Protocol Labs

Drand Security Assessment #2

Version: 1.0

Contents

Introduction

Protocol Labs is a research, development, and deployment institution for improving Internet technology. Protocol Labs leads groundbreaking internet projects, such as IPFS, a decentralized web protocol; and libp2p, a modular network stack for peer-to-peer applications.

Drand is a distributed randomness generator developed in Golang which provides unpredictable, unbiased, publicly verifiable random numbers at regular intervals, using bilinear pairings and threshold cryptography. Each node can also create locally-generated private randomness.

Sigma Prime was approached by Protocol Labs to perform a second security assessment of Drand.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the assessed system. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Drand implementation contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given, which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/-closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities found within the code.

Overview

The **Drand** network aims at providing a continuous, reliable source of randomness that can be used in various types of applications (e.g. lottery, *onchain* consensus mechanisms, elections, etc.). By relying on different advanced cyrptographic primitives such as threshold BLS signatures, *Pedersen* secret sharing, and Elliptic Curve Integrated Encryption Scheme (ECIES), participating nodes generate portions of randomness that can be combined into a single publicly verifiable random string.

The randomness generation process relies on two distinct core components:

- **Distributed Key Generation:** Initial set up phase responsible for creating a distributed secret key. Each participating node generates a public/private key pair, shared amongst the other nodes and combined in a `group.toml` file. After the successful completion of the distributed key generation phase, a collective public key is formed, along with one corresponding private key share per participant. Nodes do not have access to the actual corresponding global private key. Instead, they leverage their respective private key share to contribute to the public randomness generation process.

- **Threshold Signature Scheme:** After the Distributed Key Generation phase, Drand nodes can start producing verifiable, distributed and tamper-resistant randomness, by periodically broadcasting a partial signature over a commonly shared input (current round number and previous signature). Nodes wait to receive these partial signatures and can reconstruct the final signature as soon as the minimum threshold is received. The final signature is a regular BLS signature that can be verified against the distributed public key.

In addition to providing a distributed, continuous source of entropy to be consumed by various applications (e.g. the Filecoin network), Drand nodes can be queried locally to provide private randomness, in the form of a 32-byte hex-encoded random value (generated using the `crypto/rand` Golang package).

Security Assessment Summary

This review was initially conducted on commit 8ff83bc and targeted the following components:

- `client` package: Drand client library, containing the gRPC client and HTTP implementations;
- `net` package: `gRPC` service handlers for inter-node communication;
- `http` package: Publicly exposed HTTP server for exposing randomness;
- `lp2p` package: `gossipsub` relay/client;
- `fs` package: Utilities for state storage.

Fuzzing activities leveraging go-fuzz have been performed by the testing team in order to identify panics within the code in scope. `go-fuzz` is a coverage-guided tool which explores different code paths by mutating input to reach as many code paths possible. The aim is to find memory leaks, overflows, index out of bounds or any other panics.

Specifically, the testing team produced the following fuzzing targets:

- `chaininfojson`
- `ecieskey`
- `eciesmsg`
- `ed25519verify`
- `tblsandblsverify`
- `unmarshalbeacon`
- `unmarshalbeaconpacket`
- `unmarshalchaininfopacket`
- `unmarshalchaininforequest`
- `unmarshalcokeyrequest`
- `unmarshalcokeyresponse`
- `unmarshaldkginfopacket`
- `unmarshaldkgpacket`
- `unmarshaled25519privatekey`
- `unmarshaled25519publickey`
- `unmarshalempty`
- `unmarshalentropyinfo`
- `unmarshalfollowprogress`
- `unmarshalg1`
- `unmarshalg2`
- `unmarshalgroupinfo`
- `unmarshalgrouppacket`
- `unmarshalgrouprequest`
- `unmarshalgrouptomlresponse`
- `unmarshalhomerequest`
- `unmarshalhomeresponse`
- `unmarshalidentity`
- `unmarshalidentityrequest`
- `unmarshalinitdkgpacket`
- `unmarshalinitresharepacket`
- `unmarshalnode`
- `unmarshalpartialbeaconpacket`
- `unmarshalping`
- `unmarshalpong`
- `unmarshalprivatekeyrequest`
- `unmarshalprivatekeyresponse`

- `unmarshalprivaterandrequest`
- `unmarshalprivaterandresponse`
- `unmarshalpublickeyrequest`
- `unmarshalpublickeyresponse`
- `unmarshalpublicrandrequest`
- `unmarshalpublicrandresponse`
- `unmarshalsetupinfopacket`
- `unmarshalsharerequest`
- `unmarshalshareresponse`
- `unmarshalshutdownrequest`
- `unmarshalshutdownresponse`
- `unmarshalsignaldkgpacket`
- `unmarshalstartfollowrequest`
- `unmarshalsyncrequest`

These fuzzing targets have all been shared with the development as a byproduct of this security review. Execution and instrumentation can be done using a `Makefile`, by simply running `make run-fuzz-${TARGET-NAME}` (targets list and detailed instructions are available inside the `Makefile`).

Additionally, this assessment targeted the AWS architecture and infrastructure used by Drand, along with the deployment strategy to be followed by participants of the *League of Entropy*.

The testing team identified a total of eighteen (18) issues during this assessment, of which:

- One (1) is classified as high risk,
- Three (3) are classified as medium risk,
- Seven (7) are classified as low risk,
- Seven (7) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the scope of this review. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the code base, including comments not directly related to the security posture of Drand, are also described in this section and are labelled as *"informational"*.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team;
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk;
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
DRN-01	Unreleased Locks	High	Open
DRN-02	Node Addresses and TLS Values may be Arbitrarily Modified	Medium	Open
DRN-03	`DKGInfoPacket` May Be Replayed Between Rounds	Medium	Open
DRN-04	AWS Hardening: Identity and Access Management	Medium	Open
DRN-05	AWS Hardening: CloudTrail/CloudWatch Integration Disabled	Low	Open
DRN-06	AWS Hardening: Unencrypted CloudFront Traffic	Low	Open
DRN-07	AWS Hardening: Config Service Disabled	Low	Open
DRN-08	AWS Hardening: EBS Encryption Disabled	Low	Open
DRN-09	AWS Hardening: S3 Buckets Missing Versioning, Logging and Encryption	Low	Open
DRN-10	AWS Hardening: VPC Flow Logs Disabled	Low	Open
DRN-11	AWS Hardening: GuardDuty Disabled	Low	Open
DRN-12	Division by Zero for Small `Period`	Informational	Open
DRN-13	Inconsistent Units for Time	Informational	Resolved
DRN-14	Hard Coded Values In Filesystem	Informational	Resolved
DRN-15	Unhandled Error in `CreateSecureFolder()`	Informational	Open
DRN-16	Values Read after Releasing Lock	Informational	Open
DRN-17	External Library Issue Resolved	Informational	Open
DRN-18	Miscellaneous Observations	Informational	Open

DRN-01	Unreleased Locks		
Asset	`core/`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A range of locks are used to prevent race conditions over the core Drand chain and associated data structures. The locks are required due to multiple parallel requests from peers which may attempt to update the data concurrently.

There are three instances where these locks are not released before the code returns, thereby locking the state indefinitely. All future threads which require the lock will be waiting for a lock that will not be released.

The first lock affects `drand/core/drand_public.go` in the function `PublicRandStream()` and will not be released in the code below, if `d.beacon == nil`. This error can be reached by calling the public API `/health` before genesis.

```
76 d.state.Lock()
   if d.beacon == nil {
78   return errors.New("beacon has not started on this node yet")
   }
```

The second unreleased lock is in `drand/core/drand_control.go`. The function `setupAutomaticDKG()` will not release the lock on line [296] if it hits the return function on line [304].

The last unreleased lock can be found in `drand/core/drand_control.go` in the function `setupAutomaticResharing()`. The lock on line [375] is not released in the return statement on line [379].

Recommendations

The three sections of code mentioned above should be updated such that all locks are released even in the event or errors.

DRN-02	Node Addresses and TLS Values may be Arbitrarily Modified		
Asset	`key/keys.go`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The struct `Identity` contains a public key, node address, tls flag and a signature. The signature is only signed over the public key. Therefore the values `Addr` and `TLS` may be arbitrarily modified.

This allows the signature to be replayed for the `Identity` from previous messages/rounds.

During resharing rounds a malicious node could send a `SignalDKGPacket` on behalf of another node (using their public key and signature from a previous round) modifying the address and TLS values.

When the reshare is run, only one `PublicKey` is accepted. The public key from the first `SignalDKGPacket` with a valid signature will be the one included in the group.

If the address and TLS are set to invalid values (e.g. `Adress: 0.0.0.0`) then that node would not receive any of the messages during the reshare and be evicted from the group.

Recommendations

A possible mitigation would be to modify the signature in the `Identity` to sign over TLS and Address and add a timestamp (taking only the most recent timestamped version rather than the first), then update the group hash to ignore the signature, timestamp, tls and address fields.

Hence these fields are able to be updated without affecting the group hash, only if they have been signed by the node owning that key.

DRN-03	`DKGInfoPacket` May Be Replayed Between Rounds		
Asset	`core/group_setup.go`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The packet `DKGInfoPacket` is sent from the leader to participants during a reshare to establish the group. This packet may be replayed by any node on the network using any previous rounds packet.

The `DKGInfoPacket` includes a signature over the group hash. So long as the group is valid and the signature is from the leader this packet will validate. By resending the same `DKGInfoPacket` packet as last round to other peers, nodes will verify this packet as valid so long as the leader is the same.

The result is that the same group as the replayed round will be used for the reshare.

Recommendations

We recommend having the `DKGInfoPacket.Signature` sign over the remaining fields of the `DKGInfoPacket` and enforcing a different secret to be used each reshare.

Alternatively, add a timestamp to the packet and enforce a maximum duration between now and the timestamp to validate a packet.

DRN-04	AWS Hardening: Identity and Access Management		
Asset	Drand AWS Infrastructure		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The Identity and Access Manangemet (IAM) settings enforced on the AWS configuration are potentially exposing the Drand infrastructure to unauthorised access. Specifically:

- **AWS root account usage -** The AWS *Root* account is being used by system administrators;
- **Multi-Factor Authentication -** The *Root* accout uses a virtual MFA device;
- **Multiple Access Keys in Use -** The *willscott* account uses two different access keys.

Recommendations

- Do not use the AWS *root* account. Instead, create dedicated admin users, following the "Least Privilege" principle;
- Consider using a hardware MFA device for the *root* account;
- Restrict access key usage to one per user account.

DRN-05	AWS Hardening: CloudTrail/CloudWatch Integration Disabled		
Asset	Drand AWS Infrastructure		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Amazon CloudTrail records information about various actions performed on an AWS account. While these actions are currently being logged to an S3 bucket, the logs should also be integrated with Amazon CloudWatch so that suspicious activity will trigger alerts and system notifications. For this to be useful, appropriate metrics must be configured in CloudWatch which trigger notifications on events such as network ACL changes, policy changes, and sign-in failures.

Furthermore, CloudTrail log file integrity validation is not enabled on the Drand AWS infrastructure. CloudTrail file validation consists of a hash of the log file which can be used to ensure its integrity in the case of an account compromise, ensuring the logs generated by the AWS services have not been tampered with.

Recommendations

- Integrate Amazon CloudTrail with CloudWatch, and configure metrics so that notifications are raised upon suspicious activity;
- Enable CloudTrail log file validation.

DRN-06	AWS Hardening: Unencrypted CloudFront Traffic		
Asset	Drand AWS Infrastructure		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Traffic passed between the CloudFront edge nodes and the backend resources should be sent over HTTPS with modern protocols for all web-based origins (TLSv1.1 or higher).

The testing team observed that most CloudFront distribution endpoints are configured to only support plain, unencrypted HTTP traffic.

Additionally, CloudFront logging is disabled for the following resources:

- `arn:aws:cloudfront::373277216082:distribution/E2YEJTR553SY38`
- `arn:aws:cloudfront::373277216082:distribution/E14F4XOWX17XOR`

Recommendations

- Enable and enforce encryption in transit by requiring HTTPS connections between CloudFront and viewers;
- Enable Logging on all relevant resources.

DRN-07	AWS Hardening: Config Service Disabled		
Asset	Drand AWS Infrastructure		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The AWS Config service records the configuration of AWS resources and maintains a history of changes. This is useful for compliance and is invaluable for performing forensics after a security incident.

This service is not enabled on the Drand AWS infrastructure.

Recommendations

We recommend enabling and configuring the AWS Config service.

DRN-08	AWS Hardening: EBS Encryption Disabled		
Asset	Drand AWS Infrastructure		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Elastic Block Store (EBS) volumes in use within Drand's AWS infrastructure host unencrypted data at rest. Furthermore, EBS-backed Amazong Machine Images (AMIs) are not configured to use encryption.

Recommendations

We recommend enabling encryption of EBS volumes (Data at rest) along with configuring EBS-backed AMIs to use encryption.

DRN-09	AWS Hardening: S3 Buckets Missing Versioning, Logging and Encryption		
Asset	Drand AWS Infrastructure		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

S3 buckets on the Drand infrastructure are missing the following security hardening settings:

- **Encryption in transit -** S3 buckets allow unencrypted, plaintext HTTP connections;
- **Logging -** S3 buckets do not enable logging;
- **Versioning -** Object versioning is not enabled on S3 buckets.

Recommendations

- Add statements to the bucket policy that deny all S3 actions when `SecureTransport` is false;
- Ensure S3 bucket logging is enabled for S3 buckets;
- Enable object versioning on all S3 buckets to protect against the overwriting of objects.

DRN-10	AWS Hardening: VPC Flow Logs Disabled		
Asset	Drand AWS Infrastructure		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

VPC Flow Logs is an AWS feature that enables to capture information about the IP traffic going to/from network interfaces in a given VPC. Flow log data can be published to Amazon CloudWatch Logs or Amazon S3 buckets.

Flow logs can help with:

- Diagnosing overly restrictive security group rules;
- Monitoring the traffic that is reaching your instance;
- Determining the direction of the traffic to and from the network interfaces.

Flow log data is collected outside of the path of standard network traffic, and therefore does not affect network throughput or latency.

The testing team observed that VPC Flow Logs are disabled on the Drand AWS infrastructure.

Recommendations

We recommend enabling and configuring the AWS VPC Flow Logs.

DRN-11	AWS Hardening: GuardDuty Disabled		
Asset	Drand AWS Infrastructure		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

GuardDuty is a threat detection service for AWS that provides continuous monitoring for potentially malicious activity and unauthorized behavior.

GuardDuty analyzes logs from various AWS services (IAM, CloudTrail, VPC Flow Logs, DNS logs, etc.) to assess the security posture of an AWS cloud infrastructure and alert administrators in case any anomalies are observed.

GuardDuty is not currently enabled on the Drand AWS infrastructure.

Recommendations

Consider enabling GuardDuty on the Drand AWS infrastructure.

DRN-12	Division by Zero for Small `Period`
Asset	`core/group_setup.go`
Status	Open
Rating	Informational

Description

There is a potential division by zero when performing a remainder calculation in the function `createAndSend()`.

```
ps := int64(s.beaconPeriod.Seconds())
genesis += (ps - genesis%ps)
```
226

In the above code, if `beaconPeriod` (which is positive) is less than one second (e.g. 1ms), then the value of `ps` will be set to zero. The calculation `genesis%ps` will then perform a division by zero causing the Drand node to panic.

Recommendations

Consider disallowing setting the initial parameters to have a period of less than one second.

DRN-13	Inconsistent Units for Time
Asset	`drand/chain/time.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `TimeOfRound()` uses a combination of nanoseconds and seconds to represent time, potentially causing unexpected behaviour.

The call to `periodBits := math.Log2(float64(period))` finds the number of bits required to represent `period` in nanoseconds.

Later `period` is used as seconds in multiplication `(round - 1) * uint64(period.Seconds())`.

The check `if round > (math.MaxUint64 » int(periodBits))` is over accounting for the number of bits to prevent a multiplication overflow by $10^3 \approx 2^{30}$. Hence `periodBits` is over stated by 30 bits.

As `round` is a `uint64` this means it has a max value of about 2^{34} (34 bits). Setting a 10 second period 2^{34} rounds equates to about 608 years, which is unlikely to be an issue.

Recommendations

We recommend updating the `periodBits` to be in seconds rather than nanoseconds and accounting for potential rounding issues for casting a `float64` into an `uint64`. Furthermore, the value will later be cast into an `int64` so an additional bit must be accounted for.

Resolution

This issue has been addressed in PR #742.

DRN-14	Hard Coded Values In Filesystem
Asset	`fs/fs.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

File permissions use direct values rather than constants. For example the value `0740` is used in some cases rather than the constant `defaultDirectoryPermission = 0740`.

The permissions for read-write files (`0600`) are also hard coded and do not use constants.

Additionally, there are minor spelling and grammar issues in the comments.

Recommendations

We recommend using constants and fix spelling and grammar issues.

Resolution

This issue has been addressed in PR #738.

DRN-15	Unhandled Error in `CreateSecureFolder()`
Asset	`fs/fs.go`
Status	Open
Rating	Informational

Description

There is an unhandled error in `CreateSecureFolder()`. The error value on line [26] is not handled when `Exists()` returns `true, err`.

The likelihood of `Exists()` returning `true, err` is very low but may lead to unexpected behaviour.

Recommendations

We recommend handling all errors if it is possible for them to occur.

DRN-16	Values Read after Releasing Lock
Asset	`http/server.go`
Status	Open
Rating	Informational

Description

A value is read after a read lock is released. This has the potential to allow the data to be modified before the data is re-read.

Note that exploitation of this issue would require the data to be modified to a nil-pointer which should not occur in the code.

```
152 h.chainInfoLk.RUnlock()
    return h.chainInfo
```

Recommendations

We recommend storing the value of `h.chainInfo` in a local variable before releasing the lock.

DRN-17	External Library Issue Resolved
Asset	lp2p/ctor.go
Status	Open
Rating	Informational

Description

In `ConstructHost()`, line [81] has been commented out due to a potential race condition in `libp2p.Peerstore`. The issue was reported and fixed upstream in the lp2p repository as seen here.

Recommendations

Update the Drand codebase to reflect the fix made upstream.

DRN-18	Miscellaneous Observations
Asset	`drand/`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have a direct security implication:

- The file `net/json_marshaller.go` is not used. Consider removing it from the repository.
- Running `make test` fails as some folders do not contain tests. This has been fixed on the current master.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

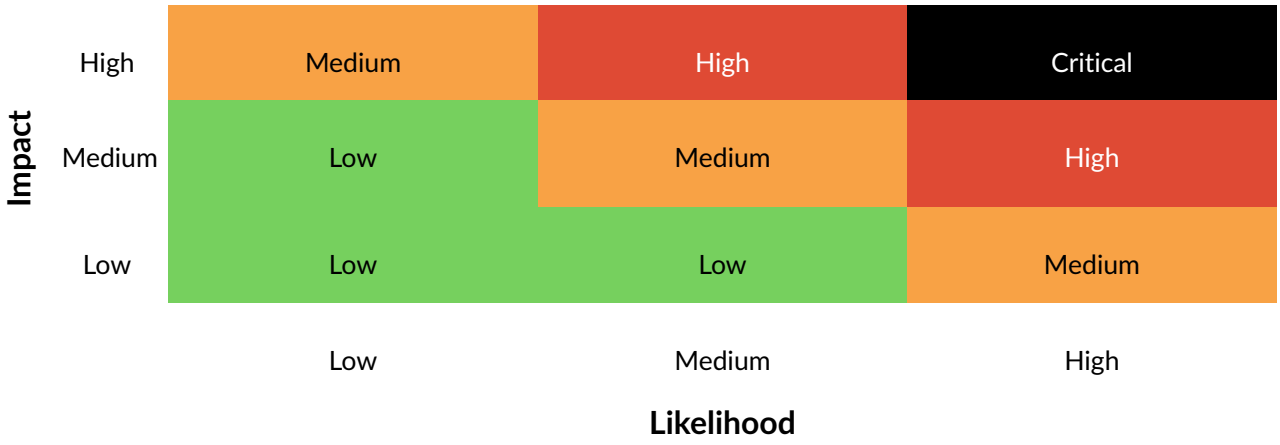

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.