[Report] Drand Audit & Fuzzing - Fuzzinglabs

Fuzzinglabs Audit Report

Introduction

Fuzzinglabs is a leading cybersecurity startup, excelling in vulnerability research, fuzzing, and blockchain security. We provide expert courses, custom security tools, and products tailored to diverse needs. Our expertise spans security audits, blockchain security, training, cryptocurrency forensics, and telecom security. Renowned for our advanced fuzzing techniques and trainings, we are dedicated to improving the security of various tech sectors, including blockchain protocols. At Fuzzinglabs, we're committed to enhancing cybersecurity through our innovative and expert solutions.

Audit Objectives

- **Objective 1:** Learn How Drand Works
  - Understand the basics of drand, what it does, and the parts it's made of.
- **Objective 2:** Search for Common Bugs
  - Look for typical mistakes or errors in Go that often happen in systems like drand.
- **Objective 3:** Make Fuzzing Tools for Important Parts
  - Create fuzzing harnesses to test important parts of drand by leveraging existing unit tests.
- **Objective 4:** Build a Malicious Network
  - Make a malicious network to attack other nodes

Team Contributors

- **Member 1**: Nabih Benazzouz
- **Member 2**: Mohammed Benhelli
- **Member 3**: Patrick Ventuzelo

Project Duration

- **Start Date:** 08/11/2023
- **End Date:** 12/01/2024
- **Total Duration:** 20 Days

Key Findings

- **Critical Severity:** 1
- **Medium Severity:** 0
- **Low Severity:** 5

- **Total Vulnerabilities Found:** 6

Tool Used

- **Go-fuzz**
- **Gofuzz**
- **Go Fuzzing Framework**
- **govulncheck**

---

1) Some information about Drand

Utility

The main goal is to provide a secure source of randomness

It’s a Randomness as a service network

Publicly verifiable & unbiased: drand periodically delivers publicly verifiable and unbiased randomness. Any third party can fetch and verify the authenticity of the randomness and by that make sure it hasn't been tampered with.

How does it work

- **Setup:**
  - Each node first generates a ***long-term public/private key pair***.
  - Then all of the public keys are written to a ***group file*** together with some further metadata required to operate the beacon.
  - After this group file has been distributed, the nodes perform a ***distributed key generation*** (DKG) protocol to create the collective public key and one private key share per server. The participants NEVER see/use the actual (distributed) private key explicitly but instead **utilize their respective private key shares** for the generation of public randomness.
- **Generation:**
  - After the setup, the nodes switch to the randomness generation mode.
  - Any of the nodes can initiate a randomness generation round by broadcasting a message which all the other participants sign using a t-of-n threshold version of the ***Boneh-Lynn-Shacham*** (BLS) signature scheme and their respective private key shares.
  - Once any node (or third-party observer) has gathered t partial signatures, it can reconstruct the full BLS signature (using Lagrange interpolation).
  - The signature is then hashed using SHA-256 to ensure that there is no bias in the byte representation of the final output. This hash corresponds to the collective random value and can be verified against the collective public key.

  A drand network is made up of a set number of nodes running the drand protocol. Before generating random numbers, they all agree on a *threshold parameter*.

Each node creates a signature, and random numbers are created by each node broadcasting a part of their signature to the rest of the network.

Each node waits and collects these signatures until it has enough signatures to match the *threshold parameter*, then this node can create the final signature.

The final signature is a regular Boneh–Lynn–Shacham signature (opens new window) that can be verified against the rest of the network. If that signature is correct, then the randomness is simply the hash of that signature.

What is a node?

A drand node is a server that runs the drand code, participates in the distributed key generation (DKG) process and the randomness generation, and can reply to public request API. It can instantiate and run multiple independent internal randomness processes, where each of them has its randomness generation frequency. The following representation is what gets embedded in each group configuration file, one per randomness process. This is what each process knows about other drand nodes on its network:

```
type Node struct {
  Key []byte  // public key on bls12-381 G1
    Addr string // publicly reachable address of the node
    TLS bool // reachable via TLS
    Signature []byte
    Index uint32 // index of the node w.r.t. to the network
}
```

What is a network?

Each group of nodes that runs a specific process (e.g., one with the same set of parameters) constitutes a separate network.
Each network will act as an independent beacon generator. ****

What is a beacon

A drand beacon is what the drand network periodically creates and that can be used to derive the randomness. A beacon contains the signature of the previous beacon generated, the round of this beacon, and the signature. See the beacon chain section for more information.

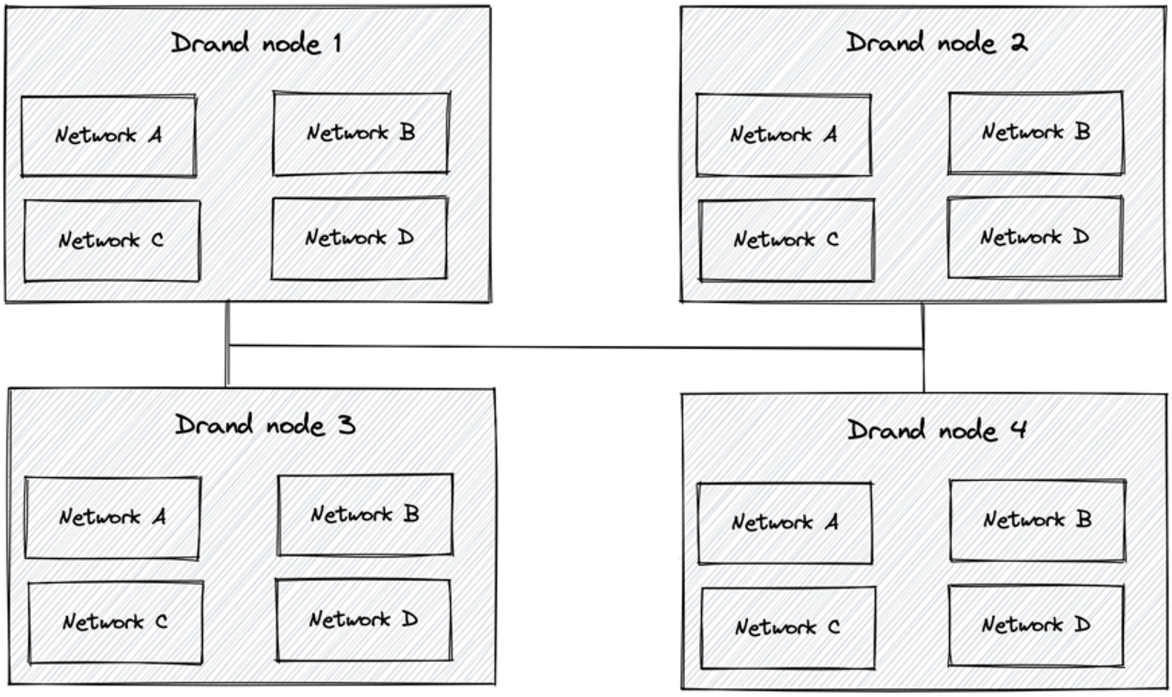
What is a scheme?

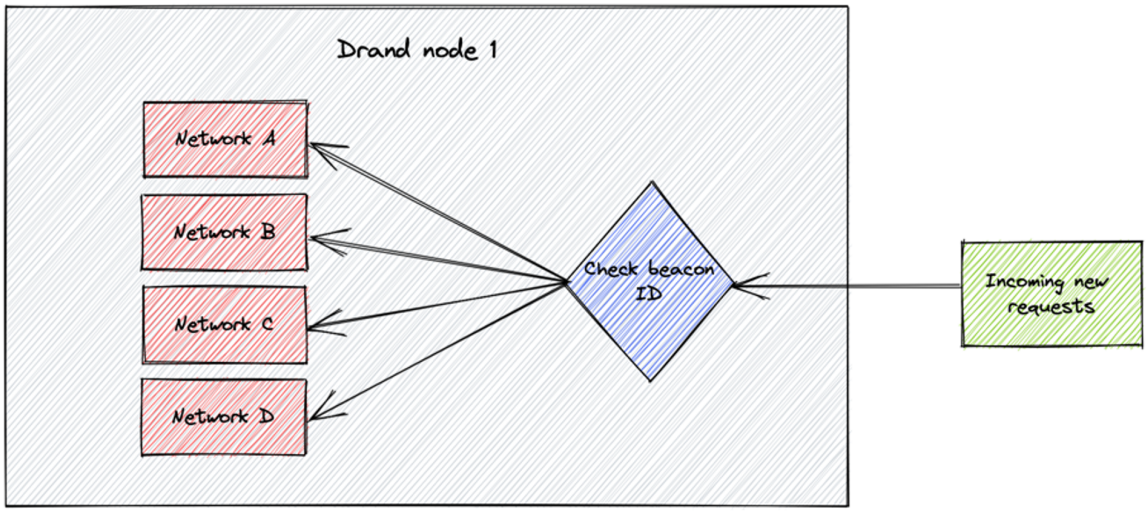
A scheme is a network-level configuration that a coordinator can set when starting a new network. There is a fixed quantity of possible values. Each one establishes a different set of parameters which affects some working aspects of the network. If no scheme is set on the network starting, a default value is used.

```
// DefaultSchemeID is the default scheme ID.
const DefaultSchemeID = "pedersen-bls-chained"

// UnchainedSchemeID is the scheme ID used to set unchained randomness on beacons.
const UnchainedSchemeID = "pedersen-bls-unchained"

// ShortSigSchemeID is the scheme ID used to set unchained randomness on beacons with a red
```

```
uced beacon size
const ShortSigSchemeID = "bls-unchained-on-g1"
```

Some schemas to better understand how it works





Drand Spec

https://drand.love/docs/specification/

2) Drand Security

https://drand.love/docs/security-model/#notations

https://drand.love/docs/security-model/#attack-vectors

Drand node

a node that is running the drand daemon and participating in the creation of the randomness in one or many networks at the same time. The **drand network** is the set of drand nodes connected. Each drand network has a long-term public key and a private share (after running the setup/resharing phase).

Relay node

a node that is connected to a drand daemon and exposes an Internet-facing interface allowing it to fetch the public randomness. The **relay network** is the set of relay nodes, partially / potentially connected.

When the type of the node is not specified in the document, it is assumed from the context - most often it refers to a drand node.

Corrupted node

a node that is in the control of an attacker. In this case, the attacker has access to all the cryptographic material this node possesses and the networking authorization. For example, if a relay node is corrupted, an attacker has a direct connection to a drand node.

Offline node

a node that is unreachable from an external point of view. It can be offline from the point of view of another drand node or a relay node. The document tries to clarify in which context when relevant.

Online node

a node that is running the binary (drand or relay depending on the context) and sends packets out to the Internet that are correctly received by the endpoint(s).

3) Drand communication

Drand currently uses gRPC as the networking protocol. All exposed services and protobuff definitions are in the protocol.proto file for the intra-nodes protocols and in the api.proto file.

Public endpoints

https://drand.love/developer/http-api/#public-endpoints

4) Understanding the repository

https://drand.love/developer/organization/#top-level-packages

- `chain` - Code for generating the sequence of beacons (implementation of which is in `chain/beacon`) after setup.
  - `boltdb` - BoltDB storage backend.

    - `errors` - common errors for the chain package.
    - `memdb` - in-memory storage backend.
    - `postgresdb` - PostgreSQL storage backend.
- `client` - The drand client library - composition utilities for fail-over and reliable abstraction.
    - `client/grpc` - The concrete gRPC client implementation.
    - `client/http` - The concrete HTTP client implementation.
    - `client/test` - Mock client implementations for testing.
- `cmd` - Binary entry points.
    - `cmd/client` - A client for fetching randomness.
    - `cmd/client/lib` - A common library for creating a client shared by `cmd/client` and `cmd/relay`.
    - `cmd/drand-cli` - The main drand group member binary.
    - `cmd/relay` - A relay that pulls randomness from a drand group member and exposes an HTTP server interface.
    - `cmd/relay-gossip` - A relay that pulls randomness from a group member and publishes it over a libp2p gossipsub topic.
- `crypto` - Holds the schemes supported by drand.
- `core` - The primary Service interface of drand comamands.
    - `core/migration` - A library for migrating drand files from single-beacon to multi-beacon versions.
- `demo` - A framework for integration testing.
- `deploy` - Records of previous drand deployments.
- `docker` - Helpers for docker image packaging.
- `docs` - Here.
- `entropy` - A common abstraction for ingesting randomness.
- `fs` - Utilities for durable state storage.
- `hooks` - Docker helper entrypoint.
- `http` - The publicly exposed HTTP server for exposing randomness.
- `key` - Validation of signatures.
- `log` - Common logging library.
- `lp2p` - Utilities for constructing a libp2p host.
- `lp2p/client` - The concrete gossip client implementation.
- `metrics` - The prometheus metrics server.
- `net` - gRPC service handlers for inter-node communication.
- `protobuf/drand` - Definitions for the wire format interface of inter-node communication.
- `test` - Testing helper utilities.
    - `test/docker` - Files and related scripts for testing drand networks on docker.

5) Old audits

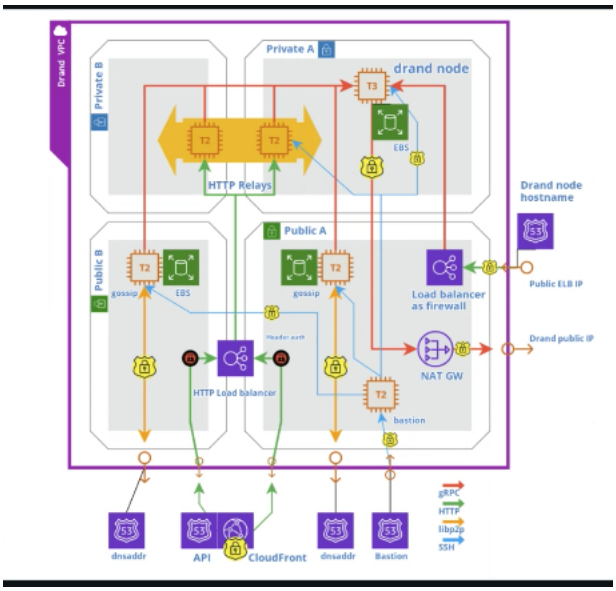
We only found one old audit

https://drive.google.com/file/d/1fCy1ynO78gJLCNbqBruzHx7bh72Tu-q2/view?pli=1

6) Meeting Yolan <> Fuzzinglabs — 23/11/2023

We did a meeting with Yolan to have some information about Drand what they are waiting for from us and which components should we focus on.

- **GRPC Communication:** Restricted to private server ports.
- **Server Configurations:** Some servers have public listen ports with a reverse proxy like Nginx (responsible for checks such as rate limiting).
- **Security Concerns:** Private servers aim to maintain proximity due to potential risks of denial-of-service attacks on other nodes.
- **Local Control Port:** Emphasized for internal control.
- **Branch Focus:** Avoidance of seeking the master branch.
- **DKG Significance:** Specifically regarding fuzzing the broadcast element.
- **Excluded Elements:** PubSub and Relays deemed out of the current scope.
- **Potentially Included:** Consideration for Relay HTTP, especially in assessing APIs taking arguments; preliminary analysis by Nabih indicates no immediate action is needed.
- **Critical Sub-Packages:** Identified as crypto, internal/dkg, handler/http, and internal/core.
- **Crypto Dependencies:** Evaluation of potential signature verification vulnerabilities leading to system crashes.
- **Fuzzing Depth:** Exploration possibility beneath the protobuff layer.
- **Specific Targets:**
  - `drand_beacon_public.go`: Multiple targets, including `BeaconProcess.PublicRandStream`.
  - `drand_daemon_public.go`: Numerous targets identified.
  - `sync_manager`: Further examination is required, encompassing `BeaconProcess.SyncChain`.
  - **BoldDB:** Verify dependencies but no fuzzing is needed.
- **Regression Testing:** Focus on the regression folder.
- **Utilization of Demo:** Recommended for a comprehensive understanding.
- **Evaluation of `sync-manager`:** Emphasized for assessment.
- **Branch Switch:** Consider moving to the following branch:
  - drand-v2.0.0 branch link



7) Beginning of the audit

Branch used: feature/drand-v2.0.0

https://github.com/drand/drand/tree/feature/drand-v2.0.0

Gosec

https://github.com/securego/gosec

Nothing critical was found by `gosec`

```
[/home/raefko/drand_fuzzing/drand-develop/internal/fs/fs.go:159] - G304 (CWE-22): Potential
file inclusion via variable (Confidence: HIGH, Severity: MEDIUM)
    158:
  > 159:    if dest, err = os.Create(destFilePath); err != nil {
    160:        return err
```

```
[/home/raefko/drand_fuzzing/drand-develop/internal/fs/fs.go:145] - G304 (CWE-22): Potential
file inclusion via variable (Confidence: HIGH, Severity: MEDIUM)
    144:
  > 145:    if src, err = os.Open(origFilePath); err != nil {
    146:        return err
```

```
[/home/raefko/drand_fuzzing/drand-develop/internal/fs/fs.go:72] - G304 (CWE-22): Potential
file inclusion via variable (Confidence: HIGH, Severity: MEDIUM)
    71:     }
  > 72:     return os.OpenFile(file, os.O_RDWR, rwFilePermission)
    73: }
```

```
[/home/raefko/drand_fuzzing/drand-develop/internal/net/control.go:71] - G104 (CWE-703): Err
ors unhandled. (Confidence: HIGH, Severity: LOW)
    70:
  > 71:     g.lis.Close()
    72: }
```

```
[/home/raefko/drand_fuzzing/drand-develop/internal/fs/fs.go:67] - G104 (CWE-703): Errors un
handled. (Confidence: HIGH, Severity: LOW)
    66:     }
  > 67:     fd.Close()
    68:     if err := os.Chmod(file, rwFilePermission); err != nil {
```

```
[/home/raefko/drand_fuzzing/drand-develop/internal/core/drand_beacon_control.go:349] - G104
(CWE-703): Errors unhandled. (Confidence: HIGH, Severity: LOW)
    348:        logger.Errorw("", "start_follow_chain", "unable to insert genesis block",
"err", err)
  > 349:        store.Close()
    350:        return fmt.Errorf("unable to insert genesis block: %w", err)
```

golangci-lint

Nothing critical was found by `golangci-lint`

https://github.com/golangci/golangci-lint

```
handler/http/server.go:97:3: string `/{` has 4 occurrences, make it a constant (goconst)
        "/{"+chainHashParamKey+"}/public/latest",
        ^
handler/http/server.go:102:52: string `.PublicRand` has 2 occurrences, make it a constant
(goconst)
        instrument(handler.PublicRand, chainHashParamKey+".PublicRand"),
                                                           ^
internal/drand-cli/control.go:399:14: string `Finished correcting faulty beacons, ` has 2 o
ccurrences, make it a constant (goconst)
                    l.Infow("Finished correcting faulty beacons, " +
                            ^
internal/drand-cli/control.go:381:4: string `   --> %.3f %% - ` has 2 occurrences, make it
a constant (goconst)
            "\\t--> %.3f %% - "+
            ^
internal/drand-cli/cli.go:215:75: string `' flag.` has 2 occurrences, make it a constant (g
oconst)
        "Requires to specify the chain-hash using the '" + hashInfoNoReq.Name + "' flag.",
                                                                                ^
internal/drand-cli/control.go:379:29: string `  synced round up to %d ` has 2 occurrences,
make it a constant (goconst)
        spin.Suffix = fmt.Sprintf("  synced round up to %d "+
                                  ^
```

```
internal/drand-cli/control.go:400:7: string `we recommend running the same command a second
time to confirm all beacons are now valid` has 2 occurrences, make it a constant (goconst)
                        "we recommend running the same command a second time to confirm all
beacons are now valid")
                        ^
internal/drand-cli/dkg_cli.go:143:3: string `To use TLS, prefix their address with 'http
s://'` has 3 occurrences, make it a constant (goconst)
        "To use TLS, prefix their address with 'https://'",
        ^
common/key/keys.go:33:2: var-naming: struct field Tls should be TLS (revive)
    Tls       bool
```

govulncheck

govulncheck command - golang.org/x/vuln/cmd/govulncheck - Go Packages
Govulncheck reports known vulnerabilities that affect Go code.
https://pkg.go.dev/golang.org/x/vuln/cmd/govulncheck

```
Scanning your code and 451 packages across 82 dependent modules for known vulnerabilities...

Vulnerability #1: GO-2023-2382
    Denial of service via chunk extensions in net/http
  More info: https://pkg.go.dev/vuln/GO-2023-2382
  Standard library
    Found in: net/http/internal@go1.21.4
    Fixed in: net/http/internal@go1.21.5
    Example traces found:
      #1: demo/node/node_inprocess.go:310:26: node.LocalNode.GetBeacon calls http.body.Read,

Vulnerability #2: GO-2023-2185
    Insecure parsing of Windows paths with a \??\ prefix in path/filepath
  More info: https://pkg.go.dev/vuln/GO-2023-2185
  Standard library
    Found in: path/filepath@go1.21.4
    Fixed in: path/filepath@go1.21.5
    Platforms: windows
    Example traces found:
      #1: demo/node/node_subprocess.go:409:13: node.NodeProc.Stop calls exec.Cmd.Run, which e
      #2: demo/node/node_subprocess.go:408:25: node.NodeProc.Stop calls exec.Command, which c
      #3: demo/node/node_subprocess.go:408:25: node.NodeProc.Stop calls exec.Command, which e
      #4: demo/node/node_subprocess.go:409:13: node.NodeProc.Stop calls exec.Cmd.Run, which e
      #5: demo/node/node_subprocess.go:409:13: node.NodeProc.Stop calls exec.Cmd.Run, which e
      #6: demo/node/node_subprocess.go:408:25: node.NodeProc.Stop calls exec.Command, which c
      #7: demo/node/node_subprocess.go:408:25: node.NodeProc.Stop calls exec.Command, which e
      #8: demo/node/node_subprocess.go:409:13: node.NodeProc.Stop calls exec.Cmd.Run, which e

Vulnerability #3: GO-2023-2153
    Denial of service from HTTP/2 Rapid Reset in google.golang.org/grpc
  More info: https://pkg.go.dev/vuln/GO-2023-2153
  Module: google.golang.org/grpc
    Found in: google.golang.org/grpc@v1.57.0
```

```
    Fixed in: google.golang.org/grpc@v1.58.3
    Example traces found:
      #1: internal/net/control.go:51:25: net.ControlListener.Start calls grpc.Server.Serve, w
      #2: internal/net/listener.go:66:30: net.NewGRPCListenerForPrivate calls grpc.NewServer
      #3: internal/net/control.go:51:25: net.ControlListener.Start calls grpc.Server.Serve

Your code is affected by 3 vulnerabilities from 1 module and the Go standard library.

Share feedback at https://go.dev/s/govulncheck-feedback.
```

semgrep

https://github.com/semgrep/semgrep

```
Semgrep rule registry URL is https://semgrep.dev/registry.
Scanning 145 files with 37 go rules.
  100%|████████████████████████████████████████████████████████████████████████████████


┌─────────┐
│ Results │
└─────────┘

Findings:

  drand-fuzzinglabs/demo/lib/orchestrator.go
     go.lang.security.audit.crypto.math_random.math-random-used
        Do not use `math/rand`. Use `crypto/rand` instead.
        Details: https://sg.run/6nK6

         ▶▶┆ Autofix ▶ crypto/rand
          9┆ "math/rand"


  drand-fuzzinglabs/internal/chain/beacon/sync_manager.go
     go.lang.security.audit.crypto.math_random.math-random-used
        Do not use `math/rand`. Use `crypto/rand` instead.
        Details: https://sg.run/6nK6

         ▶▶┆ Autofix ▶ crypto/rand
          7┆ "math/rand"


  drand-fuzzinglabs/internal/dkg/broadcast.go
     go.lang.security.audit.crypto.math_random.math-random-used
        Do not use `math/rand`. Use `crypto/rand` instead.
        Details: https://sg.run/6nK6

         ▶▶┆ Autofix ▶ crypto/rand
```

```
          8¦ "math/rand"


  drand-fuzzinglabs/internal/net/control.go
     go.grpc.security.grpc-server-insecure-connection.grpc-server-insecure-connection
        Found an insecure gRPC server without 'grpc.Creds()' or options with credentials. This
        allows for a connection without encryption to this server. A malicious attacker could ta
        with the gRPC message, which could compromise the machine. Include credentials derived f
        an SSL certificate in order to create a secure gRPC connection. You can create credentia
        using 'credentials.NewServerTLSFromFile("cert.pem", "cert.key")'.
        Details: https://sg.run/5Q5l

         32¦ grpcServer := grpc.NewServer()
```

8) Fuzzing harnesses

We did a quick code audit on Drand and we used some unit tests to create fuzzing harnesses

```
# Done #
./handler/http/server_test.go
./internal/chain/memdb/store_test.go
./internal/chain/store_test.go
./internal/chain/beacon/callbacks_test.go
./internal/net/peer_test.go
./internal/net/gateway_test.go
./internal/chain/beacon/cache_test.go
./internal/chain/beacon/store_test.go
./internal/chain/beacon/sync_manager_test.go
./internal/chain/beacon/node_test.go
./internal/core/drand_control_test.go
./internal/dkg/store_test.go
./internal/dkg/dkg_migration_test.go
./internal/dkg/state_machine_test.go
./internal/dkg/broadcast_test.go
./crypto/curve_test.go
./crypto/schemes_test.go
./common/key/keys_test.go
./common/chain/info_test.go
./common/key/encoding_test.go
./common/key/group_test.go
./internal/entropy/entropy_test.go

# DOUBT #
./internal/core/util_test.go
./internal/core/drand_beacon_test.go
./internal/core/drand_test.go
./internal/dkg/actions_active_test.go
./common/time_test.go

# No Need #
./internal/core/drand_daemon_test.go
./internal/net/control_test.go
./internal/chain/beacon/memdb_test.go
```

```
./internal/chain/beacon/boltdb_test.go
./internal/context/context_test.go
./internal/chain/postgresdb/pgdb/pgdb_test.go
./internal/chain/boltdb/store_test.go
./internal/chain/boltdb/trimmed_test.go
./internal/chain/boltdb/boltdb_test.go
./common/beacon_test.go
./common/version_test.go
./common/key/store_test.go
./internal/metrics/threshold_monitor_test.go
./internal/metrics/metrics_test.go
./internal/chain/beacon/postgresdb_test.go
./internal/core/postgresdb_test.go
./internal/core/memdb_test.go
./internal/core/boltdb_test.go
./internal/fs/fs_test.go
./common/log/log_test.go
./demo/postgresdb_test.go
./demo/demo_test.go
./demo/memdb_test.go
./demo/boltdb_test.go
./internal/drand-cli/dkg_cli_test.go
./internal/drand-cli/proposal_file_test.go
./internal/drand-cli/cli_test.go
```

During the assessment, we had to create fuzzing harnesses based on Drand **unit tests**. However, we encountered several constraints with the fuzzing process.

The majority of the code is protocol-based, with **minimal parsing involved**. This setup renders **traditional fuzzing less effective** since there's a scarcity of parsing mechanisms to target. Additionally, the system heavily relies on dynamically generated structures during execution. Engaging in fuzzing would require us to **manually generate** these structures with data, valid or invalid. Yet, this approach poses the risk of encountering **false positives**. For instance, several functions don't verify whether a field is different from **Nil** before utilizing it.

How to run the harnesses

All the harnesses are inside the `*_test.go` files.

Inside the folder `drand-fuzzinglabs` there is a `harnesses` folder.

```
drand-fuzzinglabs
├── cmd
│   └── drand
│       └── drand.go
├── codecov.yml
├── CODE_OF_CONDUCT.md
├── common
│   ├── beacon.go
│   ├── beacon_test.go
│   ├── chain
│   │   ├── convert.go
...
...
```

```
...
├── harnesses
│   ├── generate_cov.sh
│   └── list_harnesses.sh
...
...
...
```

This folder contains two bash scripts. The main one is the `list_harnesses.sh`.

You can run the script using

```
./list_harnesses.sh timeout=<number of seconds>
```

```
❯ ./list_harnesses.sh timeout=5
-------------FUZZINGLABS--------------
File Path: ../crypto/curve_test.go
Function Prototype: func FuzzBLS12381Compatv112(f *testing.F) {
----------------Fuzzing Start------------------
fuzz: elapsed: 0s, gathering baseline coverage: 0/194 completed
fuzz: elapsed: 1s, gathering baseline coverage: 194/194 completed, now fuzzing with 16 workers
fuzz: elapsed: 3s, execs: 151393 (50457/sec), new interesting: 0 (total: 194)
----------------Fuzzing End------------------
/home/raefko/drand_fuzzing/drand-fuzzinglabs/harnesses
----------------Coverage Generation Start------------------
[+] Entering ../crypto
[+] Copying FuzzBLS12381Compatv112 fuzz test files
[+] Creating coverage folder
[+] Running all test files in /home/raefko/drand_fuzzing/drand-fuzzinglabs/crypto/testdata/fuzz/FuzzBLS12381Compatv112
PASS
coverage: 11.7% of statements
ok      github.com/drand/drand/crypto   0.003s
[+] Generating html coverage report
[+] Done
----------------Coverage Generation End------------------
```

The `timeout=` option is the number of seconds each harness will run.

For this harness `FuzzBLS12381Compatv112` under `../crypto/curve_test.go` you can find the `html` coverage file inside the folder `../crypto/testdata/coverage/FuzzBLS12381Compatv112/`

The pattern will be the same for all the other harnesses.

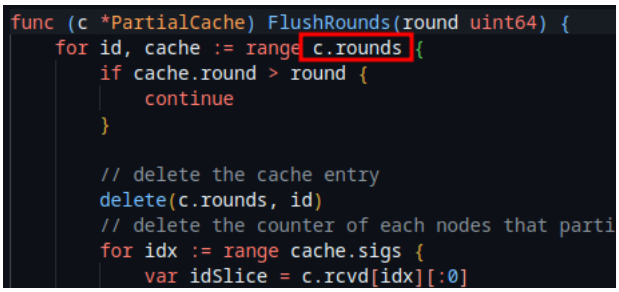
FYI: Corpus dataset

The go fuzzing framework stores all the test files of the corpus inside this folder

```
"$HOME/.cache/go-build/fuzz/github.com/drand/drand/"
```

9) Network fuzzer

Drand's code relies heavily on context or structures created during its execution. This makes it challenging to create independent fuzzing harnesses. Here's a simple example:



Our fuzzer generated a `PartialCache` struct with `round` set to nil. This leads to a false positive because the field `round` will never be nil when generated by a drand node/network.

So, we came up with a plan. By using your demo script, we got a better grasp of how communication happens within a Drand node. With a few adjustments, we managed to send custom data during the node's initialization phase. Our goal was to create something like a 'Node Chaos Monkeyware'. Following this, we modified the interceptor to get arbitrary data from the fuzzer and put it inside the requests.

Folder

Inside the folder `drand-fuzzinglabs` there is a `network_fuzzer` folder.

```
drand-fuzzinglabs
├── cmd
│   └── drand
│       └── drand.go
├── codecov.yml
├── CODE_OF_CONDUCT.md
├── common
│   ├── beacon.go
│   ├── beacon_test.go
│   ├── chain
│   │   ├── convert.go
...
...
...
├── network_fuzzer
│   ├── clean.sh
│   ├── crash_server.go
│   ├── fuzz_helpers
│   │   ├── client_fuzzing_interceptors.go
│   │   ├── constants.go
│   │   ├── crash_server_utils.go
│   │   ├── fuzzing_utils.go
│   │   ├── globals.go
│   │   ├── interceptor_utils.go
│   │   ├── packet_mutators.go
│   │   ├── panic_interceptors.go
│   │   ├── request_mutators.go
│   │   ├── response_mutators.go
│   │   └── server_fuzzing_interceptors.go
│   ├── fuzzing_orchestrator.go
│   ├── fuzz_loop.go
│   ├── fuzz_loop_test.go
│   ├── globals.go
```

```
│   ├── images
│   │   ├── fuzzing_loop.png
│   │   ├── interceptors.png
│   │   └── seed_usage.png
│   ├── launch_orchestrator_go_fuzz.sh
│   ├── launch_orchestrator_libfuzzer.sh
│   ├── launch_orchestrator_official_fuzzer.sh
│   ├── node_utils.go
│   ├── patch_go.py
│   ├── README.md
│   ├── replay_seed.py
│   ├── scenarios.go
│   ├── utils.go
│   └── visualize_coverage.sh

...
...
...
```

The network_fuzzer is a harness wrapper around a drand node initialization. You can find the main function inside `network_fuzzer/fuzz_loop.go`.

We added some files inside the drand code to avoid patching the official code

```
./internal/chain/postgresdb/schema/schema_fuzz.go
./internal/drand-cli/dkg_cli_fuzz.go
./internal/drand-cli/cli_fuzz.go
./internal/drand-cli/daemon_fuzz.go
./internal/drand-cli/control_fuzz.go
./internal/core/drand_beacon_fuzz.go
./internal/core/drand_daemon_fuzz.go
./internal/core/client_public_fuzz.go
./internal/net/listener_fuzz.go
./internal/net/client_grpc_fuzz.go
./internal/net/control_fuzz.go
./internal/net/gateway_fuzz.go
./demo/node/node_fuzz.go
./demo/node/node_inprocess_fuzz.go
./demo/node/node_subprocess_fuzz.go
./demo/lib/orchestrator_fuzz.go
```

Those files are used when building with the `//go:build fuzzydrand` directive.

Run a session

Run with go-fuzz with coverage:

```
./launch_orchestrator_go_fuzz.sh # no debug logs
FUZZ_DEBUG_LOG=y ./launch_orchestrator_go_fuzz.sh # print debug logs
./visualize_coverage.sh # see coverage
```

Run with go-fuzz with libfuzzer:

```
./launch_orchestrator_libfuzzer.sh # no debug logs
FUZZ_DEBUG_LOG=y ./launch_orchestrator_libfuzzer.sh # print debug logs
```

NOT RECOMMENDED

Run with the official go fuzzer:

```
./patch_go.py # patch go std source
./launch_orchestrator_official_fuzzer.sh # no debug logs
FUZZ_DEBUG_LOG=y ./launch_orchestrator_official_fuzzer.sh # print debug logs
```

Replay a crash

With the script

1**) get the seed**

To replay a crash search for the seed first, it will be like:

```
[+] Renew the global network_fuzzer generator with a new seed: <HEX_SEED>
```

2) **Replay the crash**

Finally, you can replay the crash with:

```
FUZZ_DEBUG_LOG=y ./replay_seed.py <HEX_SEED>
```

Sometimes you'll need to relaunch the test several times to replay the crash.

Manually

1**) get the seed**

To replay a crash search for the seed first, it will be like:

```
[+] Renew the global network_fuzzer generator with a new seed: <HEX_SEED>
```

2) **Create a test to replay the seed**

Create a test file like this one and copy the hex seed:

```
package network_fuzzer

import (
    "encoding/hex"
    "fmt"
    "github.com/drand/drand/crypto"
    "github.com/drand/drand/internal/chain"
    "github.com/drand/drand/network_fuzzer/fuzz_helpers"
    "github.com/drand/kyber/share/dkg"
    "github.com/google/gofuzz/bytesource"
    "math/rand"
    "reflect"
    "runtime"
    "testing"
)
```

```
func TestReplayFuzz(t *testing.T) {
    dataFuzz, err := hex.DecodeString("<HEX_SEED>")
    if err != nil {
       panic(err)
    }
    // `github.com/google/gofuzz` and `github.com/drand/drand` use `math/rand`, so we need to se
    bs := bytesource.New(dataFuzz)
    rand.Seed(int64(bs.Uint64()))

    go func() {
       for {
          msg := <-panicChan
          panic(fmt.Errorf("[-] Panic with hex seed %s: %s", hex.EncodeToString(dataFuzz), msg))
       }
    }()

    // in-process tricks to easily renew the seed
    fuzzing_helpers.DataFuzz = dataFuzz
    fuzzing_helpers.DataFuzzGen = fuzzing_helpers.NewFuzzGeneratorFromSeed(dataFuzz)

    fmt.Printf("[+] Renew the global network_fuzzer generator with a new seed: %s\n", hex.Encode
    resetState()
    fmt.Printf("[+] The panic log file is /tmp/%s/panic.log\n", tmpDir)
    fmt.Printf("[+] The error log file is /tmp/%s/error.log\n", tmpDir)

    var beaconID string
    fuzzGen := fuzzing_helpers.NewFuzzGeneratorFromSeed(dataFuzz)
    n := int(newRandomMinUIntFuzz(fuzzGen, 16, 9))
    nRound := int(newRandomMinUIntFuzz(fuzzGen, 5, 2))
    period := strPeriods[int(newRandomMinUIntFuzz(fuzzGen, uint(len(strPeriods)), 0))]
    fuzzGen.Fuzz(&beaconID)
    //! up to 1/3 of the nodes can be fuzzy
    fuzzyNodesCount = int(newRandomMinUIntFuzz(fuzzGen, uint(n/3), 1))
    fuzzyNodesOffset = n
    thr := dkg.MinimumT(fuzzyNodesCount + n)
    cryptoSchemes := []func() *crypto.Scheme{
       crypto.NewPedersenBLSChained,
       crypto.NewPedersenBLSUnchained,
       crypto.NewPedersenBLSUnchainedG1,
       crypto.NewPedersenBLSUnchainedSwapped,
    }
    sch := cryptoSchemes[int(newRandomMinUIntFuzz(fuzzGen, uint(len(cryptoSchemes)), 0))]()
    storages := []chain.StorageType{chain.MemDB, chain.BoltDB}
    stor := storages[int(newRandomMinUIntFuzz(fuzzGen, uint(len(storages)), 0))]

    c := computeConfig(n, thr, period, sch, beaconID, "", stor)
    orch := newOrchestrator(c)
    orch.nRound = nRound

    defer orch.Shutdown()
    setSignalForGoFuzz(orch)

    if !serverStarted {
```

```
        fmt.Println("[+] Starting crash server")
        go func() {
           if err := orch.crashServer.ListenAndServe(); err != nil {
              panic(fmt.Sprintf("[-] Error while starting crash server: %v", err))
           }
        }()
        serverStarted = true
    }

    scenario := scenarios[int(newRandomMinUIntFuzz(fuzzGen, uint(len(scenarios)), 0))]
    fmt.Printf("[+] Scenario: %s\n", runtime.FuncForPC(reflect.ValueOf(scenario).Pointer()).Name
    scenario(orch, fuzzGen)
}
```

3) **Replay the crash**

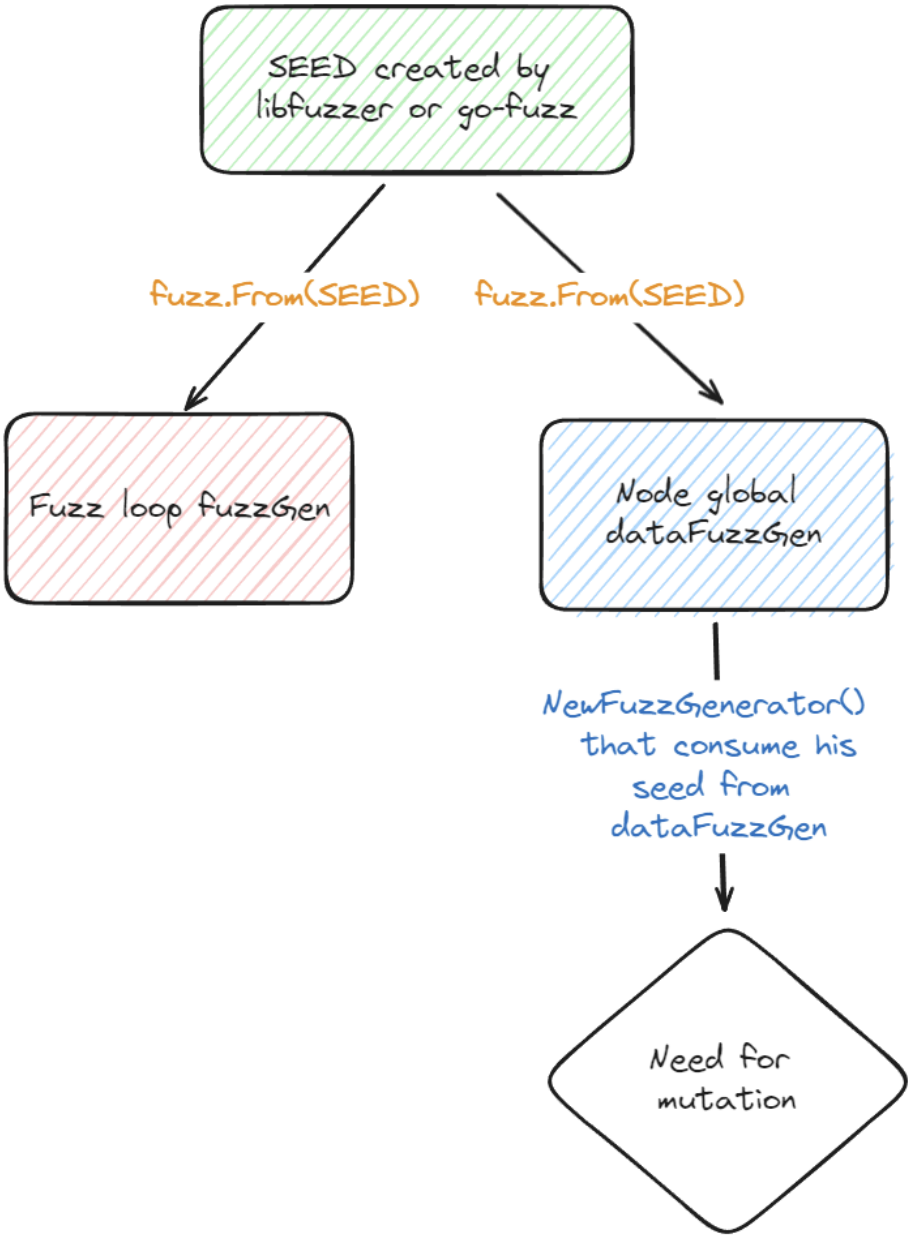
Finally, you can replay the crash with:

```
FUZZ_DEBUG_LOG=y go test -tags fuzzydrand -v -run TestReplayCrash
```

Sometimes you'll need to relaunch the test several times to replay the crash.

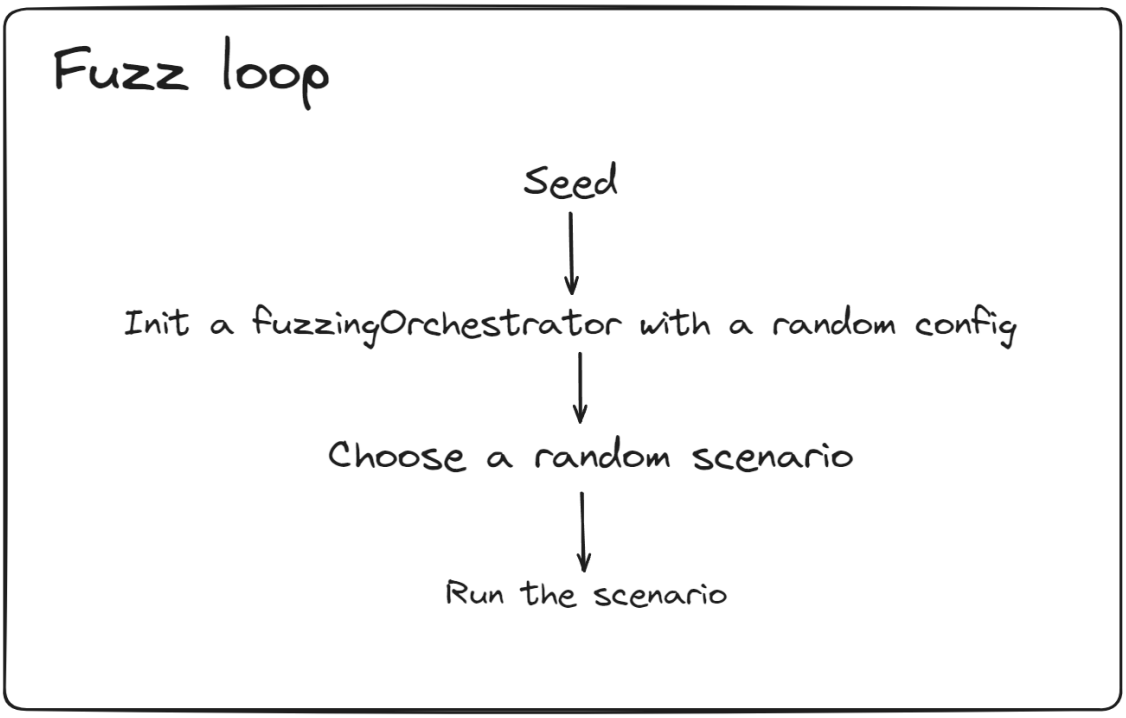
A quick review of the implementation

Seed usage through a campaign:



This choice was made because we want to be able to reproduce the same scenario but also want a good repartition of the mutations. The easiest way would have been to use the same generator but gofuzz can't handle well a highly concurrent environment.

Fuzzing loop:

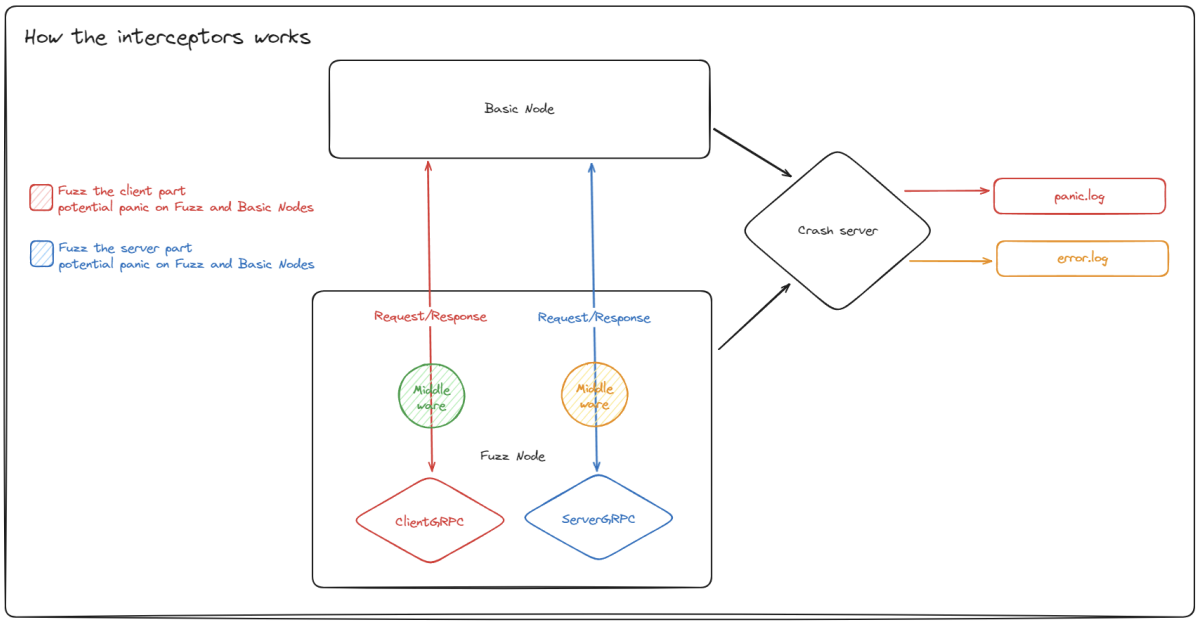


The file `fuzz_loop.go` contains the entry point for `libfuzzer` and `go-fuzz` runners.

The file `fuzz_loop_test.go` contains the entry point for the official Go fuzzer. But the issue with this runner is that it times out, and we can't control this value.

However, we made a monkey patch in `patch_go.py` if you want to use this runner. **But remember that the testing library will behave differently after a patch.**

Fuzzing interceptors:



The fuzzing interceptors are defined in `fuzz_helpers` and are used to intercept the gRPC calls client side and server side. They are used to mutate the messages before they are sent and after they are received in the fuzzy nodes.

If a method is not implemented, the interceptor will send a panic.

Crash server:

The crash server is a simple HTTP server that receives crash and error reports from the fuzzers. The logs received are stored in the `/tmp/drand-fuzzinglabs-TIMESTAMP/` directory. The files are `panic.log` and `error.log`.

How to add a new scenario:

The scenarios are defined in `scenarios.go` and are of `type fuzzingScenario func(orch *fuzzingOrchestrator, fuzzGen *fuzz.Fuzzer) int`.

You can create a new scenario by adding a new function in `scenarios.go` and adding it to the list of scenarios in `scenarios.go`.

```
...

var scenarios = []fuzzingScenario{
    ...
    myNewScenario,
}

...

func myNewScenario(orch *fuzzingOrchestrator, fuzzGen *fuzz.Fuzzer) int {
    // Your scenario here
    return 0
}
```

Potential improvements

- Use docker or Kubernetes to create environments with more metrics on bandwidth and CPU/RAM usage
- Better logging
- Emulate AWS environment using localstack
- Add support for Postgres storage
- Implement property-based checks in the scenarios and use them to detect invariants
- Add scenarios with malicious nodes to test network resilience against corruption attacks DoS, broadcast channel assumption, etc.
- Add scenarios to cover 100% of the gRPC protobuff during sessions.
- Get rid of `github.com/google/gofuzz`

10) Findings

1. GetRandom function with a size of 0 gives duplicate randomness [REPORTED]

Executive summary

During the process of auditing the code and developing fuzzing harnesses, we identified a unit test that utilizes the `GetRandom` function.

As per this `unit test`, it is expected that `GetRandom` should not produce identical randomness for two separate samples.

Upon further examination of the `GetRandom` code, we observed that the `randomBytes` generated by the `make` function do not undergo any prior verification.

This could potentially result in the generation of duplicate values if the size argument passed to `GetRandom` is `0`.

Vulnerability Details

- **Severity:** Informative
- **Affected Component:** Unit test in `internal/entropy`. Can also affect users if the package is used as a library or compiled to wasm.

Root Cause Analysis

The crash is caused by the fact that the `GetRandom` code function is being called with a size of 0. According to the report, this can lead to the generation of duplicate randomness.

However, the test function `TestNoDuplicatesDefaultCrash` is designed to fail if `GetRandom` returns the same output for two separate calls, which is indicated by the condition `bytes.Equal(random1, random2).`

So, if `GetRandom` is indeed generating duplicate randomness when called with a size of 0, this would cause the test function to call `t.Fatal`, which stops the test immediately and marks it as failed. This is why the test is "crashing".

This is due to a lack of verification before the generation

https://github.com/drand/drand/blob/6481f5be22c8afd7e74ab63f274d6918909649ff/internal/entropy/entropy.go#L16C1-L31C1

```
func GetRandom(source io.Reader, n uint32) ([]byte, error) {
    if source == nil {
        source = rand.Reader
    }

    randomBytes := make([]byte, n)
```

```
    bytesRead, err := source.Read(randomBytes)
    if err != nil || uint32(bytesRead) != n {
        // If customEntropy provides an error,
        // fallback to Golang crypto/rand generator.
        _, err := rand.Read(randomBytes)
        return randomBytes, err
    }
    return randomBytes, nil
}
```

Generating an array of size 0 will result in a `RandomBytes` with no byte inside.

It's important to note that this isn't a crash in the traditional sense (like a segmentation fault or uncaught exception), but rather the test function behaving as designed and indicating that there's a problem with the `GetRandom` function.

Reproducer

```
func TestNoDuplicatesDefaultCrash(t *testing.T) {
        random1, err := GetRandom(nil, 0)
        if err != nil {
            t.Fatal("Getting randomness failed:", err)
        }

        random2, err := GetRandom(nil, 0)
        if err != nil {
            t.Fatal("Getting randomness failed:", err)
        }
        if bytes.Equal(random1, random2) {
            t.Fatal("Randomness was the same for two samples, which is incorrect.")
        }
    }
```

2. Slice bounds OOR in dkg packet processing [REPORTED]

Executive Summary

During the audit of drand, we developed a fuzzing node that inits the beacons and intercepts every request to put arbitrary data inside. It works like a malicious node in the context of a classic network. Our tool was able to trigger a bug in this function

`func (d *Process) Packet(ctx context.Context, packet *drand.GossipPacket) (*drand.EmptyDKGResponse, error)`

Vulnerability Details

- **Severity:** Critical
- **Affected Component:** `internal/dkg/actions_passive.go`

Root Cause Analysis

There is no verification before the line 32 which leads to a `panic: runtime error: slice bounds out of range [:8] with length 6`

```
packetName := packetName(packet)
packetSig := hex.EncodeToString(packet.Metadata.Signature)
```

```
shortSig := packetSig[1:8]
```

link:
https://github.com/drand/drand/blob/84ef2452fb2847c8ed252c8fde607801dfc48847/internal/dkg/actions_passive.go#L32

Reproducer

```
package dkg

import (
    "context"
    "testing"
    "time"

    "github.com/drand/drand/protobuf/drand"
)

func TestPacketSliceCrash(t *testing.T) {
    d := &Process{}
    metadata := drand.GossipMetadata{Signature: []byte("foo")}
    gossipPacket := drand.GossipPacket{Metadata: &metadata}
    ctx, _ := context.WithTimeout(context.Background(), 10*time.Minute)

    d.Packet(ctx, &gossipPacket)

}
```

3. Panic when trying to init slices of pointer with negative size [REPORTED]

Executive Summary

During the code audit and the creation of fuzzing harnesses, we found that there is a unit test that uses the `make` function with no verification of the slice size.

The make function in Go is used to initialize slices, maps, and channels. In this case, it's being used to initialize slices of pointers to `Pair` and `Node` types. The second argument to make is the size of the slice to be created.

Vulnerability Details

- **Severity:** Informative
- **Affected Component:** Unit test in `common/key/keys_test.go`

Root Cause Analysis

The test function of `BatchIdentities` takes an integer `n` as an argument and directly uses it to create slices without any validation. If a negative value is passed for `n`, `make` will panic

Reproducer

```
func TestTriggerBugBatchIdentities(t *testing.T) {
        _, _ = BatchIdentities(t, -1)
    }
```

4. Wrong empty slice check in kyber dependency [REPORTED]

Executive Summary

During the audit of drand, we discovered an incorrect empty slice check within the `func NewDistKeyHandler(c *Config) (*DistKeyGenerator, error)`.

This particular function is invoked by `func (d *Process) startDKGExecution(...)`.

Vulnerability Details

- **Severity:** Undetermined
- **Affected Component:** `internal/dkg/execution.go`

link: https://github.com/drand/kyber/blob/master/share/dkg/dkg.go#L195

Root Cause Analysis

```
package dkg
....
func NewDistKeyHandler(c *Config) (*DistKeyGenerator, error) {
    if c.NewNodes == nil && c.OldNodes == nil {
        return nil, errors.New("dkg: can't run with empty node list")
    }
    ...
}
```

Checking an empty slice with `slice == nil` can be wrong if the slice was created using `make`.

Reproducer

```
package lab

import (
    "github.com/drand/kyber/share/dkg"
    "github.com/stretchr/testify/require"
    "testing"
)

func TestPedersenDkgCheckEmptyList(t *testing.T) {
    c := dkg.Config{
        Suite:          nil,
        Longterm:       nil,
        OldNodes:       make([]dkg.Node, 0),
        PublicCoeffs:   nil,
        NewNodes:       make([]dkg.Node, 0),
        Share:          nil,
        Threshold:      0,
        OldThreshold:   0,
        Reader:         nil,
        UserReaderOnly: false,
        FastSync:       false,
        Nonce:          nil,
        Auth:           nil,
        Log:            nil,
    }
    _, err := dkg.NewDistKeyHandler(&c)
```

```
    require.ErrorContains(t, err, "dkg: can't run with empty node list")
}
```

5. A `beaconID` containing a trailing slash will break the start command [REPORTED]

Executive Summary

During a fuzzing session of the `network_fuzzer`, we found that if the `beaconId` contains a trailing slash, `key.NewFileStore` will create more directories than expected.

Vulnerability Details

- **Severity:** Low
- **Affected Component:** `common/key/store.go`

link: https://github.com/drand/drand/blob/84ef2452fb2847c8ed252c8fde607801dfc48847/common/key/store.go#L86

Root Cause Analysis

The root cause is located in `/common/key/store.go:86`

```
func NewFileStore(baseFolder, beaconID string) Store {
    beaconID = common.GetCanonicalBeaconID(beaconID)

    store := &fileStore{baseFolder: baseFolder, beaconID: beaconID}

    keyFolder := fs.CreateSecureFolder(path.Join(baseFolder, beaconID, FolderName))
    groupFolder := fs.CreateSecureFolder(path.Join(baseFolder, beaconID, GroupFolderName))

    store.privateKeyFile = path.Join(keyFolder, keyFileName) + privateExtension
    store.publicKeyFile = path.Join(keyFolder, keyFileName) + publicExtension
    store.groupFile = path.Join(groupFolder, groupFileName)
    store.shareFile = path.Join(groupFolder, shareFileName)
    store.distKeyFile = path.Join(groupFolder, distKeyFileName)

    return store
}
```

The use of `path.Join` is vulnerable to path traversal attacks. If the `beaconId` contains a trailing slash, the function will create more directories than expected.

For example, if the `beaconId` is `back/../../../../../../home/ziion/Desktop` in the given test case.

Drand cli will create the key and group directories on the Desktop.

Reproducer

```
package lab

import (
    "github.com/drand/drand/common/key"
    "github.com/drand/drand/common/testlogger"
    "github.com/drand/drand/crypto"
    "github.com/drand/drand/internal/core"
    cli "github.com/drand/drand/internal/drand-cli"
    "github.com/drand/drand/internal/test"
```

```
    "github.com/stretchr/testify/require"
    "os"
    "path"
    "strconv"
    "testing"
)

func TestBackSlackBeaconId(t *testing.T) {
    lg := testlogger.New(t)
    sch, err := crypto.GetSchemeFromEnv()
    require.NoError(t, err)
    beaconID := "back/slack"

    tmpPath := path.Join(t.TempDir(), "drand")
    require.NoError(t, os.Mkdir(tmpPath, 0o740))

    pubPath := path.Join(tmpPath, "pub.key")
    port1, _ := strconv.Atoi(test.FreePort())
    addr := "127.0.0.1:" + strconv.Itoa(port1)

    ctrlPort1, metricsPort := test.FreePort(), test.FreePort()

    priv, err := key.NewKeyPair(addr, sch)
    require.NoError(t, err)
    require.NoError(t, key.Save(pubPath, priv.Public, false))

    config := core.NewConfig(lg, core.WithConfigFolder(tmpPath))
    fileStore := key.NewFileStore(config.ConfigFolderMB(), beaconID)
    require.NoError(t, fileStore.SaveKeyPair(priv))

    startArgs := []string{
        "drand",
        "start",
        "--private-listen", priv.Public.Address(),
        "--verbose",
        "--folder", tmpPath,
        "--control", ctrlPort1,
        "--metrics", "127.0.0.1:" + metricsPort,
    }

    if err = cli.CLI().Run(startArgs); err != nil {
        t.Errorf(err.Error())
    }
}
```

6. `del-beacon` cmd allows the use of negative `startRound` [REPORTED]

Executive Summary

During the audit of drand, we found the `del-beacon` doesn't check for negative `startRound`

Vulnerability Details

- **Severity:** Informal

- **Affected Component:** `internal/drand-cli/cli.go`

link: https://github.com/drand/drand/blob/84ef2452fb2847c8ed252c8fde607801dfc48847/internal/drand-cli/cli.go#L885

Root Cause Analysis

The root cause is located in `/internal/drand-cli/cli.go:885`

```
func deleteBeaconCmd(c *cli.Context, l log.Logger) error {
    conf := contextToConfig(c, l)
    ctx := c.Context

    startRoundStr := c.Args().First()
    sr, err := strconv.Atoi(startRoundStr)
    if err != nil {
            return fmt.Errorf("given round not valid: %d", sr)
    }

    startRound := uint64(sr)
    ...
}
```

The issue with this snippet is that the startRound is converted to `int` and then cast to `uint64` without checking if it's not negative.

Reproducer

```
package lab

import (
    "context"
    "github.com/drand/drand/common"
    "github.com/drand/drand/common/testlogger"
    "github.com/drand/drand/crypto"
    "github.com/drand/drand/internal/chain"
    "github.com/drand/drand/internal/chain/boltdb"
    "github.com/drand/drand/internal/core"
    cli "github.com/drand/drand/internal/drand-cli"
    "github.com/drand/drand/internal/fs"
    "github.com/drand/drand/internal/test"
    "github.com/stretchr/testify/require"
    "path"
    "testing"
)

func TestDeleteBeaconNegativeRound(t *testing.T) {
    beaconID := test.GetBeaconIDFromEnv()
    l := testlogger.New(t)
    ctx := context.Background()
    sch, err := crypto.GetSchemeFromEnv()
    require.NoError(t, err)
    if sch.Name == crypto.DefaultSchemeID {
        ctx = chain.SetPreviousRequiredOnContext(ctx)
    }
    tmp := path.Join(t.TempDir(), "drand")
```

```
    opt := core.WithConfigFolder(tmp)
    conf := core.NewConfig(l, opt)
    fs.CreateSecureFolder(conf.DBFolder(beaconID))
    store, err := boltdb.NewBoltStore(ctx, l, conf.DBFolder(beaconID), conf.BoltOptions())
    require.NoError(t, err)
    err = store.Put(ctx, &common.Beacon{
        Round:     1,
        Signature: []byte("Hello"),
    })
    require.NoError(t, err)
    err = store.Put(ctx, &common.Beacon{
        Round:     2,
        Signature: []byte("Hello"),
    })
    require.NoError(t, err)
    err = store.Put(ctx, &common.Beacon{
        Round:     3,
        Signature: []byte("Hello"),
    })
    require.NoError(t, err)
    err = store.Put(ctx, &common.Beacon{
        Round:     4,
        Signature: []byte("hello"),
    })
    require.NoError(t, err)
    b, err := store.Get(ctx, 3)
    require.NoError(t, err)
    require.NotNil(t, b)
    b, err = store.Get(ctx, 4)
    require.NoError(t, err)
    require.NotNil(t, b)

    err = store.Close()
    require.NoError(t, err)

    args := []string{"drand", "util", "del-beacon", "--folder", tmp, "--id", beaconID, "--", "-3
    app := cli.CLI()
    require.Error(t, app.Run(args))
}
```

7. gRPC servers are vulnerable to denial of service via HTTP/2 rapid reset [REPORTED]

Executive Summary

During the audit of drand, `govulncheck` matched the GO-2023-2153 vulnerability.

Vulnerability Details

- **Severity:** Critical
- **Affected Components:**

- `internal/net/control.go`
- `internal/net/listener.go`

links:

- https://github.com/drand/drand/blob/84ef2452fb2847c8ed252c8fde607801dfc48847/internal/net/control.go#L50
- https://github.com/drand/drand/blob/84ef2452fb2847c8ed252c8fde607801dfc48847/internal/net/listener.go#L64

Root Cause Analysis

An attacker can send HTTP/2 requests, cancel them, and send subsequent requests.

This is valid by the HTTP/2 protocol but would cause the gRPC-Go server to launch more concurrent method handlers than the configured maximum stream limit, `grpc.MaxConcurrentStreams`.

This results in a **denial of service** due to **resource consumption**.

8. Control servers allow insecure connection [REPORTED]

Executive Summary

During the audit of drand, `semgrep` matched the grpc-server-insecure-connection rule.

Vulnerability Details

- **Severity:** Undetermined
- **Affected Component:** `internal/net/control.go`

link: https://github.com/drand/drand/blob/84ef2452fb2847c8ed252c8fde607801dfc48847/internal/net/control.go#L30

Root Cause Analysis

```
package net
...
func NewGRPCListener(l log.Logger, s Service, controlAddr string) (ControlListener, error) {
    grpcServer := grpc.NewServer()
    lis, err := newListener(controlAddr)
    if err != nil {
        l.Errorw("", "grpc listener", "failure", "err", err)
        return ControlListener{}, err
    }

    control.RegisterControlServer(grpcServer, s)
    control.RegisterDKGControlServer(grpcServer, s)

    return ControlListener{log: l, conns: grpcServer, lis: lis}, nil
}
```

The `net.NewGRPCListener` function is used to create a new listener for the gRPC server. The returned listener is not configured with encryption.

A malicious attacker could tamper with the gRPC message with a man-in-the-middle attack.

9. Timeout during DKG ceremony

Due to the limited development time of the `network_fuzzer`, this crash can be a false positive.

But we think that it shouldn't happen since there are 10 normal nodes and 1 fuzzy node and a DKG timeout of 5 minutes.

Executive Summary

Vulnerability Details

- **Severity:** Undetermined
- **Affected Component:** Undetermined

Reproducer

The seed can help to reproduce these crashes just for replaying the scenario with the same config, but crashes happen randomly

Seed:
`0a10101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101`

Replay with:

```
FUZZ_DEBUG_LOG=y ./replay_seed.py e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0
```

Trace

```
[+] Renew the global network_fuzzer generator with a new seed: e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e
[+] The panic log file is /tmp/drand-fuzzinglabs-1705576082574868452/panic.log
[+] The error log file is /tmp/drand-fuzzinglabs-1705576082574868452/error.log
[+] Simulation global folder: /tmp/drand-fuzzinglabs-1705576082574868452
    - Created Basic node 127.0.0.1:40659 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:38333 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:44351 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:34947 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:41265 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:36409 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:35675 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:36607 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:33775 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Basic node 127.0.0.1:41185 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
    - Created Fuzzy node 127.0.0.1:42295 at /tmp/drand-fuzzinglabs-1705576082574868452 --> ctrl
[+] Scenario: github.com/drand/drand/network_fuzzer.nodeFailureScenario
[+] Starting nodes
[+] Starting all nodes
...
[+] Running DKG for all nodes
...
[+] Waiting for DKG completion
...
panic: [-] Error running DKG with timeout 300.000000 seconds and leaderId 5 in nodeFailureScenar
```

10. The leader cannot propose the DKG reshare ceremony

Due to the limited development time of the `network_fuzzer`, this crash can be a false positive.

But we think that it shouldn't happen since there are 10 normal nodes and 1 fuzzy node.

Executive Summary

Vulnerability Details

- **Severity:** Undetermined
- **Affected Component:** Undetermined

Reproducer

The seed can help to reproduce these crashes just for replaying the scenario with the same config, but crashes happen randomly.

Seed:

`e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e`

Replay with:

```
FUZZ_DEBUG_LOG=y ./replay_seed.py e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0
```

Trace

```
[+] Renew the global network_fuzzer generator with a new seed: e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e
[+] The panic log file is /tmp/drand-fuzzinglabs-1705594143060552082/panic.log
[+] The error log file is /tmp/drand-fuzzinglabs-1705594143060552082/error.log
[+] Simulation global folder: /tmp/drand-fuzzinglabs-1705594143060552082
    - Created Basic node 127.0.0.1:44607 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:40767 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:39963 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:38235 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:45439 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:43731 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:39269 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:35573 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Basic node 127.0.0.1:41841 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
    - Created Fuzzy node 127.0.0.1:37089 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
[+] Scenario: github.com/drand/drand/network_fuzzer.nodeDkgReshareScenario
[+] Starting nodes
[+] Starting all nodes
...
[+] Running DKG for all nodes
...
[+] Waiting for DKG completion
...
[+] Nodes finished running DKG. Checking keys...
[+] Checking if chain info is present on all nodes...
...
[+] Chain info are present on all nodes. DKG finished.
[+] Checking all created group file with collective key
...
[+] Sleeping -16 until genesis happens
[+] Sleeping 3s after genesis - leaving some time for rounds
[+] Sleeping 1s to reach round 8 [period 3.000000, current 8]
[+] Checking randomness beacon for round 8 via CLI
...
[+] Setting up 1 new nodes for resharing
    - Created Basic node 127.0.0.1:39137 at /tmp/drand-fuzzinglabs-1705594143060552082 --> ctrl
```

```
[+] Starting all nodes
...
[+] Setting up the nodes for the resharing
...
[+] Stopping old nodes
[+] Running DKG for resharing nodes
...
panic: leader.StartLeaderReshare: rpc error: code = Unknown desc = cannot make a proposal where
```

11. Invalid transition during DKG reshare ceremony

Due to the limited development time of the `network_fuzzer`, this crash can be a false positive.

But we think that it shouldn't happen since there are 12 normal nodes, 3 new normal nodes for resharing and 1 fuzzy node.

Executive Summary

Vulnerability Details

- **Severity:** Undetermined
- **Affected Component:** Undetermined

Reproducer

The seed can help to reproduce these crashes just for replaying the scenario with the same config, but crashes happen randomly.

Seed: `8264646464642b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b64646564646464646464`

Replay with:

```
FUZZ_DEBUG_LOG=y ./replay_seed.py 8264646464642b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b
```

Trace

```
[+] Renew the global network_fuzzer generator with a new seed: 8264646464642b2b2b2b2b2b2b2b2
[+] The panic log file is /tmp/drand-fuzzinglabs-1705598682079267604/panic.log
[+] The error log file is /tmp/drand-fuzzinglabs-1705598682079267604/error.log
[+] Simulation global folder: /tmp/drand-fuzzinglabs-1705598682079267604
    - Created Basic node 127.0.0.1:36149 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:41797 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:39321 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:41579 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:39629 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:46569 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:36815 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:38069 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:37343 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:39921 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:45511 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:35203 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Fuzzy node 127.0.0.1:34273 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
[+] Scenario: github.com/drand/drand/network_fuzzer.nodeDkgReshareScenario
[+] Starting nodes
```

```
[+] Starting all nodes
...
[+] Running DKG for all nodes
...
[+] Waiting for DKG completion
...
[+] Nodes finished running DKG. Checking keys...
[+] Checking if chain info is present on all nodes...
...
[+] Chain info are present on all nodes. DKG finished.
[+] Checking all created group file with collective key
...
[+] Sleeping -25 until genesis happens
[+] Sleeping 3s after genesis - leaving some time for rounds
[+] Sleeping 1s to reach round 4 [period 10.000000, current 4]
[+] Checking randomness beacon for round 4 via CLI
...
[+] Setting up 3 new nodes for resharing
    - Created Basic node 127.0.0.1:39431 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:40223 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
    - Created Basic node 127.0.0.1:43735 at /tmp/drand-fuzzinglabs-1705598682079267604 --> ctrl
[+] Starting all nodes
...
[+] Setting up the nodes for the resharing
...
[+] Stopping old nodes
[+] Running DKG for resharing nodes
...
panic: leader.StartLeaderReshare: rpc error: code = Unknown desc = invalid transition attempt fr
```

The fuzzer produce other panic like:

- `panic: Failed to ping all nodes in 120.000000 seconds.`
- `panic: [-] Error running DKG with timeout 15.000000 seconds and leaderId 1 in nodeFailureScenario: leader.WaitDKGComplete: DKG never finished`
- `panic: [-] Node 127.0.0.1:44833 has different cokey than 127.0.0.1:45623`

11) Conclusion

Project Architecture:

The project's architecture is well-organized and easily comprehensible. During our review, we identified an unnecessary folder that should be investigated further. It is recommended to validate the contents of the obsolete package, especially in the context of the significant refactoring currently underway.

Code Quality:

The codebase exhibits high-quality standards, characterized by cleanliness and brevity. Function names are clear and concise, contributing to the overall readability of the code. Additionally, the inclusion of a demo folder proved beneficial,

facilitating the creation of an attacknet project. It is noted that a majority of the unit tests were found to be useful, enhancing the robustness of the codebase.

Limit of the fuzzing harnesses

The evaluation of the project involved the creation of fuzzing harnesses based on Drand unit tests, presenting various challenges in the fuzzing process. The predominant protocol-based nature of the code, coupled with minimal parsing, hinders the effectiveness of traditional fuzzing methodologies. Notably, the system's reliance on dynamically generated structures during execution complicates the fuzzing endeavor, necessitating manual generation of these structures with potential risks of encountering false positives. Concerns were identified, such as functions not adequately verifying whether a field is different from Nil before utilization, highlighting areas for further attention and improvement in the system's robustness.

Incorporating the harnesses into your CI/CD pipeline will utilize GitHub Actions, but the expected outcomes may not justify the significant resource investment.

Fuzzy_Drand: The Network Fuzzer

The Network Fuzzer represents a substantial project initiated to identify additional vulnerabilities. The primary objective is the development of a Chaos Node, and progress is well underway in achieving this goal. Fuzzy_Drand is designed to systematically fuzz each component of drand during its execution, intercepting communication and introducing mutations. Our team is content with this implementation, appreciating its discoveries and its capability to fuzz drand components without requiring modifications to the original code. Fuzzy_Drand also possesses the capability to generate coverage reports for the fuzzing sessions, providing insights into potential scenarios that can be further explored.

12) Proposal for Tool Development: Network Fuzzer Enhancement

Objective:

To enhance the capabilities of Fuzzy_Drand, we propose the development of new features aimed at providing advanced functionalities for network testing, monitoring, and detection of abnormal behaviors.

Proposed Features:

1. **Dynamic Network Spinning:**
    - Implement the ability to spin up multiple networks on the fly.
    - Enable quick and flexible network configurations for comprehensive testing scenarios.
2. **Monitoring/Scanner for Network State and Nodes:**
    - Integrate a robust monitoring and scanning mechanism to assess the state of networks and nodes.
    - Provide real-time insights into performance metrics, health status, and potential vulnerabilities.
3. **Chaos Network Integration:**
    - Develop a Chaos Network feature to simulate real-world chaotic conditions.
    - Introduce controlled chaos scenarios to evaluate system resilience and response under adverse conditions.
    - Simulate random AWS chaos engineering during the session

4. **Behavioral Anomaly Detector:**
    - Create a detector for identifying unusual behaviors, including but not limited to:
        - High CPU usage
        - Out-of-Memory (OOM) conditions
        - Denial-of-Service (DoS) attacks
        - Unresponsive nodes or components
    - Implement intelligent algorithms to detect and report deviations from normal behavior.

Benefits:

- **Comprehensive Testing Environment:** The ability to spin up diverse networks on demand facilitates thorough testing of the system under various conditions.
- **Real-time Monitoring:** Enhanced monitoring capabilities provide immediate insights into the health and performance of networks and nodes, aiding in proactive issue identification.
- **Chaos Simulation:** The introduction of controlled chaos scenarios ensures the robustness of the system under challenging conditions, helping identify and address potential vulnerabilities.
- **Early Detection of Anomalies:** The behavioral anomaly detector adds a layer of security by promptly identifying and reporting unusual behaviors, mitigating the risk of system compromise.

Next steps:

1) Review the first version

2) Implementing new scenarios

- https://chaos-mesh.org/docs/basic-features/

3) Improving Fuzzing Speed and mutation

4) Implementing the monitoring

5) Implementing the Anomaly Detector

je suis sur que