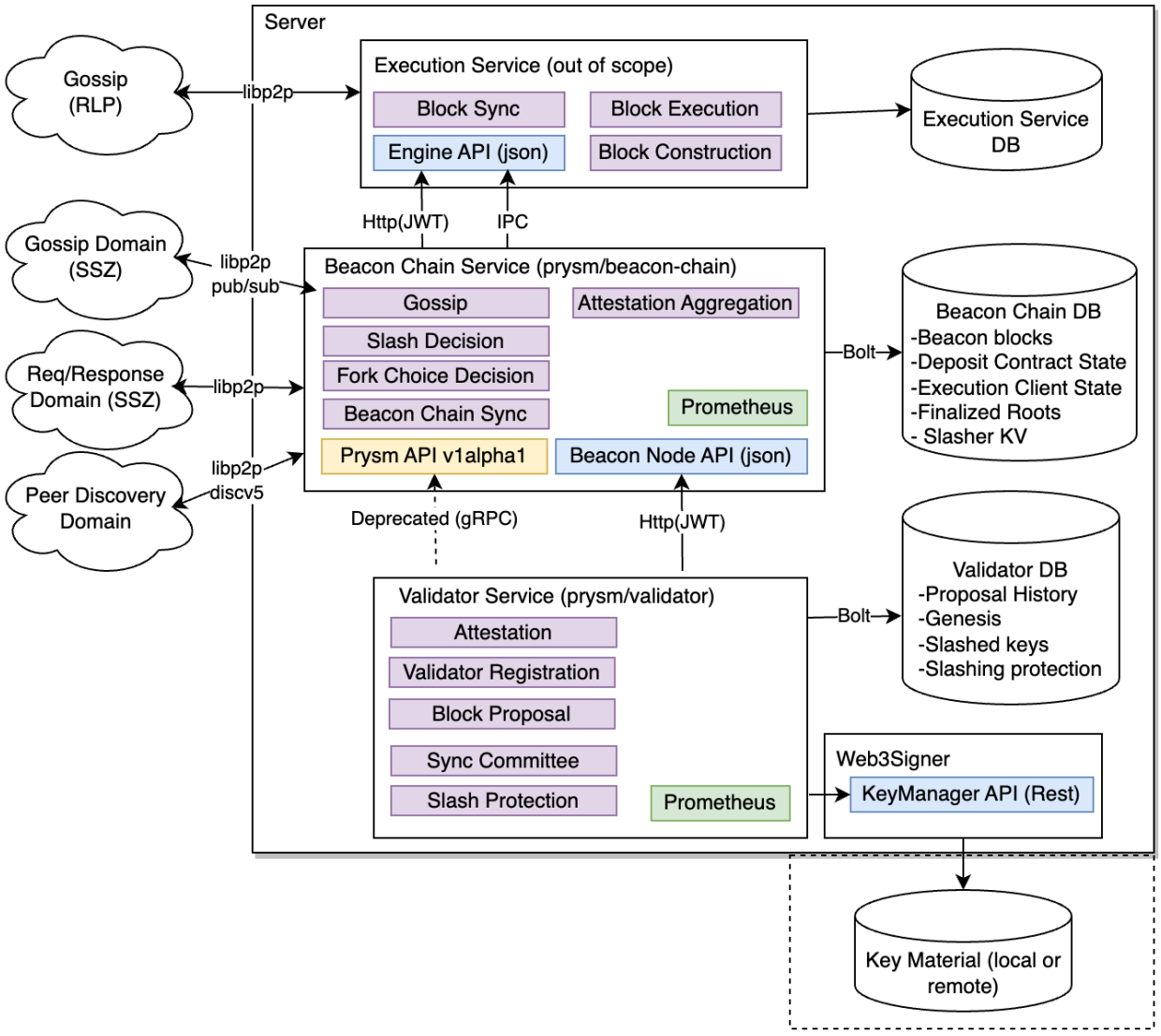

Figure C.1: System diagram of a Prysm validator

D. Security Guidance for Operators

Operating an Ethereum validator comes with technical and security risks; if the node is not adequately operated and secured, the validator may be slashed and lose funds. This appendix provides guidance on how to mitigate those technical and security risks.

This guidance builds on the Prysm team's security best practices documentation and is focused on an enterprise setting where a single actor operates multiple validators (referred to as a swarm).

Client Selection

Ethereum proof-of-stake encourages client diversity in order to produce the most robust network possible. Client diversity is incentivized using the network's "correlation penalty" calculations and "inactivity leak" penalty.

However, in practice, operators must consider various risks and factors when choosing the validator client software to use, including the following:

- The difference between potential penalties for running a buggy majority or a less frequently used client. See Correlation Penalty and Inactivity Leak below for more details.
- The potential mitigations related to social consensus; for example, a bug in a majority client that causes a large number of stakers to be slashed may lead to a larger discussion and/or actions by the ecosystem.
- The client team's reputation
- The kinds of change management practices currently used to validate new code
- The risks of compromise for the client's team

Correlation Penalty

When a validator is slashed, three penalties are levied against the validator's stake. These include an initial penalty that reduces the offender's stake by 1/32, attestation inactivity penalties that are applied until the offender's withdrawal epoch, and a correlation penalty.

The correlation penalty is used to apply extra punishment based on how many other slashing events occurred after the offender was initially slashed. The idea behind the correlation penalty is to scale slashing punishments so that coordinated attacks are much more heavily penalized than one-off events.

The correlation penalty creates an existential threat for validators using a client that is used by a large fraction of the validator network. If a bug in a validator client causes the client to commit slashing offenses, then the more validators that use that client, the more severe the slashing will be. Two theoretical situations and their consequences are outlined below.

1. A client is used by 33% of the validator network. A bug in the client causes each validator running it to commit a slashable offense. Over the following few hours, every validator using the problematic client is slashed. Due to the high correlation of slashing offenses, every slashed validator using the client will have their **entire stake** slashed.

2. A client is used by 2% of the validator network. A bug in the client causes each validator using the client to commit a slashable offense. In this situation, the correlation penalty would work out to about 6% of the validator's original stake.

Inactivity Leak

Inactivity leak protocol kicks in when the chain fails to finalize for four epochs (over one third of the validator set is offline) and penalizes validators that are offline by slashing their stake.

This means if a specific validator client is used by over 33% of the network and encounters a bug that prevents blocks from being processed, all validators using that client will be penalized by the inactivity leak.

Inactivity leak creates a danger for validator operators who are using beacon chain/execution clients that are used by a large fraction of the validator set. For this reason, validator operators are incentivized to prefer clients that are used by a minority of the network.

Current client diversity statistics can be found in figure D.1.

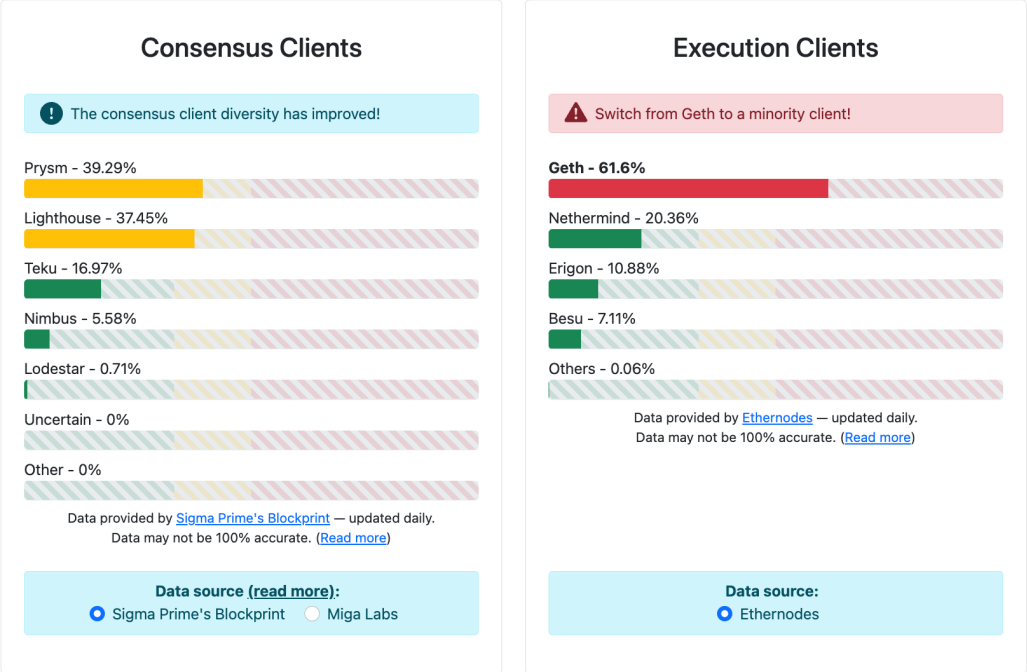

Figure D.1: Client diversity statistics as of March 30, 2023 (`https://clientdiversity.org/#distribution`)

Node Configuration

Deployment Configuration

The validator client, beacon chain client, and execution client should be deployed to a single machine or Kubernetes pod unless there is a good operational reason to do otherwise. If deployment across multiple machines or pods is required, connections between each client must be secured using TLS for encryption and JWTs for authentication. The deployment process itself should also be tested, documented, and reproducible so that it is possible to redeploy the node in the event of an emergency.

We also recommend keeping one or more additional "staging" or "test" deployments with the same exact or similar setup (but with different keys, of course) against a test network. Those additional deployments should be used for testing updates and other scenarios, like the process of node migration to another machine.

In addition, validator operators should avoid passing any sensitive values in command line flags when configuring Prysm. When passed as CLI flags, sensitive values can be leaked to other users on the system (TOB-PRYSM-3) and may be ingested by logging/monitoring software. Sensitive values related to authentication should be passed to Prysm using configuration files, and optionally stored in a remote key management solution/templated into in-memory configuration files at runtime.

Node Updates

The operators should track the node software updates through the available official channels. The new release changelog should be inspected carefully to understand all of its consequences and any steps that need to be performed in order to apply the update. The update installation should be tested first on a staging/test setup, and, if everything works as expected, then installed on the mainnet setup.

Key Management

Ethereum proof-of-stake validators use two keys: a validation key and a withdrawal key.

Validation Key

The validation key should be managed by a remote authenticated Web3Signer instance, which in turn should store the key at rest in an HSM, cloud key management solution, or software key storage solution such as HashiCorp Vault.

Given that a specific key may be needed by multiple validator nodes during a migration or backup recovery, redundant operational protections must be implemented to prevent two validator instances from attempting to use the same key in validation activities at the same time. The use of a validator key by two distinct validator clients will lead to the key being slashed.

Withdrawal Key

The withdrawal key is needed only to withdraw a validator's stake, and as such, should not be stored on validator nodes. Withdrawal keys should use threshold signatures to force withdrawals to require the consensus of multiple independent parties.

Private threshold keys should be stored in dedicated Hardware Security Modules (HSM) that are in turn kept in separate, secure geographical locations and accessible only by independent, authorized parties.

If private threshold keys are stored on cryptocurrency hardware wallets, operators should follow the 10 Rules for the Secure Use of Cryptocurrency Hardware Wallets.

Slashing Protection Database Management

The beacon chain node has several persistent databases, the most important of which is the slashing protection history. This database is **critical** for preventing a validator from being slashed and must be migrated if a validator is re-synced or deployed to another client.

If a slashing protection database is corrupted and cannot be recovered from backup, operators must wait at least two finalized epochs before restarting their validator. It must be emphasized that under no circumstances should two validator nodes use the same key at the same time. It follows that complex failover scenarios should not be attempted by validator operators without a thorough understanding of the highly skewed risk/reward

tradeoff of such a system. At this time, there are no known validator operators successfully using any such failover system, nor is there software support for such a system in any publicly available consensus client.

Slashing Nodes

Large validator operators should avoid running their validators with slashing enabled, as this requires significantly more disk space. Instead, operators should run one or more dedicated slasher nodes on different machines with static peering arrangements with the validating swarm.

Slasher nodes are not required for operating a validator or earning fees from slashing penalties. However, by *not* running a slashing node, the operator must trust that another entity on the network is running a correctly configured slasher node and that their slashing reports are propagated to the swarm's block proposers in a timely manner. Failing to include slashings in a block proposal may lead to lost earnings that would have come from the inclusion of slashing.

Monitoring and Alerting

Validator nodes must have sufficient monitoring and alerting capabilities to escalate potential incidents before losses are incurred. Prysm exposes Prometheus metrics along with various logs that should be ingested by a logging/monitoring solution. Operators should consider adding additional instrumentation to monitor system resources, such as disk utilization, network utilization, CPU utilization, and RAM utilization.

Specific guidance on the kinds of events that require manual intervention is documented under the Swarm Operation section.

Network Configuration

Validator nodes use the libp2p library for gossip/block sync/propagation, and as such, will need to expose several ports to the open internet. Operators should consider adding a DoS prevention solution to their network topology to prevent targeted attacks like TOB-PRYSM-8.

Remote management must be conducted using a dedicated control plane (accessible only from a private network) to avoid exposing remote management ports to the public.

Syncing

Fresh nodes should configure their consensus clients to sync from a recent Weak Subjectivity checkpoint. Syncing a consensus client from the genesis block should be considered unsafe due to long-range attacks. Weak Subjectivity checkpoints must be acquired by a trusted source, preferably from fully-synced nodes maintained by the same operator.

Swarm Operation

Incident Response

Large-scale validator operators must have a documented and well-understood incident response plan in place. The Incident Response Recommendations from Building Secure Contracts provides a good foundation for an enterprise incident response plan, and any operator should have answers for each question from the document.

All incident response plans should be complemented with a set of documented runbooks on how to handle different kinds of potential incidents. The Prysm team provides a mitigation worksheet that offers a good starting point for an incident response runbook. The most relevant risk events are reproduced from the Prysm documentation below, along with some additional risk events that should be considered by large-scale operators.

For each risk event, the operator should generate a plan for:

1. How each risk event is to be detected and subsequently escalated
2. A playbook for the triage/mitigation of each risk event

Risk events:

- Datacenter (where the validator node is hosted) internet connection goes offline
- Validator hardware is physically destroyed or stolen from the datacenter
- A validator's storage media or memory fails
- A validator's OS crashes
- A validator runs out of system resources (disk, RAM, etc.)
- A validator experiences unusual disk/CPU/network activity
- The consensus/execution client has a bug
- A validator instance transitions into an unexpected state
- A validator's validation keys or withdrawal keys are exposed to an attacker
- A validator in the swarm is slashed
- A validator needs to be migrated to a new machine
- A validator is failing to attest to blocks in a timely manner
- A validator is missing block proposal slots

- A validator is subject to a DoS attack
- The validator swarm needs to be migrated to a new consensus/execution client in response to a bug
- The HSM storing the validator's key is destroyed or stolen

The list of risk events above is not exhaustive, and each validator operator should internally identify additional, organization-specific risks that may justify an incident response.

Multiple members of the incident response team should be familiar with, and able to run, each playbook. The incident response team should also consider simulating risk events to practice each playbook. Some organizations use a technique called Chaos Engineering to simulate risk events in a production environment; such a technique may be especially useful for the maintenance and operation of a highly available validator swarm.

Change Management

Given the value at risk in a large validator swarm, operators should adhere to a strict change management policy. Changes to the system should be adequately tested before being deployed. Testing should be conducted in a realistic, multi-client environment such as the Ethereum public testnets.

Each change to the system should be accompanied by monitoring criteria and rollback steps to prevent changes from causing catastrophic system failures or outages.

E. Goroutines Leaking in Prysm Tests

Figure E.1 lists all the goroutines that leaked when we ran Prysm tests with the goleak goroutine leaks detector as described in finding TOB-PRYSM-6.

```
github.com/dgraph-io/ristretto.(*Cache).processItems
github.com/dgraph-io/ristretto.(*defaultPolicy).processItems
github.com/ethereum/go-ethereum/accounts/abi/bind/backends.(*filterBackend).SubscribeNewTxsEvent.func1
github.com/ethereum/go-ethereum/accounts/abi/bind/backends.(*filterBackend).SubscribePendingLogsEvent.func1
github.com/ethereum/go-ethereum/consensus/ethash.(*remoteSealer).loop
github.com/ethereum/go-ethereum/core.(*BlockChain).updateFutureBlocks
github.com/ethereum/go-ethereum/core.(*txSenderCacher).cache
github.com/ethereum/go-ethereum/core/state/snapshot.(*diskLayer).generate
github.com/ethereum/go-ethereum/eth/filters.(*EventSystem).eventLoop
github.com/ethereum/go-ethereum/metrics.(*meterArbiter).tick
github.com/ipfs/go-log/writer.(*MirrorWriter).logRoutine
github.com/libp2p/go-flow-metrics.(*sweeper).run
github.com/libp2p/go-libp2p-pubsub.(*GossipSubRouter).connector
github.com/libp2p/go-libp2p-pubsub.(*GossipSubRouter).heartbeatTimer
github.com/libp2p/go-libp2p-pubsub.(*PubSub).handleSendingMessages
github.com/libp2p/go-libp2p-pubsub.(*PubSub).processLoop
github.com/libp2p/go-libp2p-pubsub.(*Subscription).Next
github.com/libp2p/go-libp2p-pubsub.(*backoff).cleanupLoop
github.com/libp2p/go-libp2p-pubsub.(*peerScore).background
github.com/libp2p/go-libp2p-pubsub.(*validation).validateWorker
github.com/libp2p/go-libp2p/p2p/host/autonat.(*AmbientAutoNAT).background
github.com/libp2p/go-libp2p/p2p/host/basic.(*BasicHost).background
github.com/libp2p/go-libp2p/p2p/host/peerstore/pstoremem.(*memoryAddrBook).background
github.com/libp2p/go-libp2p/p2p/host/pstoremanager.(*PeerstoreManager).background
github.com/libp2p/go-libp2p/p2p/host/resource-manager.(*resourceManager).background
github.com/libp2p/go-libp2p/p2p/net/connmgr.(*BasicConnMgr).background
github.com/libp2p/go-libp2p/p2p/net/connmgr.(*decayer).process
github.com/libp2p/go-libp2p/p2p/net/swarm.(*DialBackoff).background
github.com/libp2p/go-libp2p/p2p/net/upgrader.(*listener).Accept
github.com/libp2p/go-libp2p/p2p/protocol/identify.(*ObservedAddrManager).worker
github.com/libp2p/go-libp2p/p2p/protocol/identify.(*idService).loop
github.com/libp2p/go-libp2p/p2p/transport/quicreuse.(*listener).Accept
github.com/libp2p/go-libp2p/p2p/transport/quicreuse.(*reuse).gc
github.com/libp2p/go-yamux/v4.(*Session).AcceptStream
github.com/libp2p/go-yamux/v4.(*Session).sendLoop
github.com/libp2p/go-yamux/v4.(*Session).startMeasureRTT
github.com/libp2p/go-yamux/v4.(*Stream).Read
github.com/lucas-clemente/quic-go.(*baseServer).accept
github.com/lucas-clemente/quic-go.(*baseServer).run
github.com/lucas-clemente/quic-go.(*connection).run
github.com/lucas-clemente/quic-go.(*incomingStreamsMap[...]).AcceptStream
github.com/lucas-clemente/quic-go.(*packetHandlerMap).runCloseQueue
github.com/lucas-clemente/quic-go.(*receiveStream).readImpl
github.com/lucas-clemente/quic-go.(*sendQueue).Run
github.com/patrickmn/go-cache.(*janitor).Run
github.com/paulbellamy/ratecounter.(*RateCounter).run.func1
github.com/prysmaticlabs/prysm/v3/async.RunEvery.func1
github.com/prysmaticlabs/prysm/v3/async/event.TestFeed_Send.func6.1
github.com/prysmaticlabs/prysm/v3/beacon-chain/execution.(*PowchainCollector).latestStatsUpdateLoop
github.com/prysmaticlabs/prysm/v3/beacon-chain/monitor.(*Service).monitorRoutine
github.com/prysmaticlabs/prysm/v3/beacon-chain/node.(*BeaconNode).Start.func1
github.com/prysmaticlabs/prysm/v3/beacon-chain/p2p/peers/scorers.(*Service).loop
github.com/prysmaticlabs/prysm/v3/beacon-chain/rpc/eth/events.(*Server).StreamEvents
github.com/prysmaticlabs/prysm/v3/beacon-chain/sync.(*Service).registerHandlers
github.com/prysmaticlabs/prysm/v3/beacon-chain/sync.(*Service).subscribeDynamicWithSubnets.func1
github.com/prysmaticlabs/prysm/v3/beacon-chain/sync.(*Service).verifierRoutine
github.com/prysmaticlabs/prysm/v3/beacon-chain/sync/initial-sync.(*Service).waitForStateInitialization
github.com/prysmaticlabs/prysm/v3/beacon-chain/sync/initial-sync.(*blocksFetcher).stop
github.com/prysmaticlabs/prysm/v3/container/leaky-bucket.(*Collector).PeriodicPrune.func1
```

```
github.com/prysmaticlabs/prysm/v3/testing/middleware/engine-api-proxy.(*Proxy).Start
github.com/prysmaticlabs/prysm/v3/time/slots.(*SlotTicker).Done.func1
github.com/prysmaticlabs/prysm/v3/time/slots.(*SlotTicker).start.func1
github.com/prysmaticlabs/prysm/v3/validator/db/kv.(*Store).batchAttestationWrites
github.com/syndtr/goleveldb/leveldb.(*DB).compactionError
github.com/syndtr/goleveldb/leveldb.(*DB).mCompaction
github.com/syndtr/goleveldb/leveldb.(*DB).mpoolDrain
github.com/syndtr/goleveldb/leveldb.(*DB).tCompaction
github.com/syndtr/goleveldb/leveldb.(*session).refLoop
go.opencensus.io/stats/view.(*worker).start
google.golang.org/grpc.(*addrConn).resetTransport
google.golang.org/grpc.(*ccBalancerWrapper).watcher
internal/poll.runtime_pollWait
k8s.io/klog.(*loggingT).flushDaemon
net/http.(*persistConn).writeLoop
```

Figure E.1: List of goroutines that leaked as part of one or multiple tests

F. Glossary

Consensus Client: A client that syncs & validates the Ethereum PoS beacon chain. Consensus clients may be configured to partake in consensus activities as a validator. Many implementations split the consensus client into two smaller clients - a beacon chain client and a validator client.

Beacon Chain Client: A component of a consensus client dedicated to syncing the beacon chain, validating consensus, and persisting the beacon chain.

Validator Client: A component of a consensus client dedicated to participating in consensus. Validator clients interact with beacon chain clients using the official Beacon Chain API. Validator clients are necessary only when the node is configured to participate in consensus as a validator node.

Execution Client: A client that syncs blocks, listens for transactions, and executes blocks/transactions in the EVM.

Node: A complete Ethereum PoS client that syncs the chain to a local database and may be queried for the chain's state. Nodes verify that the synced chain is the one agreed upon by consensus and that the synced chain's execution blocks are valid. Nodes consist of a consensus client and an execution client.

Slasher Node: A node that is configured to log all attestations on the network to detect slashing violations. Detected slashing violations are shared with the rest of the network via gossip.

Validator Node: A node that is participating in consensus.

G. Automated Dynamic Analysis

This appendix describes the setup of the automated dynamic analysis tools and test harnesses used during this audit.

The purpose of automated dynamic analysis

In most software, unit and integration tests are typically the extent to which testing is performed. This type of testing detects the presence of functionality, allowing developers to ensure that the given system adheres to the expected specification. However, these methods of testing do not account for other potential behaviors that an implementation may exhibit.

Fuzzing and property-based testing complement both unit and integration testing by identifying deviations in the expected behavior of a component of a system. These types of tests generate test cases and provide them to the given component as input. The tests then run the components and observe their execution for deviations from expected behaviors.

The primary difference between fuzzing and property testing is the method of generating inputs and observing behavior. Fuzzing typically attempts to provide random or randomly mutated inputs in an attempt to identify edge cases in entire components. Property testing typically provides inputs sequentially or randomly within a given format, checking to ensure a specific property of the system holds upon each execution.

By developing fuzzing and property-based testing alongside the traditional set of unit and integration tests, edge cases and unintended behaviors can be pruned during the development process, which will likely improve the overall security posture and stability of a system.

Tooling

Go supports fuzzing in its standard toolchain beginning in Go 1.18 that can be used through the `go test -fuzz` command and which is also used by the Prysm project. However, the native Go fuzzing framework still lacks some usability or user experience features. An alternative could be Trail of Bits fork of go-fuzz that extends the original go-fuzz project and for which we also have some helper tools:

- go-fuzz-utils: helper package that provides a simple interface to produce random values for various data types and can recursively populate complex structures from raw fuzz data. It can also be used with the native go fuzzing.

State of Prysm fuzzing

Since Prysm already used the native Go fuzzing, we decided to use native Go fuzzing during the audit as well. Prysm implements 18 fuzzing harnesses (figure G.1) that are run for a short period of time during their CI/CD builds. Prysm also uses the fuzzbuzz.io service to

fuzz the project for a longer period of time; however, the results or statistics of this fuzzing are not publicly available.

```
container/trie                  FuzzSparseMerkleTrie_HashTreeRoot
container/trie                  FuzzSparseMerkleTrie_MerkleProof
container/trie                  FuzzSparseMerkleTrie_Insert
container/trie                  FuzzSparseMerkleTrie_VerifyMerkleProofWithDepth
beacon-chain/sync               FuzzValidateBeaconBlockPubSub_Phase0
beacon-chain/sync               FuzzValidateBeaconBlockPubSub_Altair
beacon-chain/sync               FuzzValidateBeaconBlockPubSub_Bellatrix
beacon-chain/p2p                FuzzMsgID
beacon-chain/execution          FuzzForkChoiceResponse
beacon-chain/execution          FuzzExchangeTransitionConfiguration
beacon-chain/execution          FuzzExecutionPayload
beacon-chain/execution          FuzzExecutionBlock
beacon-chain/state/state-native FuzzPhase0StateHashTreeRoot
beacon-chain/state/state-native FuzzAltairStateHashTreeRoot
beacon-chain/state/state-native FuzzBellatrixStateHashTreeRoot
beacon-chain/state/state-native FuzzCapellaStateHashTreeRoot
beacon-chain/state/fieldtrie    FuzzFieldTrie
validator/accounts              FuzzValidateMnemonic
```

Figure G.1: Existing fuzzing harnesses in the Prysm project. The first column is the package in which the harness exists, and the second column is the fuzzing harness function name.

Scripts to fuzz Prysm

In order to run the fuzzing harnesses by ourselves, we prepared the following scripts, which can also be found in figures G.2–8:

- `get-fuzzlist.py`: a Python script that saves all fuzzing harness package paths and names into a `fuzzlist` file. It does it by parsing the output of the `go test -list .` commands executed in each package
- `fuzz-all.sh`: runs all fuzzing harnesses from the `fuzzlist` file using the `fuzz-one.sh` script
- `fuzz-one.sh`: a script to run a single fuzzing harness. It gets `<package-path>` and `<fuzzing-harness-name>` arguments. It also sets 2 CPUs for each harness
- `kill-fuzz.sh`: stops all fuzzers
- `cov-all.sh`: recompiles all harnesses from `fuzzlist` with coverage profiling and run all harnesses against all their inputs
- `cov-one.sh`: gathers coverage from a single harness. Note that it moves the fuzzer-generated corpus files to the fuzzer's `testdata/` directory so we can run all corpus inputs against the fuzzing harness. This is a workaround for the Go native fuzzer's limitation that allows it to run the harness only against inputs from the

`testdata` directory (when providing an argument). The coverage profiles are saved into the `cover/` directory.

- `gen-html.sh`: reads the coverage profiles from the `cover/` directory and renders fuzzing coverage HTML reports into the `html/` directory

These scripts are shown below:

```
import os

#os.system("""find . -type d -exec bash -c "cd {} && go test -list . && cd -" \; >
../golist.out 2>&1""")

data = open('../golist.out').read().splitlines()
fuzz = {}
harnesses = []

for line in data:
    if line.startswith('Fuzz'):
        harnesses.append(line)
    elif line.startswith('ok'):
        directory = line.split('prysm/v3/')[1].split()[0]
        fuzz[directory] = harnesses
        harnesses = []

for k, v in fuzz.items():
    for harness in v:
        print(k, harness)
```

Figure G.2: Python script to create the `fuzzlist` file. Use it as follows: `python3 ./get-fuzzlist.py > fuzzlist`

```
#!/bin/bash
IFS=$'\n'
for args in $(cat ./fuzzlist)
do
    IFS=$' '
    stringarray=($args)
    ./fuzz-one.sh ${stringarray[0]} ${stringarray[1]}
done
```

Figure G.3: The `fuzz-all.sh` script

```
#!/bin/bash
OUTPATH="$(pwd)/outputs/$2"
cd ../$1
go test -v ./ -fuzz="^$2\$" -run="^$2\$" -parallel=2 -cpu=2 > $OUTPATH 2>&1 &
```

Figure G.4: The `fuzz-one.sh` script

```
#!/bin/bash
IFS=$'\n'
for args in $(cat ./fuzzlist)
do
    IFS=$' '
    stringarray=($args)
    ./cov-one.sh ${stringarray[0]} ${stringarray[1]}
done
```

Figure G.5: The `cov-all.sh` script

```
#!/bin/bash
# $1 = dir with harness package ; $2 = harness
OUTPATH="$(pwd)/outputs/$2"
COVERPATH="$(pwd)/cover/$2.cover"

cd ../$1
FUZZPATH=./testdata/fuzz/$2
mkdir -p $FUZZPATH 2>/dev/null
mv $FUZZPATH $FUZZPATH-crashers
ln -s $(go env GOCACHE)/fuzz/github.com/prysmaticlabs/prysm/v3/$1/$2 $FUZZPATH

go test -v ./ -run="^$2/" -cover -coverprofile=$COVERPATH

rm $FUZZPATH
mv $FUZZPATH-crashers $FUZZPATH
cd -
```

Figure G.6: The cov-one.sh script

```
#!/bin/bash
killall -v -9 go
killall -v -9 --regexp '.*test'
killall -v -9 --regexp 'state-.*'
```

Figure G.7: The `kill-fuzz.sh` script

```
#!/bin/bash
for i in $(ls ./cover); do
       go tool cover -html=./cover/$i -o ./html/$i.html
done
```

Figure G.8: The `gen-html.sh` script

Our improvements and new fuzzing harnesses

In order to fuzz efficiently and increase the possibility of catching bugs, we recommend inspecting the code coverage of the fuzzers after running them for a longer period of time and checking if it is possible to improve their coverage. This can be done by either 1) adding more inputs to the initial corpus so that more paths are known for the fuzzer up front or 2) modifying the code so that inefficient paths are mocked and are not executed during fuzzing.

During the audit, we ran the existing and new fuzzing harnesses for a few days and then analyzed their code coverage by using the scripts included above. One thing we noticed is that some harnesses do not cover the code paths that process certain gossip message topics, as shown in figure G.9. We improved this by adding additional initial test cases to those fuzzing harnesses. The patch that does this can be seen in figure G.10.

Apart from this, we have also implemented three new fuzzing harnesses whose code we include in figures G.11–G.12:

- `FuzzDecodePubsubMessage`: a fuzzing harness to test the decoding of pubsub messages. Note that it does not check the full topic format but it could and it could if the used functions would validate the whole format (which could be a problem as described in TOB-PRYSM-15)

- `FuzzDeserializeRequestBodyIntoContainer`: a harness that tests the processing of data by certain API endpoints

- `FuzzDifferentialAttestingIndices`: harness that ensures that the `attestation.AttestingIndices` function returns data in the expected format

We ran those fuzzing harnesses for a few days as well, but they did not find any crashes.

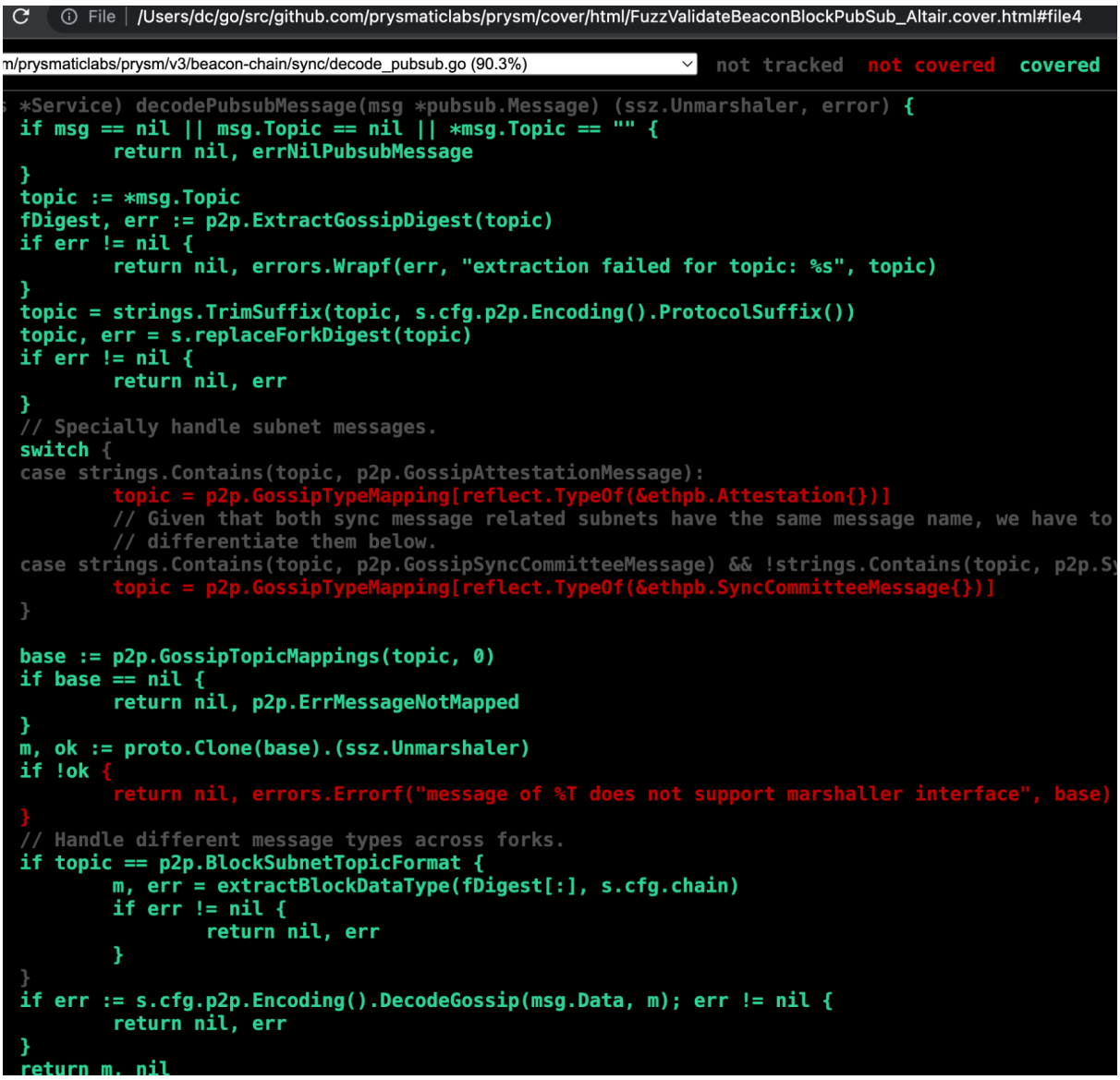

Figure G.9: Code coverage from one of the fuzzing harnesses. As shown above, we never reached code paths related to certain pubsub topics.

```
diff --git a/beacon-chain/p2p/pubsub_fuzz_test.go b/beacon-chain/p2p/pubsub_fuzz_test.go
index 6d9408114..9c61a5dfe 100644
--- a/beacon-chain/p2p/pubsub_fuzz_test.go
+++ b/beacon-chain/p2p/pubsub_fuzz_test.go
@@ -11,8 +11,10 @@ import (
 )

 func FuzzMsgID(f *testing.F) {
-       validTopic := fmt.Sprintf(p2p.BlockSubnetTopicFormat, []byte{0xb5, 0x30, 0x3f, 0x2a}) + "/" +
encoder.ProtocolSuffixSSZSnappy
-       f.Add(validTopic)
+       for _, format := range p2p.AllTopics() {
+               validTopic := fmt.Sprintf(format, []byte{0xb5, 0x30, 0x3f, 0x2a}) + "/" +
encoder.ProtocolSuffixSSZSnappy
+               f.Add(validTopic)
```

```
+        }

         f.Fuzz(func(t *testing.T, topic string) {
                 _, err := p2p.ExtractGossipDigest(topic)
diff --git a/beacon-chain/sync/sync_fuzz_test.go b/beacon-chain/sync/sync_fuzz_test.go
index 00fdbaf72..2496c687c 100644
--- a/beacon-chain/sync/sync_fuzz_test.go
+++ b/beacon-chain/sync/sync_fuzz_test.go
@@ -83,7 +83,19 @@ func FuzzValidateBeaconBlockPubSub_Phase0(f *testing.F) {
         assert.NoError(f, err)
         topic = r.addDigestToTopic(topic, digest)

+        // Use current/valid peer id as pid
+        f.Add(string(p.PeerID()), []byte("junk"), buf.Bytes(), []byte(topic))
+
+        // Add one input with invalid peer id
         f.Add("junk", []byte("junk"), buf.Bytes(), []byte(topic))
+
+        for _, topicFormats := range p2p.AllTopics() {
+                digest, err := r.currentForkDigest()
+                assert.NoError(f, err)
+                topic = r.addDigestToTopic(topicFormats, digest)
+                f.Add("junk", []byte("junk"), buf.Bytes(), []byte(topic))
+        }
+
         f.Fuzz(func(t *testing.T, pid string, from, data, topic []byte) {
                 r.cfg.p2p = p2ptest.NewFuzzTestP2P()
                 r.rateLimiter = newRateLimiter(r.cfg.p2p)
@@ -164,7 +176,19 @@ func FuzzValidateBeaconBlockPubSub_Altair(f *testing.F) {
         assert.NoError(f, err)
         topic = r.addDigestToTopic(topic, digest)

+        // Use current/valid peer id as pid
+        f.Add(string(p.PeerID()), []byte("junk"), buf.Bytes(), []byte(topic))
+
+        // Add one input with invalid peer id
         f.Add("junk", []byte("junk"), buf.Bytes(), []byte(topic))
+
+        for _, topicFormats := range p2p.AllTopics() {
+                digest, err := r.currentForkDigest()
+                assert.NoError(f, err)
+                topic = r.addDigestToTopic(topicFormats, digest)
+                f.Add("junk", []byte("junk"), buf.Bytes(), []byte(topic))
+        }
+
         f.Fuzz(func(t *testing.T, pid string, from, data, topic []byte) {
                 r.cfg.p2p = p2ptest.NewFuzzTestP2P()
                 r.rateLimiter = newRateLimiter(r.cfg.p2p)
@@ -245,7 +269,19 @@ func FuzzValidateBeaconBlockPubSub_Bellatrix(f *testing.F) {
         assert.NoError(f, err)
         topic = r.addDigestToTopic(topic, digest)

+        // Use current/valid peer id as pid
+        f.Add(string(p.PeerID()), []byte("junk"), buf.Bytes(), []byte(topic))
+
+        // Add one input with invalid peer id
         f.Add("junk", []byte("junk"), buf.Bytes(), []byte(topic))
+
+        for _, topicFormats := range p2p.AllTopics() {
+                digest, err := r.currentForkDigest()
+                assert.NoError(f, err)
+                topic = r.addDigestToTopic(topicFormats, digest)
+                f.Add("junk", []byte("junk"), buf.Bytes(), []byte(topic))
+        }
+
         f.Fuzz(func(t *testing.T, pid string, from, data, topic []byte) {
                 r.cfg.p2p = p2ptest.NewFuzzTestP2P()
                 r.rateLimiter = newRateLimiter(r.cfg.p2p)
```

Figure G.10: Patch that increases the code coverage of fuzzing harnesses that deal with gossip message topics by adding additional inputs to the initial fuzzing corpus.

```
func FuzzDeserializeRequestBodyIntoContainer(f *testing.F) {
  var bodyJson bytes.Buffer
  err := json.NewEncoder(&bodyJson).Encode(defaultRequestContainer())
  assert.NoError(f, err)

  // Add an example input
  f.Add(bodyJson.Bytes())

  f.Fuzz(func(t *testing.T, data []byte) {
    var bodyJson bytes.Buffer
    bodyJson.Write(data)

    container := &testRequestContainer{}
    err := DeserializeRequestBodyIntoContainer(&bodyJson, container)

    if err == nil {
      err2 := ProcessRequestContainerFields(container)
      if err2 == nil {
        req := http.Request{
          Header: http.Header{},
        }

        // At this point we expect no error since if we were able to
        // deserialize the struct, we should also be able to serialize it back
        err3 := SetRequestBodyToRequestContainer(container, &req)
        assert.Equal(t, nil, err3)
      }
    }
  })
}
```

Figure G.11: A fuzzing harness for some API middleware request processing logic. This harness should be added to the `prysm/api/gateway/apimiddleware/process_request_test.go` file.

```
func FuzzDifferentialAttestingIndices(f *testing.F) {
  f.Add(8, 14, 8)
  f.Fuzz(func(t *testing.T, length int, bitsToSet int, committeeLength int) {
    // Allow only up to 64 bits
    if length <= 0 || length > 64 || committeeLength <= 0 || committeeLength > 64 {
      return
    }

    bitsList := bitfield.NewBitlist(uint64(length))
    // Set random (fuzzer generated) bit indices
    var i uint64
    for i = 0; i < uint64(length); i++ {
```

```
      bit := bitsToSet & (1 << i)
      bitsList.SetBitAt(i, bit != 0)
    }

    // Create committee with sequential ids
    committee := make([]primitives.ValidatorIndex, committeeLength)
    for i = 0; i < uint64(committeeLength); i++ {
      committee[i] = primitives.ValidatorIndex(i)
    }

    got, err := attestation.AttestingIndices(bitsList, committee)

    // The only allowed error case is when the lengths do not match
    if err != nil {
      assert.NotEqual(t, length, committeeLength)
    } else {
      assert.NotNil(t, got)
      assert.Equal(t, length, committeeLength)

      // Ensure proper values were returned, according to the bitsList
      for idx, value := range bitsList.BitIndices() {
        assert.Equal(t, got[idx], uint64(value))
      }
    }
  })
}
```

Figure G.12: A differential fuzzing harness to validate the `attestation.AttestingIndices` functionality. This harness should be added to the `prysm/proto/prysm/v1alpha1/attestation/attestation_utils_test.go` file.

```
func FuzzDecodePubsubMessage(f *testing.F) {
  _, err := signing.ComputeForkDigest(params.BeaconConfig().GenesisForkVersion,
make([]byte, 32))
  require.NoError(f, err)

  // Add inputs with all known topic formats
  for _, topic := range p2p.AllTopics() {
    // TODO/FIXME: Probably format the topics accordingly
    f.Add([]byte(""), topic)
  }

  f.Fuzz(func(t *testing.T, data []byte, topic string) {
    msg := &pubsub.Message{Message: &pb.Message{}}
    msg.Message.Topic = &topic
    msg.Message.Data = data
    s := &Service{
      cfg: &config{p2p: p2ptesting.NewTestP2P(t), chain:
&mock.ChainService{ValidatorsRoot: [32]byte{}, Genesis: time.Now()}},
    }

    got, err := s.decodePubsubMessage(msg)
    // TODO/FIXME: Ideally, this should probably assert all known errors
    if got != nil {
      assert.NoError(t, err)
    } else {
      // if got is nil, there should be an error
      assert.NotNil(t, err)
    }
  })
}
```

Figure G.13: A fuzzing harness to test the `decodePubSubMessage` function. This harness should be added to the `prysm/beacon-chain/sync/decode_pubsub_test.go` file.

Further fuzzing ideas and guidance

The Prysm fuzzing harnesses can be further improved in the following ways:

1. The harnesses, similarly to unit tests, should assert as much state as possible, to be able to catch regression bugs in case of future updates. The assertions should also be designed so that the test or harness would fail if a new field were added into a structure used in the harness/test. To give some examples:

    a. The `FuzzMsgId` fuzzing harness should ensure that a non-error digest value matches the expected one, or at least that it has the expected length and conforms to the expected format.

b. The `FuzzForkChoiceResponse` fuzzing harness should perform its assertions differently, so that if a new field is added to the go-ethereum's `ForkChoiceResponse.PayloadStatus` structure it would be detected by the harness as being not handled at all (in case the tested code would set that field).

c. Tests such as `TestProcessRewardsAndPenaltiesPrecompute` should assert all balance states, not only those that are expected to be changed.

2. The state-changing protocols, such as gossip and sync, could be fuzzed in such a way that the fuzzer generates a series of operations and the initial state chain, and then it runs those protocol operations or transactions against the chain. However, in order to be effective, such a harness will likely require code modifications or mocking of certain functionality to make the code efficient and fully deterministic (e.g., not relying on time, files on disk, signing and hashing). It is also very important that such a harness leverage code coverage instrumentation so that the fuzzer reuses interesting inputs, which would increase its effectiveness in finding edge cases.

3. Differential fuzzing harnesses could be added to test the functionalities documented in the consensus specification against other consensus nodes. This, however, would require the creation of separate programs in both consensus node softwares that would trigger only the functionality to be tested and require the functionality to have a similar interface in both solutions. Such fuzzing harnesses could ensure that different nodes give the same exact results (or errors) for the same inputs.

H. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings.

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`. We used Semgrep version 1.18.0. To run Semgrep on the codebase, we ran the following in the root directory of the project:

```
semgrep --config "p/trailofbits" --sarif  --metrics=off --output semgrep.sarif
```

We also ran the tool with the following rules (configs):

- `p/ci`
- `p/security-audit`
- `r/go.lang`

We recommend integrating Semgrep into the project's CI/CD pipeline. Integrate at least the rules with HIGH confidence and those with MEDIUM confidence and HIGH impact.

We recommend using Trail of Bits's set of Semgrep rules (from the repository or less preferably from the registry) and dgryski rules which can be used via `p/semgrep-go-correctness`.

CodeQL

We installed CodeQL by following CodeQL's installation guide. We used CodeQL version 2.12.4, with Go version 1.20.0.

After installing CodeQL, we ran the following command to create the project database for the Prysm repository:

```
codeql database create codeql.db --language=go
```

We then ran the following command to query the database:

```
codeql database analyze codeql.db -o codeql.sarif --format=sarif-latest -j0 -- tob-go-all
```

We used our `tob-go-all` query pack, which includes selected queries from the built-in `codeql/go-all` query pack and dozens of Trail of Bits's private queries.

CodeQL reported a substantial number of findings that we determined were not worth reporting at the current stage of the project. However, we strongly recommend that all issues be reviewed and fixed by the Prysm team.

Moreover, we recommend integrating CodeQL into the CI/CD pipeline, using either GitHub Advanced Security or custom integration. Please note that the amount of false positives can be reduced by running only high and very-high precision queries.

Other static analysis tools

In addition to Semgrep and CodeQL, we used the following tools during the audit and highly recommend integrating them into the CI/CD pipeline.

- `Go-sec` is a static analysis utility that looks for a variety of problems in Go codebases. Notably, `go-sec` will identify potential stored credentials, unhandled errors, cryptographically troubling packages, and similar problems.
- `Go-vet` is a very popular static analysis utility that searches for more Go-specific problems within a codebase, such as mistakes pertaining to closures, marshaling, and unsafe pointers. `Go-vet` is integrated within the `go` command itself, with support for other tools through the `vettool` command line flag.
- `Staticcheck` is a static analysis utility that identifies both stylistic problems and implementation problems within a Go codebase. Note that many of the stylistic problems `staticcheck` identifies are also indicative of potential "problem areas" in a project.
- `Ineffassign` is a static analysis utility that identifies ineffectual assignments. These ineffectual assignments often identify situations in which errors go unchecked, which could lead to undefined behavior of the program due to execution in an invalid program state.
- `Errcheck` is a static analysis utility that identifies situations in which errors are not handled appropriately.
- `gokart` is a static analysis tool for Go that finds vulnerabilities using the single static assignment (SSA) form of the Go source code.
- `goleak` is a Goroutine leak detector that detects leaks by directly hooking into a project's test suite as described in its documentation (by adding the `goleak.VerifyTestMain(m)` line into all of the tests' `TestMain` functions).

Please also see our blog post on Go security assessment techniques for further discussion of the Go-related analysis tools.

I. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From September 18 to September 22, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Prysm team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 15 issues described in this report, the Prysm team has resolved eight issues and has not resolved the remaining seven issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Unhandled errors	Resolved
2	os.Create() used without checking for an existing file	Resolved
3	Passing sensitive configuration values through the command line may leak to other processes on the system	Unresolved
4	Configuration files containing potentially sensitive values are not checked for permissions	Unresolved
5	Panics by the beacon-chain and validator RPC APIs can panic are recovered but may lead to crashes due to memory exhaustion	Resolved
6	Goroutine leaks can lead to Denial of Service	Unresolved
7	Potential deadlock if the Feed.Send panic is recovered and the function is retried	Resolved
8	Block Proposer DDoS	Unresolved

9	The db backup endpoint may be triggered via SSRF or when visiting an attacker website, which may cause a DoS	**Unresolved**
10	Maximum gRPC message size of MaxInt32 (2GB) set in beacon-chain/server may lead to DoS	**Unresolved**
11	EpochParticipation.UnmarshalJSON may parse invalid data	**Resolved**
12	Uint256.UnmarshalJSON may parse invalid data	**Resolved**
13	Failed assertions in the FuzzExecutionPayload fuzzing harness	**Resolved**
14	The JWT authentication docs suggest generating the secret using third-party websites	**Resolved**
15	Potentially insufficient gossip topic validation	**Unresolved**

Detailed Fix Review Results

TOB-PRYSM-1: Unhandled errors
Resolved in PR #12578 and PR #12938. The Prysm team has addressed all identified instances of unhandled errors and modified the code to properly handle error values. Also, the configuration file for the `ineffassign` static code analysis tool has been refactored, removing the `only_files` section and retaining only the `exclude_files` section, allowing it to more effectively exclude specific files from analysis based on regular expressions.

TOB-PRYSM-2: os.Create() used without checking for an existing file
Resolved in PR #12536 The Prysm team has updated the file creation logic within the `tools/interop/convert-keys/main.go` and `tools/specs-checker/download.go` files, opting for the use of `os.OpenFile`. This choice incorporates the `os.O_CREATE` and `os.O_EXCL` flags, which guarantee the creation of a new file when it is absent and safeguards against unintentional overwrites of existing files, thereby minimizing potential data loss or conflicts during file creation.

TOB-PRYSM-3: Passing sensitive configuration values through the command line may leak to other processes on the system
Unresolved. The client provided the following context for this finding's fix status:

> *This will not be handled at the current moment as any changes to how config files are handled and how secrets are handled can break users in the short term. Also some hardening steps can cause issues for non-technical users running nodes, however we will keep on tracking this so we can find an acceptable solution to it.*

TOB-PRYSM-4: Configuration files containing potentially sensitive values are not checked for permissions
Unresolved. The client provided the following context for this finding's fix status:

> *This will not be handled at the current moment as any changes to how config files are handled and how secrets are handled can break users in the short term. Also some hardening steps can cause issues for non-technical users running nodes, however we will keep on tracking this so we can find an acceptable solution to it.*

TOB-PRYSM-5: Panics by the beacon-chain and validator RPC APIs can panic are recovered but may lead to crashes due to memory exhaustion
Resolved in PR #12932. The Prysm team has updated the page pagination logic; input validation checks were added to ensure that both `pageSize` and `totalSize` are non-negative, with informative error messages in cases of invalid input. Additionally, a validation check for the `token` value derived from the `pageToken`was implemented to prevent negative values.

TOB-PRYSM-6: Goroutine leaks can lead to Denial of Service
Unresolved. The client provided the following context for this finding's fix status:

> *We have elected to not fix this for now as a lot of these leaks are due to test-setup and handling them isn't straightforward. However we will think of a long-term solution in the future on how to handle them.*

TOB-PRYSM-7: Potential deadlock if the Feed.Send panic is recovered and the function is retried
Resolved in PR #12464. The Prysm team has updated the Send function in the `async/event/feed.go` file to include a mutex unlock before the panic is triggered.

TOB-PRYSM-8: Block Proposer DDoS
Unresolved. The client provided the following context for this finding's fix status:

> *There is no way to fix this on our end, it requires a protocol level mitigation rather than something on Prysm's end.*

TOB-PRYSM-9: The db backup endpoint may be triggered via SSRF or when visiting an attacker website, which may cause a DoS
Unresolved. The client provided the following context for this finding's fix status:

> *This can't be addressed immediately as it is a breaking change, the only time we can do this is during a major version bump (next hardfork). So we are waiting till then to enable this.*

TOB-PRYSM-10: Maximum gRPC message size of MaxInt32 (2GB) set in beacon-chain/server may lead to DoS
Unresolved. The client provided the following context for this finding's fix status:

> *The rpc endpoints are assumed to only be exposed to trusted/peers entities. For that reason we will not be changing the size as anyone with access to the rpc endpoints can cause DOS due to the many things the endpoints provide(state regeneration, returning full states back etc).*

TOB-PRYSM-11: EpochParticipation.UnmarshalJSON may parse invalid data
Resolved in PR #12534. The Prysm team has revised the section of the code for decoding Base64-encoded strings. In the original implementation, there was an assumption that the input string was enclosed in double quotes, and these quotes were removed before decoding. However, in the revised code, the team opted to directly convert the byte slice to a string and subsequently employ the `strings.Trim()` function to eliminate any surrounding double quotes.

TOB-PRYSM-12: Uint256.UnmarshalJSON may parse invalid data
Resolved in PR #12540. The Prysm team updated the `UnmarshalJSON` function in both the `api/client/builder/types.go` and `beacon-chain/rpc/apimiddleware/structs_marshalling.go` files to verify the presence and correctness of double quotes around the JSON string by examining the first and last characters of the input byte slice. If the string lacks proper double quotes or is too short, it returns an error message accordingly.

TOB-PRYSM-13: Failed assertions in the FuzzExecutionPayload fuzzing harness
Resolved in PR #12541. The Prysm team has made updates to its fuzzing execution payload target to handle capella payloads and added logic to handle the nil payload case in the capella payload marshalling. The client provided the following context for this finding's fix status:

> *We now only fuzz execution payloads for capella and compare them. Geth adds fields to their executable data, so there is no effective way to handle the fuzz target across different forks.*

TOB-PRYSM-14: The JWT authentication docs suggest generating the secret using third-party websites
Resolved in PR #790. The Prysm team has updated the JWT authentication documentation by removing the line suggesting the use of third-party websites to generate authentication secret values.

TOB-PRYSM-15: Potentially insufficient gossip topic validation
Unresolved. The client provided the following context for this finding's fix status:

> *We are not able to tackle this at the moment because it would require a large change to gossip topic validation which will take a while and brings its own risks. However, we do eventually plan to address this in the future when there is bandwidth in the team.*

J. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.